Deep Learning for Time Series Forecasting

Predict the Future with MLPs, CNNs and LSTMs in Python

MACHINE
LEARNING
MASTERY

Contents

VIII   Conclusions   555

How Far You Have Come   556

Preface

Perhaps the topic that I am most asked about is how to use deep learning methods for time series forecasting. Questions range from how to prepare data for deep learning models to what models to use for specific types of forecasting problems. This book was carefully designed to address these questions and show you exactly how to apply deep learning methods to time series forecasting problems.

In writing this book, I imagined that you were provided with a dataset and a desire to use deep learning methods to address it. I designed the chapters to walk you through the process of first establishing a baseline of performance with naive and classical methods. I then provide step-by-step tutorials to show exactly how to develop a suite of different types of neural network models for time series forecasting. After we cover these basics, I then hammer home how to use them on real-world datasets with example after example on larger projects. This is not a book for beginners. The focus on deep learning methods means that we won't focus on many other important areas of time series forecasting, such as data visualization, how classical methods work, the development of machine learning solutions, or even depth and details on how the deep learning methods work. I assume that you are familiar with these introductory topics.

Deep learning methods do offer a lot of promise for time series forecasting, specifically the automatic learning of temporal dependence and the automatic handling of temporal structures like trends and seasonality. Unfortunately, deep learning methods are often developed for univariate time series forecasting problems and often perform worse than classical and even naive forecasting methods. This has led to their general dismissal as being unsuitable for time series forecasting in general. Nevertheless, deep learning methods are effective at more complex time series forecasting problems that involve large amounts of data, multiple variates with complex relationships, and even multi-step and time series classification tasks.

A limitation I saw in the adoption of deep learning methods for time series forecasting was in the exclusive exploration of recurrent neural networks, such as LSTM networks. These methods can work well in some situations, but can often perform better when used in hybrid models with CNNs or other variations. Therefore, I have ensured that examples of each class of neural network and hybrid models were demonstrated throughout the book. In addition to providing a playbook to show you how to develop deep learning models for your own time series forecasting problems, I designed this book to highlight the areas where deep learning methods may show the most promise. Deep learning may be the future of complex and challenging time series forecasting and I think this book will help you get started and make rapid progress on your own forecasting problems. I hope that you agree and are as excited as I am about the journey ahead.

Jason Brownlee
2018

Part I

Introduction

Welcome

Welcome to *Deep Learning for Time Series Forecasting*. Deep learning methods, such as Multilayer Perceptrons, Convolutional Neural Networks, and Long Short-Term Memory Networks, can be used to automatically learn the temporal dependence structures for challenging time series forecasting problems. Neural networks may not be the best solution for all time series forecasting problems, but for those problems where classical methods fail and machine learning methods require elaborate feature engineering, deep learning methods can be used with great success. This book was designed to teach you, step-by-step, how to develop deep learning methods for time series forecasting with concrete and executable examples in Python.

Who Is This Book For?

Before we get started, let's make sure you are in the right place. This book is for developers that may know some applied machine learning. Maybe you know how to work through a predictive modeling problem end-to-end, or at least most of the main steps, with popular tools. The lessons in this book do assume a few things about you, such as:

- You know your way around basic Python for programming.

- You know some basic time series forecasting with classical methods such as naive forecasts, ARIMA, or exponential smoothing.

- You know how to work through a predictive modeling problem using machine learning or deep learning methods and libraries such as scikit-learn or Keras.

- You have some background knowledge on how neural network models work, perhaps at a high level.

- You want to learn how to develop a suite of deep learning methods to get more from new or existing time series forecasting problems.

This guide was written in the top-down and results-first machine learning style that you're used to from Machine Learning Mastery.

About Your Outcomes

This book will teach you the practical deep learning skills that you need to know for time series forecasting. After reading and working through this book, you will know:

- About the promise of neural networks and deep learning methods in general for time series forecasting.

- How to transform time series data in order to train a supervised learning algorithm, such as deep learning methods.

- How to develop baseline forecasts using naive and classical methods by which to determine whether forecasts from deep learning models have skill or not.

- How to develop Multilayer Perceptron, Convolutional Neural Network, Long Short-Term Memory Networks, and hybrid neural network models for time series forecasting.

- How to forecast univariate, multivariate, multi-step, and multivariate multi-step time series forecasting problems in general.

- How to transform sequence data into a three-dimensional structure in order to train convolutional and LSTM neural network models.

- How to grid search deep learning model hyperparameters to ensure that you are getting good performance from a given model.

- How to prepare data and develop deep learning models for forecasting a range of univariate time series problems with different temporal structures.

- How to prepare data and develop deep learning models for multi-step forecasting a real-world household electricity consumption dataset.

- How to prepare data and develop deep learning models for a real-world human activity recognition project.

This new understanding of applied deep learning methods will impact your practice of working through time series forecasting problems in the following ways:

- Confidently use naive and classical methods like SARIMA and ETS to quickly develop robust baseline models for a range of different time series forecasting problems, the performance of which can be used to challenge whether more elaborate machine learning and deep learning models are adding value.

- Transform native time series forecasting data into a form for fitting supervised learning algorithms and confidently tune the amount of lag observations and framing of the prediction problem.

- Develop MLP, CNN, RNN, and hybrid deep learning models quickly for a range of different time series forecasting problems, and confidently evaluate and interpret their performance.

This book is not a substitute for an undergraduate course in deep learning or time series forecasting, nor is it a textbook for such courses, although it could be a useful complement. For a good list of top courses, textbooks, and other resources on statistics, see the Further Reading section at the end of each tutorial.

How to Read This Book

This book was written to be read linearly, from start to finish. That being said, if you know the basics and need help with a specific method or type of problem, then you can flip straight to that section and get started. This book was designed for you to read on your workstation, on the screen, not on a tablet or eReader. My hope is that you have the book open right next to your editor and run the examples as you read about them.

This book is not intended to be read passively or be placed in a folder as a reference text. It is a playbook, a workbook, and a guidebook intended for you to learn by doing and then apply your new understanding with working Python examples. To get the most out of the book, I would recommend playing with the examples in each tutorial. Extend them, break them, then fix them. Try some of the extensions presented at the end of each lesson and let me know how you do.

About the Book Structure

This book was designed around major deep learning techniques that are directly relevant to time series forecasting. There are a lot of things you could learn about deep learning and time series forecasting, from theory to abstract concepts to APIs. My goal is to take you straight to developing an intuition for the elements you must understand with laser-focused tutorials. I designed the tutorials to focus on how to get results with deep learning methods. The tutorials give you the tools to both rapidly understand and apply each technique or operation. There is a mixture of both tutorial lessons and projects to both introduce the methods and give plenty examples and opportunity to practice using them.

Each of the tutorials are designed to take you about one hour to read through and complete, excluding the extensions and further reading. You can choose to work through the lessons one per day, one per week, or at your own pace. I think momentum is critically important, and this book is intended to be read and used, not to sit idle. I would recommend picking a schedule and sticking to it. The tutorials are divided into five parts; they are:

- **Part 1: Foundations**. Provides a gentle introduction to the promise of deep learning methods for time series forecasting, a taxonomy of the types of time series forecasting problems, how to prepare time series data for supervised learning, and a high-level procedure for getting the best performing model on time series forecasting problems in general.

- **Part 2: Deep Learning Modeling**. Provides a step-by-step introduction to deep learning methods applied to different types of time series forecasting problems with additional tutorials to better understand the 3D-structure required for some models.

- **Part 3: Univariate Forecasting**. Provides a methodical approach to univariate time series forecasting with a focus on naive and classical methods that are generally known to out-perform deep learning methods and how to grid search deep learning model hyperparameters.

- **Part 4: Multi-step Forecasting**. Provides a step-by-step series of tutorials for working through a challenging multi-step time series forecasting problem for predicting household electricity consumption using classical and deep learning methods.

- **Part 5: Time Series Classification**. Provides a step-by-step series of tutorials for working through a challenging time series classification problem for predicting human activity from accelerometer data using deep learning methods.

Each part targets a specific learning outcome, and so does each tutorial within each part. This acts as a filter to ensure you are only focused on the things you need to know to get to a specific result and do not get bogged down in the math or near-infinite number of digressions. The tutorials were not designed to teach you everything there is to know about each of the methods. They were designed to give you an understanding of how they work, how to use them, and how to interpret the results the fastest way I know how: to learn by doing.

About Python Code Examples

The code examples were carefully designed to demonstrate the purpose of a given lesson. Code examples are complete and standalone. The code for each lesson will run as-is with no code from prior lessons or third-parties required beyond the installation of the required packages. A complete working example is presented with each tutorial for you to inspect and copy-paste. All source code is also provided with the book and I would recommend running the provided files whenever possible to avoid any copy-paste issues.

The provided code was developed in a text editor and intended to be run on the command line. No special IDE or notebooks are required. If you are using a more advanced development environment and are having trouble, try running the example from the command line instead. All code examples were tested on a POSIX-compatible machine with Python 3.

About Further Reading

Each lesson includes a list of further reading resources. This may include:

- Books and book chapters.

- API documentation.

- Articles and Webpages.

Wherever possible, I try to list and link to the relevant API documentation for key functions used in each lesson so you can learn more about them. I have tried to link to books on Amazon so that you can learn more about them. I don't know everything, and if you discover a good resource related to a given lesson, please let me know so I can update the book.

About Getting Help

You might need help along the way. Don't worry; you are not alone.

- **Help with a Technique?** If you need help with the technical aspects of a specific operation or technique, see the *Further Reading* section at the end of each tutorial.

- **Help with APIs?** If you need help with using the Keras library, see the list of resources in the *Further Reading* section at the end of each lesson, and also see *Appendix A*.

- **Help with your workstation?** If you need help setting up your environment, I would recommend using Anaconda and following my tutorial in *Appendix B*.

- **Help with practice problems?** If you are looking for test sets on which to practice developing deep learning models, you can see a prepared list of standard time series forecasting problems in *Appendix A*.

- Help in general? You can shoot me an email. My details are in *Appendix A*.

Summary

Are you ready? Let's dive in! Next up you will discover a gentle introduction to the promise of deep learning methods for time series forecasting.

Part II

Foundations

Overview

This part motivates the use of deep learning for time series forecasting and provides frameworks to allow you to work through a new forecasting problems systematically and ensure that your final model is robust and defensible. After reading the chapters in this part, you will know:

- The promise that the capabilities of deep learning methods offer for addressing challenging time series forecasting problems (Chapter 1).

- How to systematically identify the properties of a time series forecasting problem using a taxonomy and framework of questions (Chapter 2).

- How to systematically work through a new time series forecasting problem to ensure that you are getting the most out of naive, classical, machine learning and deep learning forecasting methods (Chapter 3).

- How to generally transform a sequence of observations in a time series into samples with input and output elements suitable for training a supervised learning algorithm (Chapter 4).

- How naive and the top performing classical methods such as SARIMA and ETS work and how to use them prior to exploring more advanced forecasting methods (Chapter 5).

Chapter 1

Promise of Deep Learning for Time Series Forecasting

Deep learning neural networks are able to automatically learn arbitrary complex mappings from inputs to outputs and support multiple inputs and outputs. These are powerful features that offer a lot of promise for time series forecasting, particularly on problems with complex-nonlinear dependencies, multivalent inputs, and multi-step forecasting. These features along with the capabilities of more modern neural networks may offer great promise such as the automatic feature learning provided by convolutional neural networks and the native support for sequence data in recurrent neural networks. In this tutorial, you will discover the promised capabilities of deep learning neural networks for time series forecasting. After reading this tutorial, you will know:

- The focus and implicit, if not explicit, limitations on classical time series forecasting methods.

- The general capabilities of Multilayer Perceptrons and how they may be harnessed for time series forecasting.

- The added capabilities of feature learning and native support for sequences provided by Convolutional Neural Networks and Recurrent Neural Networks.

Let's get started.

1.1   Time Series Forecasting

Time series forecasting is difficult. Unlike the simpler problems of classification and regression, time series problems add the complexity of order or temporal dependence between observations. This can be difficult as specialized handling of the data is required when fitting and evaluating models. This temporal structure can also aid in modeling, providing additional structure like trends and seasonality that can be leveraged to improve model skill. Traditionally, time series forecasting has been dominated by linear methods like ARIMA because they are well understood and effective on many problems. But these classical methods also suffer from some limitations, such as:

- **Focus on complete data**: missing or corrupt data is generally unsupported.

- **Focus on linear relationships**: assuming a linear relationship excludes more complex joint distributions.

- **Focus on fixed temporal dependence**: the relationship between observations at different times, and in turn the number of lag observations provided as input, must be diagnosed and specified.

- **Focus on univariate data**: many real-world problems have multiple input variables.

- **Focus on one-step forecasts**: many real-world problems require forecasts with a long time horizon.

Machine learning methods can be effective on more complex time series forecasting problems with multiple input variables, complex nonlinear relationships, and missing data. In order to perform well, these methods often require hand-engineered features prepared by either domain experts or practitioners with a background in signal processing.

> Existing techniques often depended on hand-crafted features that were expensive to create and required expert knowledge of the field.

> — *Deep Learning for Time-Series Analysis*, 2017.

1.2 Multilayer Perceptrons for Time Series

Simpler neural networks such as the Multilayer Perceptron or MLP approximate a mapping function from input variables to output variables. This general capability is valuable for time series for a number of reasons.

- **Robust to Noise**. Neural networks are robust to noise in input data and in the mapping function and can even support learning and prediction in the presence of missing values.

- **Nonlinear**. Neural networks do not make strong assumptions about the mapping function and readily learn linear and nonlinear relationships.

  > ... one important contribution of neural networks - namely their elegant ability to approximate arbitrary nonlinear functions. This property is of high value in time series processing and promises more powerful applications, especially in the subfield of forecasting ...

  > — *Neural Networks for Time Series Processing*, 1996.

More specifically, neural networks can be configured to support an arbitrary defined but fixed number of inputs and outputs in the mapping function. This means that neural networks can directly support:

- **Multivariate Inputs**. An arbitrary number of input features can be specified, providing direct support for multivariate forecasting.

- **Multi-step Forecasts**. An arbitrary number of output values can be specified, providing direct support for multi-step and even multivariate forecasting.

For these capabilities alone, feedforward neural networks may be useful for time series forecasting. Implicit in the usage of neural networks is the requirement that there is indeed a meaningful mapping from inputs to outputs to learn. Modeling a mapping of a random walk will perform no better than a persistence model (e.g. using the last seen observation as the forecast). This expectation of a learnable mapping function also makes one of the limitations clear: the mapping function is fixed or static.

- **Fixed Inputs**. The number of lag input variables is fixed, in the same way as traditional time series forecasting methods.

- **Fixed Outputs**. The number of output variables is also fixed; although a more subtle issue, it means that for each input pattern, one output must be produced.

  Sequences pose a challenge for [deep neural networks] because they require that the dimensionality of the inputs and outputs is known and fixed.

  — *Sequence to Sequence Learning with Neural Networks*, 2014.

Feedforward neural networks do offer great capability but still suffer from this key limitation of having to specify the temporal dependence upfront in the design of the model. This dependence is almost always unknown and must be discovered and teased out from detailed analysis in a fixed form.

1.3 Convolutional Neural Networks for Time Series

Convolutional Neural Networks or CNNs are a type of neural network that was designed to efficiently handle image data. They have proven effective on challenging computer vision problems both achieving state-of-the-art results on tasks like image classification and providing a component in hybrid models for entirely new problems such as object localization, image captioning and more.

They achieve this by operating directly on raw data, such as raw pixel values, instead of domain-specific or handcrafted features derived from the raw data. The model then learns how to automatically extract the features from the raw data that are directly useful for the problem being addressed. This is called representation learning and the CNN achieves this in such a way that the features are extracted regardless of how they occur in the data, so-called transform or distortion invariance.

> Convolutional networks combine three architectural ideas to ensure some degree of shift and distortion invariance: local receptive fields, shared weights (or weight replication), and, sometimes, spatial or temporal subsampling.
>
> — *Convolutional Networks for Images, Speech, and Time-Series*, 1998.

The ability of CNNs to learn and automatically extract features from raw input data can be applied to time series forecasting problems. A sequence of observations can be treated like a one-dimensional image that a CNN model can read and distill into the most salient elements.

The key attribute of the CNN is conducting different processing units [...] Such a variety of processing units can yield an effective representation of local salience of the signals. Then, the deep architecture allows multiple layers of these processing units to be stacked, so that this deep learning model can characterize the salience of signals in different scales.

— Deep Convolutional Neural Networks On Multichannel Time Series For Human Activity Recognition, 2015.

This capability of CNNs has been demonstrated to great effect on time series classification tasks such as automatically detecting human activities based on raw accelerator sensor data from fitness devices and smartphones.

The key advantages of [CNNs for activity recognition] are: i) feature extraction is performed in task dependent and non hand-crafted manners; ii) extracted features have more discriminative power w.r.t. the classes of human activities; iii) feature extraction and classification are unified in one model so their performances are mutually enhanced.

— Deep Convolutional Neural Networks On Multichannel Time Series For Human Activity Recognition, 2015.

CNNs get the benefits of Multilayer Perceptrons for time series forecasting, namely support for multivariate input, multivariate output and learning arbitrary but complex functional relationships, but do not require that the model learn directly from lag observations. Instead, the model can learn a representation from a large input sequence that is most relevant for the prediction problem.

- **Feature Learning**. Automatic identification, extraction and distillation of salient features from raw input data that pertain directly to the prediction problem that is being modeled.

1.4   Recurrent Neural Networks for Time Series

Recurrent neural networks like the Long Short-Term Memory network or LSTM add the explicit handling of order between observations when learning a mapping function from inputs to outputs, not offered by MLPs or CNNs. They are a type of neural network that adds native support for input data comprised of sequences of observations.

- **Native Support for Sequences**. Recurrent neural networks directly add support for input sequence data.

The addition of sequence is a new dimension to the function being approximated. Instead of mapping inputs to outputs alone, the network is capable of learning a mapping function for the inputs over time to an output.

Long Short-Term Memory (LSTM) is able to solve many time series tasks unsolvable by feedforward networks using fixed size time windows.

> — *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001.

This capability of LSTMs has been used to great effect in complex natural language processing problems such as neural machine translation where the model must learn the complex inter-relationships between words both within a given language and across languages in translating form one language to another. This capability can be used in time series forecasting. In addition to the general benefits of using neural networks for time series forecasting, recurrent neural networks can also automatically learn the temporal dependence from the data.

- **Learned Temporal Dependence**. The most relevant context of input observations to the expected output is learned and can change dynamically.

In the simplest case, the network is shown one observation at a time from a sequence and can learn what observations it has seen previously are relevant and how they are relevant to forecasting. The model both learns a mapping from inputs to outputs and learns what context from the input sequence is useful for the mapping, and can dynamically change this context as needed.

> Because of this ability to learn long term correlations in a sequence, LSTM networks obviate the need for a pre-specified time window and are capable of accurately modelling complex multivariate sequences.

> — *Long Short Term Memory Networks for Anomaly Detection in Time Series*, 2015.

1.5 Promise of Deep Learning

The capabilities of deep learning neural networks suggest a good fit for time series forecasting. By definition and with enough resources, neural networks in general should be able to subsume the capabilities of classical linear forecasting methods given their ability to learn arbitrary complex mapping from inputs to outputs.

- Neural networks learn arbitrary mapping functions.

It is good practice to manually identify and remove systematic structures from time series data to make the problem easier to model (e.g. make the series stationary), and this may still be a best practice when using recurrent neural networks. But, the general capability of these networks suggests that this may not be a requirement for a skillful model. Technically, the available context of the sequence provided as input may allow neural network models to learn both trend and seasonality directly.

- Neural networks may not require a scaled or stationary time series as input

Each of the three classes of neural network models discussed, MLPs, CNNs and RNNs offer capabilities that are challenging for classical time series forecasting methods, namely:

- Neural networks support multivariate inputs.

- Neural networks support multi-step outputs.

Although MLPs can operate directly on raw observations, CNNs offer efficiency and much greater performance at automatically learning to identify, extract and distill useful features from raw data.

- Convolutional neural networks support efficient feature learning.

Although MLPs and CNNs can learn arbitrary mapping functions, the explicit addition of support for input sequences in RNNs offers efficiency and greater performance for automatically learning the temporal dependencies both within the input sequence and from the input sequence to the output.

- LSTM networks support efficient learning of temporal dependencies.

These capabilities can also be combined, such as in the use of hybrid models like CNN-LSTMs and ConvLSTMs that seek to harness the capabilities of all three model types.

- Hybrid models efficiently combine the diverse capabilities of different architectures.

Traditionally, a lot of research has been invested into using MLPs for time series forecasting with modest results (covered in Chapter 10). Perhaps the most promising area in the application of deep learning methods to time series forecasting are in the use of CNNs, LSTMs and hybrid models. These areas will be our primary focus throughput this book.

1.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Your Expectations**. List three of your own expectations of deep learning methods for time series forecasting, perhaps expectations you had before reading this tutorial.

- **Known Limitations**. Research and list or quote three limitations of deep learning methods for time series forecasting listed in the literature.

- **Successful Example**. Find one research paper that presents a successful application of deep learning methods for time series forecasting.

If you explore any of these extensions, I'd love to know.

1.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- *Deep Learning for Time-Series Analysis*, 2017.
  https://arxiv.org/abs/1701.01887

- *Neural Networks for Time Series Processing*, 1996.
  http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.45.5697

- *Sequence to Sequence Learning with Neural Networks*, 2014.
  https://arxiv.org/abs/1409.3215

- *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001.
  https://link.springer.com/chapter/10.1007/3-540-44668-0_93

- *Long Short Term Memory Networks for Anomaly Detection in Time Series*, 2015.
  https://www.elen.ucl.ac.be/Proceedings/esann/esannpdf/es2015-56.pdf

- *Convolutional Networks for Images, Speech, and Time-Series*, 1998.
  https://dl.acm.org/citation.cfm?id=303704

- *Deep Convolutional Neural Networks On Multichannel Time Series For Human Activity Recognition*, 2015.
  https://dl.acm.org/citation.cfm?id=2832806

1.8 Summary

In this tutorial, you discovered the promised capabilities of deep learning neural networks for time series forecasting. Specifically, you learned:

- The focus and implicit, if not explicit, limitations on classical time series forecasting methods.

- The general capabilities of Multilayer Perceptrons and how they may be harnessed for time series forecasting.

- The added capabilities of feature learning and native support for sequences provided by Convolutional Neural Networks and Recurrent Neural Networks.

1.8.1 Next

In the next lesson, you will discover a taxonomy that you can use to quickly learn a lot about your time series forecasting problem.

Chapter 2

Taxonomy of Time Series Forecasting Problems

When you are presented with a new time series forecasting problem, there are many things to consider. The choice that you make directly impacts each step of the project from the design of a test harness to evaluate forecast models to the fundamental difficulty of the forecast problem that you are working on. It is possible to very quickly narrow down the options by working through a series of questions about your time series forecasting problem. By considering a few themes and questions within each theme, you narrow down the type of problem, test harness, and even choice of algorithms for your project. In this tutorial, you will discover a framework that you can use to quickly understand and frame your time series forecasting problem. After reading this tutorial, you will know:

- A structured way of thinking about time series forecasting problems.

- A framework to uncover the characteristics of a given time series forecasting problem.

- A suite of specific questions, the answers to which will help to define your forecasting problem.

Let's get started.

2.1   Framework Overview

Time series forecasting involves developing and using a predictive model on data where there is an ordered relationship between observations. Before you get started on your project, you can answer a few questions and greatly improve your understanding of the structure of your forecast problem, the structure of the model requires, and how to evaluate it. The framework presented in this tutorial is divided into seven parts; they are:

1. Inputs vs. Outputs.

2. Endogenous vs. Exogenous.

3. Unstructured vs. Structured.

4. Regression vs. Classification.

5. Univariate vs. Multivariate.

6. Single-step vs. Multi-step.

7. Static vs. Dynamic.

8. Contiguous vs. Discontiguous.

I recommend working through this framework before starting any time series forecasting project. Your answers may not be crisp on the first time through and the questions may require to you study the data, the domain, and talk to experts and stakeholders. Update your answers as you learn more as it will help to keep you on track, avoid distractions, and develop the actual model that you need for your project.

2.2 Inputs vs. Outputs

Generally, a prediction problem involves using past observations to predict or forecast one or more possible future observations. The goal is to guess about what might happen in the future. When you are required to make a forecast, it is critical to think about the data that you will have available to make the forecast and what you will be guessing about the future. We can summarize this as what are the inputs and outputs of the model when making a single forecast.

- **Inputs**: Historical data provided to the model in order to make a single forecast.

- **Outputs**: Prediction or forecast for a future time step beyond the data provided as input.

The input data is not the data used to train the model. We are not at that point yet. It is the data used to make one forecast, for example the last seven days of sales data to forecast the next one day of sales data. Defining the inputs and outputs of the model forces you to think about what exactly is or may be required to make a forecast. You may not be able to be specific when it comes to input data. For example, you may not know whether one or multiple prior time steps are required to make a forecast. But you will be able to identify the variables that could be used to make a forecast.

What are the inputs and outputs for a forecast?

2.3 Endogenous vs. Exogenous

The input data can be further subdivided in order to better understand its relationship to the output variable. An input variable is endogenous if it is affected by other variables in the system and the output variable depends on it. In a time series, the observations for an input variable depend upon one another. For example, the observation at time t is dependent upon the observation at $t - 1$; $t - 1$ may depend on $t - 2$, and so on. An input variable is an exogenous variable if it is independent of other variables in the system and the output variable depends upon it. Put simply, endogenous variables are influenced by other variables in the system (including themselves) whereas as exogenous variables are not and are considered as outside the system.

- **Endogenous**: Input variables that are influenced by other variables in the system and on which the output variable depends.

- **Exogenous**: Input variables that are not influenced by other variables in the system and on which the output variable depends.

Typically, a time series forecasting problem has endogenous variables (e.g. the output is a function of some number of prior time steps) and may or may not have exogenous variables. Often, exogenous variables are ignored given the strong focus on the time series. Explicitly thinking about both variable types may help to identify easily overlooked exogenous data or even engineered features that may improve the model.

What are the endogenous and exogenous variables?

2.4   Regression vs. Classification

Regression predictive modeling problems are those where a quantity is predicted. A quantity is a numerical value; for example a price, a count, a volume, and so on. A time series forecasting problem in which you want to predict one or more future numerical values is a regression type predictive modeling problem. Classification predictive modeling problems are those where a category is predicted. A category is a label from a small well-defined set of labels; for example `hot`, `cold`, `up`, `down`, and `buy`, `sell` are categories. A time series forecasting problem in which you want to classify input time series data is a classification type predictive modeling problem.

- **Regression**: Forecast a numerical quantity.

- **Classification**: Classify as one of two or more labels.

Are you working on a regression or classification predictive modeling problem?

There is some flexibility between these types. For example, a regression problem can be reframed as classification and a classification problem can be reframed as regression. Some problems, like predicting an ordinal value, can be framed as either classification and regression. It is possible that a reframing of your time series forecasting problem may simplify it.

What are some alternate ways to frame your time series forecasting problem?

2.5   Unstructured vs. Structured

It is useful to plot each variable in a time series and inspect the plot looking for possible patterns. A time series for a single variable may not have any obvious pattern. We can think of a series with no pattern as unstructured, as in there is no discernible time-dependent structure. Alternately, a time series may have obvious patterns, such as a trend or seasonal cycles as structured. We can often simplify the modeling process by identifying and removing the obvious structures from the data, such as an increasing trend or repeating cycle. Some classical methods even allow you to specify parameters to handle these systematic structures directly.

- **Unstructured**: No obvious systematic time-dependent pattern in a time series variable.

- **Structured**: Systematic time-dependent patterns in a time series variable (e.g. trend and/or seasonality).

Are the time series variables unstructured or structured?

2.6   Univariate vs. Multivariate

A single variable measured over time is referred to as a univariate time series. Univariate means one variate or one variable. Multiple variables measured over time is referred to as a multivariate time series: multiple variates or multiple variables.

- **Univariate**: One variable measured over time.

- **Multivariate**: Multiple variables measured over time.

Are you working on a univariate or multivariate time series problem?

Considering this question with regard to inputs and outputs may add a further distinction. The number of variables may differ between the inputs and outputs, e.g. the data may not be symmetrical. For example, you may have multiple variables as input to the model and only be interested in predicting one of the variables as output. In this case, there is an assumption in the model that the multiple input variables aid and are required in predicting the single output variable.

- **Univariate and Multivariate Inputs**: One or multiple input variables measured over time.

- **Univariate and Multivariate Outputs**: One or multiple output variables to be predicted.

2.7   Single-step vs. Multi-step

A forecast problem that requires a prediction of the next time step is called a one-step forecast model. Whereas a forecast problem that requires a prediction of more than one time step is called a multi-step forecast model. The more time steps to be projected into the future, the more challenging the problem given the compounding nature of the uncertainty on each forecasted time step.

- **One-step**: Forecast the next time step.

- **Multi-step**: Forecast more than one future time steps.

Do you require a single-step or a multi-step forecast?

2.8 Static vs. Dynamic

It is possible to develop a model once and use it repeatedly to make predictions. Given that the model is not updated or changed between forecasts, we can think of this model as being static. Conversely, we may receive new observations prior to making a subsequent forecast that could be used to create a new model or update the existing model. We can think of developing a new or updated model prior to each forecasts as a dynamic problem.

For example, it if the problem requires a forecast at the beginning of the week for the week ahead, we may receive the true observation at the end of the week that we can use to update the model prior to making next weeks forecast. This would be a dynamic model. If we do not get a true observation at the end of the week or we do and choose to not re-fit the model, this would be a static model. We may prefer a dynamic model, but the constraints of the domain or limitations of a chosen algorithm may impose constraints that make this intractable.

- **Static**. A forecast model is fit once and used to make predictions.

- **Dynamic**. A forecast model is fit on newly available data prior to each prediction.

Do you require a static or a dynamically updated model?

2.9 Contiguous vs. Discontiguous

A time series where the observations are uniform over time may be described as contiguous. Many time series problems have contiguous observations, such as one observation each hour, day, month or year. A time series where the observations are not uniform over time may be described as discontiguous. The lack of uniformity of the observations may be caused by missing or corrupt values. It may also be a feature of the problem where observations are only made available sporadically or at increasingly or decreasingly spaced time intervals. In the case of non-uniform observations, specific data formatting may be required when fitting some models to make the observations uniform over time.

- **Contiguous**. Observations are made uniform over time.

- **Discontiguous**. Observations are not uniform over time.

Are your observations contiguous or discontiguous?

2.10 Framework Review

To review, the themes and questions you can ask about your problem are as follows:

1. **Inputs vs. Outputs**: What are the inputs and outputs for a forecast?

2. **Endogenous vs. Exogenous**: What are the endogenous and exogenous variables?

3. **Unstructured vs. Structured**: Are the time series variables unstructured or structured?

4. **Regression vs. Classification**: Are you working on a regression or classification predictive modeling problem? What are some alternate ways to frame your time series forecasting problem?

5. **Univariate vs. Multivariate**: Are you working on a univariate or multivariate time series problem?

6. **Single-step vs. Multi-step**: Do you require a single-step or a multi-step forecast?

7. **Static vs. Dynamic**: Do you require a static or a dynamically updated model?

8. **Contiguous vs. Discontiguous**: Are your observations contiguous or discontiguous?

2.11 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Apply Taxonomy**. Select a standard time series dataset and work through the questions in the taxonomy to learn more about the dataset.

- **Standard Form**. Transform the taxonomy into a form or spreadsheet that you can re-use on new time series forecasting projects going forward.

- **Additional Characteristic**. Brainstorm and list at least one additional characteristic of a time series forecasting problem and a question that you might used to identify it.

If you explore any of these extensions, I'd love to know.

2.12 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- *Machine Learning Strategies for Time Series Forecasting*, 2013.
  http://link.springer.com/chapter/10.1007%2F978-3-642-36318-4_3

- *Recursive and direct multi-step forecasting: the best of both worlds*, 2012.
  https://econpapers.repec.org/paper/mshebswps/2012-19.htm

2.13 Summary

In this tutorial, you discovered a framework that you can use to quickly understand and frame your time series forecasting problem. Specifically, you learned:

- A structured way of thinking about time series forecasting problems.

- A framework to uncover the characteristics of a given time series forecasting problem.

- A suite of specific questions, the answers to which will help to define your forecasting problem.

2.13.1   Next

In the next lesson, you will discover a systematic process to ensure that you get better than average performance on your next time series forecasting project.

Chapter 3

How to Develop a Skillful Forecasting Model

"Here is a dataset, now develop a forecast." This is the normal situation that most practitioners find themselves in when getting started on a new time series forecasting problem. You are not alone if you are in this situation right now. In this tutorial, I want to give you a specific and actionable procedure that you can use to work through your time series forecasting problem and get better than average performance from your model. After reading this tutorial, you will know:

- A systematic four-step process that you can use to work through any time series forecasting problem.

- A list of models to evaluate and the order in which to evaluate them.

- A methodology that allows the choice of a final model to be defensible with empirical evidence, rather than whim or fashion.

Let's get started.

3.1   The Situation

You are handed data and told to develop a forecast model. *What do you do?* This is a common situation; far more common than most people think.

- Perhaps you are sent a CSV file.

- Perhaps you are given access to a database.

- Perhaps you are starting a competition.

The problem can be reasonably well defined:

- You have or can access historical time series data.

- You know or can find out what needs to be forecasted.

- You know or can find out how what is most important in evaluating a candidate model.

So how do you tackle this problem? Unless you have been through this trial by fire, you will struggle.

- You may struggle because you are new to the fields of machine learning and time series.

- You may struggle even if you have machine learning experience because time series data is different.

- You may struggle even if you have a background in time series forecasting because machine learning methods may outperform the classical approaches on your data.

In all of these cases, you will benefit from working through the problem carefully and systematically with a reusable process.

3.2   Process Overview

The goal of this process is to get a *good enough* forecast model as fast as possible. This process may or may not deliver the best possible model, but it will deliver a good model: a model that is better than a baseline prediction, if such a model exists. Typically, this process will deliver a model that is 80% to 90% of what can be achieved on the problem.

The process is fast. As such, it focuses on automation. Hyperparameters are searched rather than specified based on careful analysis. You are encouraged to test suites of models in parallel, rapidly getting an idea of what works and what doesn't. Nevertheless, the process is flexible, allowing you to circle back or go as deep as you like on a given step if you have the time and resources. This process is divided into four parts; they are:

1. Define Problem.

2. Design Test Harness.

3. Test Models.

4. Finalize Model.

You will notice that the process is different from a classical linear work-through of a predictive modeling problem. This is because it is designed to get a working forecast model fast and then slow down and see if you can get a better model.

3.3   How to Use This Process

The biggest mistake is skipping steps. For example, the mistake that almost all beginners make is going straight to modeling without a strong idea of what problem is being solved or how to robustly evaluate candidate solutions. This almost always results in a lot of wasted time. Slow down, follow the process, and complete each step.

I recommend having separate code for each experiment that can be re-run at any time. This is important so that you can circle back when you discover a bug, fix the code, and re-run an

experiment. You are running experiments and iterating quickly, but if you are sloppy, then you cannot trust any of your results. This is especially important when it comes to the design of your test harness for evaluating candidate models. Let's take a closer look at each step of the process.

3.4 Step 1: Define Problem

Define your time series problem. Some topics to consider and motivating questions within each topic (taken from the taxonomy in Chapter 2) are as follows:

1. **Inputs vs. Outputs**: What are the inputs and outputs for a forecast?

2. **Endogenous vs. Exogenous**: What are the endogenous and exogenous variables?

3. **Unstructured vs. Structured**: Are the time series variables unstructured or structured?

4. **Regression vs. Classification**: Are you working on a regression or classification predictive modeling problem? What are some alternate ways to frame your time series forecasting problem?

5. **Univariate vs. Multivariate**: Are you working on a univariate or multivariate time series problem?

6. **Single-step vs. Multi-step**: Do you require a single-step or a multi-step forecast?

7. **Static vs. Dynamic**: Do you require a static or a dynamically updated model?

8. **Contiguous vs. Discontiguous**: Are your observations contiguous or discontiguous?

Some useful tools to help get answers include:

- Data visualizations (e.g. line plots, etc.).

- Statistical analysis (e.g. ACF/PACF plots, etc.).

- Domain experts.

- Project stakeholders.

Update your answers to these questions as you learn more.

3.5 Step 2: Design Test Harness

Design a test harness that you can use to evaluate candidate models. This includes both the method used to estimate model skill and the metric used to evaluate predictions. Below is a common time series forecasting model evaluation scheme if you are looking for ideas:

1. Split the dataset into a train and test set.

2. Fit a candidate approach on the training dataset.

3. Make predictions on the test set directly or using walk-forward validation.

4. Calculate a metric that compares the predictions to the expected values.

The test harness must be robust and you must have complete trust in the results it provides. An important consideration is to ensure that any coefficients used for data preparation are estimated from the training dataset only and then applied on the test set. This might include mean and standard deviation in the case of data standardization.

3.6   Step 3: Test Models

Test many models using your test harness. I recommend carefully designing experiments to test a suite of configurations for standard models and letting them run. Each experiment can record results to a file, to allow you to quickly discover the top three to five most skillful configurations from each run. Some common classes of methods that you can design experiments around include the following:

1. **Baseline**. Simple forecasting methods such as persistence and averages.

2. **Autoregression**. The Box-Jenkins process and methods such as SARIMA.

3. **Exponential Smoothing**. Single, double and triple exponential smoothing methods.

4. **Linear Machine Learning**. Linear regression methods and variants such as regularization.

5. **Nonlinear Machine Learning**. kNN, decision trees, support vector regression and more.

6. **Ensemble Machine Learning**. Random forest, gradient boosting, stacking and more.

7. **Deep Learning**. MLPs, CNNs, LSTMs, and Hybrid models.

This list is based on a univariate time series forecasting problem, but you can adapt it for the specifics of your problem, e.g. use VAR/VARMA/etc. in the case of multivariate time series forecasting. Slot in more of your favorite classical time series forecasting methods and machine learning methods as you see fit. Order here is important and is structured in increasing complexity from classical to modern methods. Early approaches are simple and give good results fast; later approaches are slower and more complex, but also have a higher bar to clear to be skillful.

The resulting model skill can be used in a ratchet. For example, the skill of the best persistence configuration provide a baseline skill that all other models must outperform. If an autoregression model does better than persistence, it becomes the new level to outperform in order for a method to be considered skillful. Ideally, you want to exhaust each level before moving on to the next. E.g. get the most out of Autoregression methods and use the results as a new baseline to define *skillful* before moving on to Exponential Smoothing methods. The more time and resources that you have, the more configurations that you can evaluate. For example, with more time and resources, you could:

- Search model configurations at a finer resolution around a configuration known to already perform well.

- Search more model hyperparameter configurations.

- Use analysis to set better bounds on model hyperparameters to be searched.

- Use domain knowledge to better prepare data or engineer input features.

- Explore different potentially more complex methods.

- Explore ensembles of well performing base models.

I also encourage you to include data preparation schemes as hyperparameters for model runs. Some methods will perform some basic data preparation, such as differencing in ARIMA, nevertheless, it is often unclear exactly what data preparation schemes or combinations of schemes are required to best present a dataset to a modeling algorithm. Rather than guess, grid search and decide based on real results. Some data preparation schemes to consider include:

- Differencing to remove a trend.

- Seasonal differencing to remove seasonality.

- Standardize to center.

- Normalize to rescale.

- Power Transform to make normal.

This large amount of systematic searching can be slow to execute. Some ideas to speed up the evaluation of models include:

- Use multiple machines in parallel via cloud hardware (such as Amazon EC2).

- Reduce the size of the train or test dataset to make the evaluation process faster.

- Use a more coarse grid of hyperparameters and circle back if you have time later.

- Perhaps do not refit a model for each step in walk-forward validation.

3.7   Step 4: Finalize Model

At the end of the previous time step, you know whether your time series is predictable. If it is predictable, you will have a list of the top 5 to 10 candidate models that are skillful on the problem. You can pick one or multiple models and finalize them. This involves training a new final model on all available historical data (train and test). The model is ready for use; for example:

- Make a prediction for the future.

- Save the model to file for later use in making predictions.

- Incorporate the model into software for making predictions.

If you have time, you can always circle back to the previous step and see if you can further improve upon the final model. This may be required periodically if the data changes significantly over time.

3.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Metrics**. List 3 or more metrics that you could use to evaluate the performance of evaluated time series forecasting models.

- **Modeling Algorithms**. Pick a standard time series forecasting dataset and list 3 or more algorithms that you could evaluate for each level in the *Test Models* section of the framework.

- **Framework Automation**. Design a framework that could use that automates one part or multiple parts of this process.

If you explore any of these extensions, I'd love to know.

3.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- *Forecasting: principles and practice*, 2013.
  http://amzn.to/2lln93c

- *Practical Time Series Forecasting with R: A Hands-On Guide*, 2016.
  http://amzn.to/2k3QpuV

- *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.
  http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0194889

3.10 Summary

In this tutorial, you discovered a systematic process that you can use to quickly discover a skillful predictive model for your time series forecasting problem. Specifically, you learned:

- A systematic four-step process that you can use to work through any time series forecasting problem.

- An list of models to evaluate and the order in which to evaluate them.

- A methodology that allows the choice of a final model to be defensible with empirical evidence, rather than whim or fashion.

3.10.1 Next

In the next lesson, you will discover why and how to transform a time series dataset into samples for training a supervised learning model.

Chapter 4

How to Transform Time Series to a Supervised Learning Problem

Time series forecasting can be framed as a supervised learning problem. This re-framing of your time series data allows you access to the suite of standard linear and nonlinear machine learning algorithms on your problem. In this lesson, you will discover how you can re-frame your time series problem as a supervised learning problem for machine learning. After reading this lesson, you will know:

- What supervised learning is and how it is the foundation for all predictive modeling machine learning algorithms.

- The sliding window method for framing a time series dataset and how to use it.

- How to use the sliding window for multivariate data and multi-step forecasting.

Let's get started.

4.1 Supervised Machine Learning

The majority of practical machine learning uses supervised learning. Supervised learning is where you have input variables (X) and an output variable (y) and you use an algorithm to learn the mapping function from the input to the output.

$$Y = f(X) \tag{4.1}$$

The goal is to approximate the real underlying mapping so well that when you have new input data (X), you can predict the output variables (y) for that data. Below is a contrived example of a supervised learning dataset where each row is an observation comprised of one input variable (X) and one output variable to be predicted (y).

```
X,  y
5,  0.9
4,  0.8
5,  1.0
3,  0.7
4,  0.9
```

Listing 4.1: Example of a small contrived supervised learning dataset.

It is called supervised learning because the process of an algorithm learning from the training dataset can be thought of as a teacher supervising the learning process. We know the correct answers; the algorithm iteratively makes predictions on the training data and is corrected by making updates. Learning stops when the algorithm achieves an acceptable level of performance. Supervised learning problems can be further grouped into regression and classification problems.

- **Classification**: A classification problem is when the output variable is a category, such as `red` and `blue` or `disease` and `no disease`.

- **Regression**: A regression problem is when the output variable is a real value, such as `dollars` or `weight`. The contrived example above is a regression problem.

4.2 Sliding Window

Time series data can be phrased as supervised learning. Given a sequence of numbers for a time series dataset, we can restructure the data to look like a supervised learning problem. We can do this by using previous time steps as input variables and use the next time step as the output variable. Let's make this concrete with an example. Imagine we have a time series as follows:

```
time,   measure
1,      100
2,      110
3,      108
4,      115
5,      120
```

Listing 4.2: Example of a small contrived time series dataset.

We can restructure this time series dataset as a supervised learning problem by using the value at the previous time step to predict the value at the next time step. Re-organizing the time series dataset this way, the data would look as follows:

```
X,    y
?,    100
100,  110
110,  108
108,  115
115,  120
120,  ?
```

Listing 4.3: Example of time series dataset as supervised learning.

Take a look at the above transformed dataset and compare it to the original time series. Here are some observations:

- We can see that the previous time step is the input (X) and the next time step is the output (y) in our supervised learning problem.

- We can see that the order between the observations is preserved, and must continue to be preserved when using this dataset to train a supervised model.

- We can see that we have no previous value that we can use to predict the first value in the sequence. We will delete this row as we cannot use it.

- We can also see that we do not have a known next value to predict for the last value in the sequence. We may want to delete this value while training our supervised model also.

The use of prior time steps to predict the next time step is called the sliding window method. For short, it may be called the window method in some literature. In statistics and time series analysis, this is called a lag or lag method. The number of previous time steps is called the window width or size of the lag. This sliding window is the basis for how we can turn any time series dataset into a supervised learning problem. From this simple example, we can notice a few things:

- We can see how this can work to turn a time series into either a regression or a classification supervised learning problem for real-valued or labeled time series values.

- We can see how once a time series dataset is prepared this way that any of the standard linear and nonlinear machine learning algorithms may be applied, as long as the order of the rows is preserved.

- We can see how the width sliding window can be increased to include more previous time steps.

- We can see how the sliding window approach can be used on a time series that has more than one value, or so-called multivariate time series.

We will explore some of these uses of the sliding window, starting next with using it to handle time series with more than one observation at each time step, called multivariate time series.

4.3 Sliding Window With Multiple Variates

The number of observations recorded for a given time in a time series dataset matters. Traditionally, different names are used:

- **Univariate Time Series**: These are datasets where only a single variable is observed at each time, such as temperature each hour. The example in the previous section is a univariate time series dataset.

- **Multivariate Time Series**: These are datasets where two or more variables are observed at each time.

Most time series analysis methods, and even books on the topic, focus on univariate data. This is because it is the simplest to understand and work with. Multivariate data is often more difficult to work with. It is harder to model and often many of the classical methods do not perform well.

Multivariate time series analysis considers simultaneously multiple time series. [...] It is, in general, much more complicated than univariate time series analysis

— Page 1, *Multivariate Time Series Analysis: With R and Financial Applications*, 2013.

The sweet spot for using machine learning for time series is where classical methods fall down. This may be with complex univariate time series, and is more likely with multivariate time series given the additional complexity. Below is another worked example to make the sliding window method concrete for multivariate time series. Assume we have the contrived multivariate time series dataset below with two observations at each time step. Let's also assume that we are only concerned with predicting `measure2`.

```
time,   measure1,   measure2
1,      0.2,        88
2,      0.5,        89
3,      0.7,        87
4,      0.4,        88
5,      1.0,        90
```

Listing 4.4: Example of a small contrived multivariate time series dataset.

We can re-frame this time series dataset as a supervised learning problem with a window width of one. This means that we will use the previous time step values of `measure1` and `measure2`. We will also have available the next time step value for `measure1`. We will then predict the next time step value of `measure2`. This will give us 3 input features and one output value to predict for each training pattern.

```
X1,   X2,   X3,   y
?,    ?,    0.2,  88
0.2,  88,   0.5,  89
0.5,  89,   0.7,  87
0.7,  87,   0.4,  88
0.4,  88,   1.0,  90
1.0,  90,   ?,    ?
```

Listing 4.5: Example of a multivariate time series dataset as a supervised learning problem.

We can see that as in the univariate time series example above, we may need to remove the first and last rows in order to train our supervised learning model. This example raises the question of what if we wanted to predict both `measure1` and `measure2` for the next time step? The sliding window approach can also be used in this case. Using the same time series dataset above, we can phrase it as a supervised learning problem where we predict both `measure1` and `measure2` with the same window width of one, as follows.

```
X1,   X2,   y1,   y2
?,    ?,    0.2,  88
0.2,  88,   0.5,  89
0.5,  89,   0.7,  87
0.7,  87,   0.4,  88
0.4,  88,   1.0,  90
1.0,  90,   ?,    ?
```

Listing 4.6: Example of a multivariate time series dataset as a multi-step or sequence prediction supervised learning problem.

Not many supervised learning methods can handle the prediction of multiple output values without modification, but some methods, like artificial neural networks, have little trouble. We can think of predicting more than one value as predicting a sequence. In this case, we were

predicting two different output variables, but we may want to predict multiple time steps ahead of one output variable. This is called multi-step forecasting and is covered in the next section.

4.4 Sliding Window With Multiple Steps

The number of time steps ahead to be forecasted is important. Again, it is traditional to use different names for the problem depending on the number of time steps to forecast:

- **One-step Forecast**: This is where the next time step (`t+1`) is predicted.

- **Multi-step Forecast**: This is where two or more future time steps are to be predicted.

All of the examples we have looked at so far have been one-step forecasts. There are a number of ways to model multi-step forecasting as a supervised learning problem. For now, we are focusing on framing multi-step forecast using the sliding window method. Consider the same univariate time series dataset from the first sliding window example above:

```
time, measure
1,    100
2,    110
3,    108
4,    115
5,    120
```

Listing 4.7: Example of a small contrived time series dataset.

We can frame this time series as a two-step forecasting dataset for supervised learning with a window width of one, as follows:

```
X1,   y1,   y2
?     100,  110
100,  110,  108
110,  108,  115
108,  115,  120
115,  120,  ?
120,  ?,    ?
```

Listing 4.8: Example of a univariate time series dataset as a multi-step or sequence prediction supervised learning problem.

We can see that the first row and the last two rows cannot be used to train a supervised model. It is also a good example to show the burden on the input variables. Specifically, that a supervised model only has $X1$ to work with in order to predict both $y1$ and $y2$. Careful thought and experimentation are needed on your problem to find a window width that results in acceptable model performance.

4.5 Implementing Data Preparation

In Chapter 6 we touch on data preparation an focus on how to transform time series data in order to meet the expectations of a three-dimensional structure by some deep learning methods. In Chapters 7, 8, and 9 we will explore how to implement deep learning methods for univariate,

multivariate and multi-step forecasting with a specific focus on how to prepare the data for model. In each chapter we will develop functions that can be reused to prepare time series data on future projects.

4.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Samples in Detail**. Explain how the choice of the number of time steps in a sample impacts training a model and making predictions with a trained model.

- **Work Through Dataset**. Select a standard time series forecasting dataset and demonstrate how it may be transformed into samples for supervised learning.

- **Develop Function**. Develop a function in Python to transform a time series into a samples for supervised learning.

If you explore any of these extensions, I'd love to know.

4.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- *Multivariate Time Series Analysis: With R and Financial Applications*, 2013.
  https://amzn.to/2KD4rw7

- *Machine Learning for Sequential Data: A Review*, 2002.
  https://link.springer.com/chapter/10.1007/3-540-70659-3_2

- *Machine Learning Strategies for Time Series Forecasting*, 2013.
  https://link.springer.com/chapter/10.1007/978-3-642-36318-4_3

4.8 Summary

In this lesson, you discovered how you can re-frame your time series prediction problem as a supervised learning problem for use with machine learning methods. Specifically, you learned:

- Supervised learning is the most popular way of framing problems for machine learning as a collection of observations with inputs and outputs.

- Sliding window is the way to restructure a time series dataset as a supervised learning problem.

- Multivariate and multi-step forecasting time series can also be framed as supervised learning using the sliding window method.

4.8.1 Next

In the next lesson, you will discover naive and classical time series forecasting methods that you absolutely must evaluate before investigating more sophisticated deep learning methods.

Chapter 5

Review of Simple and Classical Forecasting Methods

Machine learning and deep learning methods can achieve impressive results on challenging time series forecasting problems. Nevertheless, there are many forecasting problems where classical methods such as SARIMA and exponential smoothing readily outperform more sophisticated methods. Therefore, it is important to both understand how classical time series forecasting methods work and to evaluate them prior to exploring more advanced methods. In this tutorial, you will discover naive and classical methods for time series forecasting.

- How to develop simple forecasts for time series forecasting problems that provide a baseline for estimating model skill.

- How to develop autoregressive models for time series forecasting.

- How to develop exponential smoothing methods for time series forecasting.

Let's get started.

5.1 Simple Forecasting Methods

Establishing a baseline is essential on any time series forecasting problem. A baseline in performance gives you an idea of how well all other models will actually perform on your problem. In this section, you will discover how to develop a simple forecasting methods that you can use to calculate a baseline level of performance on your time series forecasting problem.

5.1.1 Forecast Performance Baseline

A baseline in forecast performance provides a point of comparison. It is a point of reference for all other modeling techniques on your problem. If a model achieves performance at or below the baseline, the technique should be fixed or abandoned. The technique used to generate a forecast to calculate the baseline performance must be easy to implement and naive of problem-specific details. The goal is to get a baseline performance on your time series forecast problem as quickly as possible so that you can get to work better understanding the dataset and developing more advanced models. Three properties of a good technique for making a naive forecast are:

- **Simple**: A method that requires little or no training or intelligence.

- **Fast**: A method that is fast to implement and computationally trivial to make a prediction.

- **Repeatable**: A method that is deterministic, meaning that it produces an expected output given the same input.

5.1.2   Forecast Strategies

Simple forecast strategies are those that assume little or nothing about the nature of the forecast problem and are fast to implement and calculate. If a model can perform better than the performance of a simple forecast strategy, then it can be said to be skillful. There are two main themes to simple forecast strategies; they are:

- **Naive**, or using observations values directly.

- **Average**, or using a statistic calculated on previous observations.

Let's take a closer look at both of these strategies.

5.1.3   Naive Forecasting Strategy

A naive forecast involves using the previous observation directly as the forecast without any change. It is often called the persistence forecast as the prior observation is persisted. This simple approach can be adjusted slightly for seasonal data. In this case, the observation at the same time in the previous cycle may be persisted instead. This can be further generalized to testing each possible offset into the historical data that could be used to persist a value for a forecast. For example, given the series:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listing 5.1: Example of a univariate time series.

We could persist the last observation (relative index -1) as the value 9 or persist the second last prior observation (relative index -2) as 8, and so on.

5.1.4   Average Forecast Strategy

One step above the naive forecast is the strategy of averaging prior values. All prior observations are collected and averaged, either using the mean or the median, with no other treatment to the data. In some cases, we may want to shorten the history used in the average calculation to the last few observations. We can generalize this to the case of testing each possible set of n-prior observations to be included into the average calculation. For example, given the series:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Listing 5.2: Example of a univariate time series.

We could average the last one observation (9), the last two observations (8, 9), and so on. In the case of seasonal data, we may want to average the last n-prior observations at the same time in the cycle as the time that is being forecasted. For example, given the series with a 3-step cycle:

```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

Listing 5.3: Example of a univariate time series with seasonal structure.

We could use a window size of 3 and average the last one observation (-3 or 1), the last two observations (-3 or 1, and -(3 × 2) or 1), and so on.

5.1.5 Implementing Simple Strategies

In Chapter 11 we will explore how to implement the naive and the average forecast strategy and how to grid search the hyperparameters of these strategies for univariate time series forecasting datasets. The results from these strategies provide the baseline by which the performance of more sophisticated models may be judged skillful, or not. In Chapter 17 we will explore how to develop domain-specific naive forecast strategies for making multi-step forecasts for predicting household electricity usage.

5.2 Autoregressive Methods

Autoregressive Integrated Moving Average, or ARIMA, is one of the most widely used forecasting methods for univariate time series data forecasting. Although the method can handle data with a trend, it does not support time series with a seasonal component. An extension to ARIMA that supports the direct modeling of the seasonal component of the series is called SARIMA. In this section, you will discover the Seasonal Autoregressive Integrated Moving Average, or SARIMA, method for time series forecasting with univariate data containing trends and seasonality.

5.2.1 Autoregressive Integrated Moving Average Model

An ARIMA model is a class of statistical models for analyzing and forecasting time series data. It explicitly caters to a suite of standard structures in time series data, and as such provides a simple yet powerful method for making skillful time series forecasts. ARIMA is an acronym that stands for AutoRegressive Integrated Moving Average. It is a generalization of the simpler AutoRegressive Moving Average or ARMA and adds the notion of integration. This acronym is descriptive, capturing the key aspects of the model itself. Briefly, they are:

- **AR**: *Autoregression*. A model that uses the dependent relationship between an observation and some number of lagged observations.

- **I**: *Integrated*. The use of differencing of raw observations (e.g. subtracting an observation from an observation at the previous time step) in order to make the time series stationary.

- **MA**: *Moving Average*. A model that uses the dependency between an observation and a residual error from a moving average model applied to lagged observations.

Each of these components are explicitly specified in the model as a parameter. A standard notation is used of `ARIMA(p,d,q)` where the parameters are substituted with integer values to quickly indicate the specific ARIMA model being used. A problem with ARIMA is that it does

not support seasonal data. That is a time series with a repeating cycle. ARIMA expects data that is either not seasonal or has the seasonal component removed, e.g. seasonally adjusted via methods such as seasonal differencing.

A linear regression model is constructed including the specified number and type of terms, and the data is prepared by a degree of differencing in order to make it stationary, i.e. to remove trend and seasonal structures that negatively affect the regression model. A value of 0 can be used for a parameter, which indicates to not use that element of the model. This way, the ARIMA model can be configured to perform the function of an ARMA model, and even a simple AR, I, or MA model. Adopting an ARIMA model for a time series assumes that the underlying process that generated the observations is an ARIMA process. This may seem obvious, but helps to motivate the need to confirm the assumptions of the model in the raw observations and in the residual errors of forecasts from the model.

5.2.2   What is Seasonal ARIMA

Seasonal Autoregressive Integrated Moving Average, SARIMA or Seasonal ARIMA, is an extension of ARIMA that explicitly supports univariate time series data with a seasonal component. It adds three new hyperparameters to specify the autoregression (AR), differencing (I) and moving average (MA) for the seasonal component of the series, as well as an additional parameter for the period of the seasonality.

> A seasonal ARIMA model is formed by including additional seasonal terms in the ARIMA [...] The seasonal part of the model consists of terms that are very similar to the non-seasonal components of the model, but they involve backshifts of the seasonal period.

> — Page 242, *Forecasting: Principles and Practice*, 2013.

5.2.3   How to Configure SARIMA

Configuring a SARIMA requires selecting hyperparameters for both the trend and seasonal elements of the series.

Trend Elements

There are three trend elements that require configuration. They are the same as the ARIMA model; specifically:

- **p**: Trend autoregression order.

- **d**: Trend difference order.

- **q**: Trend moving average order.

Seasonal Elements

There are four seasonal elements that are not part of ARIMA that must be configured; they are:

- **P**: Seasonal autoregressive order.

- **D**: Seasonal difference order.

- **Q**: Seasonal moving average order.

- **m**: The number of time steps for a single seasonal period.

Together, the notation for an SARIMA model is specified as: `SARIMA(p,d,q)(P,D,Q)m`. Where the specifically chosen hyperparameters for a model are specified. Importantly, the m parameter influences the P, D, and Q parameters. For example, an m of 12 for monthly data suggests a yearly seasonal cycle. A $P = 1$ would make use of the first seasonally offset observation in the model, e.g. $t - (m \times 1)$ or $t - 12$. A $P = 2$, would use the last two seasonally offset observations $t - (m \times 1)$, $t - (m \times 2)$. Similarly, a D of 1 would calculate a first order seasonal difference and a $Q = 1$ would use a first order errors in the model (e.g. moving average).

> A seasonal ARIMA model uses differencing at a lag equal to the number of seasons (s) to remove additive seasonal effects. As with lag 1 differencing to remove a trend, the lag s differencing introduces a moving average term. The seasonal ARIMA model includes autoregressive and moving average terms at lag s

> — Page 142, *Introductory Time Series with R*, 2009.

The trend elements can be chosen through careful analysis of ACF and PACF plots looking at the correlations of recent time steps (e.g. 1, 2, 3). Similarly, ACF and PACF plots can be analyzed to specify values for the seasonal model by looking at correlation at seasonal lag time steps.

> Seasonal ARIMA models can potentially have a large number of parameters and combinations of terms. Therefore, it is appropriate to try out a wide range of models when fitting to data and choose a best fitting model using an appropriate criterion ...

> — Pages 143-144, *Introductory Time Series with R*, 2009.

Alternately, a grid search can be used across the trend and seasonal hyperparameters.

5.2.4   Implementing ARIMA and SARIMA Models

In Chapter 13 we will look at how to develop a reusable framework for automatically grid searching the hyperparameters for the SARIMA model on a range of standard univariate time series forecasting problems. In Chapter 18 we will analyze ACF and PACF plots of a real-world household electricity usage dataset and develop an ARIMA to make multi-step time series forecasts.

5.3 Exponential Smoothing Methods

Exponential smoothing is a time series forecasting method for univariate data that can be extended to support data with a systematic trend or seasonal component. It is a powerful forecasting method that may be used as an alternative to the popular Box-Jenkins ARIMA family of methods. In this section, you will discover the exponential smoothing method for univariate time series forecasting.

5.3.1 What Is Exponential Smoothing?

Exponential smoothing is a time series forecasting method for univariate data. Time series methods like the Box-Jenkins ARIMA family of methods develop a model where the prediction is a weighted linear sum of recent past observations or lags. Exponential smoothing forecasting methods are similar in that a prediction is a weighted sum of past observations, but the model explicitly uses an exponentially decreasing weight for past observations. Specifically, past observations are weighted with a geometrically decreasing ratio.

> Forecasts produced using exponential smoothing methods are weighted averages of past observations, with the weights decaying exponentially as the observations get older. In other words, the more recent the observation the higher the associated weight.

> — Page 171, *Forecasting: Principles and Practice*, 2013.

Exponential smoothing methods may be considered as peers and an alternative to the popular Box-Jenkins ARIMA class of methods for time series forecasting. Collectively, the methods are sometimes referred to as ETS models, referring to the explicit modeling of Error, Trend and Seasonality. There are three main types of exponential smoothing time series forecasting methods. A simple method that assumes no systematic structure, an extension that explicitly handles trends, and the most advanced approach that add support for seasonality.

5.3.2 Single Exponential Smoothing

Single Exponential Smoothing, SES for short, also called Simple Exponential Smoothing, is a time series forecasting method for univariate data without a trend or seasonality. It requires a single parameter, called alpha (a or α), also called the smoothing factor or smoothing coefficient. This parameter controls the rate at which the influence of the observations at prior time steps decay exponentially. Alpha is often set to a value between 0 and 1. Large values mean that the model pays attention mainly to the most recent past observations, whereas smaller values mean more of the history is taken into account when making a prediction.

> A value close to 1 indicates fast learning (that is, only the most recent values influence the forecasts), whereas a value close to 0 indicates slow learning (past observations have a large influence on forecasts).

> — Page 89, *Practical Time Series Forecasting with R*, 2016.

Hyperparameters:

- **Alpha** (α): Smoothing factor for the level.

5.3.3 Double Exponential Smoothing

Double Exponential Smoothing is an extension to Exponential Smoothing that explicitly adds support for trends in the univariate time series. In addition to the alpha parameter for controlling smoothing factor for the level, an additional smoothing factor is added to control the decay of the influence of the change in trend called beta (b or β). The method supports trends that change in different ways: an additive and a multiplicative, depending on whether the trend is linear or exponential respectively. Double Exponential Smoothing with an additive trend is classically referred to as Holt's linear trend model, named for the developer of the method Charles Holt.

- **Additive Trend**: Double Exponential Smoothing with a linear trend.

- **Multiplicative Trend**: Double Exponential Smoothing with an exponential trend.

For longer range (multi-step) forecasts, the trend may continue on unrealistically. As such, it can be useful to dampen the trend over time. Dampening means reducing the size of the trend over future time steps down to a straight line (no trend).

> The forecasts generated by Holt's linear method display a constant trend (increasing or decreasing) indecently into the future. Even more extreme are the forecasts generated by the exponential trend method [...] Motivated by this observation [...] introduced a parameter that "dampens" the trend to a flat line some time in the future.

> — Page 183, *Forecasting: Principles and Practice*, 2013.

As with modeling the trend itself, we can use the same principles in dampening the trend, specifically additively or multiplicatively for a linear or exponential dampening effect. A damping coefficient Phi (p or ϕ) is used to control the rate of dampening.

- **Additive Dampening**: Dampen a trend linearly.

- **Multiplicative Dampening**: Dampen the trend exponentially.

Hyperparameters:

- **Alpha** (α): Smoothing factor for the level.

- **Beta** (β): Smoothing factor for the trend.

- **Trend Type**: Additive or multiplicative.

- **Dampen Type**: Additive or multiplicative.

- **Phi** (ϕ): Damping coefficient.

5.3.4 Triple Exponential Smoothing

Triple Exponential Smoothing is an extension of Exponential Smoothing that explicitly adds support for seasonality to the univariate time series. This method is sometimes called Holt-Winters Exponential Smoothing, named for two contributors to the method: Charles Holt and Peter Winters. In addition to the alpha and beta smoothing factors, a new parameter is added called gamma (g or γ) that controls the influence on the seasonal component. As with the trend, the seasonality may be modeled as either an additive or multiplicative process for a linear or exponential change in the seasonality.

- **Additive Seasonality**: Triple Exponential Smoothing with a linear seasonality.

- **Multiplicative Seasonality**: Triple Exponential Smoothing with an exponential seasonality.

Triple exponential smoothing is the most advanced variation of exponential smoothing and through configuration, it can also develop double and single exponential smoothing models.

> Being an adaptive method, Holt-Winter's exponential smoothing allows the level, trend and seasonality patterns to change over time.
>
> — Page 95, *Practical Time Series Forecasting with R*, 2016.

Additionally, to ensure that the seasonality is modeled correctly, the number of time steps in a seasonal period (Period) must be specified. For example, if the series was monthly data and the seasonal period repeated each year, then the `Period=12`. Hyperparameters:

- **Alpha** (α): Smoothing factor for the level.

- **Beta** (β): Smoothing factor for the trend.

- **Trend Type**: Additive or multiplicative.

- **Dampen Type**: Additive or multiplicative.

- **Phi** (ϕ): Damping coefficient.

- **Gamma** (γ): Smoothing factor for the seasonality.

- **Seasonality Type**: Additive or multiplicative.

- **Period**: Time steps in seasonal period.

5.3.5 How to Configure Exponential Smoothing

All of the model hyperparameters can be specified explicitly. This can be challenging for experts and beginners alike. Instead, it is common to use numerical optimization to search for and fund the smoothing coefficients (alpha, beta, gamma, and phi) for the model that result in the lowest error.

a more robust and objective way to obtain values for the unknown parameters included in any exponential smoothing method is to estimate them from the observed data. [...] the unknown parameters and the initial values for any exponential smoothing method can be estimated by minimizing the SSE [sum of the squared errors].

— Page 177, *Forecasting: Principles and Practice*, 2013.

The parameters that specify the type of change in the trend and seasonality, such as weather they are additive or multiplicative and whether they should be dampened, must be specified explicitly.

5.3.6   Implementing ETS Models

In Chapter 12 we will look at how to develop a reusable framework for automatically grid searching the hyperparameters for the ETS model on a range of standard univariate time series forecasting problems. You can adapt and reuse this framework on your own time series forecasting problems going forward.

5.4   Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Summarize Methods**. Write a one line summary of naive, autoregressive and exponential smoothing time series forecasting methods.

- **Examples of Algorithms**. List 3 examples of algorithms that may be used for each of the naive, autoregressive and exponential smoothing time series forecasting methods.

- **API Wrappers**. Develop a reusable wrapper function for fitting and making predictions with one time of classical time series forecasting method provided by the Statsmodels library.

If you explore any of these extensions, I'd love to know.

5.5   Further Reading

This section provides more resources on the topic if you are looking to go deeper.

5.5.1   Simple Methods

- Chapter 2, The forecaster's toolbox, *Forecasting: Principles and Practice*, 2013.
  https://amzn.to/2xlJsfV

- Forecasting, Wikipedia.
  https://en.wikipedia.org/wiki/Forecasting

5.5.2   Autoregressive Methods

- Chapter 8, ARIMA models, *Forecasting: Principles and Practice*, 2013.
  https://amzn.to/2xlJsfV

- Chapter 7, *Non-stationary Models, Introductory Time Series with R*, 2009.
  https://amzn.to/2smB9LR

- Autoregressive integrated moving average on Wikipedia.
  https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average

5.5.3   Exponential Smoothing

- Chapter 7, Exponential smoothing, *Forecasting: Principles and Practice*, 2013.
  https://amzn.to/2xlJsfV

- Section 6.4. Introduction to Time Series Analysis, *Engineering Statistics Handbook*, 2012.
  https://www.itl.nist.gov/div898/handbook/

- *Practical Time Series Forecasting with R*, 2016.
  https://amzn.to/2LGKzKm

- Exponential smoothing, Wikipedia.
  https://en.wikipedia.org/wiki/Exponential_smoothing

5.6   Summary

In this tutorial, you discovered naive and classical methods for time series forecasting. Specifically, you learned:

- How to develop simple forecasts for time series forecasting problems that provide a baseline for estimating model skill.

- How to develop autoregressive models for time series forecasting.

- How to develop exponential smoothing methods for time series forecasting.

5.6.1   Next

This is the final lesson of this part, the next part will focus on how to develop deep learning models for time series forecasting in general.

Part III

Deep Learning Methods

Overview

This part will show you exactly how to prepare data and develop MLP, CNN and LSTM deep learning models for a range of different time series forecasting problems. The goal of the modeling examples in these chapters is to provide templates that you can copy into your project and start using immediately. As such, you may find some of the examples a little repetitive. After reading the chapters in this part, you will know:

- How to transform time series data into the required three-dimensional structured expected by Convolutional and Long Short-Term Memory Neural Networks, perhaps the most confusing area for beginners (Chapter 6).

- How to develop Multilayer Perceptron models for univariate, multivariate and multi-step time series forecasting problems (Chapter 7).

- How to develop Convolutional Neural Network models for univariate, multivariate and multi-step time series forecasting problems (Chapter 8).

- How to develop Long Short-Term Memory Neural Network models for univariate, multivariate and multi-step time series forecasting problems (Chapter 9).

Chapter 6

How to Prepare Time Series Data for CNNs and LSTMs

Time series data must be transformed before it can be used to fit a supervised learning model. In this form, the data can be used immediately to fit a supervised machine learning algorithm and even a Multilayer Perceptron neural network. One further transformation is required in order to ready the data for fitting a Convolutional Neural Network (CNN) or Long Short-Term Memory (LSTM) Neural Network. Specifically, the two-dimensional structure of the supervised learning data must be transformed to a three-dimensional structure. This is perhaps the largest sticking point for practitioners looking to implement deep learning methods for time series forecasting. In this tutorial, you will discover exactly how to transform a time series data set into a three-dimensional structure ready for fitting a CNN or LSTM model. After completing this tutorial, you will know:

- How to transform a time series dataset into a two-dimensional supervised learning format.

- How to transform a two-dimensional time series dataset into a three-dimensional structure suitable for CNNs and LSTMs.

- How to step through a worked example of splitting a very long time series into subsequences ready for training a CNN or LSTM model.

Let's get started.

6.1   Overview

This tutorial is divided into four parts, they are:

1. Time Series to Supervised.

2. 3D Data Preparation Basics.

3. Univariate Worked Example.

6.2 Time Series to Supervised

Time series data requires preparation before it can be used to train a supervised learning model, such as an LSTM neural network. For example, a univariate time series is represented as a vector of observations:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Listing 6.1: Example of a univariate time series.

A supervised learning algorithm requires that data is provided as a collection of samples, where each sample has an input component (X) and an output component (y).

```
X,              y
sample input,   sample output
sample input,   sample output
sample input,   sample output
...
```

Listing 6.2: Example of a general supervised learning learning problem.

The model will learn how to map inputs to outputs from the provided examples.

$$y = f(X) \tag{6.1}$$

A time series must be transformed into samples with input and output components. The transform both informs what the model will learn and how you intend to use the model in the future when making predictions, e.g. what is required to make a prediction (X) and what prediction is made (y). For a univariate time series problem where we are interested in one-step predictions, the observations at prior time steps, so-called lag observations, are used as input and the output is the observation at the current time step. For example, the above 10-step univariate series can be expressed as a supervised learning problem with three time steps for input and one step as output, as follows:

```
X,           y
[1, 2, 3],   [4]
[2, 3, 4],   [5]
[3, 4, 5],   [6]
...
```

Listing 6.3: Example of a univariate time series converted to supervised learning.

For more on transforming your time series data into a supervised learning problem in general see Chapter 4. You can write code to perform this transform yourself and that is the general approach I teach and recommend for greater understanding of your data and control over the transformation process. The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
```

```
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 6.4: Example of a function to split a univariate series into a supervised learning problem.

For specific examples for univariate, multivariate and multi-step time series, see Chapters 7, 8 and 9. After you have transformed your data into a form suitable for training a supervised learning model it will be represented as rows and columns. Each column will represent a feature to the model and may correspond to a separate lag observation. Each row will represent a sample and will correspond to a new example with input and output components.

- **Feature**: A column in a dataset, such as a lag observation for a time series dataset.

- **Sample**: A row in a dataset, such as an input and output sequence for a time series dataset.

For example, our univariate time series may look as follows:

```
x1, x2, x3, y
 1,  2,  3,  4
 2,  3,  4,  5
 3,  4,  5,  6
...
```

Listing 6.5: Example of a univariate time series in terms of rows and columns.

The dataset will be represented in Python using a NumPy array. The array will have two dimensions. The length of each dimension is referred to as the shape of the array. For example, a time series with 3 inputs, 1 output will be transformed into a supervised learning problem with 4 columns, or really 3 columns for the input data and 1 for the output data. If we have 7 rows and 3 columns for the input data then the shape of the dataset would be [7, 3], or 7 samples and 3 features. We can make this concrete by transforming our small contrived dataset.

```
# transform univariate time series to supervised learning problem
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
```

```
  return array(X), array(y)

# define univariate time series
series = array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(series.shape)
# transform to a supervised learning problem
X, y = split_sequence(series, 3)
print(X.shape, y.shape)
# show each sample
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 6.6: Example of transforming a univariate time series into a supervised learning problem.

Running the example first prints the shape of the time series, in this case 10 time steps of observations. Next, the series is split into input and output components for a supervised learning problem. We can see that for the chosen representation that we have 7 samples for the input and output and 3 input features. The shape of the output is 7 samples represented as `(7,)` indicating that the array is a single column. It could also be represented as a two-dimensional array with 7 rows and 1 column `[7, 1]`. Finally, the input and output aspects of each sample are printed, showing the expected breakdown of the problem.

```
(10,)

(7, 3) (7,)

[1 2 3] 4
[2 3 4] 5
[3 4 5] 6
[4 5 6] 7
[5 6 7] 8
[6 7 8] 9
[7 8 9] 10
```

Listing 6.7: Example output from transforming a univariate time series into a supervised learning problem.

Data in this form can be used directly to train a simple neural network, such as a Multilayer Perceptron. The difficulty for beginners comes when trying to prepare this data for CNNs and LSTMs that require data to have a three-dimensional structure instead of the two-dimensional structure described so far.

6.3   3D Data Preparation Basics

Preparing time series data for CNNs and LSTMs requires one additional step beyond transforming the data into a supervised learning problem. This one additional step causes the most confusion for beginners. In this section we will slowly step through the basics of how and why we need to prepare three-dimensional data for CNNs and LSTMs before working through an example in the next section.

The input layer for CNN and LSTM models is specified by the `input_shape` argument on the first hidden layer of the network. This too can make things confusing for beginners as intuitively we may expect the first layer defined in the model be the input layer, not the first

hidden layer. For example, below is an example of a network with one hidden `LSTM` layer and one `Dense` output layer.

```
# lstm without an input layer
...
model = Sequential()
model.add(LSTM(32))
model.add(Dense(1))
```

Listing 6.8: Example of defining an LSTM model without an input layer.

In this example, the `LSTM()` layer must specify the shape of the input data. The input to every CNN and LSTM layer must be three-dimensional. The three dimensions of this input are:

- **Samples**. One sequence is one sample. A batch is comprised of one or more samples.

- **Time Steps**. One time step is one point of observation in the sample. One sample is comprised of multiple time steps.

- **Features**. One feature is one observation at a time step. One time step is comprised of one or more features.

This expected three-dimensional structure of input data is often summarized using the array shape notation of: `[samples, timesteps, features]`. Remember, that the two-dimensional shape of a dataset that we are familiar with from the previous section has the array shape of: `[samples, features]`. this means we are adding the new dimension of *time steps*. Except, in time series forecasting problems our features are observations at time steps. So, really, we are adding the dimension of *features*, where a univariate time series has only one feature.

When defining the input layer of your LSTM network, the network assumes you have one or more samples and requires that you specify the number of time steps and the number of features. You can do this by specifying a tuple to the `input_shape` argument. For example, the model below defines an input layer that expects 1 or more samples, 3 time steps, and 1 feature. Remember, the first layer in the network is actually the first hidden layer, so in this example 32 refers to the number of units in the first hidden layer. The number of units in the first hidden layer is completely unrelated to the number of samples, time steps or features in your input data.

```
# lstm with an input layer
...
model = Sequential()
model.add(LSTM(32, input_shape=(3, 1)))
model.add(Dense(1))
```

Listing 6.9: Example of defining an LSTM model with an input layer.

This example maps onto our univariate time series from the previous section that we split into having 3 input time steps and 1 feature. We may have loaded our time series dataset from CSV or transformed it to a supervised learning problem in memory. It will have a two-dimensional shape and we must convert it to a three-dimensional shape with some number of samples, 3 time steps per sample and 1 feature per time step, or `[?, 3, 1]`. We can do this by using the `reshape()` NumPy function. For example, if we have 7 samples and 3 time steps per sample for the input element of our time series, we can reshape it into `[7, 3, 1]` by providing a tuple to

the `reshape()` function specifying the desired new shape of `(7, 3, 1)`. The array must have enough data to support the new shape, which in this case it does as `[7, 3]` and `[7, 3, 1]` are functionally the same thing.

```
...
# transform input from [samples, features] to [samples, timesteps, features]
X = X.reshape((7, 3, 1))
```

Listing 6.10: Example of reshaping 2D data to be 3D.

A short-cut in reshaping the array is to use the known shapes, such as the number of samples and the number of times steps from the array returned from the call to the `X.shape` property of the array. For example, `X.shape[0]` refers to the number of rows in a 2D array, in this case the number of samples and `X.shape[1]` refers to the number of columns in a 2D array, in this case the number of feature that we will use as the number of time steps. The reshape can therefore be written as:

```
...
# transform input from [samples, features] to [samples, timesteps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))
```

Listing 6.11: Example of reshaping 2D data to be 3D.

We can make this concept concrete with a worked example. The complete code listing is provided below.

```
# transform univariate 2d to 3d
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define univariate time series
series = array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
print(series.shape)
# transform to a supervised learning problem
X, y = split_sequence(series, 3)
print(X.shape, y.shape)
# transform input from [samples, features] to [samples, timesteps, features]
X = X.reshape((X.shape[0], X.shape[1], 1))
print(X.shape)
```

Listing 6.12: Example of transforming a univariate time series into a three-dimensional array.

Running the example first prints the shape of the univariate time series, in this case 10 time steps. It then summarizes the shape if the input (X) and output (y) elements of each sample after the univariate series has been converted into a supervised learning problem, in this case, the data has 7 samples and the input data has 3 features per sample, which we know are actually time steps. Finally, the input element of each sample is reshaped to be three-dimensional suitable for fitting an LSTM or CNN and now has the shape `[7, 3, 1]` or 7 samples, 3 time steps, 1 feature.

```
(10,)
(7, 3) (7,)
(7, 3, 1)
```

Listing 6.13: Example output from reshaping 2D data to be 3D.

6.4 Data Preparation Example

Consider that you are in the current situation:

> *I have two columns in my data file with 5,000 rows, column 1 is time (with 1 hour interval) and column 2 is the number of sales and I am trying to forecast the number of sales for future time steps. Help me to set the number of samples, time steps and features in this data for an LSTM?*

There are few problems here:

- **Data Shape**. LSTMs expect 3D input, and it can be challenging to get your head around this the first time.

- **Sequence Length**. LSTMs don't like sequences of more than 200-400 time steps, so the data will need to be split into subsamples.

We will work through this example, broken down into the following 4 steps:

1. Load the Data

2. Drop the Time Column

3. Split Into Samples

4. Reshape Subsequences

6.4.1 Load the Data

We can load this dataset as a Pandas `Series` using the function `read_csv()`.

```
# load time series dataset
series = read_csv('filename.csv', header=0, index_col=0)
```

Listing 6.14: Example of loading a dataset as a Pandas `DataFrame`.

For this example, we will mock loading by defining a new dataset in memory with 5,000 time steps.

```python
# example of defining a dataset
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
print(data[:5, :])
print(data.shape)
```

Listing 6.15: Example of defining the dataset instead of loading it.

Running this piece both prints the first 5 rows of data and the shape of the loaded data. We can see we have 5,000 rows and 2 columns: a standard univariate time series dataset.

```
[[ 1 10]
 [ 2 20]
 [ 3 30]
 [ 4 40]
 [ 5 50]]
(5000, 2)
```

Listing 6.16: Example output from defining the dataset.

6.4.2 Drop the Time Column

If your time series data is uniform over time and there is no missing values, we can drop the time column. If not, you may want to look at imputing the missing values, resampling the data to a new time scale, or developing a model that can handle missing values. Here, we just drop the first column:

```python
# example of dropping the time dimension from the dataset
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
# drop time
data = data[:, 1]
print(data.shape)
```

Listing 6.17: Example of dropping the time column.

Running the example prints the shape of the dataset after the time column has been removed.

```
(5000,)
```

Listing 6.18: Example output from dropping the time column.

6.4.3 Split Into Samples

LSTMs need to process samples where each sample is a single sequence of observations. In this case, 5,000 time steps is too long; LSTMs work better with 200-to-400 time steps. Therefore, we need to split the 5,000 time steps into multiple shorter sub-sequences. There are many ways to do this, and you may want to explore some depending on your problem. For example, perhaps you need overlapping sequences, perhaps non-overlapping is good but your model needs state across the sub-sequences and so on. In this example, we will split the 5,000 time steps into 25 sub-sequences of 200 time steps each. Rather than using NumPy or Python tricks, we will do this the old fashioned way so you can see what is going on.

```python
# example of splitting a univariate sequence into subsequences
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
# drop time
data = data[:, 1]
# split into samples (e.g. 5000/200 = 25)
samples = list()
length = 200
# step over the 5,000 in jumps of 200
for i in range(0,n,length):
  # grab from i to i + 200
  sample = data[i:i+length]
  samples.append(sample)
print(len(samples))
```

Listing 6.19: Example of splitting the series into samples.

We now have 25 subsequences of 200 time steps each.

```
25
```

Listing 6.20: Example output from splitting the series into samples.

6.4.4 Reshape Subsequences

The LSTM needs data with the format of [samples, timesteps, features]. We have 25 samples, 200 time steps per sample, and 1 feature. First, we need to convert our list of arrays into a 2D NumPy array with the shape [25, 200].

```python
# example of creating an array of subsequence
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
```

```
# drop time
data = data[:, 1]
# split into samples (e.g. 5000/200 = 25)
samples = list()
length = 200
# step over the 5,000 in jumps of 200
for i in range(0,n,length):
  # grab from i to i + 200
  sample = data[i:i+length]
  samples.append(sample)
# convert list of arrays into 2d array
data = array(samples)
print(data.shape)
```

Listing 6.21: Example of printing the shape of the samples.

Running this piece, you should see that we have 25 rows and 200 columns. Interpreted in a machine learning context, this dataset has 25 samples and 200 features per sample.

```
(25, 200)
```

Listing 6.22: Example output from printing the shape of the samples.

Next, we can use the `reshape()` function to add one additional dimension for our single feature and use the existing columns as time steps instead.

```
# example of creating a 3d array of subsequences
from numpy import array

# define the dataset
data = list()
n = 5000
for i in range(n):
  data.append([i+1, (i+1)*10])
data = array(data)
# drop time
data = data[:, 1]
# split into samples (e.g. 5000/200 = 25)
samples = list()
length = 200
# step over the 5,000 in jumps of 200
for i in range(0,n,length):
  # grab from i to i + 200
  sample = data[i:i+length]
  samples.append(sample)
# convert list of arrays into 2d array
data = array(samples)
# reshape into [samples, timesteps, features]
data = data.reshape((len(samples), length, 1))
print(data.shape)
```

Listing 6.23: Example of reshaping the dataset into a 3D format.

And that is it. The data can now be used as an input (X) to an LSTM model, or even a CNN model.

```
(25, 200, 1)
```

Listing 6.24: Example output from reshaping the dataset into a 3D format.

6.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Explain Data Shape**. Explain in your own words the meaning of samples, time steps and features.

- **Worked Example**. Select a standard time series forecasting problem and manually reshape it into a structure suitable for training a CNN or LSTM model.

- **Develop Framework**. Develop a function to automatically reshape a time series dataset into samples and into a shape suitable for training a CNN or LSTM model.

If you explore any of these extensions, I'd love to know.

6.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- `numpy.reshape` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.reshape.html

- Keras Recurrent Layers API in Keras.
  https://keras.io/layers/recurrent/

- Keras Convolutional Layers API in Keras.
  https://keras.io/layers/convolutional/

6.7 Summary

In this tutorial, you discovered exactly how to transform a time series data set into a three-dimensional structure ready for fitting a CNN or LSTM model.

Specifically, you learned:

- How to transform a time series dataset into a two-dimensional supervised learning format.

- How to transform a two-dimensional time series dataset into a three-dimensional structure suitable for CNNs and LSTMs.

- How to step through a worked example of splitting a very long time series into subsequences ready for training a CNN or LSTM model.

6.7.1 Next

In the next lesson, you will discover how to develop Multilayer Perceptron models for time series forecasting.

Chapter 7

How to Develop MLPs for Time Series Forecasting

Multilayer Perceptrons, or MLPs for short, can be applied to time series forecasting. A challenge with using MLPs for time series forecasting is in the preparation of the data. Specifically, lag observations must be flattened into feature vectors. In this tutorial, you will discover how to develop a suite of MLP models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem. In this tutorial, you will discover how to develop a suite of Multilayer Perceptron models for a range of standard time series forecasting problems. After completing this tutorial, you will know:

- How to develop MLP models for univariate time series forecasting.

- How to develop MLP models for multivariate time series forecasting.

- How to develop MLP models for multi-step time series forecasting.

Let's get started.

7.1   Tutorial Overview

In this tutorial, we will explore how to develop a suite of MLP models for time series forecasting. The models are demonstrated on small contrived time series problems intended to give the flavor of the type of time series problem being addressed. The chosen configuration of the models is arbitrary and not optimized for each problem; that was not the goal. This tutorial is divided into four parts; they are:

1. Univariate MLP Models

2. Multivariate MLP Models

3. Multi-step MLP Models

4. Multivariate Multi-step MLP Models

Note: Traditionally, a lot of research has been invested into using MLPs for time series forecasting with modest results. Perhaps the most promising area in the application of deep learning methods to time series forecasting are in the use of CNNs, LSTMs and hybrid models. As such, we will not see more examples of straight MLP models for time series forecasting beyond this tutorial.

7.2 Univariate MLP Models

Multilayer Perceptrons, or MLPs for short, can be used to model univariate time series forecasting problems. Univariate time series are a dataset comprised of a single series of observations with a temporal ordering and a model is required to learn from the series of past observations to predict the next value in the sequence. This section is divided into two parts; they are:

1. Data Preparation

2. MLP Model

7.2.1 Data Preparation

Before a univariate series can be modeled, it must be prepared. The MLP model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the model can learn. Consider a given univariate sequence:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 7.1: Example of a univariate time series.

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,              y
10, 20, 30,     40
20, 30, 40,     50
30, 40, 50,     60
...
```

Listing 7.2: Example of a univariate time series as a supervised learning problem.

The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```python
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
```

```
    break
  # gather input and output parts of the pattern
  seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
  X.append(seq_x)
  y.append(seq_y)
return array(X), array(y)
```

Listing 7.3: Example of a function to split a univariate series into a supervised learning problem.

We can demonstrate this function on our small contrived dataset above. The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.4: Example of transforming a univariate time series into a supervised learning problem.

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Listing 7.5: Example output from transforming a univariate time series into a supervised learning problem.

Now that we know how to prepare a univariate series for modeling, let's look at developing an MLP model that can learn the mapping of inputs to outputs.

7.2.2   MLP Model

A simple MLP model has a single hidden layer of nodes, and an output layer used to make a prediction. We can define an MLP for univariate time series forecasting as follows.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 7.6: Example of defining an MLP model.

Important in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps. The number of time steps as input is the number we chose when preparing our dataset as an argument to the `split_sequence()` function. The input dimension for each sample is specified in the `input_dim` argument on the definition of first hidden layer. Technically, the model will view each time step as a separate feature instead of separate time steps.

We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: `[samples, features]`. Our `split_sequence()` function in the previous section outputs the X with the shape `[samples, features]` ready to use for modeling. The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or '`mse`', loss function. Once the model is defined, we can fit it on the training dataset.

```
# fit model
model.fit(X, y, epochs=2000, verbose=0)
```

Listing 7.7: Example of fitting an MLP model.

After the model is fit, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: `[70, 80, 90]`. And expecting the model to predict something like: `[100]`. The model expects the input shape to be two-dimensional with `[samples, features]`, therefore, we must reshape the single input sample before making the prediction, e.g with the shape `[1, 3]` for 1 sample and 3 time steps used as input features.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
```

Listing 7.8: Example of reshaping a single sample for making a prediction.

We can tie all of this together and demonstrate how to develop an MLP for univariate time series forecasting and make a single prediction.

```
# univariate mlp example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
```

```python
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.9: Example of demonstrating an MLP for univariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction. We can see that the model predicts the next value in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.0109]]
```

Listing 7.10: Example output from demonstrating an MLP for univariate time series forecasting.

For an example of an MLP applied to a real-world univariate time series forecasting problem see Chapter 14. For an example of grid searching MLP hyperparameters on a univariate time series forecasting problem, see Chapter 15.

7.3 Multivariate MLP Models

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.

2. Multiple Parallel Series.

Let's take a look at each in turn.

7.3.1 Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has an observation at the same time step. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

Listing 7.11: Example of defining a parallel time series.

We can reshape these three arrays of data as a single dataset where each row is a time step and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

Listing 7.12: Example of horizontally stacking parallel series into a dataset.

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
print(dataset)
```

Listing 7.13: Example of parallel dependent time series.

Running the example prints the dataset with one row per time step and one column for each of the two input and one output parallel time series.

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 7.14: Example output from parallel dependent time series.

As with the univariate time series, we must structure these data into samples with input and output samples. We need to split the data into samples maintaining the order of observations across the two input sequences. If we chose three input time steps, then the first sample would look as follows:
    Input:

```
10, 15
20, 25
30, 35
```

Listing 7.15: Example input data for the first data sample.

    Output:

```
65
```

Listing 7.16: Example output data for the first data sample.

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case 65. We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used. We can define a function named `split_sequences()` that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.17: Example of a function for separating parallel time series into a supervised learning problem.

We can test this function on our dataset using three time steps for each input time series as input. The complete example is listed below.

```python
# multivariate data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.18: Example of transforming a parallel dependent time series into a supervised learning problem.

Running the example first prints the shape of the X and y components. We can see that the X component has a three-dimensional structure. The first dimension is the number of samples, in this case 7. The second dimension is the number of time steps per sample, in this case 3, the value specified to the function. Finally, the last dimension specifies the number of parallel time series or the number of variables, in this case 2 for the two parallel series. We can then see that the input and output for each sample is printed, showing the three time steps for each of the two input series and the associated output for each sample.

```
(7, 3, 2) (7,)

[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

Listing 7.19: Example output from transforming a parallel dependent time series into a supervised learning problem.

MLP Model

Before we can fit an MLP on this data, we must flatten the shape of the input samples. MLPs require that the shape of the input portion of each sample is a vector. With a multivariate input, we will have multiple vectors, one for each time step. We can flatten the temporal structure of each input sample, so that:

```
[[10 15]
[20 25]
[30 35]]
```

Listing 7.20: Example input data for the first data sample.

Becomes:

```
[10, 15, 20, 25, 30, 35]
```

Listing 7.21: Example of a flattened input data for the first data sample.

First, we can calculate the length of each input vector as the number of time steps multiplied by the number of features or time series. We can then use this vector size to reshape the input.

```
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
```

Listing 7.22: Example of flattening input samples for the MLP.

We can now define an MLP model for the multivariate input where the vector length is used for the input dimension argument.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 7.23: Example of defining an MLP that expects a flattened multivariate time series as input.

When making a prediction, the model expects three time steps for two input time series. We can predict the next value in the output series proving the input values of:

```
80, 85
90, 95
100, 105
```

Listing 7.24: Example input sample for making a prediction beyond the end of the time series.

The shape of the 1 sample with 3 time steps and 2 variables would be `[1, 3, 2]`. We must again reshape this to be 1 sample with a vector of 6 elements or `[1, 6]`. We would expect the next value in the sequence to be $100 + 105$ or 205.

```
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
```

Listing 7.25: Example of reshaping the input sample for making a prediction.

The complete example is listed below.

```
# multivariate mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
```

```
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.26: Example of using an MLP to forecast a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[205.04436]]
```

Listing 7.27: Example output from using an MLP to forecast a dependent time series.

Multi-headed MLP Model

There is another more elaborate way to model the problem. Each input series can be handled by a separate MLP and the output of each of these submodels can be combined before a prediction is made for the output sequence. We can refer to this as a multi-headed input MLP model. It may offer more flexibility or better performance depending on the specifics of the problem that are being modeled. This type of model can be defined in Keras using the Keras functional API. First, we can define the first input model as an MLP with an input layer that expects vectors with n_steps features.

```
# first input model
visible1 = Input(shape=(n_steps,))
dense1 = Dense(100, activation='relu')(visible1)
```

Listing 7.28: Example of defining the first input model.

We can define the second input submodel in the same way.

```
# second input model
visible2 = Input(shape=(n_steps,))
dense2 = Dense(100, activation='relu')(visible2)
```

Listing 7.29: Example of defining the second input model.

Now that both input submodels have been defined, we can merge the output from each model into one long vector, which can be interpreted before making a prediction for the output sequence.

```
# merge input models
merge = concatenate([dense1, dense2])
output = Dense(1)(merge)
```

Listing 7.30: Example of merging the two input models.

We can then tie the inputs and outputs together.

```
# connect input and output models
model = Model(inputs=[visible1, visible2], outputs=output)
```

Listing 7.31: Example of connecting the input and output elements together.

The image below provides a schematic for how this model looks, including the shape of the inputs and outputs of each layer.
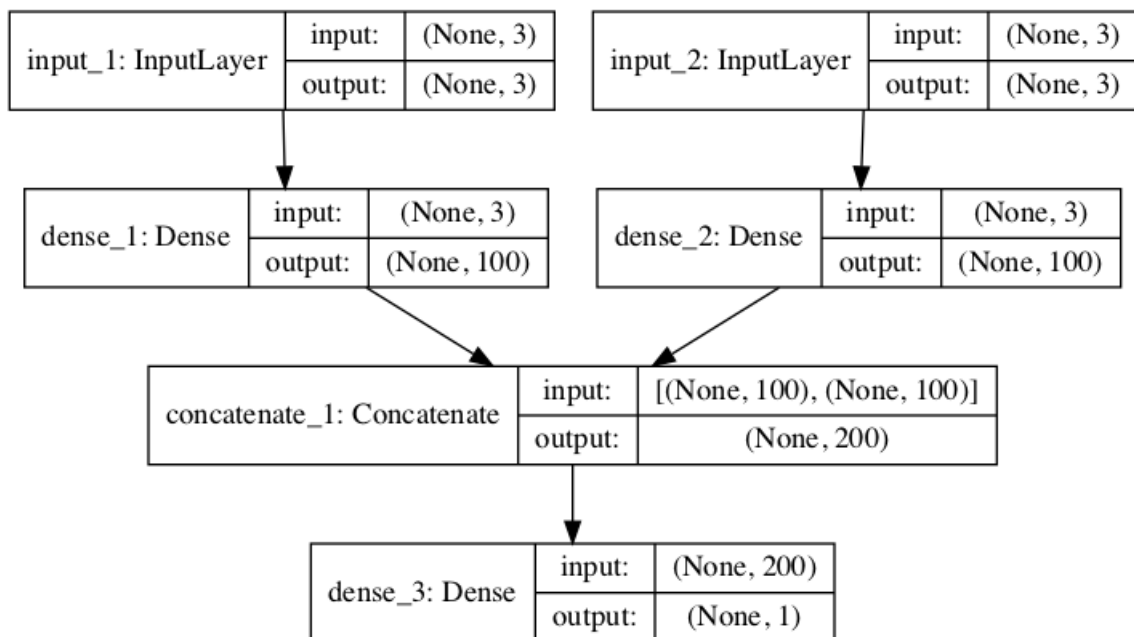
Figure 7.1: Plot of Multi-headed MLP for Multivariate Time Series Forecasting.

This model requires input to be provided as a list of two elements, where each element in the list contains data for one of the submodels. In order to achieve this, we can split the 3D input data into two separate arrays of input data: that is from one array with the shape `[7, 3, 2]` to two 2D arrays with the shape `[7, 3]`.

```
# separate input data
X1 = X[:, :, 0]
X2 = X[:, :, 1]
```

Listing 7.32: Example of separating input data for the two input models.

These data can then be provided in order to fit the model.

```
# fit model
model.fit([X1, X2], y, epochs=2000, verbose=0)
```

Listing 7.33: Example of fitting the multi-headed input model.

Similarly, we must prepare the data for a single sample as two separate two-dimensional arrays when making a single one-step prediction.

```
# reshape one sample for making a forecast
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps))
x2 = x_input[:, 1].reshape((1, n_steps))
```

Listing 7.34: Example of preparing an input sample for making a forecast.

We can tie all of this together; the complete example is listed below.

```
# multivariate mlp example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers.merge import concatenate

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
```

```
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# separate input data
X1 = X[:, :, 0]
X2 = X[:, :, 1]
# first input model
visible1 = Input(shape=(n_steps,))
dense1 = Dense(100, activation='relu')(visible1)
# second input model
visible2 = Input(shape=(n_steps,))
dense2 = Dense(100, activation='relu')(visible2)
# merge input models
merge = concatenate([dense1, dense2])
output = Dense(1)(merge)
model = Model(inputs=[visible1, visible2], outputs=output)
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit([X1, X2], y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps))
x2 = x_input[:, 1].reshape((1, n_steps))
yhat = model.predict([x1, x2], verbose=0)
print(yhat)
```

Listing 7.35: Example of using a Multi-headed MLP to forecast a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[206.05022]]
```

Listing 7.36: Example output from using a Multi-headed MLP to forecast a dependent time series.

7.3.2 Multiple Parallel Series

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each. For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 7.37: Example of a parallel time series problem.

We may want to predict the value for each of the three time series for the next time step. This might be referred to as multivariate forecasting. Again, the data must be split into input/output samples in order to train a model. The first sample of this dataset would be:

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 7.38: Example input from the first data sample.

Output:

```
40, 45, 85
```

Listing 7.39: Example output from the first data sample.

The `split_sequences()` function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```python
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.40: Example of a function for splitting a multivariate time series dataset into a supervised learning problem.

We can demonstrate this on the contrived problem; the complete example is listed below.

```python
# multivariate output data prep
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
```

```
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.41: Example of splitting a multivariate time series into a supervised learning problem.

Running the example first prints the shape of the prepared X and y components. The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3). The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3). Then, each of the samples is printed showing the input and output components of each sample.

```
(6, 3, 3) (6, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50 55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60 65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70 75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80 85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90 95 185]
```

Listing 7.42: Example output from splitting a multivariate time series into a supervised learning problem.

Vector-Output MLP Model

We are now ready to fit an MLP model on this data. As with the previous case of multivariate input, we must flatten the three dimensional structure of the input data samples to a two dimensional structure of `[samples, features]`, where lag observations are treated as features by the model.

```
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
```
Listing 7.43: Example of flattening multivariate time series for input to a MLP.

The model output will be a vector, with one element for each of the three different time series.

```
# determine the number of outputs
n_output = y.shape[1]
```
Listing 7.44: Example of defining the size of the vector to forecast.

We can now define our model, using the flattened vector length for the input layer and the number of time series as the vector length when making a prediction.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
```
Listing 7.45: Example of defining a MLP for multivariate forecasting.

We can predict the next value in each of the three parallel series by providing an input of three time steps for each series.

```
70, 75, 145
80, 85, 165
90, 95, 185
```
Listing 7.46: Example input for making an out-of-sample forecast.

The shape of the input for making a single prediction must be 1 sample, 3 time steps and 3 features, or `[1, 3, 3]`. Again, we can flatten this to `[1, 6]` to meet the expectations of the model. We would expect the vector output to be:

```
[100, 105, 205]
```
Listing 7.47: Example output for an out-of-sample forecast.

```
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
```
Listing 7.48: Example of preparing data for making an out-of-sample forecast with a MLP.

We can tie all of this together and demonstrate an MLP for multivariate output time series forecasting below.

```python
# multivariate output mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
n_output = y.shape[1]
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.49: Example of an MLP for multivariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.95039 107.541306 206.81033 ]]
```

Listing 7.50: Example output from an MLP for multivariate time series forecasting.

Multi-output MLP Model

As with multiple input series, there is another, more elaborate way to model the problem. Each output series can be handled by a separate output MLP model. We can refer to this as a multi-output MLP model. It may offer more flexibility or better performance depending on the specifics of the problem that is being modeled. This type of model can be defined in Keras using the Keras functional API. First, we can define the input model as an MLP with an input layer that expects flattened feature vectors.

```
# define model
visible = Input(shape=(n_input,))
dense = Dense(100, activation='relu')(visible)
```

Listing 7.51: Example of defining an input model.

We can then define one output layer for each of the three series that we wish to forecast, where each output submodel will forecast a single time step.

```
# define output 1
output1 = Dense(1)(dense)
# define output 2
output2 = Dense(1)(dense)
# define output 2
output3 = Dense(1)(dense)
```

Listing 7.52: Example of defining multiple output models.

We can then tie the input and output layers together into a single model.

```
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
```

Listing 7.53: Example of connecting input and output models.

To make the model architecture clear, the schematic below clearly shows the three separate output layers of the model and the input and output shapes of each layer.
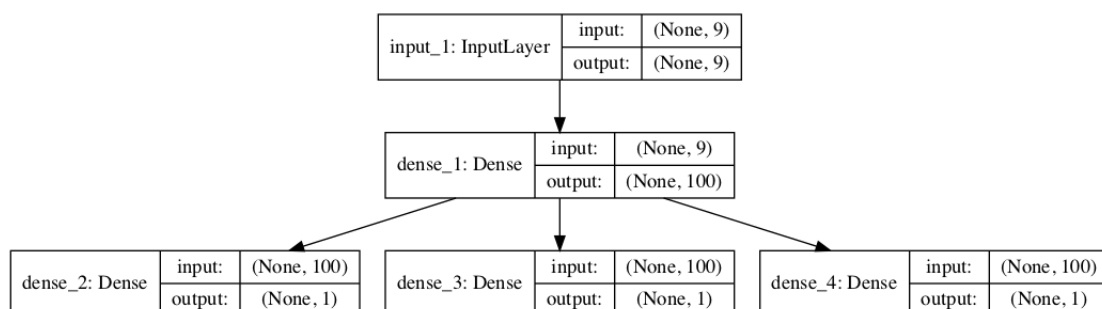
Figure 7.2: Plot of Multi-output MLP for Multivariate Time Series Forecasting.

When training the model, it will require three separate output arrays per sample. We can achieve this by converting the output training data that has the shape [7, 3] to three arrays with the shape [7, 1].

```
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
```

Listing 7.54: Example of splitting output data for the multi-output model.

These arrays can be provided to the model during training.

```
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
```

Listing 7.55: Example of fitting the multi-output MLP model.

Tying all of this together, the complete example is listed below.

```
# multivariate output mlp example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# flatten input
n_input = X.shape[1] * X.shape[2]
```

```
X = X.reshape((X.shape[0], n_input))
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
# define model
visible = Input(shape=(n_input,))
dense = Dense(100, activation='relu')(visible)
# define output 1
output1 = Dense(1)(dense)
# define output 2
output2 = Dense(1)(dense)
# define output 2
output3 = Dense(1)(dense)
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.56: Example of a multi-output MLP for multivariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[array([[100.86121]], dtype=float32),
array([[105.14738]], dtype=float32),
array([[205.97507]], dtype=float32)]
```

Listing 7.57: Example output from a multi-output MLP for multivariate time series forecasting.

7.4 Multi-step MLP Models

In practice, there is little difference to the MLP model in predicting a vector output that represents different output variables (as in the previous example) or a vector output that represents multiple time steps of one variable. Nevertheless, there are subtle and important differences in the way the training data is prepared. In this section, we will demonstrate the case of developing a multi-step forecast model using a vector model. Before we look at the specifics of the model, let's first look at the preparation of data for multi-step forecasting.

7.4.1 Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components. Both the input and output components

will be comprised of multiple time steps and may or may not have the same number of steps. For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 7.58: Example of a univariate time series.

We could use the last three time steps as input and forecast the next two time steps. The first sample would look as follows:

Input:

```
[10, 20, 30]
```

Listing 7.59: Example input for a multi-step forecast.

Output:

```
[40, 50]
```

Listing 7.60: Example output for a multi-step forecast.

The `split_sequence()` function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```python
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.61: Example of a function for preparing a univariate time series for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```python
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
```

```
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.62: Example of data preparation for multi-step time series forecasting.

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Listing 7.63: Example output from data preparation for multi-step time series forecasting.

Now that we know how to prepare data for multi-step forecasting, let's look at an MLP model that can learn this mapping.

7.4.2 Vector Output Model

The MLP can output a vector directly that can be interpreted as a multi-step forecast. This approach was seen in the previous section were one time step of each output time series was forecasted as a vector. With the number of input and output steps specified in the **n_steps_in** and **n_steps_out** variables, we can define a multi-step time-series forecasting model.

```
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps_in))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
```

Listing 7.64: Example of preparing an MLP mode for multi-step time series forecasting.

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input:

```
[70, 80, 90]
```

Listing 7.65: Example input for making an out-of-sample multi-step forecast.

We would expect the predicted output to be:

```
[100, 110]
```

Listing 7.66: Example output for an out-of-sample multi-step forecast.

As expected by the model, the shape of the single sample of input data when making the prediction must be `[1, 3]` for the 1 sample and 3 time steps (features) of the input and the single feature.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in))
yhat = model.predict(x_input, verbose=0)
```

Listing 7.67: Example of preparing data for making an out-of-sample multi-step forecast.

Tying all of this together, the MLP for multi-step forecasting with a univariate time series is listed below.

```
# univariate multi-step vector-output mlp example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_steps_in))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in))
yhat = model.predict(x_input, verbose=0)
```

```
print(yhat)
```
Listing 7.68: Example of an MLP for multi-step time series forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.572365 113.88405 ]]
```
Listing 7.69: Example output from an MLP for multi-step time series forecasting.

7.5 Multivariate Multi-step MLP Models

In the previous sections, we have looked at univariate, multivariate, and multi-step time series forecasting. It is possible to mix and match the different types of MLP models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging, particularly in preparing the data and defining the shape of inputs and outputs for the model. In this section, we will look at short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

1. Multiple Input Multi-step Output.

2. Multiple Parallel Input and Multi-step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

7.5.1 Multiple Input Multi-step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```
Listing 7.70: Example of a multivariate time series with a dependent series.

We may use three prior time steps of each of the two input time series to predict two time steps of the output time series.
        Input:

```
10, 15
20, 25
30, 35
```

Listing 7.71: Example input for a multi-step forecast for a dependent series.

Output:

```
65
85
```

Listing 7.72: Example output for a multi-step forecast for a dependent series.

The `split_sequences()` function below implements this behavior.

```python
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 7.73: Example of a function for preparing data for a multi-step dependent series.

We can demonstrate this on our contrived dataset. The complete example is listed below.

```python
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
```

```
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.74: Example of preparing data for multi-step forecasting for a dependent series.

Running the example first prints the shape of the prepared training data. We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three time steps and two variables for the two input time series. The output portion of the samples is two-dimensional for the six samples and the two time steps for each sample to be predicted. The prepared samples are then printed to confirm that the data was prepared as we specified.

```
(6, 3, 2) (6, 2)

[[10 15]
 [20 25]
 [30 35]] [65 85]
[[20 25]
 [30 35]
 [40 45]] [ 85 105]
[[30 35]
 [40 45]
 [50 55]] [105 125]
[[40 45]
 [50 55]
 [60 65]] [125 145]
[[50 55]
 [60 65]
 [70 75]] [145 165]
[[60 65]
 [70 75]
 [80 85]] [165 185]
```

Listing 7.75: Example output from preparing data for multi-step forecasting for a dependent series.

We can now develop an MLP model for multi-step predictions using a vector output. The complete example is listed below.

```
# multivariate multi-step mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
```

```python
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.76: Example of an MLP model for multi-step forecasting for a dependent series.

Running the example fits the model and predicts the next two time steps of the output sequence beyond the dataset. We would expect the next two steps to be `[185, 205]`.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[186.53822 208.41725]]
```

Listing 7.77: Example output from an MLP model for multi-step forecasting for a dependent series.

7.5.2 Multiple Parallel Input and Multi-step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 7.78: Example of a multivariate time series.

We may use the last three time steps from each of the three time series as input to the model and predict the next time steps of each of the three time series as output. The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 7.79: Example input for the first a multivariate time series sample.

Output:

```
40, 45, 85
50, 55, 105
```

Listing 7.80: Example output for the first a multivariate time series sample.

The `split_sequences()` function below implements this behavior.

```python
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
```

```
      return array(X), array(y)
```

Listing 7.81: Example of a function for preparing data for a multi-step multivariate series.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 7.82: Example of preparing data for multi-step forecasting for a multivariate series.

Running the example first prints the shape of the prepared training dataset. We can see that both the input (X) and output (Y) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively. The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

```
(5, 3, 3) (5, 2, 3)

[[10 15 25]
```

```
 [20 25 45]
 [30 35 65]] [[ 40 45 85]
 [ 50  55 105]]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [[ 50 55 105]
 [ 60  65 125]]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [[ 60 65 125]
 [ 70 75 145]]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [[ 70 75 145]
 [ 80 85 165]]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [[ 80 85 165]
 [ 90 95 185]]
```

Listing 7.83: Example output from preparing data for multi-step forecasting for a multivariate series.

We can now develop an MLP model to make multivariate multi-step forecasts. In addition to flattening the shape of the input data, as we have in prior examples, we must also flatten the three-dimensional structure of the output data. This is because the MLP model is only capable of taking vector inputs and outputs.

```python
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
```

Listing 7.84: Example of reshaping input and output data for fitting an MLP model for a multi-step multivariate series.

The complete example is listed below.

```python
# multivariate multi-step mlp example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
	X, y = list(), list()
	for i in range(len(sequences)):
		# find the end of this pattern
		end_ix = i + n_steps_in
		out_end_ix = end_ix + n_steps_out
		# check if we are beyond the dataset
		if out_end_ix > len(sequences):
			break
		# gather input and output parts of the pattern
```

```
        seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
        X.append(seq_x)
        y.append(seq_y)
    return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# flatten input
n_input = X.shape[1] * X.shape[2]
X = X.reshape((X.shape[0], n_input))
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
# define model
model = Sequential()
model.add(Dense(100, activation='relu', input_dim=n_input))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_input))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 7.85: Example of an MLP model for multi-step forecasting for a multivariate series.

Running the example fits the model and predicts the values for each of the three time steps for the next two time steps beyond the end of the dataset. We would expect the values for these series and time steps to be as follows:

```
90, 95, 185
100, 105, 205
```

Listing 7.86: Expected output for an out-of-sample multi-step multivariate forecast.

We can see that the model forecast gets reasonably close to the expected values.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[ 91.28376 96.567 188.37575 100.54482 107.9219 208.108 ]]
```

Listing 7.87: Example output from an MLP model for multi-step forecasting for a multivariate series.

7.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Problem Differences**. Explain the main changes to the MLP required when modeling each of univariate, multivariate and multi-step time series forecasting problems.

- **Experiment**. Select one example and modify it to work with your own small contrived dataset.

- **Develop Framework**. Use the examples in this chapter as the basis for a framework for automatically developing an MLP model for a given time series forecasting problem.

If you explore any of these extensions, I'd love to know.

7.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

7.7.1 Books

- *Neural Smithing: Supervised Learning in Feedforward Artificial Neural Networks*, 1999.
  https://amzn.to/2vORBWO.

- *Deep Learning*, 2016.
  https://amzn.to/2MQyLVZ

- *Deep Learning with Python*, 2017.
  https://amzn.to/2vMRiMe

7.7.2 APIs

- Keras: The Python Deep Learning library.
  https://keras.io/

- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/

- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

7.8 Summary

In this tutorial, you discovered how to develop a suite of Multilayer Perceptron, or MLP, models for a range of standard time series forecasting problems. Specifically, you learned:

- How to develop MLP models for univariate time series forecasting.

- How to develop MLP models for multivariate time series forecasting.

- How to develop MLP models for multi-step time series forecasting.

7.8.1 Next

In the next lesson, you will discover how to develop Convolutional Neural Network models for time series forecasting.

Chapter 8

How to Develop CNNs for Time Series Forecasting

Convolutional Neural Network models, or CNNs for short, can be applied to time series forecasting. There are many types of CNN models that can be used for each specific type of time series forecasting problem. In this tutorial, you will discover how to develop a suite of CNN models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem. After completing this tutorial, you will know:

- How to develop CNN models for univariate time series forecasting.

- How to develop CNN models for multivariate time series forecasting.

- How to develop CNN models for multi-step time series forecasting.

Let's get started.

8.1   Tutorial Overview

In this tutorial, we will explore how to develop CNN models for time series forecasting. The models are demonstrated on small contrived time series problems intended to give the flavor of the type of time series problem being addressed. The chosen configuration of the models is arbitrary and not optimized for each problem; that was not the goal. This tutorial is divided into four parts; they are:

1. Univariate CNN Models

2. Multivariate CNN Models

3. Multi-step CNN Models

4. Multivariate Multi-step CNN Models

8.2 Univariate CNN Models

Although traditionally developed for two-dimensional image data, CNNs can be used to model univariate time series forecasting problems. Univariate time series are datasets comprised of a single series of observations with a temporal ordering and a model is required to learn from the series of past observations to predict the next value in the sequence. This section is divided into two parts; they are:

1. Data Preparation

2. CNN Model

8.2.1 Data Preparation

Before a univariate series can be modeled, it must be prepared. The CNN model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the model can learn. Consider a given univariate sequence:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```
Listing 8.1: Example of a univariate time series.

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,          y
10, 20, 30,    40
20, 30, 40,    50
30, 40, 50,    60
...
```
Listing 8.2: Example of a univariate time series as a supervised learning problem.

The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```python
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```
Listing 8.3: Example of a function to split a univariate series into a supervised learning problem.

We can demonstrate this function on our small contrived dataset above. The complete example is listed below.

```python
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.4: Example of transforming a univariate time series into a supervised learning problem.

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Listing 8.5: Example output from transforming a univariate time series into a supervised learning problem.

Now that we know how to prepare a univariate series for modeling, let's look at developing a CNN model that can learn the mapping of inputs to outputs.

8.2.2   CNN Model

A one-dimensional CNN is a CNN model that has a convolutional hidden layer that operates over a 1D sequence. This is followed by perhaps a second convolutional layer in some cases, such as very long input sequences, and then a pooling layer whose job it is to distill the output of the convolutional layer to the most salient elements. The convolutional and pooling layers

are followed by a dense fully connected layer that interprets the features extracted by the convolutional part of the model. A flatten layer is used between the convolutional layers and the dense layer to reduce the feature maps to a single one-dimensional vector. We can define a 1D CNN Model for univariate time series forecasting as follows.

```python
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.6: Example of defining a CNN model.

Key in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features. We are working with a univariate series, so the number of features is one, for one variable. The number of time steps as input is the number we chose when preparing our dataset as an argument to the `split_sequence()` function.

The input shape for each sample is specified in the `input_shape` argument on the definition of the first hidden layer. We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: `[samples, timesteps, features]`. Our `split_sequence()` function in the previous section outputs the *X* with the shape `[samples, timesteps]`, so we can easily reshape it to have an additional dimension for the one feature.

```python
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 8.7: Example of reshaping data for the CNN.

The CNN does not actually view the data as having time steps, instead, it is treated as a sequence over which convolutional read operations can be performed, like a one-dimensional image. In this example, we define a convolutional layer with 64 filter maps and a kernel size of 2. This is followed by a max pooling layer and a dense layer to interpret the input feature. An output layer is specified that predicts a single numerical value. The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or 'mse', loss function. Once the model is defined, we can fit it on the training dataset.

```python
# fit model
model.fit(X, y, epochs=1000, verbose=0)
```

Listing 8.8: Example of fitting a CNN model.

After the model is fit, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: `[70, 80, 90]`. And expecting the model to predict something like: `[100]`. The model expects the input shape to be three-dimensional with `[samples, timesteps, features]`, therefore, we must reshape the single input sample before making the prediction.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.9: Example of reshaping data read for making a prediction.

We can tie all of this together and demonstrate how to develop a 1D CNN model for univariate time series forecasting and make a single prediction.

```
# univariate cnn example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
```

```
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.10: Example of a CNN model for univariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction. We can see that the model predicts the next value in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.67965]]
```

Listing 8.11: Example output from a CNN model for univariate time series forecasting.

For an example of a CNN applied to a real-world univariate time series forecasting problem see Chapter 14. For an example of grid searching CNN hyperparameters on a univariate time series forecasting problem, see Chapter 15.

8.3 Multivariate CNN Models

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.

2. Multiple Parallel Series.

Let's take a look at each in turn.

8.3.1 Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has observations at the same time steps. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

Listing 8.12: Example of defining multiple parallel series.

We can reshape these three arrays of data as a single dataset where each row is a time step and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

Listing 8.13: Example of defining parallel series as a dataset.

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
print(dataset)
```

Listing 8.14: Example of defining a dependent time series dataset.

Running the example prints the dataset with one row per time step and one column for each of the two input and one output parallel time series.

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.15: Example output from defining a dependent time series dataset.

As with the univariate time series, we must structure these data into samples with input and output samples. A 1D CNN model needs sufficient context to learn a mapping from an input sequence to an output value. CNNs can support parallel input time series as separate channels, like red, green, and blue components of an image. Therefore, we need to split the data into samples maintaining the order of observations across the two input sequences. If we chose three input time steps, then the first sample would look as follows:

Input:

```
10, 15
20, 25
30, 35
```

Listing 8.16: Example input from the first sample.

Output:

```
65
```

Listing 8.17: Example output from the first sample.

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case, 65. We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used. We can define a function named `split_sequences()` that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```python
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.18: Example of a function for preparing samples for a dependent time series.

We can test this function on our dataset using three time steps for each input time series as input. The complete example is listed below.

```python
# multivariate data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
```

```
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.19: Example of splitting a dependent series into samples.

Running the example first prints the shape of the X and y components. We can see that the X component has a three-dimensional structure. The first dimension is the number of samples, in this case 7. The second dimension is the number of time steps per sample, in this case 3, the value specified to the function. Finally, the last dimension specifies the number of parallel time series or the number of variables, in this case 2 for the two parallel series.

This is the exact three-dimensional structure expected by a 1D CNN as input. The data is ready to use without further reshaping. We can then see that the input and output for each sample is printed, showing the three time steps for each of the two input series and the associated output for each sample.

```
(7, 3, 2) (7,)

[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

Listing 8.20: Example output from splitting a dependent series into samples.

CNN Model

We are now ready to fit a 1D CNN model on this data, specifying the expected number of time steps and features to expect for each input sample, in this case three and two respectively.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.21: Example of defining a CNN for forecasting a dependent series.

When making a prediction, the model expects three time steps for two input time series. We can predict the next value in the output series providing the input values of:

```
80, 85
90, 95
100, 105
```

Listing 8.22: Example input for forecasting out-of-sample.

The shape of the one sample with three time steps and two variables must be `[1, 3, 2]`. We would expect the next value in the sequence to be $100 + 105$ or 205.

```
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.23: Example of preparing input for forecasting out-of-sample.

The complete example is listed below.

```
# multivariate cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
```

```python
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.24: Example of a CNN model for forecasting a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[206.0161]]
```

Listing 8.25: Example output from a CNN model for forecasting a dependent time series.

Multi-headed CNN Model

There is another, more elaborate way to model the problem. Each input series can be handled by a separate CNN and the output of each of these submodels can be combined before a prediction is made for the output sequence. We can refer to this as a multi-headed CNN model. It may offer more flexibility or better performance depending on the specifics of the problem that is

being modeled. For example, it allows you to configure each submodel differently for each input series, such as the number of filter maps and the kernel size. This type of model can be defined in Keras using the Keras functional API. First, we can define the first input model as a 1D CNN with an input layer that expects vectors with `n_steps` and 1 feature.

```
# first input model
visible1 = Input(shape=(n_steps, n_features))
cnn1 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible1)
cnn1 = MaxPooling1D(pool_size=2)(cnn1)
cnn1 = Flatten()(cnn1)
```

Listing 8.26: Example of defining the first input model.

We can define the second input submodel in the same way.

```
# second input model
visible2 = Input(shape=(n_steps, n_features))
cnn2 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible2)
cnn2 = MaxPooling1D(pool_size=2)(cnn2)
cnn2 = Flatten()(cnn2)
```

Listing 8.27: Example of defining the second input model.

Now that both input submodels have been defined, we can merge the output from each model into one long vector which can be interpreted before making a prediction for the output sequence.

```
# merge input models
merge = concatenate([cnn1, cnn2])
dense = Dense(50, activation='relu')(merge)
output = Dense(1)(dense)
```

Listing 8.28: Example of defining the output model.

We can then tie the inputs and outputs together.

```
# connect input and output models
model = Model(inputs=[visible1, visible2], outputs=output)
```

Listing 8.29: Example of connecting the input and output models.

The image below provides a schematic for how this model looks, including the shape of the inputs and outputs of each layer.
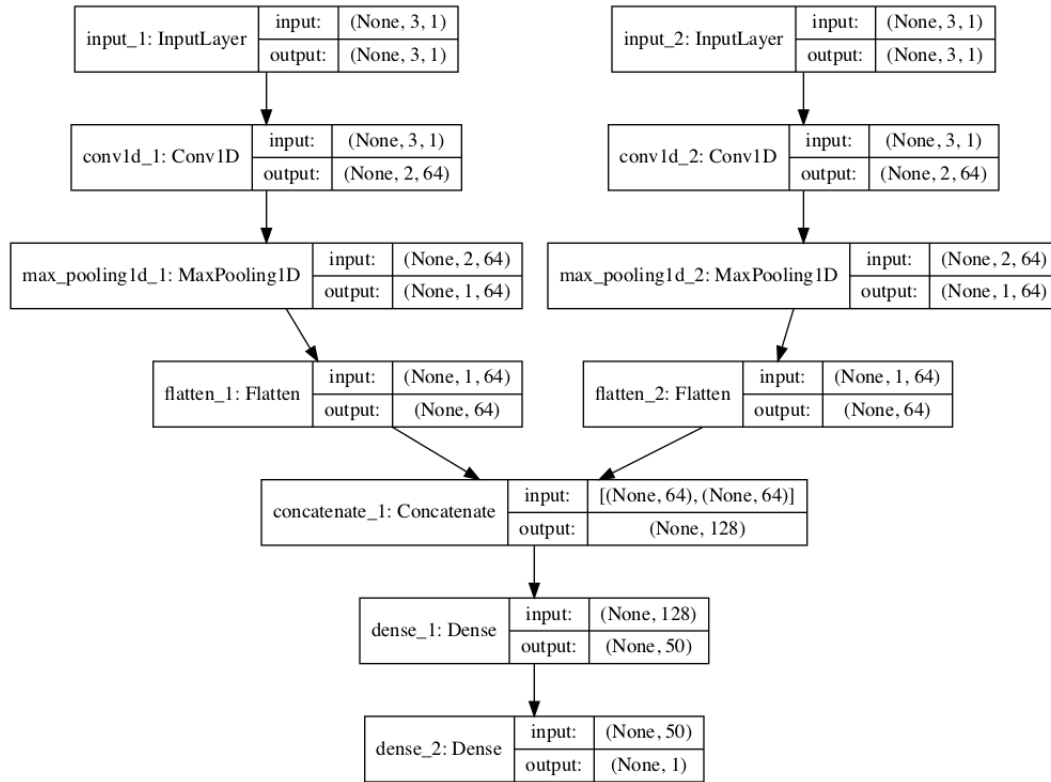
Figure 8.1: Plot of Multi-headed 1D CNN for Multivariate Time Series Forecasting.

This model requires input to be provided as a list of two elements where each element in the list contains data for one of the submodels. In order to achieve this, we can split the 3D input data into two separate arrays of input data; that is from one array with the shape [7, 3, 2] to two 3D arrays with [7, 3, 1].

```
# one time series per head
n_features = 1
# separate input data
X1 = X[:, :, 0].reshape(X.shape[0], X.shape[1], n_features)
X2 = X[:, :, 1].reshape(X.shape[0], X.shape[1], n_features)
```

Listing 8.30: Example of preparing the input data for the multi-headed model.

These data can then be provided in order to fit the model.

```
# fit model
model.fit([X1, X2], y, epochs=1000, verbose=0)
```

Listing 8.31: Example of fitting the multi-headed model.

Similarly, we must prepare the data for a single sample as two separate two-dimensional arrays when making a single one-step prediction.

```
# reshape one sample for making a prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps, n_features))
x2 = x_input[:, 1].reshape((1, n_steps, n_features))
```

Listing 8.32: Example of preparing data for forecasting with the multi-headed model.

We can tie all of this together; the complete example is listed below.

```python
# multivariate multi-headed 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.merge import concatenate

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# one time series per head
n_features = 1
# separate input data
X1 = X[:, :, 0].reshape(X.shape[0], X.shape[1], n_features)
X2 = X[:, :, 1].reshape(X.shape[0], X.shape[1], n_features)
# first input model
visible1 = Input(shape=(n_steps, n_features))
cnn1 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible1)
cnn1 = MaxPooling1D(pool_size=2)(cnn1)
cnn1 = Flatten()(cnn1)
# second input model
visible2 = Input(shape=(n_steps, n_features))
cnn2 = Conv1D(filters=64, kernel_size=2, activation='relu')(visible2)
cnn2 = MaxPooling1D(pool_size=2)(cnn2)
```

```
cnn2 = Flatten()(cnn2)
# merge input models
merge = concatenate([cnn1, cnn2])
dense = Dense(50, activation='relu')(merge)
output = Dense(1)(dense)
model = Model(inputs=[visible1, visible2], outputs=output)
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit([X1, X2], y, epochs=1000, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x1 = x_input[:, 0].reshape((1, n_steps, n_features))
x2 = x_input[:, 1].reshape((1, n_steps, n_features))
yhat = model.predict([x1, x2], verbose=0)
print(yhat)
```

Listing 8.33: Example of a Multi-headed CNN for forecasting a dependent time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[205.871]]
```

Listing 8.34: Example output from a Multi-headed CNN model for forecasting a dependent time series.

For an example of CNN models developed for a multivariate time series classification problem, see Chapter 24.

8.3.2 Multiple Parallel Series

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each. For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.35: Example of parallel time series.

We may want to predict the value for each of the three time series for the next time step. This might be referred to as multivariate forecasting. Again, the data must be split into input/output samples in order to train a model. The first sample of this dataset would be:

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 8.36: Example input from the first sample.

Output:

```
40, 45, 85
```

Listing 8.37: Example output from the first sample.

The `split_sequences()` function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.38: Example of splitting multiple parallel time series into samples.

We can demonstrate this on the contrived problem; the complete example is listed below.

```
# multivariate output data prep
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.39: Example of splitting multiple parallel series into samples.

Running the example first prints the shape of the prepared X and y components. The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3). The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3). The data is ready to use in a 1D CNN model that expects three-dimensional input and two-dimensional output shapes for the X and y components of each sample. Then, each of the samples is printed showing the input and output components of each sample.

```
(6, 3, 3) (6, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50 55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60 65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70 75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80 85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90 95 185]
```

Listing 8.40: Example output from splitting multiple parallel series into samples.

Vector-Output CNN Model

We are now ready to fit a 1D CNN model on this data. In this model, the number of time steps and parallel series (features) are specified for the input layer via the `input_shape` argument.

The number of parallel series is also used in the specification of the number of values to predict by the model in the output layer; again, this is three.

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.41: Example of defining a CNN model for forecasting multiple parallel time series.

We can predict the next value in each of the three parallel series by providing an input of three time steps for each series.

```
70, 75, 145
80, 85, 165
90, 95, 185
```

Listing 8.42: Example input for forecasting out-of-sample.

The shape of the input for making a single prediction must be 1 sample, 3 time steps, and 3 features, or [1, 3, 3].

```
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.43: Example of preparing data for forecasting out-of-sample.

We would expect the vector output to be: [100, 105, 205]. We can tie all of this together and demonstrate a 1D CNN for multivariate output time series forecasting below.

```
# multivariate output 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
```

```
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=3000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.44: Example of a CNN model for forecasting multiple parallel time series.

Running the example prepares the data, fits the model and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.11272 105.32213 205.53436]]
```

Listing 8.45: Example output from a CNN model for forecasting multiple parallel time series.

Multi-output CNN Model

As with multiple input series, there is another more elaborate way to model the problem. Each output series can be handled by a separate output CNN model. We can refer to this as a multi-output CNN model. It may offer more flexibility or better performance depending on the specifics of the problem that is being modeled. This type of model can be defined in Keras using the Keras functional API. First, we can define the first input model as a 1D CNN model.

```
# define model
visible = Input(shape=(n_steps, n_features))
cnn = Conv1D(filters=64, kernel_size=2, activation='relu')(visible)
cnn = MaxPooling1D(pool_size=2)(cnn)
cnn = Flatten()(cnn)
cnn = Dense(50, activation='relu')(cnn)
```

Listing 8.46: Example of defining the input model.

We can then define one output layer for each of the three series that we wish to forecast, where each output submodel will forecast a single time step.

```
# define output 1
output1 = Dense(1)(cnn)
# define output 2
output2 = Dense(1)(cnn)
# define output 3
output3 = Dense(1)(cnn)
```

Listing 8.47: Example of defining the output models.

We can then tie the input and output layers together into a single model.

```
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
```

Listing 8.48: Example of connecting the input and output models.

To make the model architecture clear, the schematic below clearly shows the three separate output layers of the model and the input and output shapes of each layer.
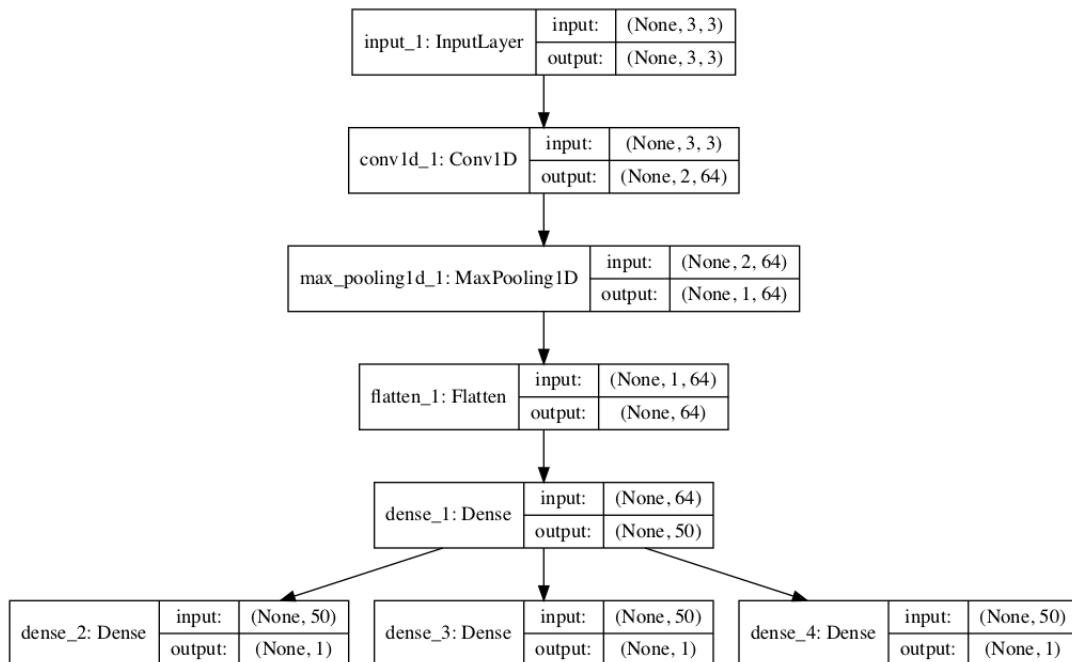
Figure 8.2: Plot of Multi-output 1D CNN for Multivariate Time Series Forecasting.

When training the model, it will require three separate output arrays per sample. We can achieve this by converting the output training data that has the shape [7, 3] to three arrays with the shape [7, 1].

```
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
```

Listing 8.49: Example of preparing the output samples for fitting the multi-output model.

These arrays can be provided to the model during training.

```
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
```

Listing 8.50: Example of fitting the multi-output model.

Tying all of this together, the complete example is listed below.

```
# multivariate output 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
```

```
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# separate output
y1 = y[:, 0].reshape((y.shape[0], 1))
y2 = y[:, 1].reshape((y.shape[0], 1))
y3 = y[:, 2].reshape((y.shape[0], 1))
# define model
visible = Input(shape=(n_steps, n_features))
cnn = Conv1D(filters=64, kernel_size=2, activation='relu')(visible)
cnn = MaxPooling1D(pool_size=2)(cnn)
cnn = Flatten()(cnn)
cnn = Dense(50, activation='relu')(cnn)
# define output 1
output1 = Dense(1)(cnn)
# define output 2
output2 = Dense(1)(cnn)
# define output 3
output3 = Dense(1)(cnn)
# tie together
model = Model(inputs=visible, outputs=[output1, output2, output3])
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, [y1,y2,y3], epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.51: Example of a Multi-output CNN model for forecasting multiple parallel time series.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[array([[100.96118]], dtype=float32),
 array([[105.502686]], dtype=float32),
 array([[205.98045]], dtype=float32)]
```

Listing 8.52: Example output from a Multi-output CNN model for forecasting multiple parallel time series.

For an example of CNN models developed for a multivariate time series forecasting problem, see Chapter 19.

8.4   Multi-step CNN Models

In practice, there is little difference to the 1D CNN model in predicting a vector output that represents different output variables (as in the previous example), or a vector output that represents multiple time steps of one variable. Nevertheless, there are subtle and important differences in the way the training data is prepared. In this section, we will demonstrate the

case of developing a multi-step forecast model using a vector model. Before we look at the specifics of the model, let's first look at the preparation of data for multi-step forecasting.

8.4.1   Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components. Both the input and output components will be comprised of multiple time steps and may or may not have the same number of steps. For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 8.53: Example of a univariate time series.

We could use the last three time steps as input and forecast the next two time steps. The first sample would look as follows:
Input:

```
[10, 20, 30]
```

Listing 8.54: Example input for the first sample.

Output:

```
[40, 50]
```

Listing 8.55: Example output for the first sample.

The `split_sequence()` function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.56: Example of function for splitting a univariate time series into samples for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
```

```python
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.57: Example transforming a time series into samples for multi-step forecasting.

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Listing 8.58: Example output from transforming a time series into samples for multi-step forecasting.

Now that we know how to prepare data for multi-step forecasting, let's look at a 1D CNN model that can learn this mapping.

8.4.2 Vector Output Model

The 1D CNN can output a vector directly that can be interpreted as a multi-step forecast. This approach was seen in the previous section were one time step of each output time series was forecasted as a vector. As with the 1D CNN models for univariate data in a prior section, the prepared samples must first be reshaped. The CNN expects data to have a three-dimensional structure of [samples, timesteps, features], and in this case, we only have one feature so the reshape is straightforward.

```python
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 8.59: Example of reshaping data for multi-step forecasting.

With the number of input and output steps specified in the `n_steps_in` and `n_steps_out` variables, we can define a multi-step time-series forecasting model.

```python
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
```

Listing 8.60: Example of defining a CNN model for multi-step forecasting.

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input: `[70, 80, 90]`. We would expect the predicted output to be: `[100, 110]`. As expected by the model, the shape of the single sample of input data when making the prediction must be `[1, 3, 1]` for the 1 sample, 3 time steps of the input, and the single feature.

```python
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 8.61: Example of reshaping data for making an out-of-sample forecast.

Tying all of this together, the 1D CNN for multi-step forecasting with a univariate time series is listed below.

```python
# univariate multi-step vector-output 1d cnn example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
```

```
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.62: Example of a vector-output CNN for multi-step forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.86651 115.08979]]
```

Listing 8.63: Example output from a vector-output CNN for multi-step forecasting.

For an example of CNN models developed for a multi-step time series forecasting problem, see Chapter 19.

8.5 Multivariate Multi-step CNN Models

In the previous sections, we have looked at univariate, multivariate, and multi-step time series forecasting. It is possible to mix and match the different types of 1D CNN models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging. In this section, we will explore short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

1. Multiple Input Multi-step Output.

2. Multiple Parallel Input and Multi-step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

8.5.1 Multiple Input Multi-step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.64: Example of a multivariate time series.

We may use three prior time steps of each of the two input time series to predict two time steps of the output time series.
Input:

```
10, 15
20, 25
30, 35
```

Listing 8.65: Example input for the first sample.

Output:

```
65
85
```

Listing 8.66: Example output for the first sample.

The `split_sequences()` function below implements this behavior.

```python
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 8.67: Example of a function for transforming a multivariate time series into samples for multi-step forecasting.

We can demonstrate this on our contrived dataset. The complete example is listed below.

```python
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.68: Example preparing a multivariate input dependent time series with multi-step forecasts.

Running the example first prints the shape of the prepared training data. We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three time steps and two variables for the two input time series. The output portion of the samples is two-dimensional for the six samples and the two time steps for each sample to be predicted. The prepared samples are then printed to confirm that the data was prepared as we specified.

```
(6, 3, 2) (6, 2)

[[10 15]
 [20 25]
 [30 35]] [65 85]
[[20 25]
```

```
 [30 35]
 [40 45]] [ 85 105]
[[30 35]
 [40 45]
 [50 55]] [105 125]
[[40 45]
 [50 55]
 [60 65]] [125 145]
[[50 55]
 [60 65]
 [70 75]] [145 165]
[[60 65]
 [70 75]
 [80 85]] [165 185]
```

Listing 8.69: Example output from preparing a multivariate input dependent time series with multi-step forecasts.

We can now develop a 1D CNN model for multi-step predictions. In this case, we will demonstrate a vector output model. The complete example is listed below.

```python
# multivariate multi-step 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

```python
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=2000, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.70: Example of a CNN model for multivariate dependent time series with multi-step forecasts.

Running the example fits the model and predicts the next two time steps of the output sequence beyond the dataset. We would expect the next two steps to be `[185, 205]`.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[185.57011 207.77893]]
```

Listing 8.71: Example output from a CNN model for multivariate dependent time series with multi-step forecasts.

8.5.2 Multiple Parallel Input and Multi-step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 8.72: Example of a multivariate time series.

We may use the last three time steps from each of the three time series as input to the model, and predict the next time steps of each of the three time series as output. The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 8.73: Example input for the first sample.

Output:

```
40, 45, 85
50, 55, 105
```

Listing 8.74: Example output for the first sample.

The `split_sequences()` function below implements this behavior.

```python
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
	X, y = list(), list()
	for i in range(len(sequences)):
		# find the end of this pattern
		end_ix = i + n_steps_in
		out_end_ix = end_ix + n_steps_out
		# check if we are beyond the dataset
		if out_end_ix > len(sequences):
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
		X.append(seq_x)
		y.append(seq_y)
	return array(X), array(y)
```

Listing 8.75: Example of a function for transforming a multivariate time series into samples for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```python
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
	X, y = list(), list()
	for i in range(len(sequences)):
		# find the end of this pattern
		end_ix = i + n_steps_in
		out_end_ix = end_ix + n_steps_out
		# check if we are beyond the dataset
		if out_end_ix > len(sequences):
			break
		# gather input and output parts of the pattern
		seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
```

```
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 8.76: Example preparing a multivariate parallel time series with multi-step forecasts.

Running the example first prints the shape of the prepared training dataset. We can see that both the input (X) and output (Y) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively. The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

```
(5, 3, 3) (5, 2, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [[ 40 45 85]
 [ 50 55 105]]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [[ 50 55 105]
 [ 60 65 125]]
[[ 30 35 65]
 [ 40 45 85]
 [ 50 55 105]] [[ 60 65 125]
 [ 70 75 145]]
[[ 40 45 85]
 [ 50 55 105]
 [ 60 65 125]] [[ 70 75 145]
 [ 80 85 165]]
[[ 50 55 105]
 [ 60 65 125]
 [ 70 75 145]] [[ 80 85 165]
 [ 90 95 185]]
```

Listing 8.77: Example output from preparing a multivariate parallel time series with multi-step forecasts.

We can now develop a 1D CNN model for this dataset. We will use a vector-output model in this case. As such, we must flatten the three-dimensional structure of the output portion of each sample in order to train the model. This means, instead of predicting two steps for each series, the model is trained on and expected to predict a vector of six numbers directly.

```python
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
```

Listing 8.78: Example of flattening output samples for training the model with vector output.

The complete example is listed below.

```python
# multivariate output multi-step 1d cnn example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# convert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# flatten output
n_output = y.shape[1] * y.shape[2]
y = y.reshape((y.shape[0], n_output))
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
```

```
# define model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=2, activation='relu', input_shape=(n_steps_in,
    n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(50, activation='relu'))
model.add(Dense(n_output))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=7000, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 8.79: Example of a CNN model for multivariate parallel time series with multi-step forecasts.

Running the example fits the model and predicts the values for each of the three time steps for the next two time steps beyond the end of the dataset. We would expect the values for these series and time steps to be as follows:

```
90, 95, 185
100, 105, 205
```

Listing 8.80: Example output for the out-of-sample forecast.

We can see that the model forecast gets reasonably close to the expected values.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[ 90.47855 95.621284 186.02629 100.48118 105.80815 206.52821 ]]
```

Listing 8.81: Example output from a CNN model for multivariate parallel time series with multi-step forecasts.

For an example of CNN models developed for a multivariate multi-step time series forecasting problem, see Chapter 19.

8.6   Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Problem Differences**. Explain the main changes to the CNN required when modeling each of univariate, multivariate and multi-step time series forecasting problems.

- **Experiment**. Select one example and modify it to work with your own small contrived dataset.

- **Develop Framework**. Use the examples in this chapter as the basis for a framework for automatically developing an CNN model for a given time series forecasting problem.

If you explore any of these extensions, I'd love to know.

8.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

8.7.1 Books

- *Deep Learning*, 2016.
  https://amzn.to/2MQyLVZ

- *Deep Learning with Python*, 2017.
  https://amzn.to/2vMRiMe

8.7.2 Papers

- *Backpropagation Applied to Handwritten Zip Code Recognition*, 1989.
  https://ieeexplore.ieee.org/document/6795724/

- *Gradient-based Learning Applied to Document Recognition*, 1998.
  https://ieeexplore.ieee.org/document/726791/

- *Very Deep Convolutional Networks for Large-Scale Image Recognition*, 2014.
  https://arxiv.org/abs/1409.1556

8.7.3 APIs

- Keras: The Python Deep Learning library.
  https://keras.io/

- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/

- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

8.8 Summary

In this tutorial, you discovered how to develop a suite of CNN models for a range of standard time series forecasting problems. Specifically, you learned:

- How to develop CNN models for univariate time series forecasting.

- How to develop CNN models for multivariate time series forecasting.

- How to develop CNN models for multi-step time series forecasting.

8.8.1 Next

In the next lesson, you will discover how to develop Recurrent Neural Network models for time series forecasting.

Chapter 9

How to Develop LSTMs for Time Series Forecasting

Long Short-Term Memory networks, or LSTMs for short, can be applied to time series forecasting. There are many types of LSTM models that can be used for each specific type of time series forecasting problem. In this tutorial, you will discover how to develop a suite of LSTM models for a range of standard time series forecasting problems. The objective of this tutorial is to provide standalone examples of each model on each type of time series problem as a template that you can copy and adapt for your specific time series forecasting problem.

After completing this tutorial, you will know:

- How to develop LSTM models for univariate time series forecasting.

- How to develop LSTM models for multivariate time series forecasting.

- How to develop LSTM models for multi-step time series forecasting.

Let's get started.

9.1 Tutorial Overview

In this tutorial, we will explore how to develop a suite of different types of LSTM models for time series forecasting. The models are demonstrated on small contrived time series problems intended to give the flavor of the type of time series problem being addressed. The chosen configuration of the models is arbitrary and not optimized for each problem; that was not the goal. This tutorial is divided into four parts; they are:

1. Univariate LSTM Models

2. Multivariate LSTM Models

3. Multi-step LSTM Models

4. Multivariate Multi-step LSTM Models

9.2 Univariate LSTM Models

LSTMs can be used to model univariate time series forecasting problems. These are problems comprised of a single series of observations and a model is required to learn from the series of past observations to predict the next value in the sequence. We will demonstrate a number of variations of the LSTM model for univariate time series forecasting. This section is divided into six parts; they are:

1. Data Preparation

2. Vanilla LSTM

3. Stacked LSTM

4. Bidirectional LSTM

5. CNN-LSTM

6. ConvLSTM

Each of these models are demonstrated for one-step univariate time series forecasting, but can easily be adapted and used as the input part of a model for other types of time series forecasting problems.

9.2.1 Data Preparation

Before a univariate series can be modeled, it must be prepared. The LSTM model will learn a function that maps a sequence of past observations as input to an output observation. As such, the sequence of observations must be transformed into multiple examples from which the LSTM can learn. Consider a given univariate sequence:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 9.1: Example of a univariate time series.

We can divide the sequence into multiple input/output patterns called samples, where three time steps are used as input and one time step is used as output for the one-step prediction that is being learned.

```
X,          y
10, 20, 30,   40
20, 30, 40,   50
30, 40, 50,   60
...
```

Listing 9.2: Example of a univariate time series as a supervised learning problem.

The `split_sequence()` function below implements this behavior and will split a given univariate sequence into multiple samples where each sample has a specified number of time steps and the output is a single time step.

```
# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.3: Example of a function to split a univariate series into a supervised learning problem.

We can demonstrate this function on our small contrived dataset above. The complete example is listed below.

```
# univariate data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.4: Example of transforming a univariate time series into a supervised learning problem.

Running the example splits the univariate series into six samples where each sample has three input time steps and one output time step.

```
[10 20 30] 40
[20 30 40] 50
[30 40 50] 60
```

```
[40 50 60] 70
[50 60 70] 80
[60 70 80] 90
```

Listing 9.5: Example output from transforming a univariate time series into a supervised learning problem.

Now that we know how to prepare a univariate series for modeling, let's look at developing LSTM models that can learn the mapping of inputs to outputs, starting with a Vanilla LSTM.

9.2.2 Vanilla LSTM

A Vanilla LSTM is an LSTM model that has a single hidden layer of LSTM units, and an output layer used to make a prediction. Key to LSTMs is that they offer native support for sequences. Unlike a CNN that reads across the entire input vector, the LSTM model reads one time step of the sequence at a time and builds up an internal state representation that can be used as a learned context for making a prediction. We can define a Vanilla LSTM for univariate time series forecasting as follows.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.6: Example of defining a Vanilla LSTM model.

Key in the definition is the shape of the input; that is what the model expects as input for each sample in terms of the number of time steps and the number of features. We are working with a univariate series, so the number of features is one, for one variable. The number of time steps as input is the number we chose when preparing our dataset as an argument to the `split_sequence()` function.

The shape of the input for each sample is specified in the `input_shape` argument on the definition of first hidden layer. We almost always have multiple samples, therefore, the model will expect the input component of training data to have the dimensions or shape: `[samples, timesteps, features]`. Our `split_sequence()` function in the previous section outputs the X with the shape `[samples, timesteps]`, so we easily reshape it to have an additional dimension for the one feature.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 9.7: Example of reshaping training data for the LSTM.

In this case, we define a model with 50 LSTM units in the hidden layer and an output layer that predicts a single numerical value. The model is fit using the efficient Adam version of stochastic gradient descent and optimized using the mean squared error, or 'mse' loss function. Once the model is defined, we can fit it on the training dataset.

```
# fit model
model.fit(X, y, epochs=200, verbose=0)
```

Listing 9.8: Example of fitting the LSTM model.

After the model is fit, we can use it to make a prediction. We can predict the next value in the sequence by providing the input: `[70, 80, 90]`. And expecting the model to predict something like: `[100]`. The model expects the input shape to be three-dimensional with `[samples, timesteps, features]`, therefore, we must reshape the single input sample before making the prediction.

```python
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.9: Example of preparing an input sample ready for making an out-of-sample forecast.

We can tie all of this together and demonstrate how to develop a Vanilla LSTM for univariate time series forecasting and make a single prediction.

```python
# univariate lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
```

```
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.10: Example of a Vanilla LSTM for univariate time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction. We can see that the model predicts the next value in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.09213]]
```

Listing 9.11: Example output from a Vanilla LSTM for univariate time series forecasting.

9.2.3 Stacked LSTM

Multiple hidden LSTM layers can be stacked one on top of another in what is referred to as a Stacked LSTM model. An LSTM layer requires a three-dimensional input and LSTMs by default will produce a two-dimensional output as an interpretation from the end of the sequence. We can address this by having the LSTM output a value for each time step in the input data by setting the `return_sequences=True` argument on the layer. This allows us to have 3D output from hidden LSTM layer as input to the next. We can therefore define a Stacked LSTM as follows.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.12: Example of defining a Stacked LSTM model.

We can tie this together; the complete code example is listed below.

```
# univariate stacked lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
```

```
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.13: Example of a Stacked LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[102.47341]]
```

Listing 9.14: Example output from a Stacked LSTM for univariate time series forecasting.

9.2.4   Bidirectional LSTM

On some sequence prediction problems, it can be beneficial to allow the LSTM model to learn the input sequence both forward and backwards and concatenate both interpretations. This is called a Bidirectional LSTM. We can implement a Bidirectional LSTM for univariate time series forecasting by wrapping the first hidden layer in a wrapper layer called Bidirectional. An example of defining a Bidirectional LSTM to read input both forward and backward is as follows.

```
# define model
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.15: Example of defining a Bidirectional LSTM model.

The complete example of the Bidirectional LSTM for univariate time series forecasting is listed below.

```python
# univariate bidirectional lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Bidirectional

# split a univariate sequence
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 3
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(Bidirectional(LSTM(50, activation='relu'), input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.16: Example of a Bidirectional LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.48093]]
```

Listing 9.17: Example output from a Bidirectional LSTM for univariate time series forecasting.

9.2.5 CNN-LSTM

A convolutional neural network, or CNN for short, is a type of neural network developed for working with two-dimensional image data. The CNN can be very effective at automatically extracting and learning features from one-dimensional sequence data such as univariate time series data. A CNN model can be used in a hybrid model with an LSTM backend where the CNN is used to interpret subsequences of input that together are provided as a sequence to an LSTM model to interpret. This hybrid model is called a CNN-LSTM.

The first step is to split the input sequences into subsequences that can be processed by the CNN model. For example, we can first split our univariate time series data into input/output samples with four steps as input and one as output. Each sample can then be split into two sub-samples, each with two time steps. The CNN can interpret each subsequence of two time steps and provide a time series of interpretations of the subsequences to the LSTM model to process as input. We can parameterize this and define the number of subsequences as `n_seq` and the number of time steps per subsequence as `n_steps`. The input data can then be reshaped to have the required structure: `[samples, subsequences, timesteps, features]`. For example:

```
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, subsequences, timesteps, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, n_steps, n_features))
```

Listing 9.18: Example of reshaping data for a CNN-LSTM model.

We want to reuse the same CNN model when reading in each sub-sequence of data separately. This can be achieved by wrapping the entire CNN model in a `TimeDistributed` wrapper that will apply the entire model once per input, in this case, once per input subsequence. The CNN model first has a convolutional layer for reading across the subsequence that requires a number of filters and a kernel size to be specified. The number of filters is the number of reads or interpretations of the input sequence. The kernel size is the number of time steps included of each *read* operation of the input sequence. The convolution layer is followed by a max pooling layer that distills the filter maps down to $\frac{1}{4}$ of their size that includes the most salient features. These structures are then flattened down to a single one-dimensional vector to be used as a single input time step to the LSTM layer.

```
# define the input cnn model
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1, activation='relu'),
    input_shape=(None, n_steps, n_features)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
```

Listing 9.19: Example of defining the CNN input model.

Next, we can define the LSTM part of the model that interprets the CNN model's read of the input sequence and makes a prediction.

```
# define the output model
```

```
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
```

Listing 9.20: Example of defining the LSTM output model.

We can tie all of this together; the complete example of a CNN-LSTM model for univariate time series forecasting is listed below.

```python
# univariate cnn lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, subsequences, timesteps, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, n_steps, n_features))
# define model
model = Sequential()
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=1, activation='relu'),
    input_shape=(None, n_steps, n_features)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(50, activation='relu'))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
```

```
x_input = x_input.reshape((1, n_seq, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.21: Example of a CNN-LSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.69263]]
```

Listing 9.22: Example output from a CNN-LSTM for univariate time series forecasting.

9.2.6 ConvLSTM

A type of LSTM related to the CNN-LSTM is the ConvLSTM, where the convolutional reading of input is built directly into each LSTM unit. The ConvLSTM was developed for reading two-dimensional spatial-temporal data, but can be adapted for use with univariate time series forecasting. The layer expects input as a sequence of two-dimensional images, therefore the shape of input data must be: `[samples, timesteps, rows, columns, features]`.

For our purposes, we can split each sample into subsequences where timesteps will become the number of subsequences, or `n_seq`, and columns will be the number of time steps for each subsequence, or `n_steps`. The number of rows is fixed at 1 as we are working with one-dimensional data. We can now reshape the prepared samples into the required structure.

```
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows, columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
```

Listing 9.23: Example of reshaping data for a ConvLSTM model.

We can define the ConvLSTM as a single layer in terms of the number of filters and a two-dimensional kernel size in terms of (`rows, columns`). As we are working with a one-dimensional series, the number of rows is always fixed to 1 in the kernel. The output of the model must then be flattened before it can be interpreted and a prediction made.

```
# define the input cnnlstm model
model.add(ConvLSTM2D(filters=64, kernel_size=(1,2), activation='relu', input_shape=(n_seq,
    1, n_steps, n_features)))
model.add(Flatten())
```

Listing 9.24: Example of defining the ConvLSTM input model.

The complete example of a ConvLSTM for one-step univariate time series forecasting is listed below.

```
# univariate convlstm example
from numpy import array
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import ConvLSTM2D

# split a univariate sequence into samples
def split_sequence(sequence, n_steps):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the sequence
    if end_ix > len(sequence)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps = 4
# split into samples
X, y = split_sequence(raw_seq, n_steps)
# reshape from [samples, timesteps] into [samples, timesteps, rows, columns, features]
n_features = 1
n_seq = 2
n_steps = 2
X = X.reshape((X.shape[0], n_seq, 1, n_steps, n_features))
# define model
model = Sequential()
model.add(ConvLSTM2D(filters=64, kernel_size=(1,2), activation='relu', input_shape=(n_seq,
    1, n_steps, n_features)))
model.add(Flatten())
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=500, verbose=0)
# demonstrate prediction
x_input = array([60, 70, 80, 90])
x_input = x_input.reshape((1, n_seq, 1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.25: Example of a ConvLSTM for univariate time series forecasting.

Running the example predicts the next value in the sequence, which we expect would be 100.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[103.68166]]
```

Listing 9.26: Example output from a ConvLSTM for univariate time series forecasting.

For an example of an LSTM applied to a real-world univariate time series forecasting problem see Chapter 14. For an example of grid searching LSTM hyperparameters on a univariate time series forecasting problem, see Chapter 15. Now that we have looked at LSTM models for univariate data, let's turn our attention to multivariate data.

9.3   Multivariate LSTM Models

Multivariate time series data means data where there is more than one observation for each time step. There are two main models that we may require with multivariate time series data; they are:

1. Multiple Input Series.

2. Multiple Parallel Series.

Let's take a look at each in turn.

9.3.1   Multiple Input Series

A problem may have two or more parallel input time series and an output time series that is dependent on the input time series. The input time series are parallel because each series has an observation at the same time steps. We can demonstrate this with a simple example of two parallel input time series where the output series is the simple addition of the input series.

```
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

Listing 9.27: Example of defining multiple input and a dependent time series.

We can reshape these three arrays of data as a single dataset where each row is a time step, and each column is a separate time series. This is a standard way of storing parallel time series in a CSV file.

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
```

Listing 9.28: Example of reshaping the parallel series into the columns of a dataset.

The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack
# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
print(dataset)
```

Listing 9.29: Example of defining a dependent series dataset.

Running the example prints the dataset with one row per time step and one column for each of the two input and one output parallel time series.

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.30: Example output from defining a dependent series dataset.

As with the univariate time series, we must structure these data into samples with input and output elements. An LSTM model needs sufficient context to learn a mapping from an input sequence to an output value. LSTMs can support parallel input time series as separate variables or features. Therefore, we need to split the data into samples maintaining the order of observations across the two input sequences. If we chose three input time steps, then the first sample would look as follows:

Input:

```
10, 15
20, 25
30, 35
```

Listing 9.31: Example input from the first sample.

Output:

```
65
```

Listing 9.32: Example Output from the first sample.

That is, the first three time steps of each parallel series are provided as input to the model and the model associates this with the value in the output series at the third time step, in this case, 65. We can see that, in transforming the time series into input/output samples to train the model, that we will have to discard some values from the output time series where we do

not have values in the input time series at prior time steps. In turn, the choice of the size of the number of input time steps will have an important effect on how much of the training data is used. We can define a function named `split_sequences()` that will take a dataset as we have defined it with rows for time steps and columns for parallel series and return input/output samples.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.33: Example of a function for transforming a dependent series dataset into samples.

We can test this function on our dataset using three time steps for each input time series as input. The complete example is listed below.

```
# multivariate data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
```

```
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.34: Example of splitting a dependent series dataset into samples.

Running the example first prints the shape of the X and y components. We can see that the X component has a three-dimensional structure. The first dimension is the number of samples, in this case 7. The second dimension is the number of time steps per sample, in this case 3, the value specified to the function. Finally, the last dimension specifies the number of parallel time series or the number of variables, in this case 2 for the two parallel series. This is the exact three-dimensional structure expected by an LSTM as input. The data is ready to use without further reshaping. We can then see that the input and output for each sample is printed, showing the three time steps for each of the two input series and the associated output for each sample.

```
(7, 3, 2) (7,)

[[10 15]
 [20 25]
 [30 35]] 65
[[20 25]
 [30 35]
 [40 45]] 85
[[30 35]
 [40 45]
 [50 55]] 105
[[40 45]
 [50 55]
 [60 65]] 125
[[50 55]
 [60 65]
 [70 75]] 145
[[60 65]
 [70 75]
 [80 85]] 165
[[70 75]
 [80 85]
 [90 95]] 185
```

Listing 9.35: Example output from splitting a dependent series dataset into samples.

We are now ready to fit an LSTM model on this data. Any of the varieties of LSTMs in the previous section can be used, such as a Vanilla, Stacked, Bidirectional, CNN, or ConvLSTM model. We will use a Vanilla LSTM where the number of time steps and parallel series (features) are specified for the input layer via the `input_shape` argument.

```
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
```

```
model.compile(optimizer='adam', loss='mse')
```

Listing 9.36: Example of defining a Vanilla LSTM for modeling a dependent series.

When making a prediction, the model expects three time steps for two input time series. We can predict the next value in the output series providing the input values of:

```
80,  85
90,  95
100, 105
```

Listing 9.37: Example input for making an out-of-sample forecast.

The shape of the one sample with three time steps and two variables must be `[1, 3, 2]`. We would expect the next value in the sequence to be $100 + 105$, or 205.

```
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.38: Example of making an out-of-sample forecast.

The complete example is listed below.

```
# multivariate lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
```

```
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(50, activation='relu', input_shape=(n_steps, n_features)))
model.add(Dense(1))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([[80, 85], [90, 95], [100, 105]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.39: Example of a Vanilla LSTM for multivariate dependent time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[208.13531]]
```

Listing 9.40: Example output from a Vanilla LSTM for multivariate dependent time series forecasting.

For an example of LSTM models developed for a multivariate time series classification problem, see Chapter 25.

9.3.2 Multiple Parallel Series

An alternate time series problem is the case where there are multiple parallel time series and a value must be predicted for each. For example, given the data from the previous section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.41: Example of a multivariate parallel time series.

We may want to predict the value for each of the three time series for the next time step. This might be referred to as multivariate forecasting. Again, the data must be split into input/output samples in order to train a model. The first sample of this dataset would be:

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 9.42: Example input from the first sample.

Output:

```
40, 45, 85
```

Listing 9.43: Example output from the first sample.

The split_sequences() function below will split multiple parallel time series with rows for time steps and one series per column into the required input/output shape.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.44: Example of a function for splitting parallel series into samples.

We can demonstrate this on the contrived problem; the complete example is listed below.

```
# multivariate output data prep
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
```

```
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.45: Example of splitting a multivariate parallel time series onto samples.

Running the example first prints the shape of the prepared X and y components. The shape of X is three-dimensional, including the number of samples (6), the number of time steps chosen per sample (3), and the number of parallel time series or features (3). The shape of y is two-dimensional as we might expect for the number of samples (6) and the number of time variables per sample to be predicted (3). The data is ready to use in an LSTM model that expects three-dimensional input and two-dimensional output shapes for the X and y components of each sample. Then, each of the samples is printed showing the input and output components of each sample.

```
(6, 3, 3) (6, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [40 45 85]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [ 50 55 105]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [ 60 65 125]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [ 70 75 145]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [ 80 85 165]
[[ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]] [ 90 95 185]
```

Listing 9.46: Example output from splitting a multivariate parallel time series onto samples.

We are now ready to fit an LSTM model on this data. Any of the varieties of LSTMs in the previous section can be used, such as a Vanilla, Stacked, Bidirectional, CNN, or ConvLSTM model. We will use a Stacked LSTM where the number of time steps and parallel series (features) are specified for the input layer via the **input_shape** argument. The number of parallel series is also used in the specification of the number of values to predict by the model in the output layer; again, this is three.

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.47: Example of defining a Stacked LSTM for parallel time series forecasting.

We can predict the next value in each of the three parallel series by providing an input of three time steps for each series.

```
70, 75, 145
80, 85, 165
90, 95, 185
```

Listing 9.48: Example input for making an out-of-sample forecast.

The shape of the input for making a single prediction must be 1 sample, 3 time steps, and 3 features, or [1, 3, 3].

```
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.49: Example of reshaping a sample for making an out-of-sample forecast.

We would expect the vector output to be:

```
[100, 105, 205]
```

Listing 9.50: Example output for an out-of-sample forecast.

We can tie all of this together and demonstrate a Stacked LSTM for multivariate output time series forecasting below.

```
# multivariate output stacked lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps
    # check if we are beyond the dataset
    if end_ix > len(sequences)-1:
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
```

```
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps = 3
# convert into input/output
X, y = split_sequences(dataset, n_steps)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_features))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=400, verbose=0)
# demonstrate prediction
x_input = array([[70,75,145], [80,85,165], [90,95,185]])
x_input = x_input.reshape((1, n_steps, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.51: Example of a Stacked LSTM for multivariate parallel time series forecasting.

Running the example prepares the data, fits the model, and makes a prediction.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[101.76599 108.730484 206.63577 ]]
```

Listing 9.52: Example output from a Stacked LSTM for multivariate parallel time series forecasting.

For an example of LSTM models developed for a multivariate time series forecasting problem, see Chapter 20.

9.4 Multi-step LSTM Models

A time series forecasting problem that requires a prediction of multiple time steps into the future can be referred to as multi-step time series forecasting. Specifically, these are problems where the forecast horizon or interval is more than one time step. There are two main types of LSTM models that can be used for multi-step forecasting; they are:

1. Vector Output Model

2. Encoder-Decoder Model

Before we look at these models, let's first look at the preparation of data for multi-step forecasting.

9.4.1   Data Preparation

As with one-step forecasting, a time series used for multi-step time series forecasting must be split into samples with input and output components. Both the input and output components will be comprised of multiple time steps and may or may not have the same number of steps. For example, given the univariate time series:

```
[10, 20, 30, 40, 50, 60, 70, 80, 90]
```

Listing 9.53: Example of a univariate time series.

We could use the last three time steps as input and forecast the next two time steps. The first sample would look as follows:

Input:

```
[10, 20, 30]
```

Listing 9.54: Example input from the first sample.

Output:

```
[40, 50]
```

Listing 9.55: Example output from the first sample.

The `split_sequence()` function below implements this behavior and will split a given univariate time series into samples with a specified number of input and output time steps.

```python
# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.56: Example of a function for splitting a univariate series into samples for multi-step forecasting.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multi-step data preparation
from numpy import array

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.57: Example of splitting a univariate series for multi-step forecasting into samples.

Running the example splits the univariate series into input and output time steps and prints the input and output components of each.

```
[10 20 30] [40 50]
[20 30 40] [50 60]
[30 40 50] [60 70]
[40 50 60] [70 80]
[50 60 70] [80 90]
```

Listing 9.58: Example output from splitting a univariate series for multi-step forecasting into samples.

Now that we know how to prepare data for multi-step forecasting, let's look at some LSTM models that can learn this mapping.

9.4.2 Vector Output Model

Like other types of neural network models, the LSTM can output a vector directly that can be interpreted as a multi-step forecast. This approach was seen in the previous section were one time step of each output time series was forecasted as a vector. As with the LSTMs for univariate data in a prior section, the prepared samples must first be reshaped. The LSTM expects data to have a three-dimensional structure of [samples, timesteps, features], and in this case, we only have one feature so the reshape is straightforward.

```
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 9.59: Example of preparing data for fitting an LSTM model.

With the number of input and output steps specified in the `n_steps_in` and `n_steps_out` variables, we can define a multi-step time-series forecasting model. Any of the presented LSTM model types could be used, such as Vanilla, Stacked, Bidirectional, CNN-LSTM, or ConvLSTM. Below defines a Stacked LSTM for multi-step forecasting.

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps_in,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.60: Example of defining a Stacked LSTM for multi-step forecasting.

The model can make a prediction for a single sample. We can predict the next two steps beyond the end of the dataset by providing the input:

```
[70, 80, 90]
```

Listing 9.61: Example input for making an out-of-sample forecast.

We would expect the predicted output to be:

```
[100, 110]
```

Listing 9.62: Expected output for making an out-of-sample forecast.

As expected by the model, the shape of the single sample of input data when making the prediction must be `[1, 3, 1]` for the 1 sample, 3 time steps of the input, and the single feature.

```
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
```

Listing 9.63: Example of preparing a sample for making an out-of-sample forecast.

Tying all of this together, the Stacked LSTM for multi-step forecasting with a univariate time series is listed below.

```
# univariate multi-step vector-output stacked lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
```

```
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps_in,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=50, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.64: Example of a Stacked LSTM for multi-step time series forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[100.98096 113.28924]]
```

Listing 9.65: Example output from a Stacked LSTM for multi-step time series forecasting.

9.4.3   Encoder-Decoder Model

A model specifically developed for forecasting variable length output sequences is called the Encoder-Decoder LSTM. The model was designed for prediction problems where there are both input and output sequences, so-called sequence-to-sequence, or seq2seq problems, such as translating text from one language to another. This model can be used for multi-step time series forecasting. As its name suggests, the model is comprised of two sub-models: the encoder and the decoder.

The encoder is a model responsible for reading and interpreting the input sequence. The output of the encoder is a fixed length vector that represents the model's interpretation of the sequence. The encoder is traditionally a Vanilla LSTM model, although other encoder models can be used such as Stacked, Bidirectional, and CNN models.

```
# define encoder model
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
```

Listing 9.66: Example of defining the encoder input model.

The decoder uses the output of the encoder as an input. First, the fixed-length output of the encoder is repeated, once for each required time step in the output sequence.

```
# repeat encoding
model.add(RepeatVector(n_steps_out))
```

Listing 9.67: Example of repeating the encoded vector.

This sequence is then provided to an LSTM decoder model. The model must output a value for each value in the output time step, which can be interpreted by a single output model.

```
# define decoder model
model.add(LSTM(100, activation='relu', return_sequences=True))
```

Listing 9.68: Example of defining the decoder model.

We can use the same output layer or layers to make each one-step prediction in the output sequence. This can be achieved by wrapping the output part of the model in a `TimeDistributed` wrapper.

```
# define model output
model.add(TimeDistributed(Dense(1)))
```

Listing 9.69: Example of defining the output model.

The full definition for an Encoder-Decoder model for multi-step time series forecasting is listed below.

```
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mse')
```

Listing 9.70: Example of defining an Encoder-Decoder LSTM for multi-step forecasting.

As with other LSTM models, the input data must be reshaped into the expected three-dimensional shape of `[samples, timesteps, features]`.

```
# reshape input training data
X = X.reshape((X.shape[0], X.shape[1], n_features))
```

Listing 9.71: Example of reshaping input samples for training the Encoder-Decoder LSTM.

In the case of the Encoder-Decoder model, the output, or y part, of the training dataset must also have this shape. This is because the model will predict a given number of time steps with a given number of features for each input sample.

```
# reshape output training data
y = y.reshape((y.shape[0], y.shape[1], n_features))
```

Listing 9.72: Example of reshaping output samples for training the Encoder-Decoder LSTM.

The complete example of an Encoder-Decoder LSTM for multi-step time series forecasting is listed below.

```
# univariate multi-step encoder-decoder lstm example
from numpy import array
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a univariate sequence into samples
def split_sequence(sequence, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequence)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the sequence
    if out_end_ix > len(sequence):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequence[i:end_ix], sequence[end_ix:out_end_ix]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
raw_seq = [10, 20, 30, 40, 50, 60, 70, 80, 90]
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# split into samples
X, y = split_sequence(raw_seq, n_steps_in, n_steps_out)
# reshape from [samples, timesteps] into [samples, timesteps, features]
n_features = 1
X = X.reshape((X.shape[0], X.shape[1], n_features))
y = y.reshape((y.shape[0], y.shape[1], n_features))
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(100, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(1)))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=100, verbose=0)
# demonstrate prediction
x_input = array([70, 80, 90])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.73: Example of an Encoder-Decoder LSTM for multi-step time series forecasting.

Running the example forecasts and prints the next two time steps in the sequence.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[[101.9736
  [116.213615]]]
```

Listing 9.74: Example output from an Encoder-Decoder LSTM for multi-step time series forecasting.

For an example of LSTM models developed for a multi-step time series forecasting problem, see Chapter 20.

9.5 Multivariate Multi-step LSTM Models

In the previous sections, we have looked at univariate, multivariate, and multi-step time series forecasting. It is possible to mix and match the different types of LSTM models presented so far for the different problems. This too applies to time series forecasting problems that involve multivariate and multi-step forecasting, but it may be a little more challenging. In this section, we will provide short examples of data preparation and modeling for multivariate multi-step time series forecasting as a template to ease this challenge, specifically:

1. Multiple Input Multi-step Output.

2. Multiple Parallel Input and Multi-step Output.

Perhaps the biggest stumbling block is in the preparation of data, so this is where we will focus our attention.

9.5.1 Multiple Input Multi-step Output

There are those multivariate time series forecasting problems where the output series is separate but dependent upon the input time series, and multiple time steps are required for the output series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.75: Example of a multivariate dependent time series.

We may use three prior time steps of each of the two input time series to predict two time steps of the output time series.

Input:

```
10, 15
20, 25
30, 35
```

Listing 9.76: Example of input from the first sample.

Output:

```
65
85
```

Listing 9.77: Example of output from the first sample.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.78: Example of a function for splitting a dependent series for multi-step forecasting into samples.

We can demonstrate this on our contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
```

```
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.79: Example splitting a parallel series for multi-step forecasting into samples.

Running the example first prints the shape of the prepared training data. We can see that the shape of the input portion of the samples is three-dimensional, comprised of six samples, with three time steps, and two variables for the 2 input time series. The output portion of the samples is two-dimensional for the six samples and the two time steps for each sample to be predicted. The prepared samples are then printed to confirm that the data was prepared as we specified.

```
(6, 3, 2) (6, 2)

[[10 15]
 [20 25]
 [30 35]] [65 85]
[[20 25]
 [30 35]
 [40 45]] [ 85 105]
[[30 35]
 [40 45]
 [50 55]] [105 125]
[[40 45]
 [50 55]
 [60 65]] [125 145]
[[50 55]
 [60 65]
 [70 75]] [145 165]
[[60 65]
 [70 75]
 [80 85]] [165 185]
```

Listing 9.80: Example output from splitting a parallel series for multi-step forecasting into samples.

We can now develop an LSTM model for multi-step predictions. A vector output or an encoder-decoder model could be used. In this case, we will demonstrate a vector output with a

Stacked LSTM. The complete example is listed below.

```python
# multivariate multi-step stacked lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out-1
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :-1], sequences[end_ix-1:out_end_ix, -1]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(100, activation='relu', return_sequences=True, input_shape=(n_steps_in,
    n_features)))
model.add(LSTM(100, activation='relu'))
model.add(Dense(n_steps_out))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=200, verbose=0)
# demonstrate prediction
x_input = array([[70, 75], [80, 85], [90, 95]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.81: Example of an Stacked LSTM for multi-step forecasting for a dependent series.

Running the example fits the model and predicts the next two time steps of the output sequence beyond the dataset. We would expect the next two steps to be: `[185, 205]`. It is a challenging framing of the problem with very little data, and the arbitrarily configured version of the model gets close.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[188.70619 210.16513]]
```

Listing 9.82: Example output from an Stacked LSTM for multi-step forecasting for a dependent series.

9.5.2 Multiple Parallel Input and Multi-step Output

A problem with parallel time series may require the prediction of multiple time steps of each time series. For example, consider our multivariate time series from a prior section:

```
[[ 10  15  25]
 [ 20  25  45]
 [ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]
 [ 80  85 165]
 [ 90  95 185]]
```

Listing 9.83: Example of a multivariate parallel time series dataset.

We may use the last three time steps from each of the three time series as input to the model and predict the next time steps of each of the three time series as output. The first sample in the training dataset would be the following.

Input:

```
10, 15, 25
20, 25, 45
30, 35, 65
```

Listing 9.84: Example of input from the first sample.

Output:

```
40, 45, 85
50, 55, 105
```

Listing 9.85: Example of output from the first sample.

The `split_sequences()` function below implements this behavior.

```
# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)
```

Listing 9.86: Example of a function for splitting a parallel dataset for multi-step forecasting into samples.

We can demonstrate this function on the small contrived dataset. The complete example is listed below.

```
# multivariate multi-step data preparation
from numpy import array
from numpy import hstack

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
```

```
print(X.shape, y.shape)
# summarize the data
for i in range(len(X)):
  print(X[i], y[i])
```

Listing 9.87: Example splitting a parallel series for multi-step forecasting into samples.

Running the example first prints the shape of the prepared training dataset. We can see that both the input (X) and output (Y) elements of the dataset are three dimensional for the number of samples, time steps, and variables or parallel time series respectively. The input and output elements of each series are then printed side by side so that we can confirm that the data was prepared as we expected.

```
(5, 3, 3) (5, 2, 3)

[[10 15 25]
 [20 25 45]
 [30 35 65]] [[ 40 45 85]
 [ 50  55 105]]
[[20 25 45]
 [30 35 65]
 [40 45 85]] [[ 50 55 105]
 [ 60  65 125]]
[[ 30  35  65]
 [ 40  45  85]
 [ 50  55 105]] [[ 60 65 125]
 [ 70 75 145]]
[[ 40  45  85]
 [ 50  55 105]
 [ 60  65 125]] [[ 70 75 145]
 [ 80 85 165]]
[[ 50  55 105]
 [ 60  65 125]
 [ 70  75 145]] [[ 80 85 165]
 [ 90 95 185]]
```

Listing 9.88: Example output from splitting a parallel series for multi-step forecasting into samples.

We can use either the Vector Output or Encoder-Decoder LSTM to model this problem. In this case, we will use the Encoder-Decoder model. The complete example is listed below.

```
# multivariate multi-step encoder-decoder lstm example
from numpy import array
from numpy import hstack
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dense
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a multivariate sequence into samples
def split_sequences(sequences, n_steps_in, n_steps_out):
  X, y = list(), list()
  for i in range(len(sequences)):
    # find the end of this pattern
```

```
    end_ix = i + n_steps_in
    out_end_ix = end_ix + n_steps_out
    # check if we are beyond the dataset
    if out_end_ix > len(sequences):
      break
    # gather input and output parts of the pattern
    seq_x, seq_y = sequences[i:end_ix, :], sequences[end_ix:out_end_ix, :]
    X.append(seq_x)
    y.append(seq_y)
  return array(X), array(y)

# define input sequence
in_seq1 = array([10, 20, 30, 40, 50, 60, 70, 80, 90])
in_seq2 = array([15, 25, 35, 45, 55, 65, 75, 85, 95])
out_seq = array([in_seq1[i]+in_seq2[i] for i in range(len(in_seq1))])
# convert to [rows, columns] structure
in_seq1 = in_seq1.reshape((len(in_seq1), 1))
in_seq2 = in_seq2.reshape((len(in_seq2), 1))
out_seq = out_seq.reshape((len(out_seq), 1))
# horizontally stack columns
dataset = hstack((in_seq1, in_seq2, out_seq))
# choose a number of time steps
n_steps_in, n_steps_out = 3, 2
# covert into input/output
X, y = split_sequences(dataset, n_steps_in, n_steps_out)
# the dataset knows the number of features, e.g. 2
n_features = X.shape[2]
# define model
model = Sequential()
model.add(LSTM(200, activation='relu', input_shape=(n_steps_in, n_features)))
model.add(RepeatVector(n_steps_out))
model.add(LSTM(200, activation='relu', return_sequences=True))
model.add(TimeDistributed(Dense(n_features)))
model.compile(optimizer='adam', loss='mse')
# fit model
model.fit(X, y, epochs=300, verbose=0)
# demonstrate prediction
x_input = array([[60, 65, 125], [70, 75, 145], [80, 85, 165]])
x_input = x_input.reshape((1, n_steps_in, n_features))
yhat = model.predict(x_input, verbose=0)
print(yhat)
```

Listing 9.89: Example of an Encoder-Decoder LSTM for multi-step forecasting for parallel series.

Running the example fits the model and predicts the values for each of the three time steps for the next two time steps beyond the end of the dataset. We would expect the values for these series and time steps to be as follows:

```
90, 95, 185
100, 105, 205
```

Listing 9.90: Expected output for an out-of-sample forecast.

We can see that the model forecast gets reasonably close to the expected values.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
[[[ 91.86044 97.77231 189.66768 ]
  [103.299355 109.18123 212.6863 ]]]
```

Listing 9.91: Example output from an Encoder-Decoder Output LSTM for multi-step forecasting for parallel series.

For an example of LSTM models developed for a multivariate multi-step time series forecasting problem, see Chapter 20.

9.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Problem Differences**. Explain the main changes to the LSTM required when modeling each of univariate, multivariate and multi-step time series forecasting problems.

- **Experiment**. Select one example and modify it to work with your own small contrived dataset.

- **Develop Framework**. Use the examples in this chapter as the basis for a framework for automatically developing an LSTM model for a given time series forecasting problem.

If you explore any of these extensions, I'd love to know.

9.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

9.7.1 Books

- *Deep Learning*, 2016.
  https://amzn.to/2MQyLVZ

- *Deep Learning with Python*, 2017.
  https://amzn.to/2vMRiMe

9.7.2 Papers

- *Long Short-Term Memory*, 1997.
  https://ieeexplore.ieee.org/document/6795963/.

- *Learning to Forget: Continual Prediction with LSTM*, 1999.
  https://ieeexplore.ieee.org/document/818041/

- *Recurrent Nets that Time and Count*, 2000.
  https://ieeexplore.ieee.org/document/861302/

- *LSTM: A Search Space Odyssey*, 2017.
  https://arxiv.org/abs/1503.04069

- *Convolutional LSTM Network: A Machine Learning Approach for Precipitation Nowcasting*, 2015.
  https://arxiv.org/abs/1506.04214v1

9.7.3 APIs

- Keras: The Python Deep Learning library.
  https://keras.io/

- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/

- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

9.8 Summary

In this tutorial, you discovered how to develop a suite of LSTM models for a range of standard time series forecasting problems. Specifically, you learned:

- How to develop LSTM models for univariate time series forecasting.

- How to develop LSTM models for multivariate time series forecasting.

- How to develop LSTM models for multi-step time series forecasting.

9.8.1 Next

This is the final lesson of this part, the next part will focus on systematically developing models for univariate time series forecasting problems.

Part IV

Univariate Forecasting

Overview

This part highlights that classical methods are known to out-perform more sophisticated methods on univariate time series forecasting problems on average and how to systematically evaluate classical and deep learning methods to ensure that you are getting the most out of them on your forecasting problem. As such, the chapters in this section focus on providing you the tools for grid searching classical and deep learning methods and demonstrating these tools on a suite of standard univariate datasets. After reading the chapters in this part, you will know:

- The results of a recent and reasonably conclusive study that classical methods out-perform machine learning and deep learning methods for univariate time series forecasting problems on average (Chapter 10).

- How to develop and grid search a suite of naive forecasting methods for univariate time series, the results of which can be used as a baseline for determining whether a model has skill (Chapter 11).

- How to develop and grid search exponential smoothing forecasting models, also known as ETS, for univariate time series forecasting problems (Chapter 12).

- How to develop and grid search Seasonal ARIMA models or SARIMA for univariate time series forecasting problems (Chapter 13).

- How to develop and evaluate MLP, CNN and LSTM deep learning models for univariate time series forecasting (Chapter 14).

- How to grid search the hyperparameters for MLP, CNN and LSTM deep learning models for univariate time series forecasting (Chapter 15).

Chapter 10

Review of Top Methods For Univariate Time Series Forecasting

Machine learning and deep learning methods are often reported to be the key solution to all predictive modeling problems. An important recent study evaluated and compared the performance of many classical and modern machine learning and deep learning methods on a large and diverse set of more than 1,000 univariate time series forecasting problems. The results of this study suggest that simple classical methods, such as linear methods and exponential smoothing, outperform complex and sophisticated methods, such as decision trees, Multilayer Perceptrons (MLP), and Long Short-Term Memory (LSTM) network models. These findings highlight the requirement to both evaluate classical methods and use their results as a baseline when evaluating any machine learning and deep learning methods for time series forecasting in order to demonstrate that their added complexity is adding skill to the forecast.

In this tutorial, you will discover the important findings of this recent study evaluating and comparing the performance of a classical and modern machine learning methods on a large and diverse set of time series forecasting datasets. After reading this tutorial, you will know:

- Classical methods like ETS and ARIMA out-perform machine learning and deep learning methods for one-step forecasting on univariate datasets.

- Classical methods like Theta and ARIMA out-perform machine learning and deep learning methods for multi-step forecasting on univariate datasets.

- Machine learning and deep learning methods do not yet deliver on their promise for univariate time series forecasting, and there is much work to do.

Let's get started.

10.1 Overview

Spyros Makridakis, et al. published a study in 2018 titled *Statistical and Machine Learning forecasting methods: Concerns and ways forward*. In this tutorial, we will take a close look at the study by Makridakis, et al. that carefully evaluated and compared classical time series forecasting methods to the performance of modern machine learning methods. This tutorial is divided into seven sections; they are:

10.2   Study Motivation

The goal of the study was to clearly demonstrate the capability of a suite of different machine learning methods as compared to classical time series forecasting methods on a very large and diverse collection of univariate time series forecasting problems. The study was a response to the increasing number of papers and claims that machine learning and deep learning methods offer superior results for time series forecasting with little objective evidence.

> Literally hundreds of papers propose new ML algorithms, suggesting methodological advances and accuracy improvements. Yet, limited objective evidence is available regarding their relative performance as a standard forecasting tool.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

The authors clearly lay out three issues with the flood of claims; they are:

- Their conclusions are based on a few, or even a single time series, raising questions about the statistical significance of the results and their generalization.

- The methods are evaluated for short-term forecasting horizons, often one-step-ahead, not considering medium and long-term ones.

- No benchmarks are used to compare the accuracy of ML methods versus alternative ones.

As a response, the study includes eight classical methods and 10 machine learning methods evaluated using one-step and multiple-step forecasts across a collection of 1,045 monthly time series. Although not definitive, the results are intended to be objective and robust.

10.3   Time Series Datasets

The time series datasets used in the study were drawn from the time series datasets used in the M3-Competition. The M3-Competition was the third in a series of competitions that sought to discover exactly what algorithms perform well in practice on real time series forecasting problems. The results of the competition were published in the 2000 paper titled *The M3-Competition: Results, Conclusions and Implications*. The datasets used in the competition were drawn from a wide range of industries and had a range of different time intervals, from hourly to annual.

The 3003 series of the M3-Competition were selected on a quota basis to include various types of time series data (micro, industry, macro, etc.) and different time intervals between successive observations (yearly, quarterly, etc.).

— *The M3-Competition: Results, Conclusions and Implications*, 2000.

The table below, taken from the paper, provides a summary of the 3,003 datasets used in the competition.

Time interval between successive observations	Types of time series data						
	Micro	Industry	Macro	Finance	Demographic	Other	Total
Yearly	146	102	83	58	245	11	645
Quarterly	204	83	336	76	57		756
Monthly	474	334	312	145	111	52	1428
Other	4			29		141	174
Total	828	519	731	308	413	204	3003

Figure 10.1: Table of Datasets, Industry and Time Interval Used in the M3-Competition. Taken from *The M3-Competition: Results, Conclusions and Implications*.

The finding of the competition was that simpler time series forecasting methods outperform more sophisticated methods, including neural network models.

This study, the previous two M-Competitions and many other empirical studies have proven, beyond the slightest doubt, that elaborate theoretical constructs or more sophisticated methods do not necessarily improve post-sample forecasting accuracy, over simple methods, although they can better fit a statistical model to the available historical data.

— *The M3-Competition: Results, Conclusions and Implications*, 2000.

The more recent study that we are reviewing in this tutorial that evaluate machine learning methods selected a subset of 1,045 time series with a monthly interval from those used in the M3 competition.

... evaluate such performance across multiple forecasting horizons using a large subset of 1045 monthly time series used in the M3 Competition.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.4   Time Series Forecasting Methods

The study evaluates the performance of eight classical (or simpler) methods and 10 machine learning methods.

... of eight traditional statistical methods and eight popular ML ones, [...], plus two more that have become popular during recent years.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

The eight classical methods evaluated were as follows:

- Naive 2, which is actually a random walk model adjusted for season.

- Simple Exponential Smoothing.

- Holt.

- Damped exponential smoothing.

- Average of SES, Holt, and Damped.

- Theta method.

- ARIMA, automatic.

- ETS, automatic.

A total of eight machine learning methods were used in an effort to reproduce and compare to results presented in the 2010 paper *An Empirical Comparison of Machine Learning Models for Time Series Forecasting*. They were:

- Multilayer Perceptron (MLP).

- Bayesian Neural Network (BNN).

- Radial Basis Functions (RBF).

- Generalized Regression Neural Networks (GRNN), also called kernel regression.

- k-Nearest Neighbor regression (KNN).

- CART regression trees (CART).

- Support Vector Regression (SVR).

- Gaussian Processes (GP).

An additional two *modern* neural network algorithms were also added to the list given the recent rise in their adoption; they were:

- Recurrent Neural Network (RNN).

- Long Short-Term Memory (LSTM).

10.5 Data Preparation

A careful data preparation methodology was used, again, based on the methodology described in the 2010 paper *An Empirical Comparison of Machine Learning Models for Time Series Forecasting*. In that paper, each time series was adjusted using a power transform, deseasonalized and detrended.

> ... before computing the 18 forecasts, they preprocessed the series in order to achieve stationarity in their mean and variance. This was done using the log transformation, then deseasonalization and finally scaling, while first differences were also considered for removing the component of trend.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

Inspired by these operations, variations of five different data transforms were applied for an MLP for one-step forecasting and their results were compared. The five transforms were:

- Original data.

- Box-Cox Power Transform.

- Deseasonalizing the data.

- Detrending the data.

- All three transforms (power, deseasonalize, detrend).

Generally, it was found that the best approach was to apply a power transform and deseasonalize the data, and perhaps detrend the series as well.

> The best combination according to sMAPE is number 7 (Box-Cox transformation, deseasonalization) while the best one according to MASE is number 10 (Box-Cox transformation, deseasonalization and detrending)

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.6 One-step Forecasting Results

All models were evaluated using one-step time series forecasting. Specifically, the last 18 time steps were used as a test set, and models were fit on all remaining observations. A separate one-step forecast was made for each of the 18 observations in the test set, presumably using a walk-forward validation method where true observations were used as input in order to make each forecast.

> The forecasting model was developed using the first n - 18 observations, where n is the length of the series. Then, 18 forecasts were produced and their accuracy was evaluated compared to the actual values not used in developing the forecasting model.

— Statistical and Machine Learning forecasting methods: Concerns and ways forward, 2018

Reviewing the results, the MLP and BNN were found to achieve the best performance from all of the machine learning methods.

> The results [...] show that MLP and BNN outperform the remaining ML methods.

— Statistical and Machine Learning forecasting methods: Concerns and ways forward, 2018.

An interesting result was that RNNs and LSTMs were found to perform poorly. This confirms a earlier finding in the application of LSTMs to univariate time series forecast reported by Gers, et al. in 2001 titled *Applying LSTM to Time Series Predictable through Time-Window Approaches*. In that study, they also reported that LSTMs are easily outperformed by simpler methods and may not be suited to simple autoregressive-type univariate time series forecasting problems.

> It should be noted that RNN is among the less accurate ML methods, demonstrating that research progress does not necessarily guarantee improvements in forecasting performance. This conclusion also applies in the performance of LSTM, another popular and more advanced ML method, which does not enhance forecasting accuracy too.

— Statistical and Machine Learning forecasting methods: Concerns and ways forward, 2018.

Comparing the performance of all methods, it was found that the machine learning methods were all out-performed by simple classical methods, where ETS and ARIMA models performed the best overall. This finding confirms the results from previous similar studies and competitions.
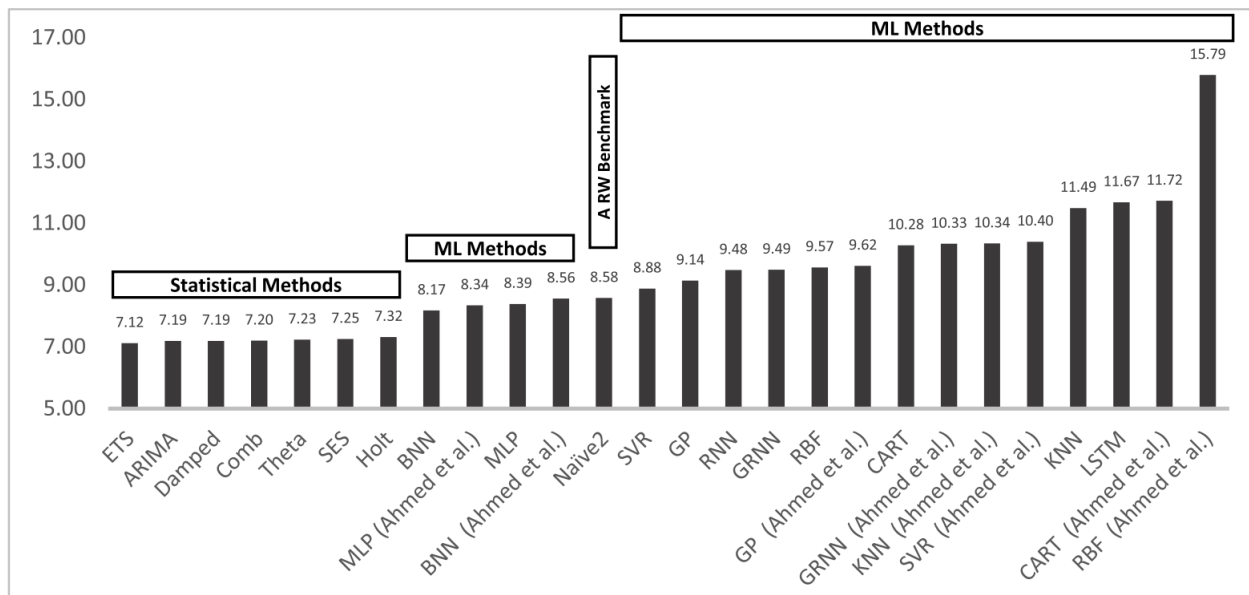

Figure 10.2: Bar Chart Comparing Model Performance (sMAPE) for One-step Forecasts. Taken from *Statistical and Machine Learning forecasting methods: Concerns and ways forward*.

10.7 Multi-step Forecasting Results

Multi-step forecasting involves predicting multiple steps ahead of the last known observation. Three approaches to multi-step forecasting were evaluated for the machine learning methods; they were:

- Iterative forecasting

- Direct forecasting

- Multi-neural network forecasting

The classical methods were found to outperform the machine learning methods again. In this case, methods such as Theta, ARIMA, and a combination of exponential smoothing (Comb) were found to achieve the best performance.

> In brief, statistical models seem to generally outperform ML methods across all forecasting horizons, with Theta, Comb and ARIMA being the dominant ones among the competitors according to both error metrics examined.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.8 Outcomes

The study provides important supporting evidence that classical methods may dominate univariate time series forecasting, at least on the types of forecasting problems evaluated. The study demonstrates the worse performance and the increase in computational cost of machine learning and deep learning methods for univariate time series forecasting for both one-step and multi-step forecasts.

These findings strongly encourage the use of classical methods, such as ETS, ARIMA, and others as a first step before more elaborate methods are explored, and requires that the results from these simpler methods be used as a baseline in performance that more elaborate methods must clear in order to justify their usage. It also highlights the need to not just consider the careful use of data preparation methods, but to actively test multiple different combinations of data preparation schemes for a given problem in order to discover what works best, even in the case of classical methods.

Machine learning and deep learning methods may still achieve better performance on specific univariate time series problems and should be evaluated. The study does not look at more complex time series problems, such as those datasets with:

- Complex irregular temporal structures.

- Missing observations

- Heavy noise.

- Complex interrelationships between multiple variates.

The study concludes with an honest puzzlement at why machine learning methods perform so poorly in practice, given their impressive performance in other areas of artificial intelligence.

> The most interesting question and greatest challenge is to find the reasons for their poor performance with the objective of improving their accuracy and exploiting their huge potential. AI learning algorithms have revolutionized a wide range of applications in diverse fields and there is no reason that the same cannot be achieved with the ML methods in forecasting. Thus, we must find how to be applied to improve their ability to forecast more accurately.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

Comments are made by the authors regarding LSTMs and RNNs, that are generally believed to be the deep learning approach for sequence prediction problems in general, and in this case their clearly poor performance in practice.

> ... one would expect RNN and LSTM, which are more advanced types of NNs, to be far more accurate than the ARIMA and the rest of the statistical methods utilized.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

They comment that LSTMs appear to be more suited at fitting or overfitting the training dataset rather than forecasting it.

> Another interesting example could be the case of LSTM that compared to simpler NNs like RNN and MLP, report better model fitting but worse forecasting accuracy

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

There is work to do and machine learning methods and deep learning methods hold the promise of better learning time series data than classical statistical methods, and even doing so directly on the raw observations via automatic feature learning.

> Given their ability to learn, ML methods should do better than simple benchmarks, like exponential smoothing. Accepting the problem is the first step in devising workable solutions and we hope that those in the field of AI and ML will accept the empirical findings and work to improve the forecasting accuracy of their methods.

— *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.

10.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Develop Methodology**. Develop a methodology for working through a univariate time series forecasting problem (e.g. what algorithms in what order) based on the results presented in this lesson and the referenced study.

- **Find Example**. Research and find an example of a research paper that presents the results of a deep learning model for time series foresting without comparing it to a naive forecast or a classical method.

- **Additional Limitations**. Research and find another research paper that lists the limitations of machine learning or deep learning methods for univariate time series forecasting problems.

If you explore any of these extensions, I'd love to know.

10.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Makridakis Competitions, Wikipedia.
  https://en.wikipedia.org/wiki/Makridakis_Competitions

- *The M3-Competition: Results, Conclusions and Implications*, 2000.
  https://www.sciencedirect.com/science/article/pii/S0169207000000571

- *The M4 Competition: Results, findings, conclusion and way forward*, 2018.
  https://www.sciencedirect.com/science/article/pii/S0169207018300785

- *Statistical and Machine Learning forecasting methods: Concerns and ways forward*, 2018.
  http://journals.plos.org/plosone/article?id=10.1371/journal.pone.0194889

- *An Empirical Comparison of Machine Learning Models for Time Series Forecasting*, 2010.
  https://www.tandfonline.com/doi/abs/10.1080/07474938.2010.481556

- *Applying LSTM to Time Series Predictable through Time-Window Approaches*, 2001.
  https://link.springer.com/chapter/10.1007/3-540-44668-0_93

10.11 Summary

In this tutorial, you discovered the important findings of a recent study evaluating and comparing the performance of classical and modern machine learning methods on a large and diverse set of time series forecasting datasets. Specifically, you learned:

- Classical methods like ETS and ARIMA out-perform machine learning and deep learning methods for one-step forecasting on univariate datasets.

- Classical methods like Theta and ARIMA out-perform machine learning and deep learning methods for multi-step forecasting on univariate datasets.

- Machine learning and deep learning methods do not yet deliver on their promise for univariate time series forecasting and there is much work to do.

10.11.1 Next

In the next lesson, you will discover how to develop robust naive forecasting models for univariate time series forecasting problems.

Chapter 11

How to Develop Simple Methods for Univariate Forecasting

Simple forecasting methods include naively using the last observation as the prediction or an average of prior observations. It is important to evaluate the performance of simple forecasting methods on univariate time series forecasting problems before using more sophisticated methods as their performance provides a lower-bound and point of comparison that can be used to determine of a model has skill or not for a given problem. Although simple, methods such as the naive and average forecast strategies can be tuned to a specific problem in terms of the choice of which prior observation to persist or how many prior observations to average. Often, tuning the hyperparameters of these simple strategies can provide a more robust and defensible lower bound on model performance, as well as surprising results that may inform the choice and configuration of more sophisticated methods.

In this tutorial, you will discover how to develop a framework from scratch for grid searching simple naive and averaging strategies for time series forecasting with univariate data. After completing this tutorial, you will know:

- How to develop a framework for grid searching simple models from scratch using walk-forward validation.

- How to grid search simple model hyperparameters for daily time series data for births.

- How to grid search simple model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

Let's get started.

11.1 Tutorial Overview

This tutorial is divided into six parts; they are:

1. Simple Forecasting Strategies

2. Develop a Grid Search Framework

3. Case Study 1: No Trend or Seasonality

11.2 Simple Forecasting Strategies

It is important and useful to test simple forecast strategies prior to testing more complex models. Simple forecast strategies are those that assume little or nothing about the nature of the forecast problem and are fast to implement and calculate. The results can be used as a baseline in performance and used as a point of a comparison. If a model can perform better than the performance of a simple forecast strategy, then it can be said to be skillful. There are two main themes to simple forecast strategies; they are:

- **Naive**, or using observations values directly.

- **Average**, or using a statistic calculated on previous observations.

For more information on simple forecasting strategies, see Chapter 5.

11.3 Develop a Grid Search Framework

In this section, we will develop a framework for grid searching the two simple forecast strategies described in the previous section, namely the naive and average strategies. We can start off by implementing a naive forecast strategy. For a given dataset of historical observations, we can persist any value in that history, that is from the previous observation at index -1 to the first observation in the history at `-(len(data))`. The `naive_forecast()` function below implements the naive forecast strategy for a given offset from 1 to the length of the dataset.

```python
# one-step naive forecast
def naive_forecast(history, n):
  return history[-n]
```

Listing 11.1: Example of a function for making a persistence forecast.

We can test this function out on a small contrived dataset.

```python
# example of a one-step naive forecast
def naive_forecast(history, n):
  return history[-n]

# define dataset
data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
print(data)
# test naive forecast
for i in range(1, len(data)+1):
  print(naive_forecast(data, i))
```

Listing 11.2: Example of making a persistence forecast.

Running the example first prints the contrived dataset, then the naive forecast for each offset in the historical dataset.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
100.0
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
```

Listing 11.3: Example output from making a persistence forecast.

We can now look at developing a function for the average forecast strategy. Averaging the last n observations is straight-forward; for example:

```python
# mean forecast
from numpy import mean
result = mean(history[-n:])
```

Listing 11.4: Example of averaging prior observations.

We may also want to test out the median in those cases where the distribution of observations is non-Gaussian.

```python
# median forecast
from numpy import median
result = median(history[-n:])
```

Listing 11.5: Example of calculating the median prior observations.

The `average_forecast()` function below implements this taking the historical data and a config array or tuple that specifies the number of prior values to average as an integer, and a string that describe the way to calculate the average (`mean` or `median`).

```python
# one-step average forecast
def average_forecast(history, config):
  n, avg_type = config
  # mean of last n values
  if avg_type is 'mean':
    return mean(history[-n:])
  # median of last n values
  return median(history[-n:])
```

Listing 11.6: Example of a function for calculating an average forecast.

The complete example on a small contrived dataset is listed below.

```python
# example of an average forecast
from numpy import mean
from numpy import median

# one-step average forecast
def average_forecast(history, config):
```

```
  n, avg_type = config
  # mean of last n values
  if avg_type is 'mean':
    return mean(history[-n:])
  # median of last n values
  return median(history[-n:])

# define dataset
data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
print(data)
# test naive forecast
for i in range(1, len(data)+1):
  print(average_forecast(data, (i, 'mean')))
```

Listing 11.7: Example of making an average forecast.

Running the example forecasts the next value in the series as the mean value from contiguous subsets of prior observations from -1 to -10, inclusively.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
100.0
95.0
90.0
85.0
80.0
75.0
70.0
65.0
60.0
55.0
```

Listing 11.8: Example output from making an average forecast.

We can update the function to support averaging over seasonal data, respecting the seasonal offset. An offset argument can be added to the function that when not set to 1 will determine the number of prior observations backwards to count before collecting values from which to include in the average. For example, if n=1 and offset=3, then the average is calculated from the single value at $n \times$ offset or $1 \times 3 = -3$. If $n = 2$ and offset $= 3$, then the average is calculated from the values at 1×3 or -3 and 2×3 or -6. We can also add some protection to raise an exception when a seasonal configuration ($n \times$ offset) extends beyond the end of the historical observations. The updated function is listed below.

```
# one-step average forecast
def average_forecast(history, config):
  n, offset, avg_type = config
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
```

```
  # mean of last n values
  if avg_type is 'mean':
    return mean(values)
  # median of last n values
  return median(values)
```

Listing 11.9: Example of a function for calculating an average forecast with support for seasonality.

We can test out this function on a small contrived dataset with a seasonal cycle. The complete example is listed below.

```
# example of an average forecast for seasonal data
from numpy import mean
from numpy import median

# one-step average forecast
def average_forecast(history, config):
  n, offset, avg_type = config
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # mean of last n values
  if avg_type is 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# define dataset
data = [10.0, 20.0, 30.0, 10.0, 20.0, 30.0, 10.0, 20.0, 30.0]
print(data)
# test naive forecast
for i in [1, 2, 3]:
  print(average_forecast(data, (i, 3, 'mean')))
```

Listing 11.10: Example of making an average forecast with seasonality.

Running the example calculates the mean values of [10], [10, 10] and [10, 10, 10].

```
[10.0, 20.0, 30.0, 10.0, 20.0, 30.0, 10.0, 20.0, 30.0]
10.0
10.0
10.0
```

Listing 11.11: Example output from making an average forecast with seasonality.

It is possible to combine both the naive and the average forecast strategies together into the same function. There is a little overlap between the methods, specifically the n-offset into the history that is used to either persist values or determine the number of values to average.

It is helpful to have both strategies supported by one function so that we can test a suite of configurations for both strategies at once as part of a broader grid search of simple models. The simple_forecast() function below combines both strategies into a single function.

```
# one-step simple forecast
def simple_forecast(history, config):
 n, offset, avg_type = config
 # persist value, ignore other config
 if avg_type == 'persist':
  return history[-n]
 # collect values to average
 values = list()
 if offset == 1:
  values = history[-n:]
 else:
  # skip bad configs
  if n*offset > len(history):
   raise Exception('Config beyond end of data: %d %d' % (n,offset))
  # try and collect n values using offset
  for i in range(1, n+1):
   ix = i * offset
   values.append(history[-ix])
 # check if we can average
 if len(values) < 2:
  raise Exception('Cannot calculate average')
 # mean of last n values
 if avg_type == 'mean':
  return mean(values)
 # median of last n values
 return median(values)
```

Listing 11.12: Example of a function that combines persistence and average forecasts.

Next, we need to build up some functions for fitting and evaluating a model repeatedly via walk-forward validation, including splitting a dataset into train and test sets and evaluating one-step forecasts. We can split a list or NumPy array of data using a slice given a specified size of the split, e.g. the number of time steps to use from the data in the test set. The train_test_split() function below implements this for a provided dataset and a specified number of time steps to use in the test set.

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

Listing 11.13: Example of a function for splitting data into train and test sets.

After forecasts have been made for each step in the test dataset, they need to be compared to the test set in order to calculate an error score. There are many popular error scores for time series forecasting. In this case, we will use root mean squared error (RMSE), but you can change this to your preferred measure, e.g. MAPE, MAE, etc. The measure_rmse() function below will calculate the RMSE given a list of actual (the test set) and predicted values.

```
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```

Listing 11.14: Example of a function for calculating RMSE.

We can now implement the walk-forward validation scheme. This is a standard approach to evaluating a time series forecasting model that respects the temporal ordering of observations. First, a provided univariate time series dataset is split into train and test sets using the `train_test_split()` function. Then the number of observations in the test set are enumerated. For each we fit a model on all of the history and make a one step forecast. The true observation for the time step is then added to the history, and the process is repeated. The `simple_forecast()` function is called in order to fit a model and make a prediction. Finally, an error score is calculated by comparing all one-step forecasts to the actual test set by calling the `measure_rmse()` function. The `walk_forward_validation()` function below implements this, taking a univariate time series, a number of time steps to use in the test set, and an array of model configuration.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error
```

Listing 11.15: Example of a function for performing walk-forward validation.

If you are interested in making multi-step predictions, you can change the call to `predict()` in the `simple_forecast()` function and also change the calculation of error in the `measure_rmse()` function. We can call `walk_forward_validation()` repeatedly with different lists of model configurations. One possible issue is that some combinations of model configurations may not be called for the model and will throw an exception.

We can trap exceptions and ignore warnings during the grid search by wrapping all calls to `walk_forward_validation()` with a try-except and a block to ignore warnings. We can also add debugging support to disable these protections in the case we want to see what is really going on. Finally, if an error does occur, we can return a `None` result; otherwise, we can print some information about the skill of each model evaluated. This is helpful when a large number of models are evaluated. The `score_model()` function below implements this and returns a tuple of (key and result), where the key is a string version of the tested model configuration.

```
# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
```

```
# show all warnings and fail on exception if debugging
if debug:
  result = walk_forward_validation(data, n_test, cfg)
else:
  # one failure during model validation suggests an unstable config
  try:
    # never show warnings when grid searching, too noisy
    with catch_warnings():
      filterwarnings("ignore")
      result = walk_forward_validation(data, n_test, cfg)
  except:
    error = None
# check for an interesting result
if result is not None:
  print(' > Model[%s] %.3f' % (key, result))
return (key, result)
```

Listing 11.16: Example of a function the robust evaluation of a model.

Next, we need a loop to test a list of different model configurations. This is the main function that drives the grid search process and will call the score_model() function for each model configuration. We can dramatically speed up the grid search process by evaluating model configurations in parallel. One way to do that is to use the Joblib library[1]. We can define a Parallel object with the number of cores to use and set it to the number of scores detected in your hardware.

```
# define executor
executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
```

Listing 11.17: Example of preparing a Joblib executor.

We can then create a list of tasks to execute in parallel, which will be one call to the score_model() function for each model configuration we have.

```
# define list of tasks
tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
```

Listing 11.18: Example of preparing a task list for a Joblib executor.

Finally, we can use the Parallel object to execute the list of tasks in parallel.

```
# execute list of tasks
scores = executor(tasks)
```

Listing 11.19: Example of running a Joblib executor.

On some systems, such as windows that do not support the fork() function, it is necessary to add a check to ensure that the entry point of the script is only executed by the main process and not child processes.

```
...
if __name__ == '__main__':
  ...
```

Listing 11.20: Example of wrapping the entry point into the script in a check for the main process.

---

[1]Note, you may have to install Joblib: pip install joblib

That's it. We can also provide a non-parallel version of evaluating all model configurations in case we want to debug something.

```
# execute list of tasks sequentially
scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
```
Listing 11.21: Example of evaluating a suite of configurations in sequence.

The result of evaluating a list of configurations will be a list of tuples, each with a name that summarizes a specific model configuration and the error of the model evaluated with that configuration as either the RMSE or `None` if there was an error. We can filter out all scores set to `None`.

```
# order scores
scores = [r for r in scores if r[1] != None]
```
Listing 11.22: Example of filtering out scores for invalid configurations.

We can then sort all tuples in the list by the score in ascending order (best are first), then return this list of scores for review. The `grid_search()` function below implements this behavior given a univariate time series dataset, a list of model configurations (list of lists), and the number of time steps to use in the test set. An optional parallel argument allows the evaluation of models across all cores to be tuned on or off, and is on by default.

```
# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores
```
Listing 11.23: Example of a function for grid searching configurations.

We're nearly done. The only thing left to do is to define a list of model configurations to try for a dataset. We can define this generically. The only parameter we may want to specify is the periodicity of the seasonal component in the series (offset), if one exists. By default, we will assume no seasonal component. The `simple_configs()` function below will create a list of model configurations to evaluate. The function only requires the maximum length of the historical data as an argument and optionally the periodicity of any seasonal component, which is defaulted to 1 (no seasonal component).

```
# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
```

```
        cfg = [i, o, t]
        configs.append(cfg)
  return configs
```

Listing 11.24: Example of a function for defining simple forecast configurations to grid search.

We now have a framework for grid searching simple model hyperparameters via one-step walk-forward validation. It is generic and will work for any in-memory univariate time series provided as a list or NumPy array. We can make sure all the pieces work together by testing it on a contrived 10-step dataset. The complete example is listed below.

```python
# grid search simple forecasts
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error

# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

```python
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
```

```python
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # define dataset
  data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
  # data split
  n_test = 4
  # model configs
  max_length = len(data) - n_test
  cfg_list = simple_configs(max_length)
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 11.25: Example of demonstrating the grid search infrastructure.

Running the example first prints the contrived time series dataset. Next, the model configurations and their errors are reported as they are evaluated. Finally, the configurations and the error for the top three configurations are reported. We can see that the persistence model with a configuration of 1 (e.g. persist the last observation) achieves the best performance of the simple models tested, as would be expected.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]

...
> Model[[4, 1, 'mean']] 25.000
> Model[[3, 1, 'median']] 20.000
> Model[[6, 1, 'mean']] 35.000
> Model[[5, 1, 'median']] 30.000
> Model[[6, 1, 'median']] 35.000
done

[1, 1, 'persist'] 10.0
[2, 1, 'mean'] 15.0
[2, 1, 'median'] 15.0
```

Listing 11.26: Example output from demonstrating the grid search infrastructure.

Now that we have a robust framework for grid searching simple model hyperparameters, let's test it out on a suite of standard univariate time series datasets. All datasets in this tutorial were drawn from the Time Series Dataset Library on the DataMarket website[2]. The results

---

[2]https://datamarket.com/data/list/?q=provider:tsdl

demonstrated on each dataset provide a baseline of performance that can be used to compare more sophisticated methods, such as SARIMA, ETS, and even machine learning methods.

11.4   Case Study 1: No Trend or Seasonality

The *daily female births* dataset summarizes the daily total female births in California, USA in 1959. You can download the dataset directly from here:

- daily-total-female-births.csv [3]

Save the file with the filename daily-total-female-births.csv in your current working directory. We can load this dataset as a Pandas Series using the function read_csv() and summarize the shape of the dataset.

```
# load
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.27: Example of loading the daily female births dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.28: Example of plotting the daily female births dataset.

The complete example is listed below.

```
# load and plot daily births dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.29: Example of loading and plotting the daily female births dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has one year, or 365 observations. We will use the first 200 for training and the remaining 165 as the test set.

---

[3] https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-total-female-births.csv

```
(365, 1)
```

Listing 11.30: Example output from loading and summarizing the shape of the daily female births dataset

A line plot of the series is also created. We can see that there is no obvious trend or seasonality.
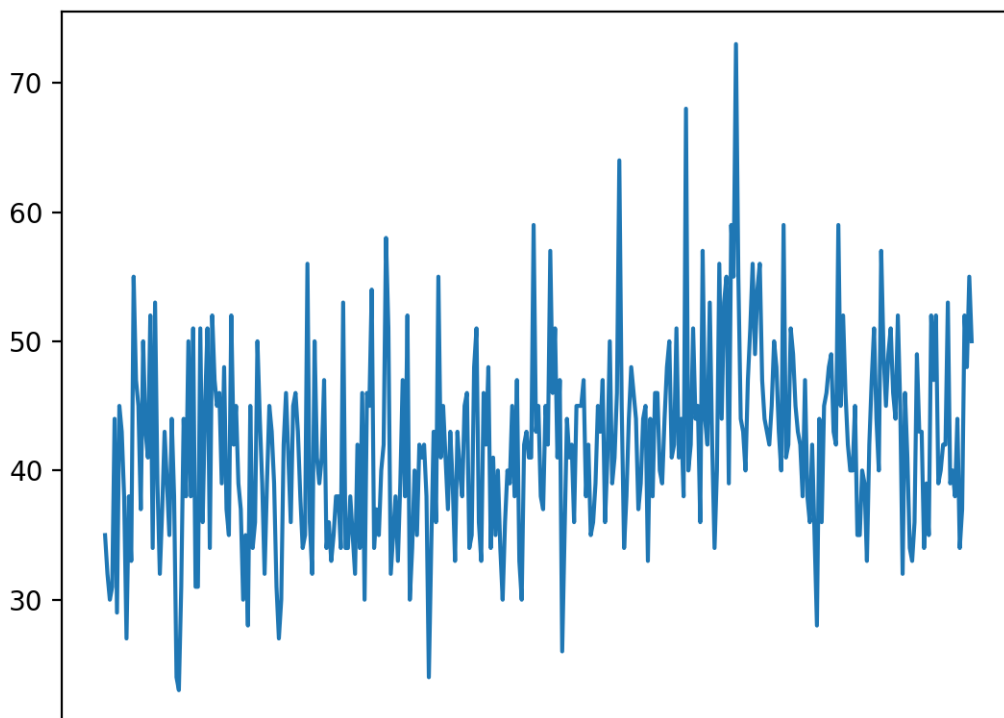
Figure 11.1: Line Plot of the Daily Female Births Dataset.

We can now grid search naive models for the dataset. The complete example grid searching the daily female univariate time series forecasting problem is listed below.

```python
# grid search simple forecast for daily female births
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv
```

```python
# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
```

```python
    result = None
    # convert config to a key
    key = str(cfg)
    # show all warnings and fail on exception if debugging
    if debug:
        result = walk_forward_validation(data, n_test, cfg)
    else:
        # one failure during model validation suggests an unstable config
        try:
            # never show warnings when grid searching, too noisy
            with catch_warnings():
                filterwarnings("ignore")
                result = walk_forward_validation(data, n_test, cfg)
        except:
            error = None
    # check for an interesting result
    if result is not None:
        print(' > Model[%s] %.3f' % (key, result))
    return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
    scores = None
    if parallel:
        # execute configs in parallel
        executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
        tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
        scores = executor(tasks)
    else:
        scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
    # remove empty results
    scores = [r for r in scores if r[1] != None]
    # sort configs by error, asc
    scores.sort(key=lambda tup: tup[1])
    return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
    configs = list()
    for i in range(1, max_length+1):
        for o in offsets:
            for t in ['persist', 'mean', 'median']:
                cfg = [i, o, t]
                configs.append(cfg)
    return configs

if __name__ == '__main__':
    # define dataset
    series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
    data = series.values
    # data split
    n_test = 165
    # model configs
    max_length = len(data) - n_test
    cfg_list = simple_configs(max_length)
    # grid search
```

```
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 11.31: Example of grid searching naive models for the daily female births dataset.

Running the example prints the model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[186, 1, 'mean']] 7.523
> Model[[200, 1, 'median']] 7.681
> Model[[186, 1, 'median']] 7.691
> Model[[187, 1, 'persist']] 11.137
> Model[[187, 1, 'mean']] 7.527
done

[22, 1, 'mean'] 6.930411499775709
[23, 1, 'mean'] 6.932293117115201
[21, 1, 'mean'] 6.951918385845375
```

Listing 11.32: Example output from grid searching naive models for the daily female births dataset.

We can see that the best result was an RMSE of about 6.93 births with the following configuration:

- **Strategy**: Average

- **n**: 22

- **function**: mean()

This is surprising given the lack of trend or seasonality, I would have expected either a persistence of -1 or an average of the entire historical dataset to result in the best performance.

11.5 Case Study 2: Trend

The *monthly shampoo sales* dataset summarizes the monthly sales of shampoo over a three-year period. You can download the dataset directly from here:

- `monthly-shampoo-sales.csv` [4]

Save the file with the filename `monthly-shampoo-sales.csv` in your current working directory. We can load this dataset as a Pandas `Series` using the function `read_csv()` and summarize the shape of the dataset.

---

[4]https://raw.githubusercontent.com/jbrownlee/Datasets/master/shampoo.csv

```
# load
series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.33: Example of loading the monthly shampoo sales dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.34: Example of plotting the monthly shampoo sales dataset.

The complete example is listed below.

```
# load and plot monthly shampoo sales dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.35: Example of loading and plotting the monthly shampoo sales dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has three years, or 36 observations. We will use the first 24 for training and the remaining 12 as the test set.

```
(36, 1)
```

Listing 11.36: Example output from loading and summarizing the shape of the monthly shampoo sales dataset

A line plot of the series is also created. We can see that there is an obvious trend and no obvious seasonality.
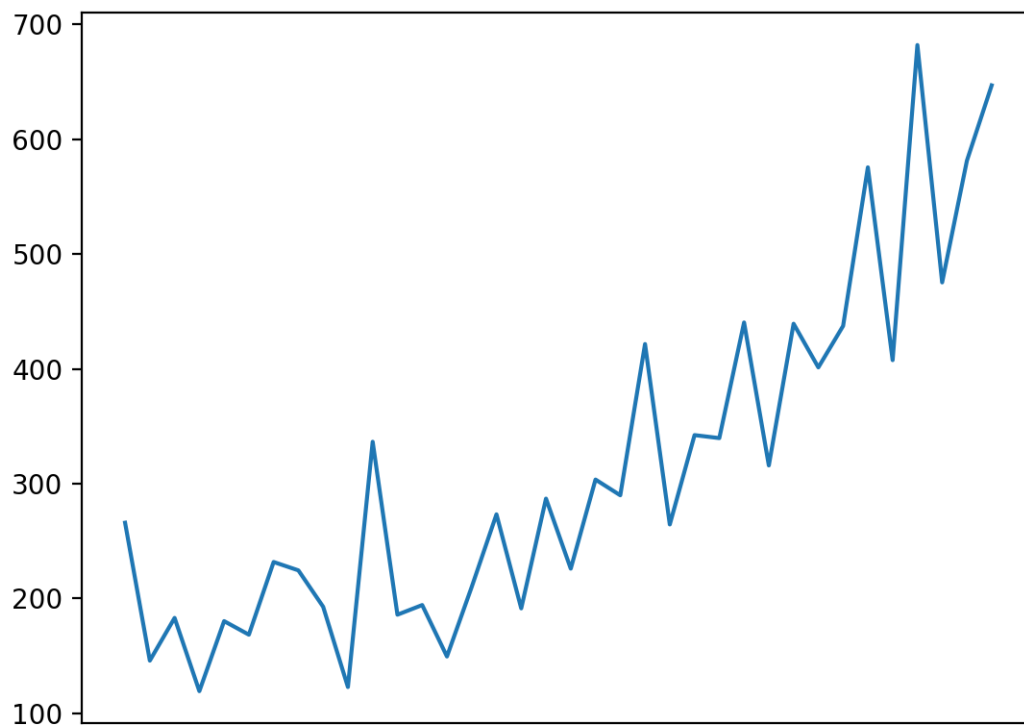
Figure 11.2: Line Plot of the Monthly Shampoo Sales Dataset.

We can now grid search naive models for the dataset. The complete example grid searching the shampoo sales univariate time series forecasting problem is listed below.

```python
# grid search simple forecast for monthly shampoo sales
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
```

```python
      values = history[-n:]
    else:
      # skip bad configs
      if n*offset > len(history):
        raise Exception('Config beyond end of data: %d %d' % (n,offset))
      # try and collect n values using offset
      for i in range(1, n+1):
        ix = i * offset
        values.append(history[-ix])
    # check if we can average
    if len(values) < 2:
      raise Exception('Cannot calculate average')
    # mean of last n values
    if avg_type == 'mean':
      return mean(values)
    # median of last n values
    return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
```

```python
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  max_length = len(data) - n_test
  cfg_list = simple_configs(max_length)
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 11.37: Example of grid searching naive models for the monthly shampoo sales dataset.

Running the example prints the configurations and the RMSE are printed as the models are

evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[23, 1, 'mean']] 209.782
> Model[[23, 1, 'median']] 221.863
> Model[[24, 1, 'persist']] 305.635
> Model[[24, 1, 'mean']] 213.466
> Model[[24, 1, 'median']] 226.061
done

[2, 1, 'persist'] 95.69454007413378
[2, 1, 'mean'] 96.01140340258198
[2, 1, 'median'] 96.01140340258198
```

Listing 11.38: Example output from grid searching naive models for the monthly shampoo sales dataset.

We can see that the best result was an RMSE of about 95.69 sales with the following configuration:

- **Strategy**: Persist

- **n**: 2

This is surprising as the trend structure of the data would suggest that persisting the previous value (-1) would be the best approach, not persisting the second last value.

11.6   Case Study 3: Seasonality

The *monthly mean temperatures* dataset summarizes the monthly average air temperatures in Nottingham Castle, England from 1920 to 1939 in degrees Fahrenheit. You can download the dataset directly from here:

- monthly-mean-temp.csv [5]

Save the file with the filename `monthly-mean-temp.csv` in your current working directory. We can load this dataset as a Pandas `Series` using the function `read_csv()` and summarize the shape of the dataset.

```
# load
series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.39: Example of loading the monthly mean temperature dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

---

[5]https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-mean-temp.csv

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.40: Example of plotting the monthly mean temperature dataset.

The complete example is listed below.

```
# load and plot monthly mean temp dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.41: Example of loading and plotting the monthly mean temperature dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has 20 years, or 240 observations.

```
(240, 1)
```

Listing 11.42: Example output from loading and summarizing the shape of the monthly mean temperature dataset

A line plot of the series is also created. We can see that there is no obvious trend and an obvious seasonality structure.
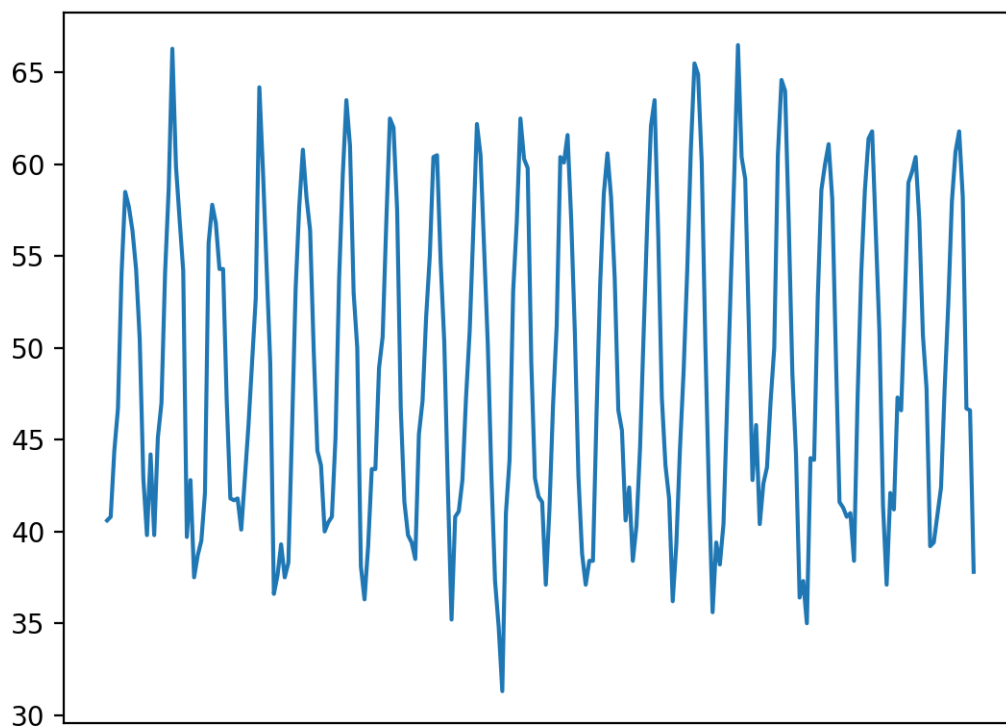
Figure 11.3: Line Plot of the Monthly Mean Temperatures Dataset.

We will trim the dataset to the last five years of data (60 observations) in order to speed up the model evaluation process and use the last year or 12 observations for the test set.

```
# trim dataset to 5 years
data = data[-(5*12):]
```
Listing 11.43: Example of reducing the size of the dataset.

The period of the seasonal component is about one year, or 12 observations. We will use this as the seasonal period in the call to the `simple_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = simple_configs(seasonal=[0, 12])
```
Listing 11.44: Example of specifying some seasonal configurations.

We can now grid search naive models for the dataset. The complete example grid searching the monthly mean temperature time series forecasting problem is listed below.

```
# grid search simple forecast for monthly mean temperature
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
```

```python
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
```

```python
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # define dataset
  series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
```

```
data = series.values
# data split
n_test = 12
# model configs
max_length = len(data) - n_test
cfg_list = simple_configs(max_length, offsets=[1,12])
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 11.45: Example of grid searching naive models for the monthly mean temperature dataset.

Running the example prints the model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[227, 12, 'persist']] 5.365
> Model[[228, 1, 'persist']] 2.818
> Model[[228, 1, 'mean']] 8.258
> Model[[228, 1, 'median']] 8.361
> Model[[228, 12, 'persist']] 2.818
done

[4, 12, 'mean'] 1.5015616870445234
[8, 12, 'mean'] 1.5794579766489512
[13, 12, 'mean'] 1.586186052546763
```

Listing 11.46: Example output from grid searching naive models for the monthly mean temperature dataset.

We can see that the best result was an RMSE of about 1.50 degrees with the following configuration:

- **Strategy**: Average

- **n**: 4

- **offset**: 12

- **function**: mean()

This finding is not too surprising. Given the seasonal structure of the data, we would expect a function of the last few observations at prior points in the yearly cycle to be effective.

11.7 Case Study 4: Trend and Seasonality

The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. You can download the dataset directly from here:

- monthly-car-sales.csv [6]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. We can load this dataset as a Pandas `Series` using the function `read_csv()` and summarize the shape of the dataset.

```
# load
series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
```

Listing 11.47: Example of loading the monthly car sales dataset.

We can then create a line plot of the series and inspect it for systematic structures like trends and seasonality.

```
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.48: Example of plotting the monthly car sales dataset.

The complete example is listed below.

```
# load and plot monthly car sales dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 11.49: Example of loading and plotting the monthly car sales dataset.

Running the example first summarizes the shape of the loaded dataset. The dataset has 9 years, or 108 observations. We will use the last year or 12 observations as the test set.

```
(108, 1)
```

Listing 11.50: Example output from loading and summarizing the shape of the monthly shampoo sales dataset

A line plot of the series is also created. We can see that there is an obvious trend and seasonal components.

---

[6]https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-car-sales.csv
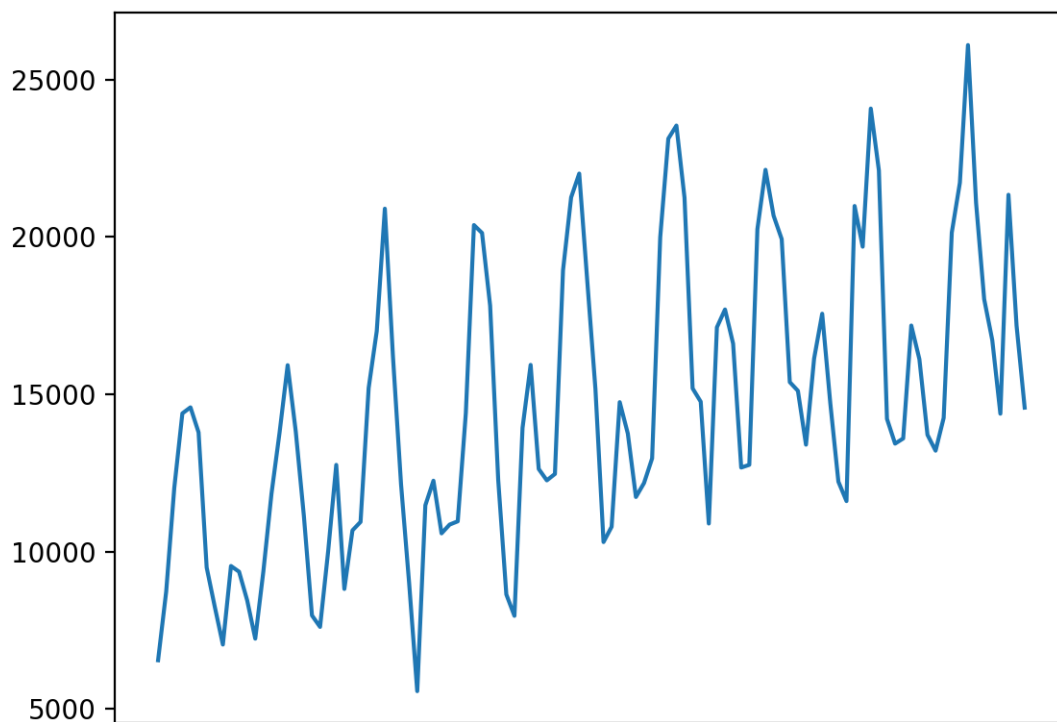
Figure 11.4: Line Plot of the Monthly Car Sales Dataset.

The period of the seasonal component could be six months or 12 months. We will try both as the seasonal period in the call to the `simple_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = simple_configs(seasonal=[0,6,12])
```

Listing 11.51: Example of specifying some seasonal configurations.

We can now grid search naive models for the dataset. The complete example grid searching the monthly car sales time series forecasting problem is listed below.

```
# grid search simple forecast for monthly car sales
from math import sqrt
from numpy import mean
from numpy import median
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from sklearn.metrics import mean_squared_error
from pandas import read_csv
```

```python
# one-step simple forecast
def simple_forecast(history, config):
  n, offset, avg_type = config
  # persist value, ignore other config
  if avg_type == 'persist':
    return history[-n]
  # collect values to average
  values = list()
  if offset == 1:
    values = history[-n:]
  else:
    # skip bad configs
    if n*offset > len(history):
      raise Exception('Config beyond end of data: %d %d' % (n,offset))
    # try and collect n values using offset
    for i in range(1, n+1):
      ix = i * offset
      values.append(history[-ix])
  # check if we can average
  if len(values) < 2:
    raise Exception('Cannot calculate average')
  # mean of last n values
  if avg_type == 'mean':
    return mean(values)
  # median of last n values
  return median(values)

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = simple_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
```

```python
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of simple configs to try
def simple_configs(max_length, offsets=[1]):
  configs = list()
  for i in range(1, max_length+1):
    for o in offsets:
      for t in ['persist', 'mean', 'median']:
        cfg = [i, o, t]
        configs.append(cfg)
  return configs

if __name__ == '__main__':
  # define dataset
  series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  max_length = len(data) - n_test
  cfg_list = simple_configs(max_length, offsets=[1,12])
  # grid search
```

```
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 11.52: Example of grid searching naive models for the monthly car sales dataset.

Running the example prints the model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

```
...
> Model[[79, 1, 'median']] 5124.113
> Model[[91, 12, 'persist']] 9580.149
> Model[[79, 12, 'persist']] 8641.529
> Model[[92, 1, 'persist']] 9830.921
> Model[[92, 1, 'mean']] 5148.126
done

[3, 12, 'median'] 1841.1559321976688
[3, 12, 'mean'] 2115.198495632485
[4, 12, 'median'] 2184.37708988932
```

Listing 11.53: Example output from grid searching naive models for the monthly car sales dataset.

We can see that the best result was an RMSE of about 1841.15 sales with the following configuration:

- `Strategy`: Average

- `n`: 3

- `offset`: 12

- `function`: median()

It is not surprising that the chosen model is a function of the last few observations at the same point in prior cycles, although the use of the median instead of the mean may not have been immediately obvious and the results were much better than the mean.

11.8 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Plot Forecast**. Update the framework to re-fit a model with the best configuration and forecast the entire test dataset, then plot the forecast compared to the actual observations in the test set.

- **Drift Method**. Implement the drift method for simple forecasts and compare the results to the average and naive methods.

- **Another Dataset**. Apply the developed framework to an additional univariate time series problem (e.g. from the Time Series Dataset Library).

If you explore any of these extensions, I'd love to know.

11.9 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

11.9.1 Datasets

- Time Series Dataset Library, DataMarket.
  https://datamarket.com/data/list/?q=provider:tsdl

- Daily Female Births Dataset, DataMarket.
  https://datamarket.com/data/set/235k/daily-total-female-births-in-california-1959

- Monthly Shampoo Sales Dataset, DataMarket.
  https://datamarket.com/data/set/22r0/sales-of-shampoo-over-a-three-year-period

- Monthly Mean Temperature Dataset, DataMarket.
  https://datamarket.com/data/set/22li/mean-monthly-air-temperature-deg-f-nottingham-

- Monthly Car Sales Dataset, DataMarket.
  https://datamarket.com/data/set/22n4/monthly-car-sales-in-quebec-1960-1968

11.9.2 APIs

- `numpy.mean` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.mean.html

- `numpy.median` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.median.html

- `sklearn.metrics.mean_squared_error` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

- `pandas.read_csv` API.
  https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

- Joblib: running Python functions as pipeline jobs.
  https://pythonhosted.org/joblib/

11.9.3 Articles

- Forecasting, Wikipedia.
  https://en.wikipedia.org/wiki/Forecasting

11.10 Summary

In this tutorial, you discovered how to develop a framework from scratch for grid searching simple naive and averaging strategies for time series forecasting with univariate data. Specifically, you learned:

- How to develop a framework for grid searching simple models from scratch using walk-forward validation.

- How to grid search simple model hyperparameters for daily time series data for births.

- How to grid search simple model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

11.10.1 Next

In the next lesson, you will discover how to develop exponential smoothing models for univariate time series forecasting problems.

Chapter 12

How to Develop ETS Models for Univariate Forecasting

Exponential smoothing is a time series forecasting method for univariate data that can be extended to support data with a systematic trend or seasonal component. It is common practice to use an optimization process to find the model hyperparameters that result in the exponential smoothing model with the best performance for a given time series dataset. This practice applies only to the coefficients used by the model to describe the exponential structure of the level, trend, and seasonality. It is also possible to automatically optimize other hyperparameters of an exponential smoothing model, such as whether or not to model the trend and seasonal component and if so, whether to model them using an additive or multiplicative method.

In this tutorial, you will discover how to develop a framework for grid searching all of the exponential smoothing model hyperparameters for univariate time series forecasting. After completing this tutorial, you will know:

- How to develop a framework for grid searching ETS models from scratch using walk-forward validation.

- How to grid search ETS model hyperparameters for daily time series data for female births.

- How to grid search ETS model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

Let's get started.

12.1   Tutorial Overview

This tutorial is divided into five parts; they are:

1. Develop a Grid Search Framework

2. Case Study 1: No Trend or Seasonality

3. Case Study 2: Trend

4. Case Study 3: Seasonality

5. Case Study 4: Trend and Seasonality

12.2 Develop a Grid Search Framework

In this section, we will develop a framework for grid searching exponential smoothing model hyperparameters for a given univariate time series forecasting problem. For more information on exponential smoothing for time series forecasting, also called ETS, see Chapter 5. We will use the implementation of Holt-Winters Exponential Smoothing provided by the Statsmodels library. This model has hyperparameters that control the nature of the exponential performed for the series, trend, and seasonality, specifically:

- `smoothing_level` (alpha): the smoothing coefficient for the level.

- `smoothing_slope` (beta): the smoothing coefficient for the trend.

- `smoothing_seasonal` (gamma): the smoothing coefficient for the seasonal component.

- `damping_slope` (phi): the coefficient for the damped trend.

All four of these hyperparameters can be specified when defining the model. If they are not specified, the library will automatically tune the model and find the optimal values for these hyperparameters (e.g. `optimized=True`). There are other hyperparameters that the model will not automatically tune that you may want to specify; they are:

- `trend`: The type of trend component, as either `add` for additive or `mul` for multiplicative. Modeling the trend can be disabled by setting it to `None`.

- `damped`: Whether or not the trend component should be damped, either True or False.

- `seasonal`: The type of seasonal component, as either `add` for additive or `mul` for multiplicative. Modeling the seasonal component can be disabled by setting it to `None`.

- `seasonal_periods`: The number of time steps in a seasonal period, e.g. 12 for 12 months in a yearly seasonal structure.

- `use_boxcox`: Whether or not to perform a power transform of the series (True/False) or specify the lambda for the transform.

If you know enough about your problem to specify one or more of these parameters, then you should specify them. If not, you can try grid searching these parameters. We can start-off by defining a function that will fit a model with a given configuration and make a one-step forecast. The `exp_smoothing_forecast()` below implements this behavior. The function takes an array or list of contiguous prior observations and a list of configuration parameters used to configure the model. The configuration parameters in order are: the trend type, the dampening type, the seasonality type, the seasonal period, whether or not to use a Box-Cox transform, and whether or not to remove the bias when fitting the model.

```
# one-step Holt Winter's Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
 t,d,s,p,b,r = config
 # define model
 history = array(history)
 model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
```

```
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]
```

Listing 12.1: Example of a function for making an ETS forecast.

In this tutorial, we will use the grid searching framework developed in Chapter 11 for tuning and evaluating naive forecasting methods. One important modification to the framework is the function used to perform the walk-forward validation of the model named `walk_forward_validation()`. This function must be updated to call the function for making an ETS forecast. The updated version of the function is listed below.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error
```

Listing 12.2: Example of a function for walk-forward validation with ETS forecasts.

We're nearly done. The only thing left to do is to define a list of model configurations to try for a dataset. We can define this generically. The only parameter we may want to specify is the periodicity of the seasonal component in the series, if one exists. By default, we will assume no seasonal component. The `exp_smoothing_configs()` function below will create a list of model configurations to evaluate. An optional list of seasonal periods can be specified, and you could even change the function to specify other elements that you may know about your time series. In theory, there are 72 possible model configurations to evaluate, but in practice, many will not be valid and will result in an error that we will trap and ignore.

```
# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
```

```
    for t in t_params:
      for d in d_params:
        for s in s_params:
          for p in p_params:
            for b in b_params:
              for r in r_params:
                cfg = [t,d,s,p,b,r]
                models.append(cfg)
  return models
```

Listing 12.3: Example of a function for defining configurations for ETS models to grid search.

We now have a framework for grid searching triple exponential smoothing model hyperparameters via one-step walk-forward validation. It is generic and will work for any in-memory univariate time series provided as a list or NumPy array. We can make sure all the pieces work together by testing it on a contrived 10-step dataset. The complete example is listed below.

```
# grid search holt winter's exponential smoothing
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
```

```python
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
```

```python
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # define dataset
  data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
  print(data)
  # data split
  n_test = 4
  # model configs
  cfg_list = exp_smoothing_configs()
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.4: Example of demonstrating the grid search infrastructure.

Running the example first prints the contrived time series dataset. Next, the model configurations and their errors are reported as they are evaluated. Finally, the configurations and the error for the top three configurations are reported.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]

 > Model[[None, False, None, None, True, True]] 1.380
 > Model[[None, False, None, None, True, False]] 10.000
 > Model[[None, False, None, None, False, True]] 2.563
 > Model[[None, False, None, None, False, False]] 10.000
done

[None, False, None, None, True, True] 1.379824445857423
[None, False, None, None, False, True] 2.5628662672606612
[None, False, None, None, False, False] 10.0
```

Listing 12.5: Example output from demonstrating the grid search infrastructure.

We do not report the model parameters optimized by the model itself. It is assumed that you can achieve the same result again by specifying the broader hyperparameters and allow the library to find the same internal parameters. You can access these internal parameters by refitting a standalone model with the same configuration and printing the contents of the `params` attribute on the model fit; for example:

```
# access model parameters
```

```
print(model_fit.params)
```
Listing 12.6: Example of accessing the automatically fit model parameters.

Now that we have a robust framework for grid searching ETS model hyperparameters, let's test it out on a suite of standard univariate time series datasets. The datasets were chosen for demonstration purposes; I am not suggesting that an ETS model is the best approach for each dataset, and perhaps an SARIMA or something else would be more appropriate in some cases.

12.3 Case Study 1: No Trend or Seasonality

The *daily female births* dataset summarizes the daily total female births in California, USA in 1959. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- daily-total-female-births.csv [1]

Save the file with the filename daily-total-female-births.csv in your current working directory. The dataset has one year, or 365 observations. We will use the first 200 for training and the remaining 165 as the test set. The complete example grid searching the daily female univariate time series forecasting problem is listed below.

```python
# grid search ets models for daily female births
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```

---

[1]https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-total-female-births.csv

```python
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
```

```
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 165
  # model configs
  cfg_list = exp_smoothing_configs()
  # grid search
  scores = grid_search(data[:,0], cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.7: Example of grid searching ETS models for the daily female births dataset.

Running the example may take a few minutes as fitting each ETS model can take about a minute on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > Model[['mul', False, None, None, True, False]] 6.985
 > Model[[None, False, None, None, True, True]] 7.169
```

```
 > Model[[None, False, None, None, True, False]] 7.212
 > Model[[None, False, None, None, False, True]] 7.117
 > Model[[None, False, None, None, False, False]] 7.126
done

['mul', False, None, None, True, True] 6.960703917145126
['mul', False, None, None, True, False] 6.984513598720297
['add', False, None, None, True, True] 7.081359856193836
```

Listing 12.8: Example output from grid searching ETS models for the daily female births dataset.

We can see that the best result was an RMSE of about 6.96 births. A naive model achieved an RMSE of 6.93 births, meaning that the best performing ETS model is not skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend**: Multiplicative

- **Damped**: False

- **Seasonal**: None

- **Seasonal Periods**: None

- **Box-Cox Transform**: True

- **Remove Bias**: True

What is surprising is that a model that assumed an multiplicative trend performed better than one that didn't. We would not know that this is the case unless we threw out assumptions and grid searched models.

12.4 Case Study 2: Trend

The *monthly shampoo sales* dataset summarizes the monthly sales of shampoo over a three-year period. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-shampoo-sales.csv [2]

Save the file with the filename `monthly-shampoo-sales.csv` in your current working directory. The dataset has three years, or 36 observations. We will use the first 24 for training and the remaining 12 as the test set. The complete example grid searching the shampoo sales univariate time series forecasting problem is listed below.

```
# grid search ets models for monthly shampoo sales
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
```

---

[2]https://raw.githubusercontent.com/jbrownlee/Datasets/master/shampoo.csv

```python
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
```

```python
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
```

```
cfg_list = exp_smoothing_configs()
# grid search
scores = grid_search(data[:,0], cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 12.9: Example of grid searching ETS models for the monthly shampoo sales dataset.

Running the example is fast given there are a small number of observations. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > Model[['mul', True, None, None, False, False]] 102.152
 > Model[['mul', False, None, None, False, True]] 86.406
 > Model[['mul', False, None, None, False, False]] 83.747
 > Model[[None, False, None, None, False, True]] 99.416
 > Model[[None, False, None, None, False, False]] 108.031
done

['mul', False, None, None, False, False] 83.74666940175238
['mul', False, None, None, False, True] 86.40648953786152
['mul', True, None, None, False, True] 95.33737598817238
```

Listing 12.10: Example output from grid searching ETS models for the monthly shampoo sales dataset.

We can see that the best result was an RMSE of about 83.74 sales. A naive model achieved an RMSE of 95.69 sales on this dataset, meaning that the best performing ETS model is skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend**: Multiplicative

- **Damped**: False

- **Seasonal**: None

- **Seasonal Periods**: None

- **Box-Cox Transform**: False

- **Remove Bias**: False

12.5   Case Study 3: Seasonality

The *monthly mean temperatures* dataset summarizes the monthly average air temperatures in Nottingham Castle, England from 1920 to 1939 in degrees Fahrenheit. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-mean-temp.csv [3]

Save the file with the filename `monthly-mean-temp.csv` in your current working directory. The dataset has 20 years, or 240 observations. We will trim the dataset to the last five years of data (60 observations) in order to speed up the model evaluation process and use the last year or 12 observations for the test set.

```
# trim dataset to 5 years
data = data[-(5*12):]
```

Listing 12.11: Example of reducing the size of the dataset.

The period of the seasonal component is about one year, or 12 observations. We will use this as the seasonal period in the call to the `exp_smoothing_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = exp_smoothing_configs(seasonal=[0, 12])
```

Listing 12.12: Example of specifying some seasonal configurations.

The complete example grid searching the monthly mean temperature time series forecasting problem is listed below.

```
# grid search ets hyperparameters for monthly mean temp dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
```

---

[3]https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-mean-temp.csv

```python
    return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
```

```
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
          for b in b_params:
            for r in r_params:
              cfg = [t,d,s,p,b,r]
              models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
  data = series.values
  # trim dataset to 5 years
  data = data[-(5*12):]
  # data split
  n_test = 12
  # model configs
  cfg_list = exp_smoothing_configs(seasonal=[0,12])
  # grid search
  scores = grid_search(data[:,0], cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.13: Example of grid searching ETS models for the monthly mean temperature dataset.

Running the example is relatively slow given the large amount of data. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[['mul', True, None, 12, False, False]] 4.593
 > Model[['mul', False, 'add', 12, True, True]] 4.230
 > Model[['mul', False, 'add', 12, True, False]] 4.157
 > Model[['mul', False, 'add', 12, False, True]] 1.538
 > Model[['mul', False, 'add', 12, False, False]] 1.520
done

[None, False, 'add', 12, False, False] 1.5015527325330889
[None, False, 'add', 12, False, True] 1.5015531225114707
[None, False, 'mul', 12, False, False] 1.501561363221282
```

Listing 12.14: Example output from grid searching ETS models for the monthly mean temperature dataset.

We can see that the best result was an RMSE of about 1.50 degrees. This is the same RMSE found by a naive model on this problem, suggesting that the best ETS model sits on the border of being unskillful. We can unpack the configuration of the best performing model as follows:

- **Trend**: None

- **Damped**: False

- **Seasonal**: Additive

- **Seasonal Periods**: 12

- **Box-Cox Transform**: False

- **Remove Bias**: False

12.6   Case Study 4: Trend and Seasonality

The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-car-sales.csv [4]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. The dataset has 9 years, or 108 observations. We will use the last year or 12 observations as the test set. The period of the seasonal component could be six months or 12 months. We will try both as the seasonal period in the call to the `exp_smoothing_configs()` function when preparing the model configurations.

---

[4]https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-car-sales.csv

```
# model configs
cfg_list = exp_smoothing_configs(seasonal=[0,6,12])
```

Listing 12.15: Example of specifying some seasonal configurations.

The complete example grid searching the monthly car sales time series forecasting problem is listed below.

```
# grid search ets models for monthly car sales
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.holtwinters import ExponentialSmoothing
from sklearn.metrics import mean_squared_error
from pandas import read_csv
from numpy import array

# one-step Holt Winters Exponential Smoothing forecast
def exp_smoothing_forecast(history, config):
  t,d,s,p,b,r = config
  # define model
  history = array(history)
  model = ExponentialSmoothing(history, trend=t, damped=d, seasonal=s, seasonal_periods=p)
  # fit model
  model_fit = model.fit(optimized=True, use_boxcox=b, remove_bias=r)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = exp_smoothing_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
```

```python
    error = measure_rmse(test, predictions)
    return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of exponential smoothing configs to try
def exp_smoothing_configs(seasonal=[None]):
  models = list()
  # define config lists
  t_params = ['add', 'mul', None]
  d_params = [True, False]
  s_params = ['add', 'mul', None]
  p_params = seasonal
  b_params = [True, False]
  r_params = [True, False]
  # create config instances
  for t in t_params:
    for d in d_params:
      for s in s_params:
        for p in p_params:
```

```
        for b in b_params:
          for r in r_params:
            cfg = [t,d,s,p,b,r]
            models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 12
  # model configs
  cfg_list = exp_smoothing_configs(seasonal=[0,6,12])
  # grid search
  scores = grid_search(data[:,0], cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 12.16: Example of grid searching ETS models for the monthly car sales dataset.

Running the example is slow given the large amount of data. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[['mul', True, 'add', 6, False, False]] 3745.142
 > Model[['mul', True, 'add', 12, True, True]] 2203.354
 > Model[['mul', True, 'add', 12, True, False]] 2284.172
 > Model[['mul', True, 'add', 12, False, True]] 2842.605
 > Model[['mul', True, 'add', 12, False, False]] 2086.899
done

['add', False, 'add', 12, False, True] 1672.5539372356582
['add', False, 'add', 12, False, False] 1680.845043013083
['add', True, 'add', 12, False, False] 1696.1734099400082
```

Listing 12.17: Example output from grid searching ETS models for the monthly car sales dataset.

We can see that the best result was an RMSE of about 1,672 sales. A naive model achieved an RMSE of 1841.15 sales on this problem, suggesting that the best performing ETS model is skillful. We can unpack the configuration of the best performing model as follows:

- **Trend**: Additive

- **Damped**: False

- **Seasonal**: Additive

- **Seasonal Periods**: 12

- **Box-Cox Transform**: False

- **Remove Bias**: True

This is a little surprising as I would have guessed that a six-month seasonal model would be the preferred approach.

12.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Transforms**. Update the framework to support configurable data transforms such as normalization and standardization.

- **Plot Forecast**. Update the framework to re-fit a model with the best configuration and forecast the entire test dataset, then plot the forecast compared to the actual observations in the test set.

- **Tune Amount of History**. Update the framework to tune the amount of historical data used to fit the model (e.g. in the case of the 10 years of max temperature data).

If you explore any of these extensions, I'd love to know.

12.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

12.8.1 Books

- Chapter 7 Exponential smoothing, *Forecasting: principles and practice*, 2013.
  https://amzn.to/2xlJsfV

- Section 6.4. Introduction to Time Series Analysis, *Engineering Statistics Handbook*, 2012.
  https://www.itl.nist.gov/div898/handbook/

- *Practical Time Series Forecasting with R*, 2016.
  https://amzn.to/2LGKzKm

12.8.2 APIs

- statsmodels.tsa.holtwinters.ExponentialSmoothing API.

- statsmodels.tsa.holtwinters.HoltWintersResults API.

12.8.3   Articles

- Exponential smoothing, Wikipedia.
  https://en.wikipedia.org/wiki/Exponential_smoothing

12.9   Summary

In this tutorial, you discovered how to develop a framework for grid searching all of the exponential smoothing model hyperparameters for univariate time series forecasting. Specifically, you learned:

- How to develop a framework for grid searching ETS models from scratch using walk-forward validation.

- How to grid search ETS model hyperparameters for daily time series data for births.

- How to grid search ETS model hyperparameters for monthly time series data for shampoo sales, car sales and temperature.

12.9.1   Next

In the next lesson, you will discover how to develop autoregressive models for univariate time series forecasting problems.

Chapter 13

How to Develop SARIMA Models for Univariate Forecasting

The Seasonal Autoregressive Integrated Moving Average, or SARIMA, model is an approach for modeling univariate time series data that may contain trend and seasonal components. It is an effective approach for time series forecasting, although it requires careful analysis and domain expertise in order to configure the seven or more model hyperparameters. An alternative approach to configuring the model that makes use of fast and parallel modern hardware is to grid search a suite of hyperparameter configurations in order to discover what works best. Often, this process can reveal non-intuitive model configurations that result in lower forecast error than those configurations specified through careful analysis.

In this tutorial, you will discover how to develop a framework for grid searching all of the SARIMA model hyperparameters for univariate time series forecasting. After completing this tutorial, you will know:

- How to develop a framework for grid searching SARIMA models from scratch using walk-forward validation.

- How to grid search SARIMA model hyperparameters for daily time series data for births.

- How to grid search SARIMA model hyperparameters for monthly time series data for shampoo sales, car sales, and temperature.

Let's get started.

13.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Develop a Grid Search Framework

2. Case Study 1: No Trend or Seasonality

3. Case Study 2: Trend

4. Case Study 3: Seasonality

5. Case Study 4: Trend and Seasonality

13.2 Develop a Grid Search Framework

In this section, we will develop a framework for grid searching SARIMA model hyperparameters for a given univariate time series forecasting problem. For more information on SARIMA for time series forecasting, see Chapter 5. We will use the implementation of SARIMA provided by the Statsmodels library. This model has hyperparameters that control the nature of the model performed for the series, trend and seasonality, specifically:

- `order`: A tuple p, d, and q parameters for the modeling of the trend.

- `seasonal_order`: A tuple of P, D, Q, and m parameters for the modeling the seasonality

- `trend`: A parameter for controlling a model of the deterministic trend as one of 'n', 'c', 't', and 'ct' for no trend, constant, linear, and constant with linear trend, respectively.

If you know enough about your problem to specify one or more of these parameters, then you should specify them. If not, you can try grid searching these parameters. We can start-off by defining a function that will fit a model with a given configuration and make a one-step forecast. The `sarima_forecast()` below implements this behavior.

The function takes an array or list of contiguous prior observations and a list of configuration parameters used to configure the model, specifically two tuples and a string for the trend order, seasonal order trend, and parameter. We also try to make the model robust by relaxing constraints, such as that the data must be stationary and that the MA transform be invertible.

```
# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]
```

Listing 13.1: Example of a function for making a SARIMA forecast.

In this tutorial, we will use the grid searching framework developed in Chapter 11 for tuning and evaluating naive forecasting methods. One important modification to the framework is the function used to perform the walk-forward validation of the model named `walk_forward_validation()`. This function must be updated to call the function for making an SARIMA forecast. The updated version of the function is listed below.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
```

```
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error
```

Listing 13.2: Example of a function for walk-forward validation with SARIMA forecasts.

We're nearly done. The only thing left to do is to define a list of model configurations to try for a dataset. We can define this generically. The only parameter we may want to specify is the periodicity of the seasonal component in the series, if one exists. By default, we will assume no seasonal component. The `sarima_configs()` function below will create a list of model configurations to evaluate.

The configurations assume each of the AR, MA, and I components for trend and seasonality are low order, e.g. off (0) or in `[1,2]`. You may want to extend these ranges if you believe the order may be higher. An optional list of seasonal periods can be specified, and you could even change the function to specify other elements that you may know about your time series. In theory, there are 1,296 possible model configurations to evaluate, but in practice, many will not be valid and will result in an error that we will trap and ignore.

```
# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
                  cfg = [(p,d,q), (P,D,Q,m), t]
                  models.append(cfg)
  return models
```

Listing 13.3: Example of a function for defining configurations for SARIMA models to grid search.

We now have a framework for grid searching SARIMA model hyperparameters via one-step walk-forward validation. It is generic and will work for any in-memory univariate time series

provided as a list or NumPy array. We can make sure all the pieces work together by testing it on a contrived 10-step dataset. The complete example is listed below.

```python
# grid search sarima hyperparameters
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
```

```python
    key = str(cfg)
    # show all warnings and fail on exception if debugging
    if debug:
        result = walk_forward_validation(data, n_test, cfg)
    else:
        # one failure during model validation suggests an unstable config
        try:
            # never show warnings when grid searching, too noisy
            with catch_warnings():
                filterwarnings("ignore")
                result = walk_forward_validation(data, n_test, cfg)
        except:
            error = None
    # check for an interesting result
    if result is not None:
        print(' > Model[%s] %.3f' % (key, result))
    return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
    scores = None
    if parallel:
        # execute configs in parallel
        executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
        tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
        scores = executor(tasks)
    else:
        scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
    # remove empty results
    scores = [r for r in scores if r[1] != None]
    # sort configs by error, asc
    scores.sort(key=lambda tup: tup[1])
    return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
    models = list()
    # define config lists
    p_params = [0, 1, 2]
    d_params = [0, 1]
    q_params = [0, 1, 2]
    t_params = ['n','c','t','ct']
    P_params = [0, 1, 2]
    D_params = [0, 1]
    Q_params = [0, 1, 2]
    m_params = seasonal
    # create config instances
    for p in p_params:
        for d in d_params:
            for q in q_params:
                for t in t_params:
                    for P in P_params:
                        for D in D_params:
                            for Q in Q_params:
                                for m in m_params:
                                    cfg = [(p,d,q), (P,D,Q,m), t]
```

```
                models.append(cfg)
  return models

if __name__ == '__main__':
  # define dataset
  data = [10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]
  print(data)
  # data split
  n_test = 4
  # model configs
  cfg_list = sarima_configs()
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.4: Example of demonstrating the grid search infrastructure.

Running the example first prints the contrived time series dataset. Next, the model configurations and their errors are reported as they are evaluated, truncated below for brevity. Finally, the configurations and the error for the top three configurations are reported. We can see that many models achieve perfect performance on this simple linearly increasing contrived time series problem.

```
[10.0, 20.0, 30.0, 40.0, 50.0, 60.0, 70.0, 80.0, 90.0, 100.0]

...
 > Model[[(2, 0, 0), (2, 0, 0, 0), 'ct']] 0.001
 > Model[[(2, 0, 0), (2, 0, 1, 0), 'ct']] 0.000
 > Model[[(2, 0, 1), (0, 0, 0, 0), 'n']] 0.000
 > Model[[(2, 0, 1), (0, 0, 1, 0), 'n']] 0.000
done

[(2, 1, 0), (1, 0, 0, 0), 'n'] 0.0
[(2, 1, 0), (2, 0, 0, 0), 'n'] 0.0
[(2, 1, 1), (1, 0, 1, 0), 'n'] 0.0
```

Listing 13.5: Example output from demonstrating the grid search infrastructure.

Now that we have a robust framework for grid searching SARIMA model hyperparameters, let's test it out on a suite of standard univariate time series datasets. The datasets were chosen for demonstration purposes; I am not suggesting that a SARIMA model is the best approach for each dataset; perhaps an ETS or something else would be more appropriate in some cases.

13.3 Case Study 1: No Trend or Seasonality

The *daily female births* dataset summarizes the daily total female births in California, USA in 1959. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- `daily-total-female-births.csv` [1]

Save the file with the filename `daily-total-female-births.csv` in your current working directory. The dataset has one year, or 365 observations. We will use the first 200 for training and the remaining 165 as the test set. The complete example grid searching the daily female univariate time series forecasting problem is listed below.

```python
# grid search sarima hyperparameters for daily female dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
```

---

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/daily-total-female-births.csv

```python
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
```

```
      for d in d_params:
        for q in q_params:
          for t in t_params:
            for P in P_params:
              for D in D_params:
                for Q in Q_params:
                  for m in m_params:
                    cfg = [(p,d,q), (P,D,Q,m), t]
                    models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('daily-total-female-births.csv', header=0, index_col=0)
  data = series.values
  # data split
  n_test = 165
  # model configs
  cfg_list = sarima_configs()
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.6: Example of grid searching SARIMA models for the daily female births dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[[(2, 1, 2), (1, 0, 1, 0), 'ct']] 6.905
> Model[[(2, 1, 2), (2, 0, 0, 0), 'ct']] 7.031
> Model[[(2, 1, 2), (2, 0, 1, 0), 'ct']] 6.985
> Model[[(2, 1, 2), (1, 0, 2, 0), 'ct']] 6.941
> Model[[(2, 1, 2), (2, 0, 2, 0), 'ct']] 7.056
done

[(1, 0, 2), (1, 0, 1, 0), 't'] 6.770349800255089
[(0, 1, 2), (1, 0, 2, 0), 'ct'] 6.773217122759515
[(2, 1, 1), (2, 0, 2, 0), 'ct'] 6.886633191752254
```

Listing 13.7: Example output from grid searching SARIMA models for the daily female births dataset.

We can see that the best result was an RMSE of about 6.77 births. A naive model achieved an RMSE of 6.93 births suggesting that the best performing SARIMA model is skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Order**: (1, 0, 2)

- **Seasonal Order**: (1, 0, 1, 0)

- **Trend Parameter**: 't' for linear trend

It is surprising that a configuration with some seasonal elements resulted in the lowest error. I would not have guessed at this configuration and would have likely stuck with an ARIMA model.

13.4 Case Study 2: Trend

The *monthly shampoo sales* dataset summarizes the monthly sales of shampoo over a three-year period. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-shampoo-sales.csv [2]

Save the file with the filename monthly-shampoo-sales.csv in your current working directory. The dataset has three years, or 36 observations. We will use the first 24 for training and the remaining 12 as the test set. The complete example grid searching the shampoo sales univariate time series forecasting problem is listed below.

```python
# grid search sarima hyperparameters for monthly shampoo sales dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

---

[2]https://raw.githubusercontent.com/jbrownlee/Datasets/master/shampoo.csv

```python
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
```

```
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
 models = list()
 # define config lists
 p_params = [0, 1, 2]
 d_params = [0, 1]
 q_params = [0, 1, 2]
 t_params = ['n','c','t','ct']
 P_params = [0, 1, 2]
 D_params = [0, 1]
 Q_params = [0, 1, 2]
 m_params = seasonal
 # create config instances
 for p in p_params:
   for d in d_params:
     for q in q_params:
       for t in t_params:
         for P in P_params:
           for D in D_params:
             for Q in Q_params:
               for m in m_params:
                 cfg = [(p,d,q), (P,D,Q,m), t]
                 models.append(cfg)
 return models

if __name__ == '__main__':
 # load dataset
 series = read_csv('monthly-shampoo-sales.csv', header=0, index_col=0)
 data = series.values
 # data split
 n_test = 12
 # model configs
 cfg_list = sarima_configs()
 # grid search
 scores = grid_search(data, cfg_list, n_test)
 print('done')
 # list top 3 configs
 for cfg, error in scores[:3]:
   print(cfg, error)
```

Listing 13.8: Example of grid searching SARIMA models for the monthly shampoo sales dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[[(2, 1, 2), (1, 0, 1, 0), 'ct']] 68.891
> Model[[(2, 1, 2), (2, 0, 0, 0), 'ct']] 75.406
```

```
> Model[[(2, 1, 2), (1, 0, 2, 0), 'ct']] 80.908
> Model[[(2, 1, 2), (2, 0, 1, 0), 'ct']] 78.734
> Model[[(2, 1, 2), (2, 0, 2, 0), 'ct']] 82.958
done

[(0, 1, 2), (2, 0, 2, 0), 't'] 54.767582003072874
[(0, 1, 1), (2, 0, 2, 0), 'ct'] 58.69987083057107
[(1, 1, 2), (0, 0, 1, 0), 't'] 58.709089340600094
```

Listing 13.9: Example output from grid searching naive models for the monthly shampoo sales dataset.

We can see that the best result was an RMSE of about 54.76 sales. A naive model achieved an RMSE of 95.69 sales on this dataset, meaning that the best performing SARIMA model is skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend Order**: (0, 1, 2)

- **Seasonal Order**: (2, 0, 2, 0)

- **Trend Parameter**: '`t`' (linear trend)

13.5 Case Study 3: Seasonality

The *monthly mean temperatures* dataset summarizes the monthly average air temperatures in Nottingham Castle, England from 1920 to 1939 in degrees Fahrenheit. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-mean-temp.csv [3]

Save the file with the filename `monthly-mean-temp.csv` in your current working directory. The dataset has 20 years, or 240 observations. We will trim the dataset to the last five years of data (60 observations) in order to speed up the model evaluation process and use the last year or 12 observations for the test set.

```
# trim dataset to 5 years
data = data[-(5*12):]
```

Listing 13.10: Example of reducing the size of the dataset.

The period of the seasonal component is about one year, or 12 observations. We will use this as the seasonal period in the call to the `sarima_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = sarima_configs(seasonal=[0, 12])
```

Listing 13.11: Example of specifying some seasonal configurations.

---

[3]https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-mean-temp.csv

The complete example grid searching the monthly mean temperature time series forecasting problem is listed below.

```python
# grid search sarima hyperparameters for monthly mean temp dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
```

```python
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
  models = list()
  # define config lists
  p_params = [0, 1, 2]
  d_params = [0, 1]
  q_params = [0, 1, 2]
  t_params = ['n','c','t','ct']
  P_params = [0, 1, 2]
  D_params = [0, 1]
  Q_params = [0, 1, 2]
  m_params = seasonal
  # create config instances
  for p in p_params:
    for d in d_params:
      for q in q_params:
        for t in t_params:
          for P in P_params:
            for D in D_params:
              for Q in Q_params:
                for m in m_params:
```

```
                    cfg = [(p,d,q), (P,D,Q,m), t]
                    models.append(cfg)
  return models

if __name__ == '__main__':
  # load dataset
  series = read_csv('monthly-mean-temp.csv', header=0, index_col=0)
  data = series.values
  # trim dataset to 5 years
  data = data[-(5*12):]
  # data split
  n_test = 12
  # model configs
  cfg_list = sarima_configs(seasonal=[0, 12])
  # grid search
  scores = grid_search(data, cfg_list, n_test)
  print('done')
  # list top 3 configs
  for cfg, error in scores[:3]:
    print(cfg, error)
```

Listing 13.12: Example of grid searching SARIMA models for the monthly mean temperature dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> Model[[(2, 1, 2), (2, 1, 0, 12), 't']] 4.599
> Model[[(2, 1, 2), (1, 1, 0, 12), 'ct']] 2.477
> Model[[(2, 1, 2), (2, 0, 0, 12), 'ct']] 2.548
> Model[[(2, 1, 2), (2, 0, 1, 12), 'ct']] 2.893
> Model[[(2, 1, 2), (2, 1, 0, 12), 'ct']] 5.404
done

[(0, 0, 0), (1, 0, 1, 12), 'n'] 1.5577613610905712
[(0, 0, 0), (1, 1, 0, 12), 'n'] 1.6469530713847962
[(0, 0, 0), (2, 0, 0, 12), 'n'] 1.7314448163607488
```

Listing 13.13: Example output from grid searching SARIMA models for the monthly mean temperature dataset.

We can see that the best result was an RMSE of about 1.55 degrees. A naive model achieved an RMSE of 1.50 degrees, suggesting that the best performing SARIMA model is not skillful on this problem. We can unpack the configuration of the best performing model as follows:

- **Trend Order**: (0, 0, 0)

- **Seasonal Order**: (1, 0, 1, 12)

- **Trend Parameter**: 'n' (no trend)

As we would expect, the model has no trend component and a 12-month seasonal ARIMA component.

13.6   Case Study 4: Trend and Seasonality

The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-car-sales.csv [4]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. The dataset has 9 years, or 108 observations. We will use the last year or 12 observations as the test set. The period of the seasonal component could be six months or 12 months. We will try both as the seasonal period in the call to the `sarima_configs()` function when preparing the model configurations.

```
# model configs
cfg_list = sarima_configs(seasonal=[0,6,12])
```

Listing 13.14: Example of specifying some seasonal configurations.

The complete example grid searching the monthly car sales time series forecasting problem is listed below.

```
# grid search sarima hyperparameters for monthly car sales dataset
from math import sqrt
from multiprocessing import cpu_count
from joblib import Parallel
from joblib import delayed
from warnings import catch_warnings
from warnings import filterwarnings
from statsmodels.tsa.statespace.sarimax import SARIMAX
from sklearn.metrics import mean_squared_error
from pandas import read_csv

# one-step sarima forecast
def sarima_forecast(history, config):
  order, sorder, trend = config
  # define model
  model = SARIMAX(history, order=order, seasonal_order=sorder, trend=trend,
      enforce_stationarity=False, enforce_invertibility=False)
  # fit model
  model_fit = model.fit(disp=False)
  # make one step forecast
  yhat = model_fit.predict(len(history), len(history))
  return yhat[0]

# root mean squared error or rmse
```

```python
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = sarima_forecast(history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  return error

# score a model, return None on failure
def score_model(data, n_test, cfg, debug=False):
  result = None
  # convert config to a key
  key = str(cfg)
  # show all warnings and fail on exception if debugging
  if debug:
    result = walk_forward_validation(data, n_test, cfg)
  else:
    # one failure during model validation suggests an unstable config
    try:
      # never show warnings when grid searching, too noisy
      with catch_warnings():
        filterwarnings("ignore")
        result = walk_forward_validation(data, n_test, cfg)
    except:
      error = None
  # check for an interesting result
  if result is not None:
    print(' > Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test, parallel=True):
  scores = None
  if parallel:
    # execute configs in parallel
    executor = Parallel(n_jobs=cpu_count(), backend='multiprocessing')
    tasks = (delayed(score_model)(data, n_test, cfg) for cfg in cfg_list)
    scores = executor(tasks)
```

```python
  else:
    scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # remove empty results
  scores = [r for r in scores if r[1] != None]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a set of sarima configs to try
def sarima_configs(seasonal=[0]):
 models = list()
 # define config lists
 p_params = [0, 1, 2]
 d_params = [0, 1]
 q_params = [0, 1, 2]
 t_params = ['n','c','t','ct']
 P_params = [0, 1, 2]
 D_params = [0, 1]
 Q_params = [0, 1, 2]
 m_params = seasonal
 # create config instances
 for p in p_params:
   for d in d_params:
     for q in q_params:
       for t in t_params:
         for P in P_params:
           for D in D_params:
             for Q in Q_params:
               for m in m_params:
                 cfg = [(p,d,q), (P,D,Q,m), t]
                 models.append(cfg)
 return models

if __name__ == '__main__':
 # load dataset
 series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
 data = series.values
 # data split
 n_test = 12
 # model configs
 cfg_list = sarima_configs(seasonal=[0,6,12])
 # grid search
 scores = grid_search(data, cfg_list, n_test)
 print('done')
 # list top 3 configs
 for cfg, error in scores[:3]:
   print(cfg, error)
```

Listing 13.15: Example of grid searching SARIMA models for the monthly car sales dataset.

Running the example may take a while on modern hardware. Model configurations and the RMSE are printed as the models are evaluated. The top three model configurations and their error are reported at the end of the run. A truncated example of the results from running the hyperparameter grid search are listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider

running the example a few times.

```
...
> Model[[(2, 1, 2), (2, 0, 2, 12), 'ct']] 10710.462
> Model[[(2, 1, 2), (2, 1, 2, 6), 'ct']] 2183.568
> Model[[(2, 1, 2), (2, 1, 0, 12), 'ct']] 2105.800
> Model[[(2, 1, 2), (2, 1, 1, 12), 'ct']] 2330.361
> Model[[(2, 1, 2), (2, 1, 2, 12), 'ct']] 31580326686.803
done

[(0, 0, 0), (1, 1, 0, 12), 't'] 1551.8423920342414
[(0, 0, 0), (2, 1, 1, 12), 'c'] 1557.334614575545
[(0, 0, 0), (1, 1, 0, 12), 'c'] 1559.3276311282675
```

Listing 13.16: Example output from grid searching SARIMA models for the monthly car sales dataset.

We can see that the best result was an RMSE of about 1,551.84 sales. A naive model achieved an RMSE of 1,841.15 sales on this problem, suggesting that the best performing SARIMA model is skillful. We can unpack the configuration of the best performing model as follows:

- **Trend Order**: (0, 0, 0)

- **Seasonal Order**: (1, 1, 0, 12)

- **Trend Parameter**: '`t`' (linear trend)

13.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Transforms**. Update the framework to support configurable data transforms such as normalization and standardization.

- **Plot Forecast**. Update the framework to re-fit a model with the best configuration and forecast the entire test dataset, then plot the forecast compared to the actual observations in the test set.

- **Tune Amount of History**. Update the framework to tune the amount of historical data used to fit the model (e.g. in the case of the 10 years of max temperature data).

If you explore any of these extensions, I'd love to know.

13.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

13.8.1 Books

- Chapter 8 ARIMA models, *Forecasting: principles and practice*, 2013.
  https://amzn.to/2xlJsfV

- Chapter 7, Non-stationary Models, *Introductory Time Series with R*, 2009.
  https://amzn.to/2smB9LR

13.8.2 APIs

- Statsmodels Time Series Analysis by State Space Methods.
  http://www.statsmodels.org/dev/statespace.html

- `statsmodels.tsa.statespace.sarimax.SARIMAX` API.
  http://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAX.html

- `statsmodels.tsa.statespace.sarimax.SARIMAXResults` API.
  http://www.statsmodels.org/dev/generated/statsmodels.tsa.statespace.sarimax.SARIMAXResults.html

- Statsmodels SARIMAX Notebook.
  http://www.statsmodels.org/dev/examples/notebooks/generated/statespace_sarimax_stata.html

13.8.3 Articles

- Autoregressive integrated moving average, Wikipedia.
  https://en.wikipedia.org/wiki/Autoregressive_integrated_moving_average

13.9 Summary

In this tutorial, you discovered how to develop a framework for grid searching all of the SARIMA model hyperparameters for univariate time series forecasting. Specifically, you learned:

- How to develop a framework for grid searching SARIMA models from scratch using walk-forward validation.

- How to grid search SARIMA model hyperparameters for daily time series data for births.

- How to grid search SARIMA model hyperparameters for monthly time series data for shampoo sales, car sales and temperature.

13.9.1 Next

In the next lesson, you will discover how to develop deep learning models for univariate time series forecasting problems.

Chapter 14

How to Develop MLPs, CNNs and LSTMs for Univariate Forecasting

Deep learning neural networks are capable of automatically learning and extracting features from raw data. This feature of neural networks can be used for time series forecasting problems, where models can be developed directly on the raw observations without the direct need to scale the data using normalization and standardization or to make the data stationary by differencing. Impressively, simple deep learning neural network models are capable of making skillful forecasts as compared to naive models and tuned SARIMA models on univariate time series forecasting problems that have both trend and seasonal components with no pre-processing.

In this tutorial, you will discover how to develop a suite of deep learning models for univariate time series forecasting. After completing this tutorial, you will know:

- How to develop a robust test harness using walk-forward validation for evaluating the performance of neural network models.

- How to develop and evaluate simple Multilayer Perceptron and convolutional neural networks for time series forecasting.

- How to develop and evaluate LSTMs, CNN-LSTMs, and ConvLSTM neural network models for time series forecasting.

Let's get started.

14.1   Tutorial Overview

This tutorial is divided into five parts; they are:

1. Time Series Problem

2. Model Evaluation Test Harness

3. Multilayer Perceptron Model

4. Convolutional Neural Network Model

5. Recurrent Neural Network Models

14.2 Time Series Problem

In this tutorial we will focus on one dataset and use it as the context to demonstrate the development of a range of deep learning models for univariate time series forecasting. We will use the *monthly car sales* dataset as this context as it includes the complexity of both trend and seasonal elements. The *monthly car sales* dataset summarizes the monthly car sales in Quebec, Canada between 1960 and 1968. For more information on this dataset, see Chapter 11 where it was introduced. You can download the dataset directly from here:

- monthly-car-sales.csv [1]

Save the file with the filename `monthly-car-sales.csv` in your current working directory. The dataset is monthly and has nine years, or 108 observations. In our testing, will use the last year, or 12 observations, as the test set. A line plot is created. The dataset has an obvious trend and seasonal component. The period of the seasonal component could be six months or 12 months. From prior experiments, we know that a naive model can achieve a root mean squared error, or RMSE, of 1,841.155 by taking the median of the observations at the three prior years for the month being predicted (see Chapter 11); for example:

```
yhat = median(-12, -24, -36)
```

Listing 14.1: Example of an effective naive forecast model.

Where the negative indexes refer to observations in the series relative to the end of the historical data for the month being predicted. From prior experiments, we know that a SARIMA model can achieve an RMSE of 1,551.84 with the configuration of `SARIMA(0, 0, 0),(1, 1, 0),12` where no elements are specified for the trend and a seasonal difference with a period of 12 is calculated and an AR model of one season is used (see Chapter 13).

The performance of the naive model provides a lower bound on a model that is considered skillful. Any model that achieves a predictive performance of lower than 1,841.15 on the last 12 months has skill. The performance of the SARIMA model provides a measure of a good model on the problem. Any model that achieves a predictive performance lower than 1,551.84 on the last 12 months should be adopted over a SARIMA model. Now that we have defined our problem and expectations of model skill, we can look at defining the test harness.

14.3 Model Evaluation Test Harness

In this section, we will develop a test harness for developing and evaluating different types of neural network models for univariate time series forecasting. This test harness is a modified version of the framework presented in Chapter 11 and may cover much of the same of the same ground. This section is divided into the following parts:

1. Train-Test Split

2. Series as Supervised Learning

3. Walk-Forward Validation

---

[1]https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-car-sales.csv

4. Repeat Evaluation

5. Summarize Performance

6. Worked Example

14.3.1 Train-Test Split

The first step is to split the loaded series into train and test sets. We will use the first eight years (96 observations) for training and the last 12 for the test set. The `train_test_split()` function below will split the series taking the raw observations and the number of observations to use in the test set as arguments.

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

Listing 14.2: Example of a function for splitting data into train and test sets.

14.3.2 Series as Supervised Learning

Next, we need to be able to frame the univariate series of observations as a supervised learning problem so that we can train neural network models (covered in Chapter 4). A supervised learning framing of a series means that the data needs to be split into multiple examples that the model learn from and generalize across. Each sample must have both an input component and an output component. The input component will be some number of prior observations, such as three years or 36 time steps. The output component will be the total sales in the next month because we are interested in developing a model to make one-step forecasts.

We can implement this using the `shift()` function on the Pandas `DataFrame`. It allows us to shift a column down (forward in time) or back (backward in time). We can take the series as a column of data, then create multiple copies of the column, shifted forward or backward in time in order to create the samples with the input and output elements we require. When a series is shifted down, `NaN` values are introduced because we don't have values beyond the start of the series. For example, the series defined as a column:

```
(t)
1
2
3
4
```

Listing 14.3: Example of a time series as a column of data.

Can be shifted and inserted as a column beforehand:

```
(t-1),    (t)
Nan,      1
1,        2
2,        3
3,        4
4,        NaN
```

Listing 14.4: Example of column time series data with an added shifted column.

We can see that on the second row, the value 1 is provided as input as an observation at the prior time step, and 2 is the next value in the series that can be predicted, or learned by the model to be predicted when 1 is presented as input. Rows with `NaN` values can be removed. The `series_to_supervised()` function below implements this behavior, allowing you to specify the number of lag observations to use in the input and the number to use in the output for each sample. It will also remove rows that have `NaN` values as they cannot be used to train or test a model.

```python
# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values
```

Listing 14.5: Example of a function for transforming a univariate series into a supervised learning dataset.

Note, this is a more generic way of transforming a time series dataset into samples than the specialized methods presented in Chapters 7, 8, and 9.

14.3.3 Walk-Forward Validation

Time series forecasting models can be evaluated on a test set using walk-forward validation. Walk-forward validation is an approach where the model makes a forecast for each observation in the test dataset one at a time. After each forecast is made for a time step in the test dataset, the true observation for the forecast is added to the test dataset and made available to the model. Simpler models can be refit with the observation prior to making the subsequent prediction. More complex models, such as neural networks, are not refit given the much greater computational cost. Nevertheless, the true observation for the time step can then be used as part of the input for making the prediction on the next time step. First, the dataset is split into train and test sets. We will call the `train_test_split()` function to perform this split and pass in the pre-specified number of observations to use as the test data.

A model will be fit once on the training dataset for a given configuration. We will define a generic `model_fit()` function to perform this operation that can be filled in for the given type of neural network that we may be interested in later. The function takes the training dataset and the model configuration and returns the fit model ready for making predictions.

```python
# fit a model
def model_fit(train, config):
  return None
```

Listing 14.6: Example of a function for fitting a model with a configuration.

Each time step of the test dataset is enumerated. A prediction is made using the fit model. Again, we will define a generic function named `model_predict()` that takes the fit model, the history, and the model configuration and makes a single one-step prediction.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  return 0.0
```
Listing 14.7: Example of a function for making a forecast with a fit model.

The prediction is added to a list of predictions and the true observation from the test set is added to a list of observations that was seeded with all observations from the training dataset. This list is built up during each step in the walk-forward validation, allowing the model to make a one-step prediction using the most recent history. All of the predictions can then be compared to the true values in the test set and an error measure calculated. We will calculate the root mean squared error, or RMSE, between predictions and the true values.

RMSE is calculated as the square root of the average of the squared differences between the forecasts and the actual values. The `measure_rmse()` implements this below using the `mean_squared_error()` scikit-learn function to first calculate the mean squared error, or MSE, before calculating the square root.

```
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```
Listing 14.8: Example of a function for calculating RMSE for a forecast.

The complete `walk_forward_validation()` function that ties all of this together is listed below. It takes the dataset, the number of observations to use as the test set, and the configuration for the model, and returns the RMSE for the model performance on the test set.

```
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
 predictions = list()
 # split dataset
 train, test = train_test_split(data, n_test)
 # fit model
 model = model_fit(train, cfg)
 # seed history with training dataset
 history = [x for x in train]
 # step over each time step in the test set
 for i in range(len(test)):
   # fit model and make forecast for history
   yhat = model_predict(model, history, cfg)
   # store forecast in list of predictions
   predictions.append(yhat)
   # add actual observation to history for the next loop
   history.append(test[i])
 # estimate prediction error
 error = measure_rmse(test, predictions)
 print(' > %.3f' % error)
 return error
```
Listing 14.9: Example of a function for walk-forward validation of a forecast model configuration.

14.3.4 Repeat Evaluation

Neural network models are stochastic. This means that, given the same model configuration and the same training dataset, a different internal set of weights will result each time the model is trained that will in turn have a different performance. This is a benefit, allowing the model to be adaptive and find high performing configurations to complex problems. It is also a problem when evaluating the performance of a model and in choosing a final model to use to make predictions.

To address model evaluation, we will evaluate a model configuration multiple times via walk-forward validation and report the error as the average error across each evaluation. This is not always possible for large neural networks and may only make sense for small networks that can be fit in minutes or hours. The `repeat_evaluate()` function below implements this and allows the number of repeats to be specified as an optional parameter that defaults to 30 and returns a list of model performance scores: in this case, RMSE values.

```
# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores
```

Listing 14.10: Example of a function for the repeated evaluation of a forecast model configuration.

14.3.5 Summarize Performance

Finally, we need to summarize the performance of a model from the multiple repeats. We will summarize the performance first using summary statistics, specifically the mean and the standard deviation. We will also plot the distribution of model performance scores using a box and whisker plot to help get an idea of the spread of performance. The `summarize_scores()` function below implements this, taking the name of the model that was evaluated and the list of scores from each repeated evaluation, printing the summary and showing a plot.

```
# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()
```

Listing 14.11: Example of a function for summarizing the performance of a forecast model.

14.3.6 Worked Example

Now that we have defined the elements of the test harness, we can tie them all together and define a simple persistence model. Specifically, we will calculate the median of a subset of prior observations relative to the time to be forecasted. We do not need to fit a model so the `model_fit()` function will be implemented to simply return `None`.

```
# fit a model
def model_fit(train, config):
  return None
```

Listing 14.12: Example of a function for fitting a forecast model.

We will use the config to define a list of index offsets in the prior observations relative to the time to be forecasted that will be used as the prediction. For example, 12 will use the observation 12 months ago (-12) relative to the time to be forecasted.

```
# define config
config = [12, 24, 36]
```

Listing 14.13: Example of a configuration to evaluate.

The model_predict() function can be implemented to use this configuration to collect the observations, then return the median of those observations.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  values = list()
  for offset in config:
    values.append(history[-offset])
  return median(values)
```

Listing 14.14: Example of a function for making a naive forecast.

The complete example of using the framework with a simple persistence model is listed below.

```
# persistence forecast for monthly car sales dataset
from math import sqrt
from numpy import median
from numpy import mean
from numpy import std
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]

# fit a model
def model_fit(train, config):
  return None

# forecast with a pre-fit model
def model_predict(model, history, config):
```

```python
  values = list()
  for offset in config:
    values.append(history[-offset])
  return median(values)

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [12, 24, 36]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('persistence', scores)
```

Listing 14.15: Example of demonstrating the forecast infrastructure with a naive model.

Running the example prints the RMSE of the model evaluated using walk-forward validation

on the final 12 months of data. The model is evaluated 30 times, although, because the model has no stochastic element, the score is the same each time.

```
...
 > 1841.156
 > 1841.156
 > 1841.156
 > 1841.156
 > 1841.156
persistence: 1841.156 RMSE (+/- 0.000)
```

Listing 14.16: Example output from demonstrating the forecast infrastructure with a naive model.

We can see that the RMSE of the model is 1841 sales, providing a lower-bound of performance by which we can evaluate whether a model is skillful or not on the problem. A box and whisker plot is also created, but is not reproduced here because there is no distribution to summarize, e.g. the plot is not interesting in this case as all skill scores have the same value. Now that we have a robust test harness, we can use it to evaluate a suite of neural network models.

14.4   Multilayer Perceptron Model

The first network that we will evaluate is a Multilayer Perceptron, or MLP for short. This is a simple feedforward neural network model that should be evaluated before more elaborate models are considered. MLPs can be used for time series forecasting by taking multiple observations at prior time steps, called lag observations, and using them as input features and predicting one or more time steps from those observations. This is exactly the framing of the problem provided by the `series_to_supervised()` function in the previous section. The training dataset is therefore a list of samples, where each sample has some number of observations from months prior to the time being forecasted, and the forecast is the next month in the sequence. For example:

```
X,                      y
month1, month2, month3,  month4
month2, month3, month4,  month5
month3, month4, month5,  month6
...
```

Listing 14.17: Example framing of the forecast problem for training a model.

The model will attempt to generalize over these samples, such that when a new sample is provided beyond what is known by the model, it can predict something useful; for example:

```
X,                      y
month4, month5, month6,  ???
```

Listing 14.18: Example framing of the forecast problem for making a forecast.

We will implement a simple MLP using the Keras deep learning library. For more details on modeling a univariate time series with an MLP, see Chapter 7. The model will have an input layer with some number of prior observations. This can be specified using the `input_dim` argument when we define the first hidden layer. The model will have a single hidden layer with some number of nodes, then a single output layer. We will use the rectified linear activation function on the hidden layer as it performs well. We will use a linear activation function (the

default) on the output layer because we are predicting a continuous value. The loss function for the network will be the mean squared error loss, or MSE, and we will use the efficient Adam flavor of stochastic gradient descent to train the network.

```
# define model
model = Sequential()
model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

Listing 14.19: Example of defining an MLP forecast model.

The model will be fit for some number of training epochs (exposures to the training data) and batch size can be specified to define how often the weights are updated within each epoch. The `model_fit()` function for fitting an MLP model on the training dataset is listed below. The function expects the config to be a list with the following configuration hyperparameters:

- `n_input`: The number of lag observations to use as input to the model.

- `n_nodes`: The number of nodes to use in the hidden layer.

- `n_epochs`: The number of times to expose the model to the whole training dataset.

- `n_batch`: The number of samples within an epoch after which the weights are updated.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.20: Example of a function for defining and fitting a MLP forecast model.

Making a prediction with a fit MLP model is as straightforward as calling the `predict()` function and passing in one sample worth of input values required to make the prediction.

```
# make a prediction
yhat = model.predict(x_input, verbose=0)
```

Listing 14.21: Example of making a prediction with a fit MLP model.

In order to make a prediction beyond the limit of known data, this requires that the last n known observations are taken as an array and used as input. The `predict()` function expects one or more samples of inputs when making a prediction, so providing a single sample requires the array to have the shape `[1, n_input]`, where `n_input` is the number of time steps that the

model expects as input. Similarly, the `predict()` function returns an array of predictions, one for each sample provided as input. In the case of one prediction, there will be an array with one value. The `model_predict()` function below implements this behavior, taking the model, the prior observations, and model configuration as arguments, formulating an input sample and making a one-step prediction that is then returned.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape(1, n_input)
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.22: Example of a function for making a forecast with a fit MLP model.

We now have everything we need to evaluate an MLP model on the monthly car sales dataset. Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- **n_input**: 24 (e.g. 24 months)

- **n_nodes**: 500

- **n_epochs**: 100

- **n_batch**: 100

This configuration can be defined as a list:

```
# define config
config = [24, 500, 100, 100]
```

Listing 14.23: Example of good configuration for an MLP forecast model.

Note that when the training data is framed as a supervised learning problem, there are only 72 samples that can be used to train the model. Using a batch size of 72 or more means that the model is being trained using batch gradient descent instead of mini-batch gradient descent. This is often used for small datasets and means that weight updates and gradient calculations are performed at the end of each epoch, instead of multiple times within each epoch. The complete code example is listed below.

```
# evaluate mlp for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
```

```python
from keras.layers import Dense
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape(1, n_input)
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
```

```
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [24, 500, 100, 100]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('mlp', scores)
```

Listing 14.24: Example of an MLP model for forecasting monthly car sales.

Running the example prints the RMSE for each of the 30 repeated evaluations of the model. At the end of the run, the average and standard deviation RMSE are reported of about 1,526 sales. We can see that, on average, the chosen configuration has better performance than both the naive model (1,841.155) and the SARIMA model (1,551.842). This is impressive given that the model operated on the raw data directly without scaling or the data being made stationary.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > 1458.993
 > 1643.383
 > 1457.925
 > 1558.934
 > 1708.278

mlp: 1526.688 RMSE (+/- 134.789)
```

Listing 14.25: Example output from an MLP model for forecasting monthly car sales.

A box and whisker plot of the RMSE scores is created to summarize the spread of the performance for the model. This helps to understand the spread of the scores. We can see that although on average the performance of the model is impressive, the spread is large. The standard deviation is a little more than 134 sales, meaning a worse case model run that is 2 or 3 standard deviations in error from the mean error may be worse than the naive model. A challenge in using the MLP model is in harnessing the higher skill and minimizing the variance of the model across multiple runs.

This problem applies generally for neural networks. There are many strategies that you could use, but perhaps the simplest is simply to train multiple final models on all of the available data and use them in an ensemble when making predictions, e.g. the prediction is the average of 10-to-30 models.
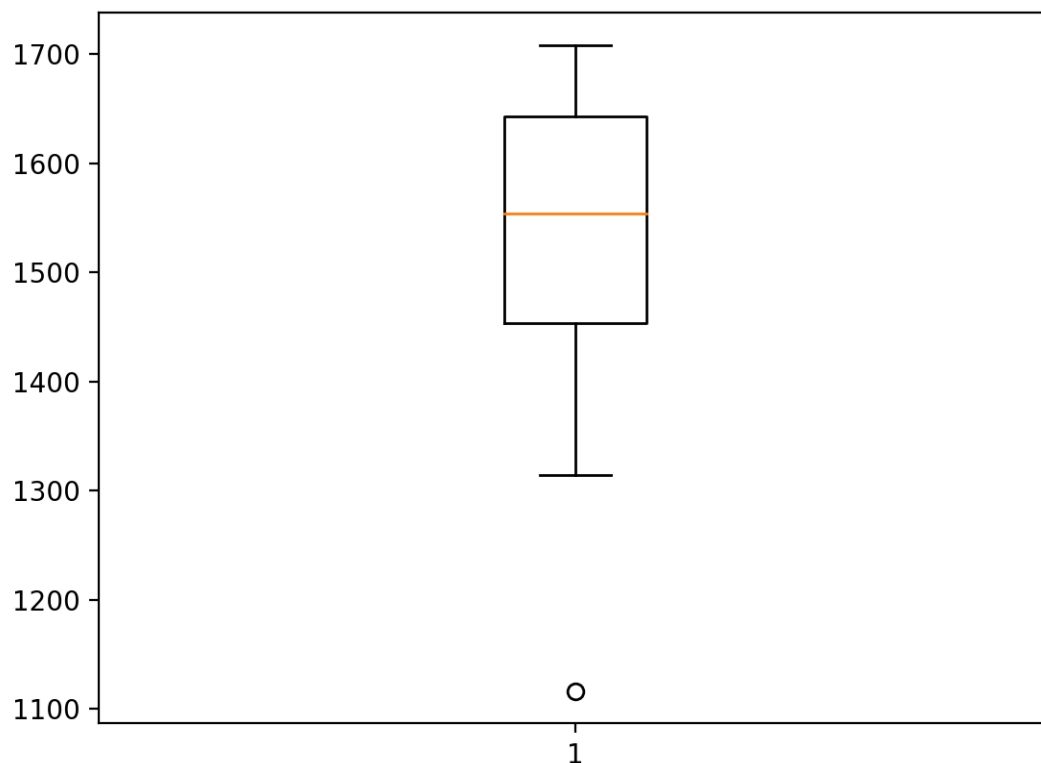
Figure 14.1: Box and Whisker Plot of Multilayer Perceptron RMSE Forecasting Car Sales.

14.5   Convolutional Neural Network Model

Convolutional Neural Networks, or CNNs, are a type of neural network developed for two-dimensional image data, although they can be used for one-dimensional data such as sequences of text and time series. When operating on one-dimensional data, the CNN reads across a sequence of lag observations and learns to extract features that are relevant for making a prediction. For more information on using CNNs for univariate time series forecasting, see Chapter 8. We will define a CNN with two convolutional layers for extracting features from the input sequences. Each will have a configurable number of filters and kernel size and will use the rectified linear activation function. The number of filters determines the number of parallel fields on which the weighted inputs are read and projected. The kernel size defines the number of time steps read within each snapshot as the network reads along the input sequence.

```
# define convolutional layers
model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
    input_shape=(n_input, 1)))
model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu'))
```

Listing 14.26: Example of defining convolutional layers.

A max pooling layer is used after convolutional layers to distill the weighted input features

into those that are most salient, reducing the input size by 1/4. The pooled inputs are flattened to one long vector before being interpreted and used to make a one-step prediction.

```
# define pooling and output layers
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(1))
```

Listing 14.27: Example of defining pooling, flatten and output layers.

The CNN model expects input data to be in the form of multiple samples, where each sample has multiple input time steps, the same as the MLP in the previous section. One difference is that the CNN can support multiple features or types of observations at each time step, which are interpreted as channels of an image. We only have a single feature at each time step, therefore the required three-dimensional shape of the input data will be `[n_samples, n_input, 1]`.

```
# reshape training data
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
```

Listing 14.28: Example of reshaping input data for the CNN model.

The `model_fit()` function for fitting the CNN model on the training dataset is listed below. The model takes the following five configuration parameters as a list:

- `n_input`: The number of lag observations to use as input to the model.

- `n_filters`: The number of parallel filters.

- `n_kernel`: The number of time steps considered in each read of the input sequence.

- `n_epochs`: The number of times to expose the model to the whole training dataset.

- `n_batch`: The number of samples within an epoch after which the weights are updated.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, 1)))
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.29: Example of a function for defining and fitting a CNN forecast model.

Making a prediction with the fit CNN model is very much like making a prediction with the fit MLP model in the previous section. The one difference is in the requirement that we specify the number of features observed at each time step, which in this case is 1. Therefore, when making a single one-step prediction, the shape of the input array must be: `[1, n_input, 1]`. The `model_predict()` function below implements this behavior.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.30: Example of a function for making a forecast with a fit CNN model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_input`: 36 (e.g. 3 years or 3×12)

- `n_filters`: 256

- `n_kernel`: 3

- `n_epochs`: 100

- `n_batch`: 100 (e.g. batch gradient descent)

This can be specified as a list as follows:

```
# define config
config = [36, 256, 3, 100, 100]
```

Listing 14.31: Example of good configuration for a CNN forecast model.

Tying all of this together, the complete example is listed below.

```
# evaluate cnn for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from matplotlib import pyplot
```

```python
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch = config
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, 1)))
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, _ = config
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

```python
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [36, 256, 3, 100, 100]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('cnn', scores)
```

Listing 14.32: Example of a CNN model for forecasting monthly car sales.

Running the example first prints the RMSE for each repeated evaluation of the model. At the end of the run, we can see that indeed the model is skillful, achieving an average RMSE of 1,524.06, which is better than the naive model, the SARIMA model, and even the MLP model in the previous section. This is impressive given that the model operated on the raw data directly without scaling or the data being made stationary.

The standard deviation of the score is large, at about 57 sales, but is $\frac{1}{3}$ the size of the standard deviation observed with the MLP model in the previous section. We have some confidence that in a bad-case scenario (3 standard deviations), the model RMSE will remain below (better than) the performance of the naive model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
> 1489.795
> 1652.620
> 1537.349
> 1443.777
> 1567.179

cnn: 1524.067 RMSE (+/- 57.148)
```

Listing 14.33: Example output from a CNN model for forecasting monthly car sales.

A box and whisker plot of the scores is created to help understand the spread of error across the runs. We can see that the spread does seem to be biased towards larger error values, as we would expect, although the upper whisker of the plot (in this case, the largest error that are not outliers) is still limited at an RMSE of 1,650 sales.
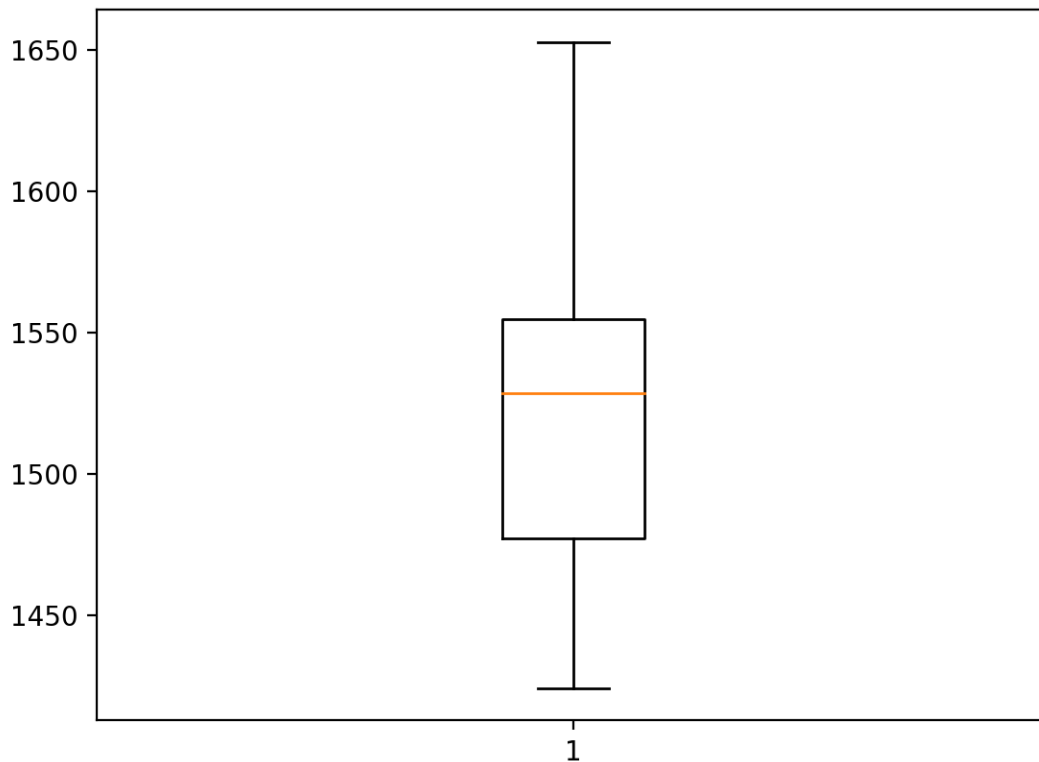
Figure 14.2: Box and Whisker Plot of Convolutional Neural Network RMSE Forecasting Car Sales.

14.6 Recurrent Neural Network Models

Recurrent neural networks, or RNNs, are those types of neural networks that use an output of the network from a prior step as an input in attempt to automatically learn across sequence data. The Long Short-Term Memory, or LSTM, network is a type of RNN whose implementation addresses the general difficulties in training RNNs on sequence data that results in a stable model. It achieves this by learning the weights for internal gates that control the recurrent connections within each node. Although developed for sequence data, LSTMs have not proven effective on time series forecasting problems where the output is a function of recent observations, e.g. an autoregressive type forecasting problem, such as the car sales dataset. Nevertheless, we can develop LSTM models for autoregressive problems and use them as a point of comparison with other neural network models. For more information on LSTMs for univariate time series forecasting, see Chapter 9. In this section, we will explore three variations on the LSTM model for univariate time series forecasting; they are:

- **Vanilla LSTM**: The LSTM network as-is.

- **CNN-LSTM**: A CNN network that learns input features and an LSTM that interprets them.

- **ConvLSTM**: A combination of CNNs and LSTMs where the LSTM units read input data using the convolutional process of a CNN.

14.6.1 LSTM

The LSTM neural network can be used for univariate time series forecasting. As an RNN, it will read each time step of an input sequence one step at a time. The LSTM has an internal memory allowing it to accumulate internal state as it reads across the steps of a given input sequence. At the end of the sequence, each node in a layer of hidden LSTM units will output a single value. This vector of values summarizes what the LSTM learned or extracted from the input sequence. This can be interpreted by a fully connected layer before a final prediction is made.

```
# define model
model = Sequential()
model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, 1)))
model.add(Dense(n_nodes, activation='relu'))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

Listing 14.34: Example of defining an LSTM forecast model.

Like the CNN, the LSTM can support multiple variables or features at each time step. As the car sales dataset only has one value at each time step, we can fix this at 1, both when defining the input to the network in the `input_shape` argument `[n_input, 1]`, and in defining the shape of the input samples.

```
# reshape input samples
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
```

Listing 14.35: Example of reshaping input data for the LSTM model.

Unlike the MLP and CNN that do not read the sequence data one-step at a time, the LSTM does perform better if the data is stationary. This means that difference operations are performed to remove the trend and seasonal structure. In the case of the car sales dataset, we can make the data stationery by performing a seasonal adjustment, that is subtracting the value from one year ago from each observation.

```
# seasonal differencing
adjusted = value - value[-12]
```

Listing 14.36: Example of seasonal differencing.

This can be performed systematically for the entire training dataset. It also means that the first year of observations must be discarded as we have no prior year of data to difference them with. The `difference()` function below will difference a provided dataset with a provided offset, called the difference order, e.g. 12 for one year of months prior.

```
# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]
```

Listing 14.37: Example of a function for differencing a series.

We can make the difference order a hyperparameter to the model and only perform the operation if a value other than zero is provided. The `model_fit()` function for fitting an LSTM model is provided below. The model expects a list of five model hyperparameters; they are:

- `n_input`: The number of lag observations to use as input to the model.

- `n_nodes`: The number of LSTM units to use in the hidden layer.

- `n_epochs`: The number of times to expose the model to the whole training dataset.

- `n_batch`: The number of samples within an epoch after which the weights are updated.

- `n_diff`: The difference order or 0 if not used.

```python
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, 1)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.38: Example of a function for fitting an LSTM forecast model.

Making a prediction with the LSTM model is the same as making a prediction with a CNN model. A single input must have the three-dimensional structure of samples, time steps, and features, which in this case we only have 1 sample and 1 feature: `[1, n_input, 1]`. If the difference operation was performed, we must add back the value that was subtracted after the model has made a forecast. We must also difference the historical data prior to formulating the single input used to make a prediction. The `model_predict()` function below implements this behavior.

```python
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

```
	# forecast
	yhat = model.predict(x_input, verbose=0)
	return correction + yhat[0]
```

Listing 14.39: Example of a function for making a forecast with a fit LSTM model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_input`: 36 (i.e. 3 years or 3×12)

- `n_nodes`: 50

- `n_epochs`: 100

- `n_batch`: 100 (i.e. batch gradient descent)

- `n_diff`: 12 (i.e. seasonal difference)

This can be specified as a list:

```
# define config
config = [36, 50, 100, 100, 12]
```

Listing 14.40: Example of good configuration for an LSTM forecast model.

Tying all of this together, the complete example is listed below.

```
# evaluate lstm for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
	return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
	df = DataFrame(data)
	cols = list()
	# input sequence (t-n, ... t-1)
	for i in range(n_in, 0, -1):
		cols.append(df.shift(i))
	# forecast sequence (t, t+1, ... t+n)
	for i in range(0, n_out):
```

```python
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, 1)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
```

```python
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [36, 50, 100, 100, 12]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('lstm', scores)
```

Listing 14.41: Example of an LSTM model for forecasting monthly car sales.

Running the example, we can see the RMSE for each repeated evaluation of the model. At the end of the run, we can see that the average RMSE is about 2,109, which is worse than the naive model. This suggests that the chosen model is not skillful, and it was the best that could be found given the same resources used to find model configurations in the previous sections. This provides further evidence (although weak evidence) that LSTMs, at least alone, are perhaps a bad fit for autoregressive-type sequence prediction problems.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
```

```
> 2266.130
> 2105.043
> 2128.549
> 1952.002
> 2188.287

lstm: 2109.779 RMSE (+/- 81.373)
```

Listing 14.42: Example output from an LSTM model for forecasting monthly car sales.

A box and whisker plot is also created summarizing the distribution of RMSE scores. Even the base case for the model did not achieve the performance of a naive model.
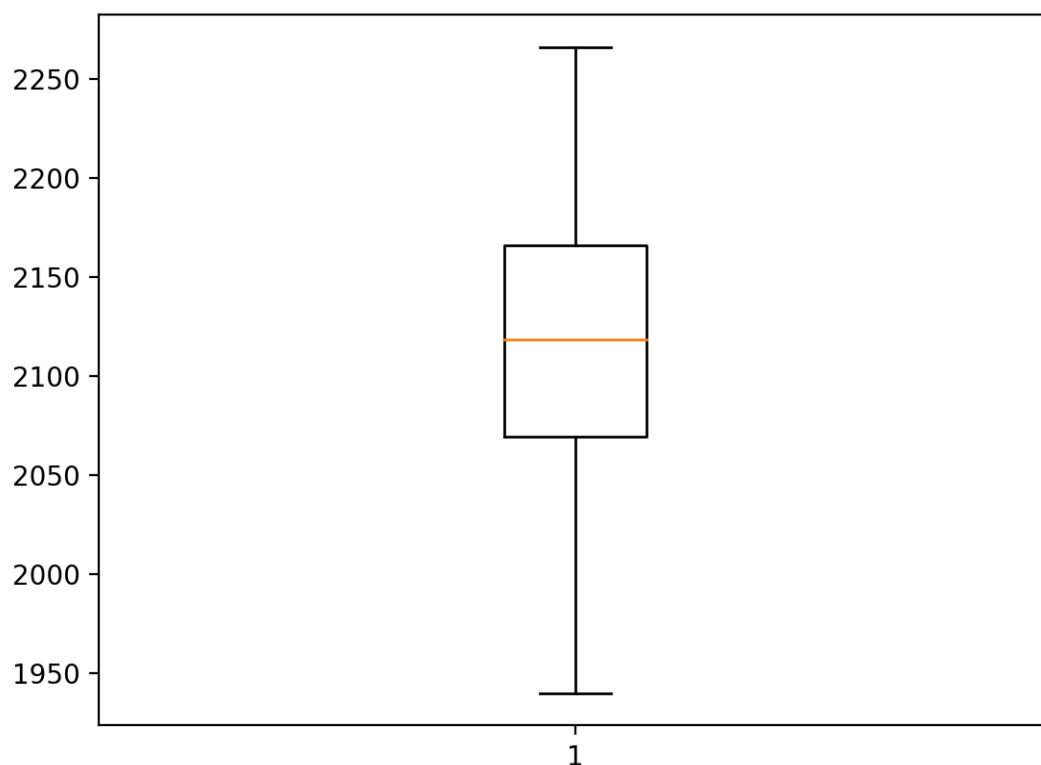

Figure 14.3: Box and Whisker Plot of Long Short-Term Memory Neural Network RMSE Forecasting Car Sales.

14.6.2   CNN LSTM

We have seen that the CNN model is capable of automatically learning and extracting features from the raw sequence data without scaling or differencing. We can combine this capability with the LSTM where a CNN model is applied to sub-sequences of input data, the results of which together form a time series of extracted features that can be interpreted by an LSTM model. This combination of a CNN model used to read multiple subsequences over time by an LSTM is called a CNN-LSTM model. The model requires that each input sequence, e.g. 36

months, is divided into multiple subsequences, each read by the CNN model, e.g. 3 subsequence of 12 time steps. It may make sense to divide the sub-sequences by years, but this is just a hypothesis, and other splits could be used, such as six subsequences of six time steps. Therefore, this splitting is parameterized with the `n_seq` and `n_steps` for the number of subsequences and number of steps per subsequence parameters.

```
# reshape input samples
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], 1))
```

Listing 14.43: Example of reshaping input data for the CNN-LSTM model.

The number of lag observations per sample is simply ($n\_seq \times n\_steps$). This is a 4-dimensional input array now with the dimensions: `[samples, subsequences, timesteps, features]`. The same CNN model must be applied to each input subsequence. We can achieve this by wrapping the entire CNN model in a `TimeDistributed` layer wrapper.

```
# define CNN input model
model = Sequential()
model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
    activation='relu', input_shape=(None,n_steps,1))))
model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
    activation='relu')))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
```

Listing 14.44: Example of defining the CNN input model.

The output of one application of the CNN submodel will be a vector. The output of the submodel to each input subsequence will be a time series of interpretations that can be interpreted by an LSTM model. This can be followed by a fully connected layer to interpret the outcomes of the LSTM and finally an output layer for making one-step predictions.

```
# define LSTM and output model
model.add(LSTM(n_nodes, activation='relu'))
model.add(Dense(n_nodes, activation='relu'))
model.add(Dense(1))
```

Listing 14.45: Example of defining the LSTM output model.

The complete `model_fit()` function is listed below. The model expects a list of seven hyperparameters; they are:

- `n_seq`: The number of subsequences within a sample.

- `n_steps`: The number of time steps within each subsequence.

- `n_filters`: The number of parallel filters.

- `n_kernel`: The number of time steps considered in each read of the input sequence.

- `n_nodes`: The number of LSTM units to use in the hidden layer.

- `n_epochs`: The number of times to expose the model to the whole training dataset.

- `n_batch`: The number of samples within an epoch after which the weights are updated.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, n_steps, 1))
  # define model
  model = Sequential()
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu', input_shape=(None,n_steps,1))))
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu')))
  model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(n_nodes, activation='relu'))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.46: Example of a function for defining and fitting a CNN-LSTM model.

Making a prediction with the fit model is much the same as the LSTM or CNN, although with the addition of splitting each sample into subsequences with a given number of time steps.

```
# prepare data
x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
```

Listing 14.47: Example of reshaping data for making a prediction.

The updated `model_predict()` function is listed below.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.48: Example of defining a function for making a forecast with a fit CNN-LSTM model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- n_seq: 3 (i.e. 3 years)

- n_steps: 12 (i.e. 1 year of months)

- n_filters: 64

- n_kernel: 3

- n_nodes: 100

- n_epochs: 200

- n_batch: 100 (i.e. batch gradient descent)

We can define the configuration as a list; for example:

```
# define config
config = [3, 12, 64, 3, 100, 200, 100]
```

Listing 14.49: Example of defining a good configuration for the CNN-LSTM model.

The complete example of evaluating the CNN-LSTM model for forecasting the univariate monthly car sales is listed below.

```
# evaluate cnn-lstm for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import TimeDistributed
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
```

```python
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, n_steps, 1))
  # define model
  model = Sequential()
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu', input_shape=(None,n_steps,1))))
  model.add(TimeDistributed(Conv1D(filters=n_filters, kernel_size=n_kernel,
      activation='relu')))
  model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(n_nodes, activation='relu'))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
```

```
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [3, 12, 64, 3, 100, 200, 100]
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('cnn-lstm', scores)
```

Listing 14.50: Example of a CNN-LSTM model for forecasting monthly car sales.

Running the example prints the RMSE for each repeated evaluation of the model. The final averaged RMSE is reported at the end of about 1,626, which is lower than the naive model, but still higher than a SARIMA model. The standard deviation of this score is also very large, suggesting that the chosen configuration may not be as stable as the standalone CNN model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > 1289.794
 > 1685.976
 > 1498.123
 > 1618.627
 > 1448.361

cnn-lstm: 1626.735 RMSE (+/- 279.850)
```

Listing 14.51: Example output from a CNN-LSTM model for forecasting monthly car sales.

A box and whisker plot is also created summarizing the distribution of RMSE scores. The plot shows one single outlier of very poor performance just below 3,000 sales.
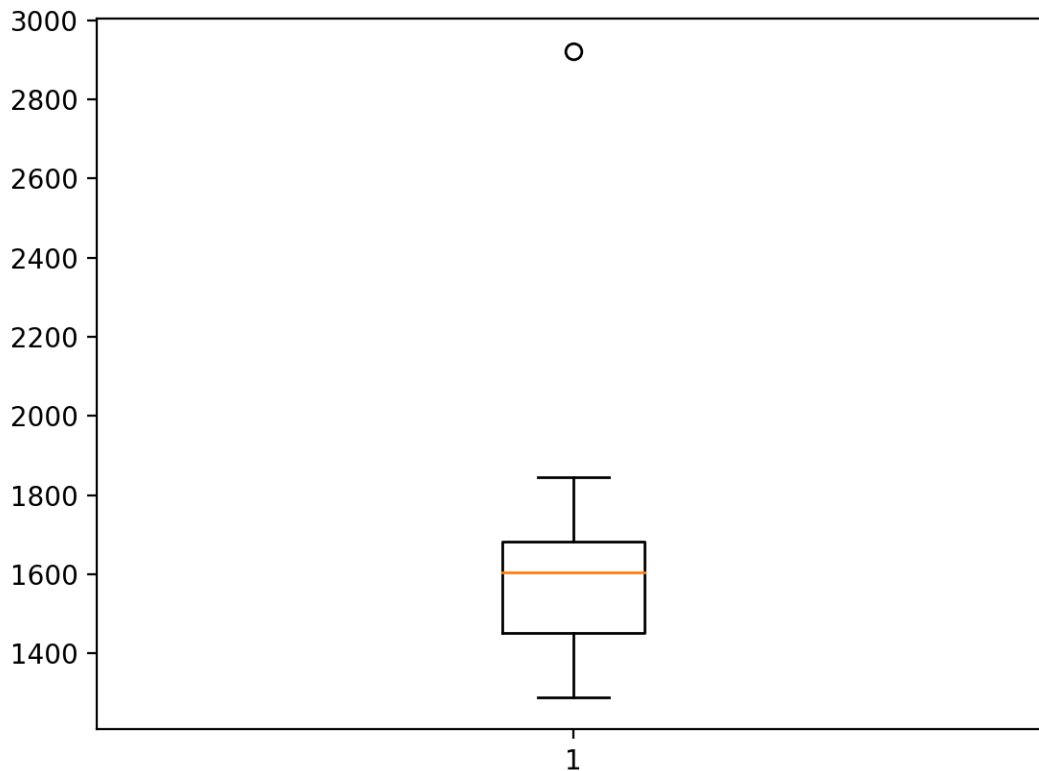

Figure 14.4: Box and Whisker Plot of CNN-LSTM RMSE Forecasting Car Sales.

14.6.3 ConvLSTM

It is possible to perform a convolutional operation as part of the read of the input sequence within each LSTM unit. This means, rather than reading a sequence one step at a time, the LSTM would read a block or subsequence of observations at a time using a convolutional process, like a CNN. This is different to first reading an extracting features with an LSTM and interpreting the result with an LSTM; this is performing the CNN operation at each time step as part of the LSTM.

This type of model is called a Convolutional LSTM, or ConvLSTM for short. It is provided in Keras as a layer called `ConvLSTM2D` for 2D data. We can configure it for use with 1D sequence data by assuming that we have one row with multiple columns. As with the CNN-LSTM, the input data is split into subsequences where each subsequence has a fixed number of time steps, although we must also specify the number of rows in each subsequence, which in this case is fixed at 1.

```
# reshape input samples
train_x = train_x.reshape((train_x.shape[0], n_seq, 1, n_steps, 1))
```

The shape is five-dimensional, with the dimensions: `[samples, subsequences, rows, columns, features]`.

Like the CNN, the ConvLSTM layer allows us to specify the number of filter maps and the size of the kernel used when reading the input sequences.

```
# define convlstm layer
model.add(ConvLSTM2D(filters=n_filters, kernel_size=(1,n_kernel), activation='relu',
    input_shape=(n_seq, 1, n_steps, 1)))
```

Listing 14.53: Example of defining a ConvLSTM layer for univariate data.

The output of the layer is a sequence of filter maps that must first be flattened before it can be interpreted and followed by an output layer. The model expects a list of seven hyperparameters, the same as the CNN-LSTM; they are:

- `n_seq`: The number of subsequences within a sample.

- `n_steps`: The number of time steps within each subsequence.

- `n_filters`: The number of parallel filters.

- `n_kernel`: The number of time steps considered in each read of the input sequence.

- `n_nodes`: The number of LSTM units to use in the hidden layer.

- `n_epochs`: The number of times to expose the model to the whole training dataset.

- `n_batch`: The number of samples within an epoch after which the weights are updated.

The `model_fit()` function that implements all of this is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, 1, n_steps, 1))
  # define model
  model = Sequential()
  model.add(ConvLSTM2D(filters=n_filters, kernel_size=(1,n_kernel), activation='relu',
      input_shape=(n_seq, 1, n_steps, 1)))
  model.add(Flatten())
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 14.54: Example of a function for defining and fitting a ConvLSTM forecast model.

A prediction is made with the fit model in the same way as the CNN-LSTM, although with the additional rows dimension that we fix to 1.

```
# prepare data
x_input = array(history[-n_input:]).reshape((1, n_seq, 1, n_steps, 1))
```

Listing 14.55: Example of reshaping input data for making a forecast with a ConvLSTM model.

The `model_predict()` function for making a single one-step prediction is listed below.

```
# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, 1, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]
```

Listing 14.56: Example of a function for making a forecast with a fit ConvLSTM model.

Model hyperparameters were chosen with a little trial and error and are listed below. The model may not be optimal for the problem and improvements could be made via grid searching. For details on how, see Chapter 15.

- `n_seq`: 3 (i.e. 3 years)

- `n_steps`: 12 (i.e. 1 year of months)

- `n_filters`: 256

- `n_kernel`: 3

- `n_nodes`: 200

- `n_epochs`: 200

- `n_batch`: 100 (i.e. batch gradient descent)

We can define the configuration as a list; for example:

```
# define config
config = [3, 12, 256, 3, 200, 200, 100]
```

Listing 14.57: Example of defining a good configuration for the ConvLSTM model.

We can tie all of this together. The complete code listing for the ConvLSTM model evaluated for one-step forecasting of the monthly car sales dataset is listed below.

```
# evaluate convlstm for monthly car sales dataset
from math import sqrt
from numpy import array
from numpy import mean
from numpy import std
from pandas import DataFrame
```

```python
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import ConvLSTM2D
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, interval):
  return [data[i] - data[i - interval] for i in range(interval, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_seq, n_steps, n_filters, n_kernel, n_nodes, n_epochs, n_batch = config
  n_input = n_seq * n_steps
  # prepare data
  data = series_to_supervised(train, n_input)
  train_x, train_y = data[:, :-1], data[:, -1]
  train_x = train_x.reshape((train_x.shape[0], n_seq, 1, n_steps, 1))
  # define model
  model = Sequential()
  model.add(ConvLSTM2D(filters=n_filters, kernel_size=(1,n_kernel), activation='relu',
      input_shape=(n_seq, 1, n_steps, 1)))
  model.add(Flatten())
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
```

```python
  return model

# forecast with a pre-fit model
def model_predict(model, history, config):
  # unpack config
  n_seq, n_steps, _, _, _, _, _ = config
  n_input = n_seq * n_steps
  # prepare data
  x_input = array(history[-n_input:]).reshape((1, n_seq, 1, n_steps, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# repeat evaluation of a config
def repeat_evaluate(data, config, n_test, n_repeats=30):
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  return scores

# summarize model performance
def summarize_scores(name, scores):
  # print a summary
  scores_m, score_std = mean(scores), std(scores)
  print('%s: %.3f RMSE (+/- %.3f)' % (name, scores_m, score_std))
  # box and whisker plot
  pyplot.boxplot(scores)
  pyplot.show()

series = read_csv('monthly-car-sales.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# define config
config = [3, 12, 256, 3, 200, 200, 100]
```

```
# grid search
scores = repeat_evaluate(data, config, n_test)
# summarize scores
summarize_scores('convlstm', scores)
```

Listing 14.58: Example of a ConvLSTM model for forecasting monthly car sales.

Running the example prints the RMSE for each repeated evaluation of the model. The final averaged RMSE is reported at the end of about 1,660, which is lower than the naive model, but still higher than a SARIMA model. It is a result that is perhaps on par with the CNN-LSTM model. The standard deviation of this score is also very large, suggesting that the chosen configuration may not be as stable as the standalone CNN model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
 > 1653.084
 > 1650.430
 > 1291.353
 > 1558.616
 > 1653.231

convlstm: 1660.840 RMSE (+/- 248.826)
```

Listing 14.59: Example output from a ConvLSTM model for forecasting monthly car sales.

A box and whisker plot is also created, summarizing the distribution of RMSE scores.
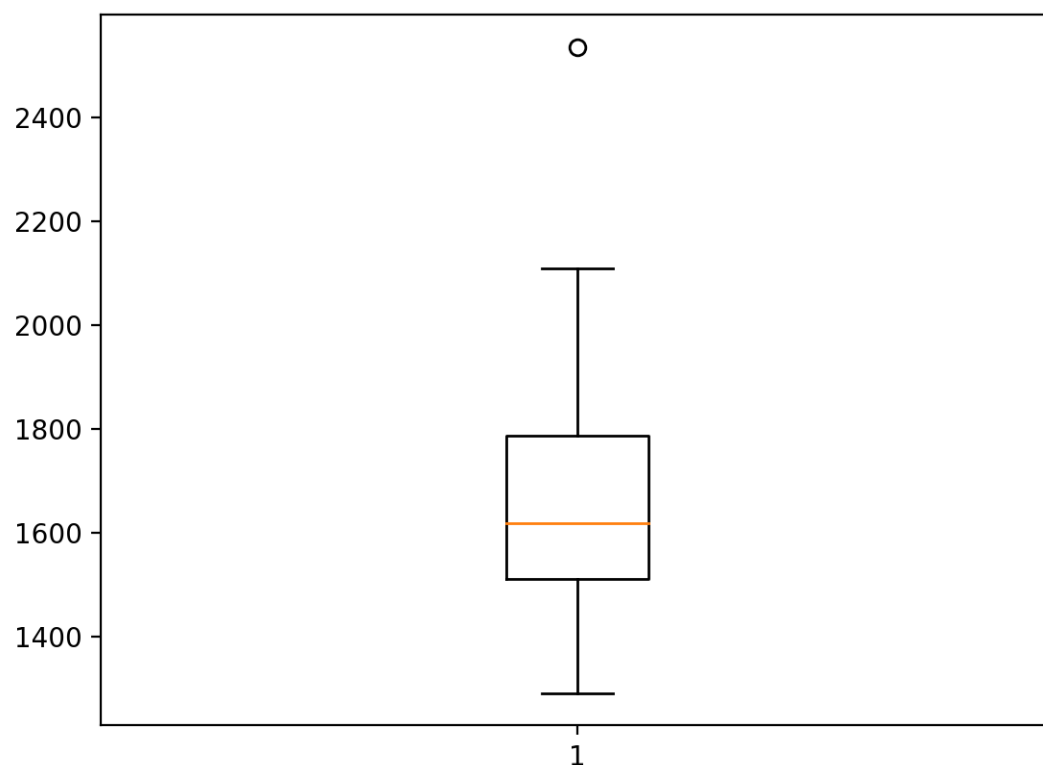
Figure 14.5: Box and Whisker Plot of ConvLSTM RMSE Forecasting Car Sales.

14.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Preparation**. Explore whether data preparation, such as normalization, standardization, and/or differencing can list the performance of any of the models.

- **Grid Search Hyperparameters**. Implement a grid search of the hyperparameters for one model to see if you can further lift performance.

- **Learning Curve Diagnostics**. Create a single fit of one model and review the learning curves on train and validation splits of the dataset, then use the diagnostics of the learning curves to further tune the model hyperparameters in order to improve model performance.

- **History Size**. Explore different amounts of historical data (lag inputs) for one model to see if you can further improve model performance

- **Reduce Variance of Final Model**. Explore one or more strategies to reduce the variance for one of the neural network models.

- **Update During Walk-Forward**. Explore whether re-fitting or updating a neural network model as part of walk-forward validation can further improve model performance.

- **More Parameterization**. Explore adding further model parameterization for one model, such as the use of additional layers.

If you explore any of these extensions, I'd love to know.

14.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- `pandas.DataFrame.shift` API.
  http://pandas-docs.github.io/pandas-docs-travis/generated/pandas.DataFrame.shift.html

- `matplotlib.pyplot.boxplot` API.
  https://matplotlib.org/api/_as_gen/matplotlib.pyplot.boxplot.html

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

14.9 Summary

In this tutorial, you discovered how to develop a suite of deep learning models for univariate time series forecasting. Specifically, you learned:

- How to develop a robust test harness using walk-forward validation for evaluating the performance of neural network models.

- How to develop and evaluate simple Multilayer Perceptron and convolutional neural networks for time series forecasting.

- How to develop and evaluate LSTMs, CNN-LSTMs, and ConvLSTM neural network models for time series forecasting.

14.9.1 Next

In the next lesson, you will discover how to develop a framework to grid search deep learning models for univariate time series forecasting problems.

Chapter 15

How to Grid Search Deep Learning Models for Univariate Forecasting

Grid searching is generally not an operation that we can perform with deep learning methods. This is because deep learning methods often require large amounts of data and large models, together resulting in models that take hours, days, or weeks to train. In those cases where the datasets are smaller, such as univariate time series, it may be possible to use a grid search to tune the hyperparameters of a deep learning model. In this tutorial, you will discover how to develop a framework to grid search hyperparameters for deep learning models. After completing this tutorial, you will know:

- How to develop a generic grid searching framework for tuning model hyperparameters.

- How to grid search hyperparameters for a Multilayer Perceptron model on the airline passengers univariate time series forecasting problem.

- How to adapt the framework to grid search hyperparameters for convolutional and long short-term memory neural networks.

Let's get started.

15.1   Tutorial Overview

This tutorial is divided into five parts; they are:

1. Time Series Problem

2. Grid Search Framework

3. Multilayer Perceptron Model

4. Convolutional Neural Network Model

5. Long Short-Term Memory Network Model

15.2 Time Series Problem

In this tutorial we will focus on one dataset and use it as the context to demonstrate the development of a grid searching framework for range of deep learning models for univariate time series forecasting. We will use the *monthly airline passenger* dataset as this context as it includes the complexity of both trend and seasonal elements. The *monthly airline passenger* dataset summarizes the monthly total number of international passengers in thousands on for an airline from 1949 to 1960. Download the dataset directly from here:

- monthly-airline-passengers.csv [1]

Save the file with the filename `monthly-airline-passengers.csv` in your current working directory. We can load this dataset as a Pandas `DataFrame` using the function `read_csv()`.

```
# load
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
```

Listing 15.1: Load the dataset.

Once loaded, we can summarize the shape of the dataset in order to determine the number of observations.

```
# summarize shape
print(series.shape)
```

Listing 15.2: Summarize the shape of the dataset.

We can then create a line plot of the series to get an idea of the structure of the series.

```
# plot
pyplot.plot(series)
pyplot.show()
```

Listing 15.3: Create a line plot of the dataset.

We can tie all of this together; the complete example is listed below.

```
# load and plot monthly airline passengers dataset
from pandas import read_csv
from matplotlib import pyplot
# load
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
# summarize shape
print(series.shape)
# plot
pyplot.plot(series)
pyplot.xticks([])
pyplot.show()
```

Listing 15.4: Example of loading and plotting the monthly airline passengers dataset.

Running the example first prints the shape of the dataset.

```
(144, 1)
```

Listing 15.5: Example output from loading and plotting the monthly airline passengers dataset.

---

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/airline-passengers.csv

The dataset is monthly and has 12 years, or 144 observations. In our testing, we will use the last year, or 12 observations, as the test set. A line plot is created. The dataset has an obvious trend and seasonal component. The period of the seasonal component is 12 months.
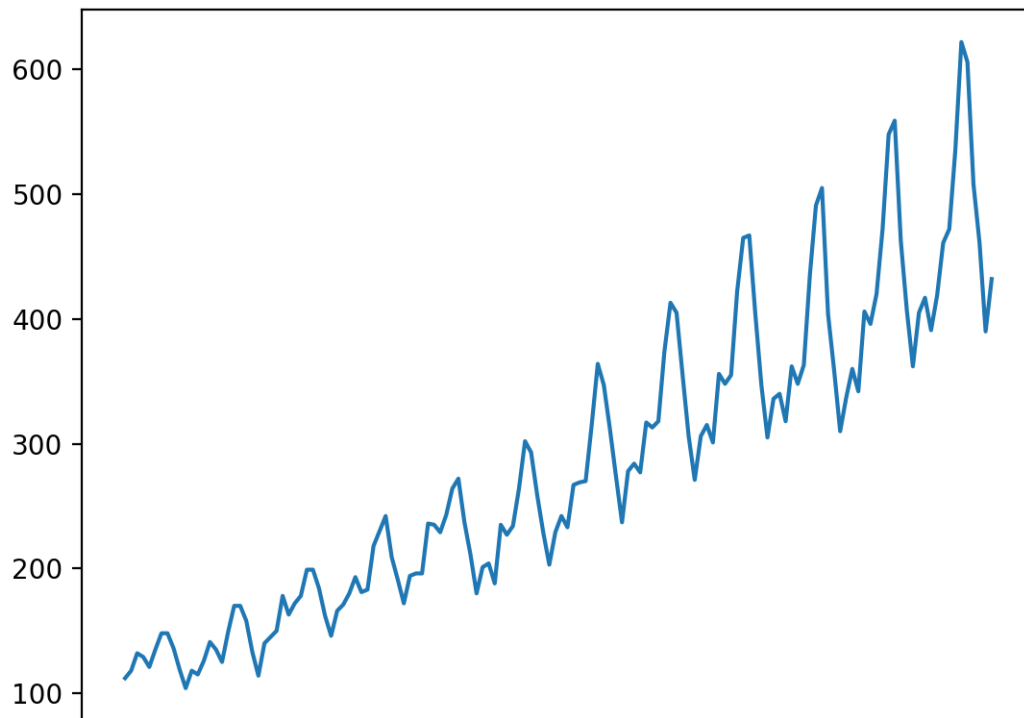

Figure 15.1: Line Plot of Monthly International Airline Passengers.

In this tutorial, we will introduce the tools for grid searching, but we will not optimize the model hyperparameters for this problem. Instead, we will demonstrate how to grid search the deep learning model hyperparameters generally and find models with some skill compared to a naive model. From prior experiments, a naive model can achieve a root mean squared error, or RMSE, of 50.70 (remember the units are thousands of passengers) by persisting the value from 12 months ago (relative index -12). The performance of this naive model provides a bound on a model that is considered skillful for this problem. Any model that achieves a predictive performance of lower than 50.70 on the last 12 months has skill.

It should be noted that a tuned ETS model can achieve an RMSE of 17.09 and a tuned SARIMA can achieve an RMSE of 13.89. These provide a lower bound on the expectations of a well-tuned deep learning model for this problem. Now that we have defined our problem and expectations of model skill, we can look at defining the grid search test harness.

15.3 Develop a Grid Search Framework

In this section, we will develop a grid search test harness that can be used to evaluate a range of hyperparameters for different neural network models, such as MLPs, CNNs, and LSTMs. This section is divided into the following parts:

1. Train-Test Split

2. Series as Supervised Learning

3. Walk-Forward Validation

4. Repeat Evaluation

5. Summarize Performance

6. Worked Example

Note, much of this framework was presented already in Chapter 14. Some elements are duplicated here given the changes needed to adapt it for grid searching model hyperparameters.

15.3.1 Train-Test Split

The first step is to split the loaded series into train and test sets. We will use the first 11 years (132 observations) for training and the last 12 for the test set. The `train_test_split()` function below will split the series taking the raw observations and the number of observations to use in the test set as arguments.

```
# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

Listing 15.6: Example of a function to split the dataset into train and test sets.

15.3.2 Series as Supervised Learning

Next, we need to be able to frame the univariate series of observations as a supervised learning problem so that we can train neural network models. A supervised learning framing of a series means that the data needs to be split into multiple examples that the model learns from and generalizes across. Each sample must have both an input component and an output component. The input component will be some number of prior observations, such as three years, or 36 time steps.

The output component will be the total sales in the next month because we are interested in developing a model to make one-step forecasts. We can implement this using the `shift()` function on the Pandas `DataFrame`. It allows us to shift a column down (forward in time) or back (backward in time). We can take the series as a column of data, then create multiple copies of the column, shifted forward or backward in time in order to create the samples with the input and output elements we require. When a series is shifted down, `NaN` values are introduced because we don't have values beyond the start of the series.

```
(t)
1
2
3
4
```

Listing 15.7: Example of a time series as a column.

This column can be shifted and inserted as a column beforehand:

```
(t-1),    (t)
Nan,      1
1,        2
2,        3
3,        4
4,        NaN
```

Listing 15.8: Example of an added columns with the shifted time series.

We can see that on the second row, the value 1 is provided as input as an observation at the prior time step, and 2 is the next value in the series that can be predicted, or learned by the model to be predicted when 1 is presented as input. Rows with `NaN` values can be removed. The `series_to_supervised()` function below implements this behavior, allowing you to specify the number of lag observations to use in the input and the number to use in the output for each sample. It will also remove rows that have `NaN` values as they cannot be used to train or test a model.

```python
# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values
```

Listing 15.9: Example of a function to transform a time series into samples.

Note, this is a more generic way of transforming a time series dataset into samples than the specialized methods presented in Chapters 7, 8, and 9.

15.3.3   Walk-Forward Validation

Time series forecasting models can be evaluated on a test set using walk-forward validation. Walk-forward validation is an approach where the model makes a forecast for each observation in the test dataset one at a time. After each forecast is made for a time step in the test dataset, the true observation for the forecast is added to the test dataset and made available to

the model. Simpler models can be refit with the observation prior to making the subsequent prediction. More complex models, such as neural networks, are not refit given the much greater computational cost. Nevertheless, the true observation for the time step can then be used as part of the input for making the prediction on the next time step.

First, the dataset is split into train and test sets. We will call the `train_test_split()` function to perform this split and pass in the pre-specified number of observations to use as the test data. A model will be fit once on the training dataset for a given configuration. We will define a generic `model_fit()` function to perform this operation that can be filled in for the given type of neural network that we may be interested in later. The function takes the training dataset and the model configuration and returns the fit model ready for making predictions.

```python
# fit a model
def model_fit(train, config):
  return None
```

Listing 15.10: Example of a dummy function for fitting a model.

Each time step of the test dataset is enumerated. A prediction is made using the fit model. Again, we will define a generic function named `model_predict()` that takes the fit model, the history, and the model configuration and makes a single one-step prediction.

```python
# forecast with a pre-fit model
def model_predict(model, history, config):
  return 0.0
```

Listing 15.11: Example of a dummy function for making a prediction with a fit model.

The prediction is added to a list of predictions and the true observation from the test set is added to a list of observations that was seeded with all observations from the training dataset. This list is built up during each step in the walk-forward validation, allowing the model to make a one-step prediction using the most recent history. All of the predictions can then be compared to the true values in the test set and an error measure calculated. We will calculate the root mean squared error, or RMSE, between predictions and the true values.

RMSE is calculated as the square root of the average of the squared differences between the forecasts and the actual values. The `measure_rmse()` implements this below using the `mean_squared_error()` scikit-learn function to first calculate the mean squared error, or MSE, before calculating the square root.

```python
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))
```

Listing 15.12: Example of a function for calculating the error for a forecast.

The complete `walk_forward_validation()` function that ties all of this together is listed below. It takes the dataset, the number of observations to use as the test set, and the configuration for the model, and returns the RMSE for the model performance on the test set.

```python
# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
```

```
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error
```

Listing 15.13: Example of a function for the walk-forward evaluation of a deep learning model.

15.3.4   Repeat Evaluation

Neural network models are stochastic. This means that, given the same model configuration and the same training dataset, a different internal set of weights will result each time the model is trained that will, in turn, have a different performance. This is a benefit, allowing the model to be adaptive and find high performing configurations to complex problems. It is also a problem when evaluating the performance of a model and in choosing a final model to use to make predictions.

To address model evaluation, we will evaluate a model configuration multiple times via walk-forward validation and report the error as the average error across each evaluation. This is not always possible for large neural networks and may only make sense for small networks that can be fit in minutes or hours. The repeat_evaluate() function below implements this and allows the number of repeats to be specified as an optional parameter that defaults to 10 and returns the mean RMSE score from all repeats.

```
# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)
```

Listing 15.14: Example of a function for the repeated evaluation of a model.

15.3.5   Grid Search

We now have all the pieces of the framework. All that is left is a function to drive the search. We can define a grid_search() function that takes the dataset, a list of configurations to search, and the number of observations to use as the test set and perform the search. Once mean scores

are calculated for each config, the list of configurations is sorted in ascending order so that the best scores are listed first. The complete function is listed below.

```
# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [score_model(data, n_test, cfg) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores
```

Listing 15.15: Example of a function for coordinating the grid search of model hyperparameters.

15.3.6 Worked Example

Now that we have defined the elements of the test harness, we can tie them all together and define a simple persistence model. We do not need to fit a model so the `model_fit()` function will be implemented to simply return `None`.

```
# fit a model
def model_fit(train, config):
  return None
```

Listing 15.16: Example of a dummy function for fitting a model.

We will use the config to define a list of index offsets in the prior observations relative to the time to be forecasted that will be used as the prediction. For example, 12 will use the observation 12 months ago (-12) relative to the time to be forecasted.

```
# define config
cfg_list = [1, 6, 12, 24, 36]
```

Listing 15.17: Example of a set of configurations to search.

The `model_predict()` function can be implemented to use this configuration to persist the value at the negative relative offset.

```
# forecast with a pre-fit model
def model_predict(model, history, offset):
  history[-offset]
```

Listing 15.18: Example of a function for making persistence forecasts.

The complete example of using the framework with a simple persistence model is listed below.

```
# grid search persistence models for monthly airline passengers dataset
from math import sqrt
from numpy import mean
from pandas import read_csv
from sklearn.metrics import mean_squared_error

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

```python
# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# fit a model
def model_fit(train, config):
  return None

# forecast with a pre-fit model
def model_predict(model, history, offset):
  return history[-offset]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
```

```
# data split
n_test = 12
# model configs
cfg_list = [1, 6, 12, 24, 36]
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 10 configs
for cfg, error in scores[:10]:
  print(cfg, error)
```

Listing 15.19: Example of demonstrating the grid search framework for evaluating a persistence model on the airline passengers dataset.

Running the example prints the RMSE of the model evaluated using walk-forward validation on the final 12 months of data. Each model configuration is evaluated 10 times, although, because the model has no stochastic element, the score is the same each time. At the end of the run, the configurations and RMSE for the top three performing model configurations are reported. We can see, as we might have expected, that persisting the value from one year ago (relative offset -12) resulted in the best performance for the persistence model.

```
...
 > 110.274
 > 110.274
 > 110.274
> Model[36] 110.274
done

12 50.708316214732804
1 53.1515129919491
24 97.10990337413241
36 110.27352356753639
6 126.73495965991387
```

Listing 15.20: Example output from demonstrating the grid search framework for evaluating a persistence model on the airline passengers dataset.

Now that we have a robust test harness for grid searching model hyperparameters, we can use it to evaluate a suite of neural network models.

15.4 Multilayer Perceptron Model

In this section we will grid search hyperparameters for an MLPs for univariate time series forecasting. For more details on modeling a univariate time series with an MLP, see Chapter 7. There are many aspects of the MLP that we may wish to tune. We will define a very simple model with one hidden layer and define five hyperparameters to tune. They are:

- n_input: The number of prior inputs to use as input for the model (e.g. 12 months).

- n_nodes: The number of nodes to use in the hidden layer (e.g. 50).

- n_epochs: The number of training epochs (e.g. 1000).

- **n_batch**: The number of samples to include in each mini-batch (e.g. 32).

- **n_diff**: The difference order (e.g. 0 or 12).

Modern neural networks can handle raw data with little pre-processing, such as scaling and differencing. Nevertheless, when it comes to time series data, sometimes differencing the series can make a problem easier to model. Recall that differencing is the transform of the data such that a value of a prior observation is subtracted from the current observation, removing trend or seasonality structure. We will add support for differencing to the grid search test harness, just in case it adds value to your specific problem. It does add value for the internal airline passengers dataset. The `difference()` function below will calculate the difference of a given order for the dataset.

```
# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]
```

Listing 15.21: Example of a function for differencing a dataset.

Differencing will be optional, where an order of 0 suggests no differencing, whereas an order 1 or order 12 will require that the data be differenced prior to fitting the model and that the predictions of the model will need the differencing reversed prior to returning the forecast. We can now define the elements required to fit the MLP model in the test harness. First, we must unpack the list of hyperparameters.

```
# unpack config
n_input, n_nodes, n_epochs, n_batch, n_diff = config
```

Listing 15.22: Example of unpacking MLP hyperparameters.

Next, we must prepare the data, including the differencing, transforming the data to a supervised format and separating out the input and output aspects of the data samples.

```
# prepare data
if n_diff > 0:
  train = difference(train, n_diff)
# transform series into supervised format
data = series_to_supervised(train, n_input)
# separate inputs and outputs
train_x, train_y = data[:, :-1], data[:, -1]
```

Listing 15.23: Example of preparing data for fitting the MLP model.

We can now define and fit the model with the provided configuration.

```
# define model
model = Sequential()
model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
# fit model
model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
```

Listing 15.24: Example of defining an MLP model.

The complete implementation of the `model_fit()` function is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 15.25: Example of a function for fitting an MLP model for a given configuration.

The five chosen hyperparameters are by no means the only or best hyperparameters of the model to tune. You may modify the function to tune other parameters, such as the addition and size of more hidden layers and much more. Once the model is fit, we can use it to make forecasts. If the data was differenced, the difference must be inverted for the prediction of the model. This involves adding the value at the relative offset from the history back to the value predicted by the model.

```
# invert difference
correction = 0.0
if n_diff > 0:
  correction = history[-n_diff]
...
# correct forecast if it was differenced
return correction + yhat[0]
```

Listing 15.26: Example of inverting any differencing performed.

It also means that the history must be differenced so that the input data used to make the prediction has the expected form.

```
# calculate difference
history = difference(history, n_diff)
```

Listing 15.27: Example of differencing the history prior to making a prediction.

Once prepared, we can use the history data to create a single sample as input to the model for making a one-step prediction. The shape of one sample must be [1, n_input] where n_input is the chosen number of lag observations to use.

```
# shape input for model
x_input = array(history[-n_input:]).reshape((1, n_input))
```

Listing 15.28: Example of preparing one sample ready for making a forecast.

Finally, a prediction can be made.

```
# make forecast
yhat = model.predict(x_input, verbose=0)
```

Listing 15.29: Example of making a forecast with a single sample of data.

The complete implementation of the `model_predict()` function is listed below. Next, we must define the range of values to try for each hyperparameter. We can define a `model_configs()` function that creates a list of the different combinations of parameters to try. We will define a small subset of configurations to try as an example, including a differencing of 12 months, which we expect will be required. You are encouraged to experiment with standalone models, review learning curve diagnostic plots, and use information about the domain to set ranges of values of the hyperparameters to grid search.

You are also encouraged to repeat the grid search to narrow in on ranges of values that appear to show better performance. An implementation of the `model_configs()` function is listed below.

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [50, 100]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
  for i in n_input:
    for j in n_nodes:
      for k in n_epochs:
        for l in n_batch:
          for m in n_diff:
            cfg = [i, j, k, l, m]
            configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs
```

Listing 15.30: Example of a function for preparing a list of model configurations to evaluate.

We now have all of the pieces needed to grid search MLP models for a univariate time series forecasting problem. The complete example is listed below.

```
# grid search mlps for monthly airline passengers dataset
from math import sqrt
from numpy import array
from numpy import mean
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]
```

```python
# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_in=n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # define model
  model = Sequential()
  model.add(Dense(n_nodes, activation='relu', input_dim=n_input))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  # shape input for model
  x_input = array(history[-n_input:]).reshape((1, n_input))
```

```python
  # make forecast
  yhat = model.predict(x_input, verbose=0)
  # correct forecast if it was differenced
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [50, 100]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
```

```
  for i in n_input:
    for j in n_nodes:
      for k in n_epochs:
        for l in n_batch:
          for m in n_diff:
            cfg = [i, j, k, l, m]
            configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# model configs
cfg_list = model_configs()
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 3 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 15.31: Example of demonstrating the grid search framework for evaluating a MLP model configurations on the airline passengers dataset.

Running the example, we can see that there are a total of eight configurations to be evaluated by the framework. Each config will be evaluated 10 times; that means 10 models will be created and evaluated using walk-forward validation to calculate an RMSE score before an average of those 10 scores is reported and used to score the configuration. The scores are then sorted and the top 3 configurations with the lowest RMSE are reported at the end. A skillful model configuration was found as compared to a naive model that reported an RMSE of 50.70. We can see that the best RMSE of 18.98 was achieved with a configuration of `[12, 100, 100, 1, 12]`, which we know can be interpreted as:

- n_input: 12

- n_nodes: 100

- n_epochs: 100

- n_batch: 1

- n_diff: 12

A truncated example output of the grid search is listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Total configs: 8
 > 20.707
 > 29.111
 > 17.499
 > 18.918
 > 28.817
...
 > 21.015
 > 20.208
 > 18.503
> Model[[12, 100, 100, 150, 12]] 19.674
done

[12, 100, 100, 1, 12] 18.982720013625606
[12, 50, 100, 150, 12] 19.33004059448595
[12, 100, 100, 1, 0] 19.5389405532858
```

Listing 15.32: Example output from demonstrating the grid search framework for evaluating a MLP model configurations on the airline passengers dataset.

15.5 Convolutional Neural Network Model

We can now adapt the framework to grid search CNN models. For more details on modeling a univariate time series with a CNN, see Chapter 8. Much the same set of hyperparameters can be searched as with the MLP model, except the number of nodes in the hidden layer can be replaced by the number of filter maps and kernel size in the convolutional layers. The chosen set of hyperparameters to grid search in the CNN model are as follows:

- `n_input`: The number of prior inputs to use as input for the model (e.g. 12 months).

- `n_filters`: The number of filter maps in the convolutional layer (e.g. 32).

- `n_kernel`: The kernel size in the convolutional layer (e.g. 3).

- `n_epochs`: The number of training epochs (e.g. 1000).

- `n_batch`: The number of samples to include in each mini-batch (e.g. 32).

- `n_diff`: The difference order (e.g. 0 or 12).

Some additional hyperparameters that you may wish to investigate are the use of two convolutional layers before a pooling layer, the repetition of the convolutional and pooling layer pattern, the use of dropout, and more. We will define a very simple CNN model with one convolutional layer and one max pooling layer.

```
# define model
model = Sequential()
model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
    input_shape=(n_input, n_features)))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
```

```
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
```

Listing 15.33: Example of defining a CNN model.

The data must be prepared in much the same way as for the MLP. Unlike the MLP that expects the input data to have the shape [samples, features], the 1D CNN model expects the data to have the shape [samples, timesteps, features] where features maps onto channels and in this case 1 for the one variable we measure each month.

```
# reshape input data into [samples, timesteps, features]
n_features = 1
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
```

Listing 15.34: Example of reshaping data for the CNN model.

The complete implementation of the model_fit() function is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, n_features)))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 15.35: Example of a function for fitting a CNN model for a given configuration.

Making a prediction with a fit CNN model is very much like making a prediction with a fit MLP. Again, the only difference is that the one sample worth of input data must have a three-dimensional shape.

```
x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

Listing 15.36: Example of reshaping one sample for making a forecast.

The complete implementation of the model_predict() function is listed below.

```
# forecast with the fit model
def model_predict(model, history, config):
```

```
  # unpack config
  n_input, _, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]
```

Listing 15.37: Example of a function for making a forecast with a fit CNN model.

Finally, we can define a list of configurations for the model to evaluate. As before, we can do this by defining lists of hyperparameter values to try that are combined into a list. We will try a small number of configurations to ensure the example executes in a reasonable amount of time. The complete `model_configs()` function is listed below.

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_filters = [64]
  n_kernels = [3, 5]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
  for a in n_input:
    for b in n_filters:
      for c in n_kernels:
        for d in n_epochs:
          for e in n_batch:
            for f in n_diff:
              cfg = [a,b,c,d,e,f]
              configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs
```

Listing 15.38: Example of a function for preparing a list of model configurations to evaluate.

We now have all of the elements needed to grid search the hyperparameters of a convolutional neural network for univariate time series forecasting. The complete example is listed below.

```
# grid search cnn for monthly airline passengers dataset
from math import sqrt
from numpy import array
from numpy import mean
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
```

```python
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_filters, n_kernel, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_in=n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=n_kernel, activation='relu',
      input_shape=(n_input, n_features)))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

```python
# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores
```

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_filters = [64]
  n_kernels = [3, 5]
  n_epochs = [100]
  n_batch = [1, 150]
  n_diff = [0, 12]
  # create configs
  configs = list()
  for a in n_input:
    for b in n_filters:
      for c in n_kernels:
        for d in n_epochs:
          for e in n_batch:
            for f in n_diff:
              cfg = [a,b,c,d,e,f]
              configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# model configs
cfg_list = model_configs()
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 10 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 15.39: Example of demonstrating the grid search framework for evaluating a CNN model configurations on the airline passengers dataset.

Running the example, we can see that only eight distinct configurations are evaluated. We can see that a configuration of [12, 64, 5, 100, 1, 12] achieved an RMSE of 18.89, which is skillful as compared to a naive forecast model that achieved 50.70. We can unpack this configuration as:

- n_input: 12

- n_filters: 64

- n_kernel: 5

- n_epochs: 100

- n_batch: 1

- n_diff: 12

A truncated example output of the grid search is listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Total configs: 8
 > 23.372
 > 28.317
 > 31.070
...
 > 20.923
 > 18.700
 > 18.210
> Model[[12, 64, 5, 100, 150, 12]] 19.152
done

[12, 64, 5, 100, 1, 12] 18.89593462072732
[12, 64, 5, 100, 150, 12] 19.152486150334234
[12, 64, 3, 100, 150, 12] 19.44680151564605
```

Listing 15.40: Example output from demonstrating the grid search framework for evaluating a CNN model configurations on the airline passengers dataset.

15.6 Long Short-Term Memory Network Model

We can now adopt the framework for grid searching the hyperparameters of an LSTM model. For more details on modeling a univariate time series with an LSTM network, see Chapter 9. The hyperparameters for the LSTM model will be the same five as the MLP; they are:

- `n_input`: The number of prior inputs to use as input for the model (e.g. 12 months).

- `n_nodes`: The number of nodes to use in the hidden layer (e.g. 50).

- `n_epochs`: The number of training epochs (e.g. 1000).

- `n_batch`: The number of samples to include in each mini-batch (e.g. 32).

- `n_diff`: The difference order (e.g. 0 or 12).

We will define a simple LSTM model with a single hidden LSTM layer and the number of nodes specifying the number of units in this layer.

```
# define model
model = Sequential()
model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, n_features)))
model.add(Dense(n_nodes, activation='relu'))
model.add(Dense(1))
model.compile(loss='mse', optimizer='adam')
# fit model
model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
```

Listing 15.41: Example of defining an LSTM model.

It may be interesting to explore tuning additional configurations such as the use of a bidirectional input layer, stacked LSTM layers, and even hybrid models with CNN or ConvLSTM input models. As with the CNN model, the LSTM model expects input data to have a three-dimensional shape for the samples, time steps, and features.

```
# reshape input data into [samples, timesteps, features]
n_features = 1
train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
```

Listing 15.42: Example of reshaping training data for the LSTM model.

The complete implementation of the `model_fit()` function is listed below.

```
# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, n_features)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model
```

Listing 15.43: Example of a function for fitting an LSTM model.

Also like the CNN, the single input sample used to make a prediction must also be reshaped into the expected three-dimensional structure.

```
# reshape sample into [samples, timesteps, features]
x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

Listing 15.44: Example of reshaping a single sample for making a prediction.

The complete `model_predict()` function is listed below.

```
# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
```

```
  # reshape sample into [samples, timesteps, features]
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]
```

Listing 15.45: Example of a function for making a forecast with a fit LSTM model.

We can now define the function used to create the list of model configurations to evaluate. The LSTM model is quite a bit slower to train than MLP and CNN models; as such, you may want to evaluate fewer configurations per run. We will define a very simple set of two configurations to explore: stochastic and batch gradient descent.

```
# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [100]
  n_epochs = [50]
  n_batch = [1, 150]
  n_diff = [12]
  # create configs
  configs = list()
  for i in n_input:
    for j in n_nodes:
      for k in n_epochs:
        for l in n_batch:
          for m in n_diff:
            cfg = [i, j, k, l, m]
            configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs
```

Listing 15.46: Example of a function for defining model configurations to evaluate.

We now have everything we need to grid search hyperparameters for the LSTM model for univariate time series forecasting. The complete example is listed below.

```
# grid search lstm for monthly airline passengers dataset
from math import sqrt
from numpy import array
from numpy import mean
from pandas import DataFrame
from pandas import concat
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

# split a univariate dataset into train/test sets
def train_test_split(data, n_test):
  return data[:-n_test], data[-n_test:]

# transform list into supervised learning format
def series_to_supervised(data, n_in, n_out=1):
  df = DataFrame(data)
```

```
  cols = list()
  # input sequence (t-n, ... t-1)
  for i in range(n_in, 0, -1):
    cols.append(df.shift(i))
  # forecast sequence (t, t+1, ... t+n)
  for i in range(0, n_out):
    cols.append(df.shift(-i))
  # put it all together
  agg = concat(cols, axis=1)
  # drop rows with NaN values
  agg.dropna(inplace=True)
  return agg.values

# root mean squared error or rmse
def measure_rmse(actual, predicted):
  return sqrt(mean_squared_error(actual, predicted))

# difference dataset
def difference(data, order):
  return [data[i] - data[i - order] for i in range(order, len(data))]

# fit a model
def model_fit(train, config):
  # unpack config
  n_input, n_nodes, n_epochs, n_batch, n_diff = config
  # prepare data
  if n_diff > 0:
    train = difference(train, n_diff)
  # transform series into supervised format
  data = series_to_supervised(train, n_in=n_input)
  # separate inputs and outputs
  train_x, train_y = data[:, :-1], data[:, -1]
  # reshape input data into [samples, timesteps, features]
  n_features = 1
  train_x = train_x.reshape((train_x.shape[0], train_x.shape[1], n_features))
  # define model
  model = Sequential()
  model.add(LSTM(n_nodes, activation='relu', input_shape=(n_input, n_features)))
  model.add(Dense(n_nodes, activation='relu'))
  model.add(Dense(1))
  model.compile(loss='mse', optimizer='adam')
  # fit model
  model.fit(train_x, train_y, epochs=n_epochs, batch_size=n_batch, verbose=0)
  return model

# forecast with the fit model
def model_predict(model, history, config):
  # unpack config
  n_input, _, _, _, n_diff = config
  # prepare data
  correction = 0.0
  if n_diff > 0:
    correction = history[-n_diff]
    history = difference(history, n_diff)
  # reshape sample into [samples, timesteps, features]
  x_input = array(history[-n_input:]).reshape((1, n_input, 1))
```

```python
  # forecast
  yhat = model.predict(x_input, verbose=0)
  return correction + yhat[0]

# walk-forward validation for univariate data
def walk_forward_validation(data, n_test, cfg):
  predictions = list()
  # split dataset
  train, test = train_test_split(data, n_test)
  # fit model
  model = model_fit(train, cfg)
  # seed history with training dataset
  history = [x for x in train]
  # step over each time-step in the test set
  for i in range(len(test)):
    # fit model and make forecast for history
    yhat = model_predict(model, history, cfg)
    # store forecast in list of predictions
    predictions.append(yhat)
    # add actual observation to history for the next loop
    history.append(test[i])
  # estimate prediction error
  error = measure_rmse(test, predictions)
  print(' > %.3f' % error)
  return error

# score a model, return None on failure
def repeat_evaluate(data, config, n_test, n_repeats=10):
  # convert config to a key
  key = str(config)
  # fit and evaluate the model n times
  scores = [walk_forward_validation(data, n_test, config) for _ in range(n_repeats)]
  # summarize score
  result = mean(scores)
  print('> Model[%s] %.3f' % (key, result))
  return (key, result)

# grid search configs
def grid_search(data, cfg_list, n_test):
  # evaluate configs
  scores = scores = [repeat_evaluate(data, cfg, n_test) for cfg in cfg_list]
  # sort configs by error, asc
  scores.sort(key=lambda tup: tup[1])
  return scores

# create a list of configs to try
def model_configs():
  # define scope of configs
  n_input = [12]
  n_nodes = [100]
  n_epochs = [50]
  n_batch = [1, 150]
  n_diff = [12]
  # create configs
  configs = list()
  for i in n_input:
```

```
      for j in n_nodes:
        for k in n_epochs:
          for l in n_batch:
            for m in n_diff:
              cfg = [i, j, k, l, m]
              configs.append(cfg)
  print('Total configs: %d' % len(configs))
  return configs

# define dataset
series = read_csv('monthly-airline-passengers.csv', header=0, index_col=0)
data = series.values
# data split
n_test = 12
# model configs
cfg_list = model_configs()
# grid search
scores = grid_search(data, cfg_list, n_test)
print('done')
# list top 10 configs
for cfg, error in scores[:3]:
  print(cfg, error)
```

Listing 15.47: Example of demonstrating the grid search framework for evaluating an LSTM model configurations on the airline passengers dataset.

Running the example, we can see that only two distinct configurations are evaluated. We can see that a configuration of [12, 100, 50, 1, 12] achieved an RMSE of 21.24, which is skillful as compared to a naive forecast model that achieved 50.70. The model requires a lot more tuning and may do much better with a hybrid configuration, such as having a CNN model as input. We can unpack this configuration as:

- n_input: 12

- n_nodes: 100

- n_epochs: 50

- n_batch: 1

- n_diff: 12

A truncated example output of the grid search is listed below.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Total configs: 2
 > 20.488
 > 17.718
 > 21.213
...
 > 22.300
 > 20.311
```

```
 > 21.322
> Model[[12, 100, 50, 150, 12]] 21.260
done

[12, 100, 50, 1, 12] 21.243775750634093
[12, 100, 50, 150, 12] 21.259553398553606
```

Listing 15.48: Example output from demonstrating the grid search framework for evaluating an LSTM model configurations on the airline passengers dataset.

15.7   Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **More Configurations**. Explore a large suite of configurations for one of the models and see if you can find a configuration that results in better performance.

- **Data Scaling**. Update the grid search framework to also support the scaling (normalization and/or standardization) of data both before fitting the model and inverting the transform for predictions.

- **Network Architecture**. Explore the grid searching larger architectural changes for a given model, such as the addition of more hidden layers.

- **New Dataset**. Explore the grid search of a given model in a new univariate time series dataset.

- **Multivariate**. Update the grid search framework to support small multivariate time series datasets, e.g. datasets with multiple input variables.

If you explore any of these extensions, I'd love to know.

15.8   Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

15.9 Summary

In this tutorial, you discovered how to develop a framework to grid search hyperparameters for deep learning models. Specifically, you learned:

- How to develop a generic grid searching framework for tuning model hyperparameters.

- How to grid search hyperparameters for a Multilayer Perceptron model on the airline passengers univariate time series forecasting problem.

- How to adapt the framework to grid search hyperparameters for convolutional and long short-term memory neural networks.

15.9.1 Next

This is the final lesson of this part, the next part will focus how to systematically work through a real-world multivariate multi-step time series problem to forecast household energy usage.

Part V

Multi-step Forecasting

Overview

This part focuses on the real-world problem of multi-step forecasting for household electricity consumption and how to develop naive methods, classical autoregression models, Convolutional and Long Short-Term Memory neural network models for the problem. The tutorials in this part do not seek to demonstrate the best way to solve the problem, instead the dataset provides a context on which each of the specific methods can be demonstrated. As such, the performance of each method on the dataset are not compared directly. After reading the chapters in this part, you will know:

- How to load, summarize and visualize a multivariate time series dataset describing household electricity consumption over many years (Chapter 16).

- How to develop naive forecasting models for forecasting household electricity usage, the results of which can be used as a baseline to determine whether a more sophisticated model has skill (Chapter 17).

- How to diagnose the autocorrelation and develop autoregressive models for forecasting household electricity usage (Chapter 18).

- How to develop a suite of convolutional neural network models for forecasting household electricity usage (Chapter 19).

- How to develop a suite of Long Short-Term Memory neural network models for forecasting household electricity usage (Chapter 20).

Chapter 16

How to Load and Explore Household Energy Usage Data

Given the rise of smart electricity meters and the wide adoption of electricity generation technology like solar panels, there is a wealth of electricity usage data available. This data represents a multivariate time series of power-related variables, that in turn could be used to model and even forecast future electricity consumption. In this tutorial, you will discover a household power consumption dataset for multi-step time series forecasting and how to better understand the raw data using exploratory analysis. This dataset will provided the basis for the remaining tutorials in this part of the book. After completing this tutorial, you will know:

- The household power consumption dataset that describes electricity usage for a single house over four years.

- How to explore and understand the dataset using a suite of line plots for the series data and histogram for the data distributions.

- How to use the new understanding of the problem to consider different framings of the prediction problem, ways the data may be prepared, and modeling methods that may be used.

Let's get started.

16.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Power Consumption Dataset

2. Load Dataset

3. Patterns in Observations Over Time

4. Time Series Data Distributions

5. Ideas on Modeling

Note: The data visualizations in this tutorial are intended to be quick-and-dirty to rapidly learn about the dataset, rather than used as the basis for presentation. As such they are more utilitarian and less aesthetic.

16.2 Household Power Consumption Dataset

The Household Power Consumption dataset is a multivariate time series dataset that describes the electricity consumption for a single household over four years. The data was collected between December 2006 and November 2010 and observations of power consumption within the household were collected every minute. It is a multivariate series comprised of seven variables (besides the date and time); they are:

- `global_active_power`: The total active power consumed by the household (kilowatts).

- `global_reactive_power`: The total reactive power consumed by the household (kilowatts).

- `voltage`: Average voltage (volts).

- `global_intensity`: Average current intensity (amps).

- `sub_metering_1`: Active energy for kitchen (watt-hours of active energy).

- `sub_metering_2`: Active energy for laundry (watt-hours of active energy).

- `sub_metering_3`: Active energy for climate control systems (watt-hours of active energy).

Active and reactive energy refer to the technical details of alternative current. In general terms, the active energy is the real power consumed by the household, whereas the reactive energy is the unused power in the lines. We can see that the dataset provides the active power as well as some division of the active power by main circuit in the house, specifically the kitchen, laundry, and climate control. These are not all the circuits in the household. The remaining watt-hours can be calculated from the active energy by first converting the active energy to watt-hours then subtracting the other sub-metered active energy in watt-hours, as follows:

$$\text{remainder} = (\frac{\text{global\_act\_pwr} \times 1000}{60}) - (\text{sub\_met\_1} + \text{sub\_met\_2} + \text{sub\_met\_3}) \quad (16.1)$$

The dataset seems to have been provided without a seminal reference paper. Nevertheless, this dataset has become a standard for evaluating time series forecasting and machine learning methods for multi-step forecasting, specifically for forecasting active power. Further, it is not clear whether the other features in the dataset may benefit a model in forecasting active power.

16.3 Load Dataset

The dataset is described and has been made freely available on the UCI Machine Learning repository[1]. The dataset can be downloaded as a single 20 megabyte zip file. A direct download link is provided blow:

---

[1] https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption

- household_power_consumption.zip [2]

Download the dataset and unzip it into your current working directory. You will now have the file household_power_consumption.txt that is about 127 megabytes in size and contains all of the observations Inspect the data file. Below are the first five rows of data (and the header) from the raw data file.

```
Date;Time;Global_active_power;Global_reactive_power;Voltage;Global_intensity;...
16/12/2006;17:24:00;4.216;0.418;234.840;18.400;0.000;1.000;17.000
16/12/2006;17:25:00;5.360;0.436;233.630;23.000;0.000;1.000;16.000
16/12/2006;17:26:00;5.374;0.498;233.290;23.000;0.000;2.000;17.000
16/12/2006;17:27:00;5.388;0.502;233.740;23.000;0.000;1.000;17.000
16/12/2006;17:28:00;3.666;0.528;235.680;15.800;0.000;1.000;17.000
...
```

Listing 16.1: Sample of the household power consumption dataset.

We can see that the data columns are separated by semicolons (';'). The data is reported to have one row for each day in the time period. The data does have missing values; for example, we can see 2-3 days worth of missing data around 28/4/2007.

```
...
28/4/2007;00:20:00;0.492;0.208;236.240;2.200;0.000;0.000;0.000
28/4/2007;00:21:00;?;?;?;?;?;?;
28/4/2007;00:22:00;?;?;?;?;?;?;
28/4/2007;00:23:00;?;?;?;?;?;?;
28/4/2007;00:24:00;?;?;?;?;?;?;
...
```

Listing 16.2: Sample of the household power consumption dataset with missing values.

We can start-off by loading the data file as a Pandas DataFrame and summarize the loaded data. We can use the read_csv() function to load the data. It is easy to load the data with this function, but a little tricky to load it correctly. Specifically, we need to do a few custom things:

- Specify the separate between columns as a semicolon (sep=';')

- Specify that line 0 has the names for the columns (header=0)

- Specify that we have lots of RAM to avoid a warning that we are loading the data as an array of objects instead of an array of numbers, because of the '?' values for missing data (low_memory=False).

- Specify that it is okay for Pandas to try to infer the date-time format when parsing dates, which is way faster (infer_datetime_format=True)

- Specify that we would like to parse the date and time columns together as a new column called 'datetime' (parse_dates='datetime':[0,1])

- Specify that we would like our new datetime column to be the index for the DataFrame (index_col=['datetime']).

Putting all of this together, we can now load the data and summarize the loaded shape and first few rows.

```python
# load all data
dataset = read_csv('household_power_consumption.txt', sep=';', header=0, low_memory=False,
    infer_datetime_format=True, parse_dates={'datetime':[0,1]}, index_col=['datetime'])
# summarize
print(dataset.shape)
print(dataset.head())
```

Listing 16.3: Example of loading and summarizing the dataset.

Next, we can mark all missing values indicated with a '?' character with a `NaN` value, which is a float. This will allow us to work with the data as one array of floating point values rather than mixed types, which is less efficient.

```python
# mark all missing values
dataset.replace('?', nan, inplace=True)
```

Listing 16.4: Example of replacing missing values with `NaN`.

Now we can create a new column that contains the remainder of the sub-metering, using the calculation from the previous section.

```python
# add a column for for the remainder of sub metering
values = dataset.values.astype('float32')
dataset['sub_metering_4'] = (values[:,0] * 1000 / 60) - (values[:,4] + values[:,5] +
    values[:,6])
```

Listing 16.5: Example of calculating the remaining sub-metered data.

We can now save the cleaned-up version of the dataset to a new file; in this case we will just change the file extension to `.csv` and save the dataset as `household_power_consumption.csv`.

```python
# save updated dataset
dataset.to_csv('household_power_consumption.csv')
```

Listing 16.6: Example of saving the dataset to a new file.

To confirm that we have not messed-up, we can re-load the dataset and summarize the first five rows.

```python
# load the new file
dataset = read_csv('household_power_consumption.csv', header=None)
print(dataset.head())
```

Listing 16.7: Example of loading the newly saved data file

Tying all of this together, the complete example of loading, cleaning-up, and saving the dataset is listed below.

```python
# load and clean-up power usage data
from numpy import nan
from pandas import read_csv
# load all data
dataset = read_csv('household_power_consumption.txt', sep=';', header=0, low_memory=False,
    infer_datetime_format=True, parse_dates={'datetime':[0,1]}, index_col=['datetime'])
# summarize
print(dataset.shape)
```

```
print(dataset.head())
# mark all missing values
dataset.replace('?', nan, inplace=True)
# add a column for for the remainder of sub metering
values = dataset.values.astype('float32')
dataset['sub_metering_4'] = (values[:,0] * 1000 / 60) - (values[:,4] + values[:,5] +
    values[:,6])
# save updated dataset
dataset.to_csv('household_power_consumption.csv')
# load the new dataset and summarize
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
print(dataset.head())
```

Listing 16.8: Example of loading and preparing the power consumption dataset.

Running the example first loads the raw data and summarizes the shape and first five rows of the loaded data.

```
(2075259, 7)

                    Global_active_power    ...      Sub_metering_3
datetime                                   ...
2006-12-16 17:24:00              4.216     ...                17.0
2006-12-16 17:25:00              5.360     ...                16.0
2006-12-16 17:26:00              5.374     ...                17.0
2006-12-16 17:27:00              5.388     ...                17.0
2006-12-16 17:28:00              3.666     ...                17.0
```

Listing 16.9: Example of the loaded raw dataset.

The dataset is then cleaned up and saved to a new file. We load this new file and again print the first five rows, showing the removal of the date and time columns and addition of the new sub-metered column.

```
                    Global_active_power    ...        sub_metering_4
datetime                                   ...
2006-12-16 17:24:00              4.216     ...             52.266670
2006-12-16 17:25:00              5.360     ...             72.333336
2006-12-16 17:26:00              5.374     ...             70.566666
2006-12-16 17:27:00              5.388     ...             71.800000
2006-12-16 17:28:00              3.666     ...             43.100000
```

Listing 16.10: Example of the prepared dataset.

We can peek inside the new `household_power_consumption.csv` file and check that the missing observations are marked with an empty column, that Pandas will correctly read as `NaN`, for example around row 190,499:

```
...
2007-04-28 00:20:00,0.492,0.208,236.240,2.200,0.000,0.000,0.0,8.2
2007-04-28 00:21:00,,,,,,,,
2007-04-28 00:22:00,,,,,,,,
2007-04-28 00:23:00,,,,,,,,
2007-04-28 00:24:00,,,,,,,,
2007-04-28 00:25:00,,,,,,,,
...
```

Listing 16.11: Sample of the prepared dataset with missing values.

Now that we have a cleaned-up version of the dataset, we can investigate it further using visualizations.

16.4 Patterns in Observations Over Time

The data is a multivariate time series and the best way to understand a time series is to create line plots. We can start off by creating a separate line plot for each of the eight variables. The complete example is listed below.

```
# line plots for power usage dataset
from pandas import read_csv
from matplotlib import pyplot
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# line plot for each variable
pyplot.figure()
for i in range(len(dataset.columns)):
  # create subplot
  pyplot.subplot(len(dataset.columns), 1, i+1)
  # get variable name
  name = dataset.columns[i]
  # plot data
  pyplot.plot(dataset[name])
  # set title
  pyplot.title(name, y=0)
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
pyplot.show()
```

Listing 16.12: Example of creating line plots from the prepared dataset.

Running the example creates a single image with eight subplots, one for each variable. This gives us a really high level of the four years of one minute observations. We can see that something interesting was going on in Sub_metering_3 (environmental control) that may not directly map to hot or cold years. Perhaps new systems were installed.

Interestingly, the contribution of sub_metering_4 seems to decrease with time, or show a downward trend, perhaps matching up with the solid increase in seen towards the end of the series for Sub_metering_3. These observations do reinforce the need to honor the temporal ordering of subsequences of this data when fitting and evaluating any model. We might be able to see the wave of a seasonal effect in the Global_active_power and some other variates. There is some spiky usage that may match up with a specific period, such as weekends.
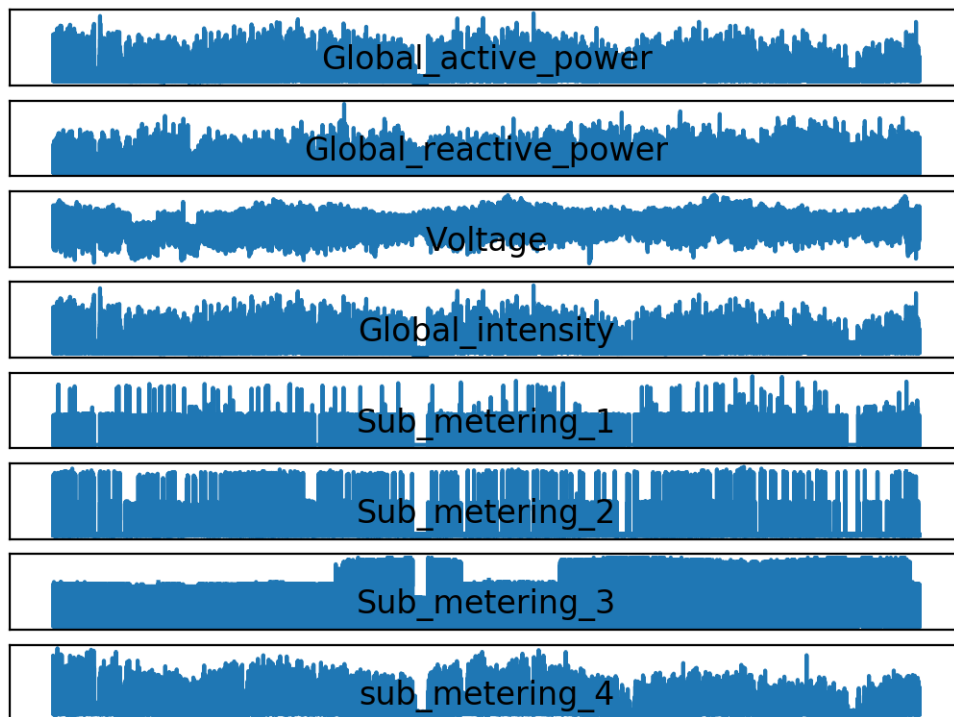
Figure 16.1: Line Plots of Each Variable in the Power Consumption Dataset.

Let's zoom in and focus on the `Global_active_power`, or *active power* for short. We can create a new plot of the active power for each year to see if there are any common patterns across the years. The first year, 2006, has less than one month of data, so will remove it from the plot. The complete example is listed below.

```python
# yearly line plots for power usage dataset
from pandas import read_csv
from matplotlib import pyplot
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# plot active power for each year
years = ['2007', '2008', '2009', '2010']
pyplot.figure()
for i in range(len(years)):
  # prepare subplot
  ax = pyplot.subplot(len(years), 1, i+1)
  # determine the year to plot
  year = years[i]
  # get all observations for the year
  result = dataset[str(year)]
  # plot the active power for the year
  pyplot.plot(result['Global_active_power'])
```

```
  # add a title to the subplot
  pyplot.title(str(year), y=0, loc='left')
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
pyplot.show()
```

Listing 16.13: Example of creating line plots of power consumption per year.

Running the example creates one single image with four line plots, one for each full year (or mostly full years) of data in the dataset. We can see some common gross patterns across the years, such as around Feb-Mar and around Aug-Sept where we see a marked decrease in consumption. We also seem to see a downward trend over the summer months (middle of the year in the northern hemisphere) and perhaps more consumption in the winter months towards the edges of the plots. These may show an annual seasonal pattern in consumption. We can also see a few patches of missing data in at least the first, third, and fourth plots.
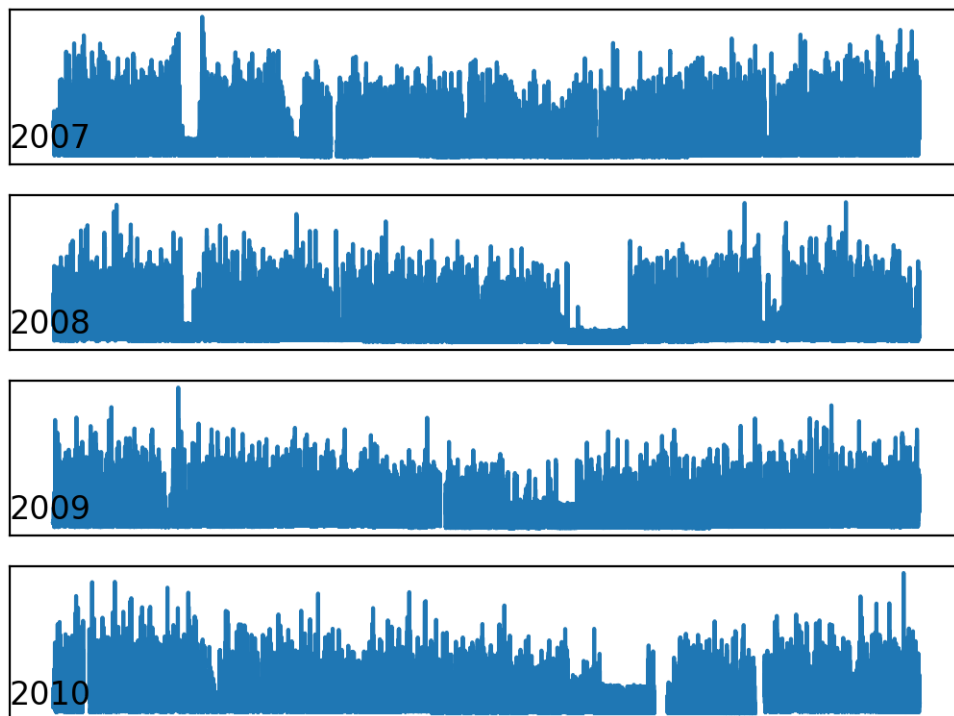

Figure 16.2: Line Plots of Active Power for Most Years.

We can continue to zoom in on consumption and look at active power for each of the 12 months of 2007. This might help tease out gross structures across the months, such as daily and weekly patterns. The complete example is listed below.

```
# monthly line plots for power usage dataset
```

```
from pandas import read_csv
from matplotlib import pyplot
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# plot active power for each year
months = [x for x in range(1, 13)]
pyplot.figure()
for i in range(len(months)):
  # prepare subplot
  ax = pyplot.subplot(len(months), 1, i+1)
  # determine the month to plot
  month = '2007-' + str(months[i])
  # get all observations for the month
  result = dataset[month]
  # plot the active power for the month
  pyplot.plot(result['Global_active_power'])
  # add a title to the subplot
  pyplot.title(month, y=0, loc='left')
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
pyplot.show()
```

Listing 16.14: Example of creating line plots of power consumption per month.

Running the example creates a single image with 12 line plots, one for each month in 2007. We can see the sign-wave of power consumption of the days within each month. This is good as we would expect some kind of daily pattern in power consumption. We can see that there are stretches of days with very minimal consumption, such as in August and in April. These may represent vacation periods where the home was unoccupied and where power consumption was minimal.
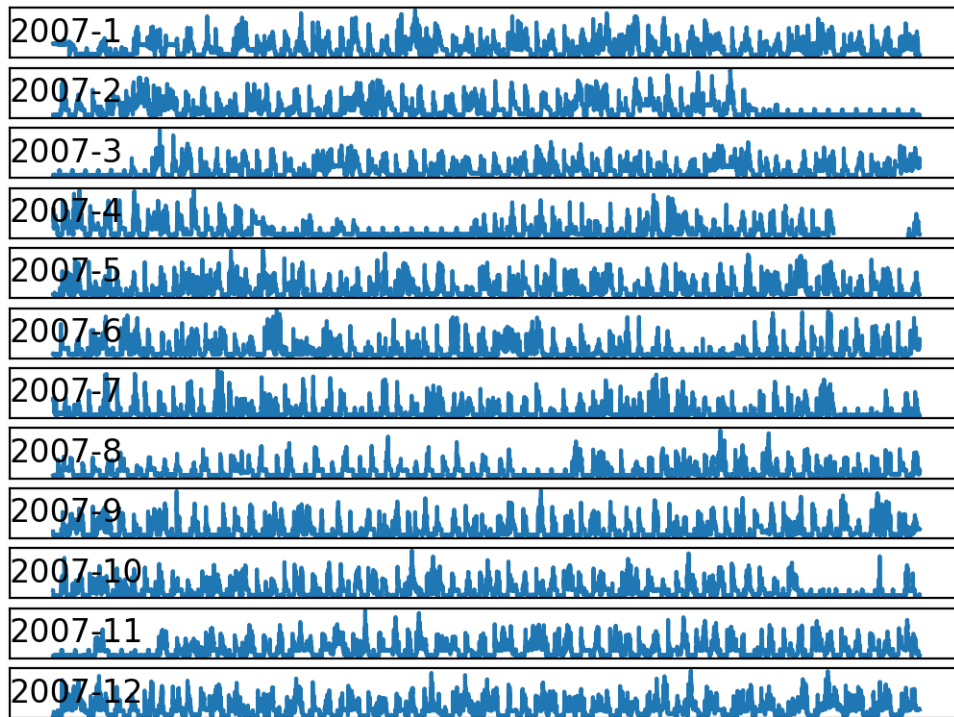
Figure 16.3: Line Plots for Active Power for All Months in One Year.

Finally, we can zoom in one more level and take a closer look at power consumption at the daily level. We would expect there to be some pattern to consumption each day, and perhaps differences in days over a week. The complete example is listed below.

```python
# daily line plots for power usage dataset
from pandas import read_csv
from matplotlib import pyplot
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# plot active power for each year
days = [x for x in range(1, 20)]
pyplot.figure()
for i in range(len(days)):
  # prepare subplot
  ax = pyplot.subplot(len(days), 1, i+1)
  # determine the day to plot
  day = '2007-01-' + str(days[i])
  # get all observations for the day
  result = dataset[day]
  # plot the active power for the day
  pyplot.plot(result['Global_active_power'])
  # add a title to the subplot
  pyplot.title(day, y=0, loc='left', size=6)
```

```
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
pyplot.show()
```

Listing 16.15: Example of creating line plots of power consumption per day.

Running the example creates a single image with 20 line plots, one for the first 20 days in January 2007. There is commonality across the days; for example, many days consumption starts early morning, around 6-7AM. Some days show a drop in consumption in the middle of the day, which might make sense if most occupants are out of the house. We do see some strong overnight consumption on some days, that in a northern hemisphere January may match up with a heating system being used. Time of year, specifically the season and the weather that it brings, will be an important factor in modeling this data, as would be expected.
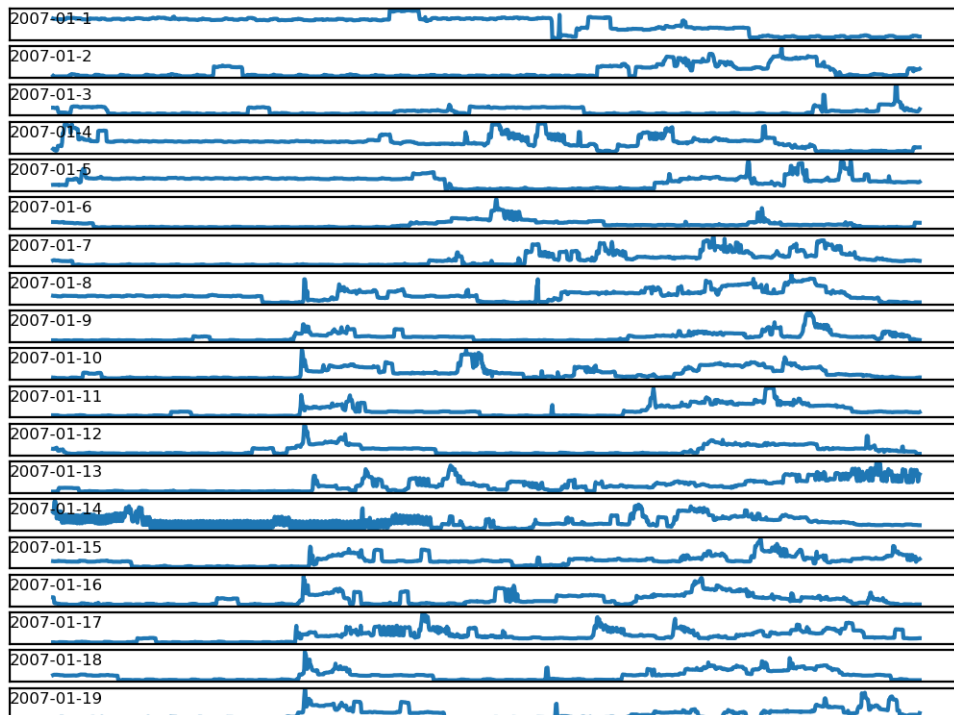

Figure 16.4: Line Plots for Active Power for 20 Days in One Month.

16.5   Time Series Data Distributions

Another important area to consider is the distribution of the variables. For example, it may be interesting to know if the distributions of observations are Gaussian or some other distribution. We can investigate the distributions of the data by reviewing histograms. We can start-off by creating a histogram for each variable in the time series. The complete example is listed below.

```
# histogram plots for power usage dataset
from pandas import read_csv
from matplotlib import pyplot
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# histogram plot for each variable
pyplot.figure()
for i in range(len(dataset.columns)):
  # create subplot
  pyplot.subplot(len(dataset.columns), 1, i+1)
  # get variable name
  name = dataset.columns[i]
  # create histogram
  dataset[name].hist(bins=100)
  # set title
  pyplot.title(name, y=0, loc='right')
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
pyplot.show()
```

Listing 16.16: Example of creating histograms for each variable.

Running the example creates a single figure with a separate histogram for each of the 8 variables. We can see that active and reactive power, intensity, as well as the sub-metered power are all skewed distributions down towards small watt-hour or kilowatt values. We can also see that distribution of voltage data is strongly Gaussian.
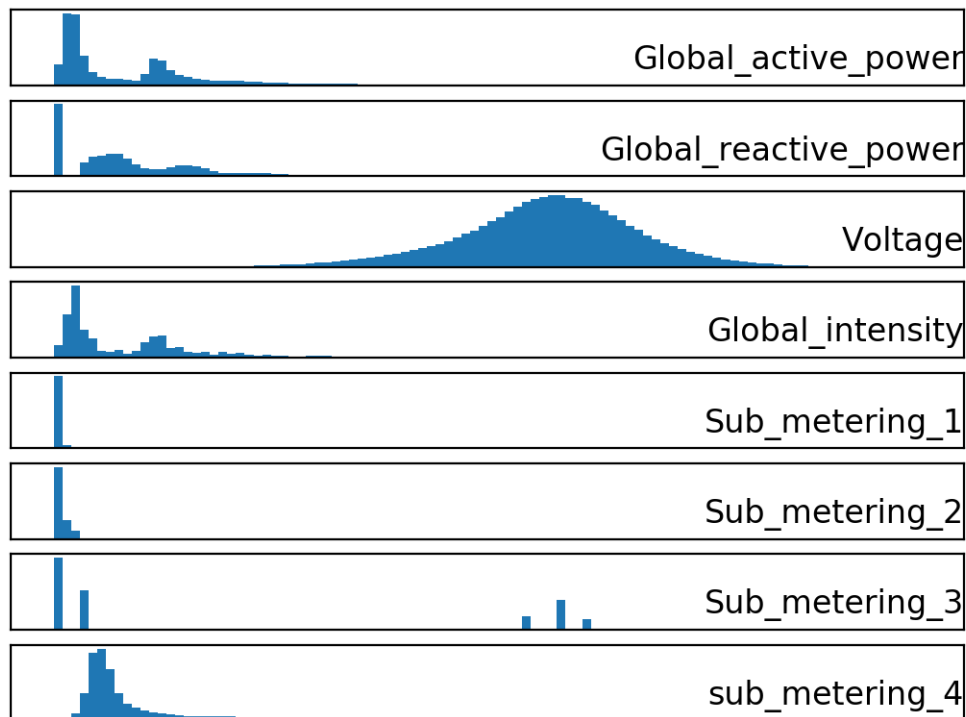
Figure 16.5: Histogram plots for Each Variable in the Power Consumption Dataset.

The distribution of active power appears to be bi-modal, meaning it looks like it has two mean groups of observations. We can investigate this further by looking at the distribution of active power consumption for the four full years of data. The complete example is listed below.

```python
# yearly histogram plots for power usage dataset
from pandas import read_csv
from matplotlib import pyplot
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# plot active power for each year
years = ['2007', '2008', '2009', '2010']
pyplot.figure()
for i in range(len(years)):
  # prepare subplot
  ax = pyplot.subplot(len(years), 1, i+1)
  # determine the year to plot
  year = years[i]
  # get all observations for the year
  result = dataset[str(year)]
  # plot the active power for the year
  result['Global_active_power'].hist(bins=100)
  # zoom in on the distribution
  ax.set_xlim(0, 5)
```

```
  # add a title to the subplot
  pyplot.title(str(year), y=0, loc='right')
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
pyplot.show()
```

Listing 16.17: Example of creating histograms of power consumption per year.

Running the example creates a single plot with four figures, one for each of the years between 2007 to 2010. We can see that the distribution of active power consumption across those years looks very similar. The distribution is indeed bimodal with one peak around 0.3 KW and perhaps another around 1.3 KW. There is a long tail on the distribution to higher kilowatt values. It might open the door to notions of discretizing the data and separating it into peak 1, peak 2 or long tail. These groups or clusters for usage on a day or hour may be helpful in developing a predictive model.
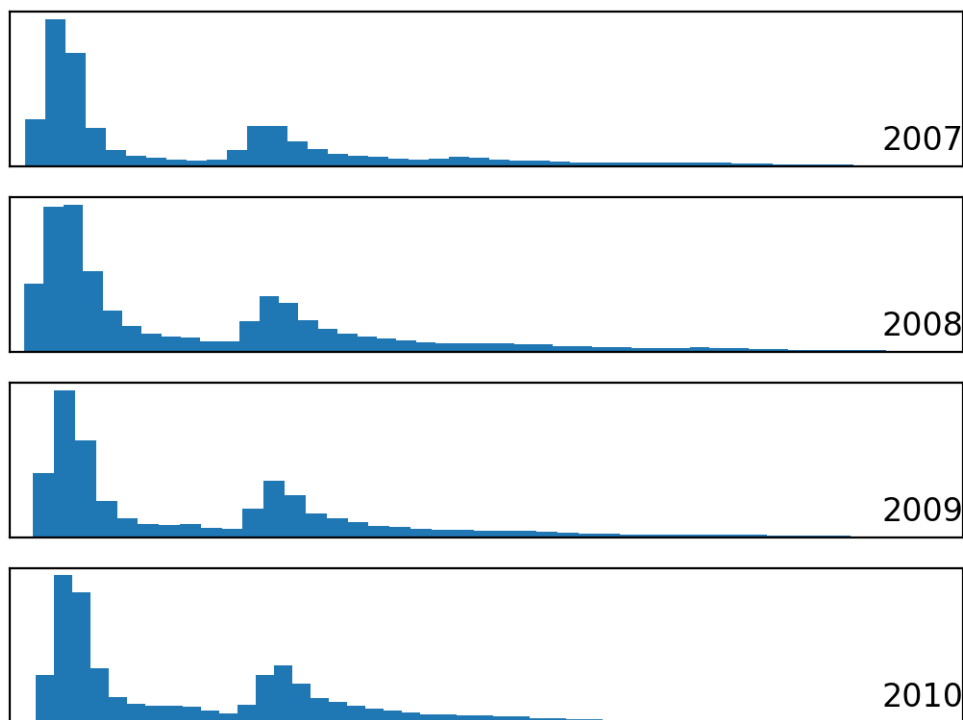

Figure 16.6: Histogram Plots of Active Power for Most Years.

It is possible that the identified groups may vary over the seasons of the year. We can investigate this by looking at the distribution for active power for each month in a year. The complete example is listed below.

```
# monthly histogram plots for power usage dataset
```

```python
from pandas import read_csv
from matplotlib import pyplot
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# plot active power for each year
months = [x for x in range(1, 13)]
pyplot.figure()
for i in range(len(months)):
  # prepare subplot
  ax = pyplot.subplot(len(months), 1, i+1)
  # determine the month to plot
  month = '2007-' + str(months[i])
  # get all observations for the month
  result = dataset[month]
  # plot the active power for the month
  result['Global_active_power'].hist(bins=100)
  # zoom in on the distribution
  ax.set_xlim(0, 5)
  # add a title to the subplot
  pyplot.title(month, y=0, loc='right')
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
pyplot.show()
```

Listing 16.18: Example of creating histograms of power consumption per month.

Running the example creates an image with 12 plots, one for each month in 2007. We can see generally the same data distribution each month. The axes for the plots appear to align (given the similar scales), and we can see that the peaks are shifted down in the warmer northern hemisphere months and shifted up for the colder months. We can also see a thicker or more prominent tail toward larger kilowatt values for the cooler months of December through to March.
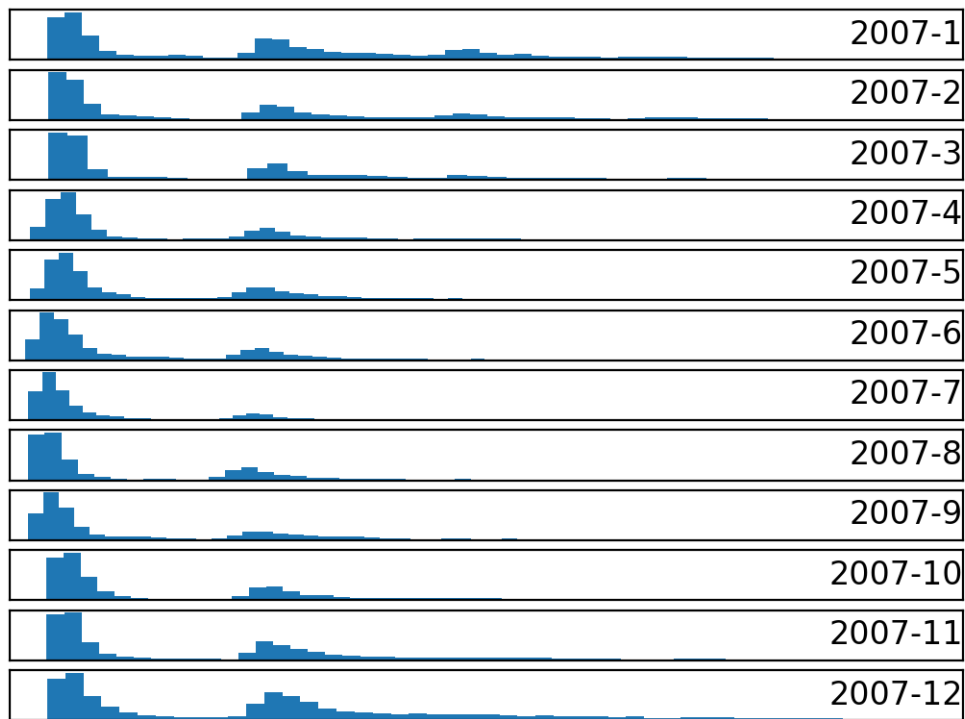
Figure 16.7: Histogram Plots for Active Power for All Months in One Year.

16.6   Ideas on Modeling

Now that we know how to load and explore the dataset, we can pose some ideas on how to model the dataset. In this section, we will take a closer look at three main areas when working with the data; they are:

1. Problem Framing

2. Data Preparation

3. Modeling Methods

16.6.1   Problem Framing

There does not appear to be a seminal publication for the dataset to demonstrate the intended way to frame the data in a predictive modeling problem. We are therefore left to guess at possibly useful ways that this data may be used. The data is only for a single household, but perhaps effective modeling approaches could be generalized across to similar households. Perhaps the most useful framing of the dataset is to forecast an interval of future active power consumption.

Four examples include:

- Forecast hourly consumption for the next day.

- Forecast daily consumption for the next week.

- Forecast daily consumption for the next month.

- Forecast monthly consumption for the next year.

Generally, these types of forecasting problems are referred to as multi-step forecasting. Models that make use of all of the variables might be referred to as a multivariate multi-step forecasting models. Each of these models is not limited to forecasting the minutely data, but instead could model the problem at or below the chosen forecast resolution. Forecasting consumption in turn, at scale, could aid in a utility company forecasting demand, which is a widely studied and important problem.

16.6.2   Data Preparation

There is a lot of flexibility in preparing this data for modeling. The specific data preparation methods and their benefit really depend on the chosen framing of the problem and the modeling methods. Nevertheless, below is a list of general data preparation methods that may be useful:

- Daily differencing may be useful to adjust for the daily cycle in the data.

- Annual differencing may be useful to adjust for any yearly cycle in the data.

- Normalization may aid in reducing the variables with differing units to the same scale.

There are many simple human factors that may be helpful in engineering features from the data, that in turn may make specific days easier to forecast. Some examples include:

- Indicating the time of day, to account for the likelihood of people being home or not.

- Indicating whether a day is a weekday or weekend.

- Indicating whether a day is a North American public holiday or not.

These factors may be significantly less important for forecasting monthly data, and perhaps to a degree for weekly data. More general features may include:

- Indicating the season, which may lead to the type or amount environmental control systems being used.

16.6.3   Modeling Methods

There are perhaps four classes of methods that might be interesting to explore on this problem; they are:

1. Naive Methods.

2. Classical Linear Methods.

3. Machine Learning Methods.

4. Deep Learning Methods.

16.6.4   Naive Methods

Naive methods would include methods that make very simple, but often very effective assumptions. Some examples include:

- Tomorrow will be the same as today.

- Tomorrow will be the same as this day last year.

- Tomorrow will be an average of the last few days.

Classical Linear Methods

Classical linear methods include techniques are very effective for univariate time series forecasting. Two important examples include:

- SARIMA.

- ETS (triple exponential smoothing).

They would require that the additional variables be discarded and the parameters of the model be configured or tuned to the specific framing of the dataset. Concerns related to adjusting the data for daily and seasonal structures can also be supported directly.

Machine Learning Methods

Machine learning methods require that the problem be framed as a supervised learning problem. This would require that lag observations for a series be framed as input features, discarding the temporal relationship in the data. A suite of nonlinear and ensemble methods could be explored, including:

- k-Nearest Neighbors.

- Support Vector Machines.

- Decision Trees.

- Random Forest.

- Gradient Boosting Machines.

Careful attention is required to ensure that the fitting and evaluation of these models preserved the temporal structure in the data. This is important so that the method is not able to *cheat* by harnessing observations from the future. These methods are often agnostic to large numbers of variables and may aid in teasing out whether the additional variables can be harnessed and add value to predictive models.

Deep Learning Methods

Generally, neural networks have not proven very effective at autoregression type problems. Nevertheless, techniques such as convolutional neural networks are able to automatically learn complex features from raw data, including one-dimensional signal data. And recurrent neural networks, such as the long short-term memory network, are capable of directly learning across multiple parallel sequences of input data. Further, combinations of these methods, such as CNN-LSTM and ConvLSTM, have proven effective on time series classification tasks. It is possible that these methods may be able to harness the large volume of minute-based data and multiple input variables.

16.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **List Intuitions**. Outline the method that you believe may be most effective in making forecasts with on this dataset.

- **Apply Taxonomy**. Use the taxonomy in Chapter 2 to describe the dataset presented in this chapter.

- **Additional Analysis**. Use summary statistics and/or plots to explore one more aspect of the dataset that may provide insight into modeling this problem.

If you explore any of these extensions, I'd love to know.

16.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

16.8.1 APIs

- `pandas.read_csv` API.
  https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

- `matplotlib.pyplot` API.
  https://matplotlib.org/api/pyplot_api.html

16.8.2 Articles

- Household Power Consumption Dataset, UCI Machine Learning Repository.
  https://archive.ics.uci.edu/ml/datasets/individual+household+electric+power+consumption

- AC power, Wikipedia.
  https://en.wikipedia.org/wiki/AC_power#Active,_reactive,_and_apparent_power

16.9   Summary

In this tutorial, you discovered a household power consumption dataset for multi-step time series forecasting and how to better understand the raw data using exploratory analysis. Specifically, you learned:

- The household power consumption dataset that describes electricity usage for a single house over four years.

- How to explore and understand the dataset using a suite of line plots for the series data and histogram for the data distributions.

- How to use the new understanding of the problem to consider different framings of the prediction problem, ways the data may be prepared, and modeling methods that may be used.

16.9.1   Next

In the next lesson, you will discover how to develop robust naive models for forecasting the household power usage problem.

Chapter 17

How to Develop Naive Models for Multi-step Energy Usage Forecasting

Given the rise of smart electricity meters and the wide adoption of electricity generation technology like solar panels, there is a wealth of electricity usage data available. This data represents a multivariate time series of power-related variables that in turn could be used to model and even forecast future electricity consumption. In this tutorial, you will discover how to develop a test harness for the *Household Power Consumption* dataset and evaluate three naive forecast strategies that provide a baseline for more sophisticated algorithms. After completing this tutorial, you will know:

- How to load, prepare, and downsample the household power consumption dataset ready for developing models.

- How to develop metrics, dataset split, and walk-forward validation elements for a robust test harness for evaluating forecasting models.

- How to develop and evaluate and compare the performance a suite of naive persistence forecasting methods.

Let's get started.

17.1   Tutorial Overview

This tutorial is divided into four parts; they are:

1. Problem Description

2. Load and Prepare Dataset

3. Model Evaluation

4. Develop Naive Forecast Models

17.2   Problem Description

The *Household Power Consumption* dataset is a multivariate time series dataset that describes the electricity consumption for a single household over four years. The data was collected between December 2006 and November 2010 and observations of power consumption within the household were collected every minute. It is a multivariate series comprised of seven variables (besides the date and time); they are:

- `global_active_power`: The total active power consumed by the household (kilowatts).

- `global_reactive_power`: The total reactive power consumed by the household (kilowatts).

- `voltage`: Average voltage (volts).

- `global_intensity`: Average current intensity (amps).

- `sub_metering_1`: Active energy for kitchen (watt-hours of active energy).

- `sub_metering_2`: Active energy for laundry (watt-hours of active energy).

- `sub_metering_3`: Active energy for climate control systems (watt-hours of active energy).

Active and reactive energy refer to the technical details of alternative current. A fourth sub-metering variable can be created by subtracting the sum of three defined sub-metering variables from the total active energy. This dataset was introduced and analyzed in Chapter 16. Refer to that chapter for more details if needed.

17.3   Load and Prepare Dataset

The dataset can be downloaded as a single 20 megabyte zip file. A direct download link is provided blow:

- household_power_consumption.zip [1]

Download the dataset and unzip it into your current working directory. You will now have the file `household_power_consumption.txt` that is about 127 megabytes in size and contains all of the observations. We can use the `read_csv()` function to load the data and combine the first two columns into a single date-time column that we can use as an index.

```
# load all data
dataset = read_csv('household_power_consumption.txt', sep=';', header=0, low_memory=False,
    infer_datetime_format=True, parse_dates={'datetime':[0,1]}, index_col=['datetime'])
```

Listing 17.1: Example of loading the dataset.

Next, we can mark all missing values indicated with a '?' character with a `NaN` value, which is a float. This will allow us to work with the data as one array of floating point values rather than mixed types (less efficient.)

---

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/household_power_consumption.zip

```
# mark all missing values
dataset.replace('?', nan, inplace=True)
# make dataset numeric
dataset = dataset.astype('float32')
```

Listing 17.2: Example of marking missing values.

We also need to fill in the missing values now that they have been marked. A very simple approach would be to copy the observation from the same time the day before. We can implement this in a function named `fill_missing()` that will take the NumPy array of the data and copy values from exactly 24 hours ago.

```
# fill missing values with a value at the same time one day ago
def fill_missing(values):
  one_day = 60 * 24
  for row in range(values.shape[0]):
    for col in range(values.shape[1]):
      if isnan(values[row, col]):
        values[row, col] = values[row - one_day, col]
```

Listing 17.3: Example of a function for filling missing values.

We can apply this function directly to the data within the `DataFrame`.

```
# fill missing
fill_missing(dataset.values)
```

Listing 17.4: Example filling missing values.

Now we can create a new column that contains the remainder of the sub-metering, using the calculation from the previous section.

```
# add a column for for the remainder of sub metering
values = dataset.values
dataset['sub_metering_4'] = (values[:,0] * 1000 / 60) - (values[:,4] + values[:,5] +
    values[:,6])
```

Listing 17.5: Example of calculating the remaining sub-metered power.

We can now save the cleaned-up version of the dataset to a new file; in this case we will just change the file extension to `.csv` and save the dataset as `household_power_consumption.csv`.

```
# save updated dataset
dataset.to_csv('household_power_consumption.csv')
```

Listing 17.6: Example of saving the prepared dataset to file.

Tying all of this together, the complete example of loading, cleaning-up, and saving the dataset is listed below.

```
# load and clean-up the power usage dataset
from numpy import nan
from numpy import isnan
from pandas import read_csv

# fill missing values with a value at the same time one day ago
def fill_missing(values):
  one_day = 60 * 24
```

```
  for row in range(values.shape[0]):
    for col in range(values.shape[1]):
      if isnan(values[row, col]):
        values[row, col] = values[row - one_day, col]

# load all data
dataset = read_csv('household_power_consumption.txt', sep=';', header=0, low_memory=False,
    infer_datetime_format=True, parse_dates={'datetime':[0,1]}, index_col=['datetime'])
# mark all missing values
dataset.replace('?', nan, inplace=True)
# make dataset numeric
dataset = dataset.astype('float32')
# fill missing
fill_missing(dataset.values)
# add a column for for the remainder of sub metering
values = dataset.values
dataset['sub_metering_4'] = (values[:,0] * 1000 / 60) - (values[:,4] + values[:,5] +
    values[:,6])
# save updated dataset
dataset.to_csv('household_power_consumption.csv')
```

Listing 17.7: Example of preparing the dataset for modeling.

Running the example creates the new file `household_power_consumption.csv` that we can use as the starting point for our modeling project.

17.4 Model Evaluation

In this section, we will consider how we can develop and evaluate predictive models for the household power dataset. This section is divided into four parts; they are:

1. Problem Framing

2. Evaluation Metric

3. Train and Test Sets

4. Walk-Forward Validation

17.4.1 Problem Framing

There are many ways to harness and explore the household power consumption dataset. In this tutorial, we will use the data to explore a very specific question; that is: *Given recent power consumption, what is the expected power consumption for the week ahead?* This requires that a predictive model forecast the total active power for each day over the next seven days. Technically, this framing of the problem is referred to as a multi-step time series forecasting problem, given the multiple forecast steps. A model that makes use of multiple input variables may be referred to as a multivariate multi-step time series forecasting model.

A model of this type could be helpful within the household in planning expenditures. It could also be helpful on the supply side for planning electricity demand for a specific household. This framing of the dataset also suggests that it would be useful to downsample the per-minute

observations of power consumption to daily totals. This is not required, but makes sense, given that we are interested in total power per day. We can achieve this easily using the `resample()` function on the Pandas `DataFrame`. Calling this function with the argument 'D' allows the loaded data indexed by date-time to be grouped by day (see all offset aliases). We can then calculate the sum of all observations for each day and create a new dataset of daily power consumption data for each of the eight variables. The complete example is listed below.

```
# resample minute data to total for each day for the power usage dataset
from pandas import read_csv
# load the new file
dataset = read_csv('household_power_consumption.csv', header=0, infer_datetime_format=True,
    parse_dates=['datetime'], index_col=['datetime'])
# resample data to daily
daily_groups = dataset.resample('D')
daily_data = daily_groups.sum()
# summarize
print(daily_data.shape)
print(daily_data.head())
# save
daily_data.to_csv('household_power_consumption_days.csv')
```

Listing 17.8: Example of resampling the dataset to daily.

Running the example creates a new daily total power consumption dataset and saves the result into a separate file named `household_power_consumption_days.csv`. We can use this as the dataset for fitting and evaluating predictive models for the chosen framing of the problem.

17.4.2 Evaluation Metric

A forecast will be comprised of seven values, one for each day of the week ahead. It is common with multi-step forecasting problems to evaluate each forecasted time step separately. This is helpful for a few reasons:

- To comment on the skill at a specific lead time (e.g. +1 day vs +3 days).

- To contrast models based on their skills at different lead times (e.g. models good at +1 day vs models good at days +5).

The units of the total power are kilowatts and it would be useful to have an error metric that was also in the same units. Both Root Mean Squared Error (RMSE) and Mean Absolute Error (MAE) fit this bill, although RMSE is more commonly used and will be adopted in this tutorial. Unlike MAE, RMSE is more punishing of forecast errors. The performance metric for this problem will be the RMSE for each lead time from day 1 to day 7. As a short-cut, it may be useful to summarize the performance of a model using a single score in order to aide in model selection. One possible score that could be used would be the RMSE across all forecast days. The function `evaluate_forecasts()` below will implement this behavior and return the performance of a model based on multiple seven-day forecasts.

```
# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
```

```
for i in range(actual.shape[1]):
  # calculate mse
  mse = mean_squared_error(actual[:, i], predicted[:, i])
  # calculate rmse
  rmse = sqrt(mse)
  # store
  scores.append(rmse)
# calculate overall RMSE
s = 0
for row in range(actual.shape[0]):
  for col in range(actual.shape[1]):
    s += (actual[row, col] - predicted[row, col])**2
score = sqrt(s / (actual.shape[0] * actual.shape[1]))
return score, scores
```

Listing 17.9: Example of a function for evaluating forecasts.

Running the function will first return the overall RMSE regardless of day, then an array of RMSE scores for each day.

17.4.3 Train and Test Sets

We will use the first three years of data for training predictive models and the final year for evaluating models. The data in a given dataset will be divided into standard weeks. These are weeks that begin on a Sunday and end on a Saturday. This is a realistic and useful way for using the chosen framing of the model, where the power consumption for the week ahead can be predicted. It is also helpful with modeling, where models can be used to predict a specific day (e.g. Wednesday) or the entire sequence.

We will split the data into standard weeks, working backwards from the test dataset. The final year of the data is in 2010 and the first Sunday for 2010 was January 3rd. The data ends in mid November 2010 and the closest final Saturday in the data is November 20th. This gives 46 weeks of test data. The first and last rows of daily data for the test dataset are provided below for confirmation.

```
2010-01-03,2083.4539999999984,191.61000000000055,350992.12000000034,8703.600000000033,...
...
2010-11-20,2197.006000000004,153.76800000000028,346475.9999999998,9320.20000000002,...
```

Listing 17.10: First and last rows of the daily dataset.

The daily data starts in late 2006. The first Sunday in the dataset is December 17th, which is the second row of data. Organizing the data into standard weeks gives 159 full standard weeks for training a predictive model.

```
2006-12-17,3390.46,226.0059999999994,345725.32000000024,14398.59999999998,2033.0,4187.0,...
...
2010-01-02,1309.2679999999998,199.54600000000016,352332.8399999997,5489.7999999999865,...
```

Listing 17.11: Observations that define the boundary of training and test sets.

The function split_dataset() below splits the daily data into train and test sets and organizes each into standard weeks. Specific row offsets are used to split the data using knowledge of the dataset. The split datasets are then organized into weekly data using the NumPy split() function.

```
# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test
```

Listing 17.12: Example of a function for splitting the data into train and test sets.

We can test this function out by loading the daily dataset and printing the first and last rows of data from both the train and test sets to confirm they match the expectations above. The complete code example is listed below.

```
# split the power usage dataset into standard weeks
from numpy import split
from numpy import array
from pandas import read_csv

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
train, test = split_dataset(dataset.values)
# validate train data
print(train.shape)
print(train[0, 0, 0], train[-1, -1, 0])
# validate test
print(test.shape)
print(test[0, 0, 0], test[-1, -1, 0])
```

Listing 17.13: Example of splitting the data into train and test sets.

Running the example shows that indeed the train dataset has 159 weeks of data, whereas the test dataset has 46 weeks. We can see that the total active power for the train and test dataset for the first and last rows match the data for the specific dates that we defined as the bounds on the standard weeks for each set.

```
(159, 7, 8)
3390.46 1309.2679999999998
(46, 7, 8)
2083.4539999999984 2197.006000000004
```

Listing 17.14: Sample output from splitting the data into train and test sets.

17.4.4 Walk-Forward Validation

Models will be evaluated using a scheme called walk-forward validation. This is where a model is required to make a one week prediction, then the actual data for that week is made available to the model so that it can be used as the basis for making a prediction on the subsequent week. This is both realistic for how the model may be used in practice and beneficial to the models allowing them to make use of the best available data. We can demonstrate this below with separation of input data and output/predicted data.

```
Input,                  Predict
[Week1]                 Week2
[Week1 + Week2]         Week3
[Week1 + Week2 + Week3] Week4
...
```

Listing 17.15: Example of weekly walk-forward validation.

The walk-forward validation approach to evaluating predictive models on this dataset is implement below, named `evaluate_model()`. The name of a function is provided for the model as the argument `model_func`. This function is responsible for defining the model, fitting the model on the training data, and making a one-week forecast. The forecasts made by the model are then evaluated against the test dataset using the previously defined `evaluate_forecasts()` function.

```python
# evaluate a single model
def evaluate_model(model_func, train, test):
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = model_func(history)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  predictions = array(predictions)
  # evaluate predictions days for each week
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores
```

Listing 17.16: Example of a function for walk-forward validation.

Once we have the evaluation for a model, we can summarize the performance. The function below named `summarize_scores()` will display the performance of a model as a single line for easy comparison with other models.

```python
# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))
```

Listing 17.17: Example of a function for summarizing model performance.

We now have all of the elements to begin evaluating predictive models on the dataset.

17.5 Develop Naive Forecast Models

It is important to test naive forecast models on any new prediction problem. The results from naive models provide a quantitative idea of how difficult the forecast problem is and provide a baseline performance by which more sophisticated forecast methods can be evaluated. In this section, we will develop and compare three naive forecast methods for the household power prediction problem; they are:

1. Daily Persistence Forecast.

2. Weekly Persistent Forecast.

3. Weekly One-Year-Ago Persistent Forecast.

For more information on simple forecast strategies generally, see Chapter 5.

17.5.1 Daily Persistence Forecast

The first naive forecast that we will develop is a daily persistence model. This model takes the active power from the last day prior to the forecast period (e.g. Saturday) and uses it as the value of the power for each day in the forecast period (Sunday to Saturday). The `daily_persistence()` function below implements the daily persistence forecast strategy.

```
# daily persistence model
def daily_persistence(history):
  # get the data for the prior week
  last_week = history[-1]
  # get the total active power for the last day
  value = last_week[-1, 0]
  # prepare 7 day forecast
  forecast = [value for _ in range(7)]
  return forecast
```

Listing 17.18: Example of a function for a daily persistence model.

17.5.2 Weekly Persistent Forecast

Another good naive forecast when forecasting a standard week is to use the entire prior week as the forecast for the week ahead. It is based on the idea that next week will be very similar to this week. The `weekly_persistence()` function below implements the weekly persistence forecast strategy.

```
# weekly persistence model
def weekly_persistence(history):
  # get the data for the prior week
  last_week = history[-1]
  return last_week[:, 0]
```

Listing 17.19: Example of a function for a weekly persistence model.

17.5.3 Weekly One-Year-Ago Persistent Forecast

Similar to the idea of using last week to forecast next week is the idea of using the same week last year to predict next week. That is, use the week of observations from 52 weeks ago as the forecast, based on the idea that next week will be similar to the same week one year ago. The `week_one_year_ago_persistence()` function below implements the week one year ago forecast strategy.

```
# week one year ago persistence model
def week_one_year_ago_persistence(history):
  # get the data for the prior week
  last_week = history[-52]
  return last_week[:, 0]
```

Listing 17.20: Example of a function for a week one year ago persistence model.

17.5.4 Naive Model Comparison

We can compare each of the forecast strategies using the test harness developed in the previous section. First, the dataset can be loaded and split into train and test sets.

```
# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
```

Listing 17.21: Load and split the daily dataset.

Each of the strategies can be stored in a dictionary against a unique name. This name can be used in printing and in creating a plot of the scores.

```
# define the names and functions for the models we wish to evaluate
models = dict()
models['daily'] = daily_persistence
models['weekly'] = weekly_persistence
models['week-oya'] = week_one_year_ago_persistence
```

Listing 17.22: Specify the model functions to evaluate.

We can then enumerate each of the strategies, evaluating it using walk-forward validation, printing the scores, and adding the scores to a line plot for visual comparison.

```
# evaluate each model
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
for name, func in models.items():
  # evaluate and get scores
  score, scores = evaluate_model(func, train, test)
  # summarize scores
  summarize_scores('daily persistence', score, scores)
  # plot scores
  pyplot.plot(days, scores, marker='o', label=name)
```

Listing 17.23: Evaluate and plot the performance of each model.

Tying all of this together, the complete example evaluating the three naive forecast strategies is listed below.

```python
# naive forecast strategies for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot

# split a univariate dataset into train/test sets
def split_dataset(data):
	# split into standard weeks
	train, test = data[1:-328], data[-328:-6]
	# restructure into windows of weekly data
	train = array(split(train, len(train)/7))
	test = array(split(test, len(test)/7))
	return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
	scores = list()
	# calculate an RMSE score for each day
	for i in range(actual.shape[1]):
		# calculate mse
		mse = mean_squared_error(actual[:, i], predicted[:, i])
		# calculate rmse
		rmse = sqrt(mse)
		# store
		scores.append(rmse)
	# calculate overall RMSE
	s = 0
	for row in range(actual.shape[0]):
		for col in range(actual.shape[1]):
			s += (actual[row, col] - predicted[row, col])**2
	score = sqrt(s / (actual.shape[0] * actual.shape[1]))
	return score, scores

# summarize scores
def summarize_scores(name, score, scores):
	s_scores = ', '.join(['%.1f' % s for s in scores])
	print('%s: [%.3f] %s' % (name, score, s_scores))

# evaluate a single model
def evaluate_model(model_func, train, test):
	# history is a list of weekly data
	history = [x for x in train]
	# walk-forward validation over each week
	predictions = list()
	for i in range(len(test)):
		# predict the week
		yhat_sequence = model_func(history)
		# store the predictions
		predictions.append(yhat_sequence)
		# get real observation and add to history for predicting the next week
		history.append(test[i, :])
	predictions = array(predictions)
```

```python
  # evaluate predictions days for each week
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# daily persistence model
def daily_persistence(history):
  # get the data for the prior week
  last_week = history[-1]
  # get the total active power for the last day
  value = last_week[-1, 0]
  # prepare 7 day forecast
  forecast = [value for _ in range(7)]
  return forecast

# weekly persistence model
def weekly_persistence(history):
  # get the data for the prior week
  last_week = history[-1]
  return last_week[:, 0]

# week one year ago persistence model
def week_one_year_ago_persistence(history):
  # get the data for the prior week
  last_week = history[-52]
  return last_week[:, 0]

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# define the names and functions for the models we wish to evaluate
models = dict()
models['daily'] = daily_persistence
models['weekly'] = weekly_persistence
models['week-oya'] = week_one_year_ago_persistence
# evaluate each model
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
for name, func in models.items():
  # evaluate and get scores
  score, scores = evaluate_model(func, train, test)
  # summarize scores
  summarize_scores(name, score, scores)
  # plot scores
  pyplot.plot(days, scores, marker='o', label=name)
# show plot
pyplot.legend()
pyplot.show()
```

Listing 17.24: Example of evaluating and comparing naive forecast methods.

Running the example first prints the total and daily scores for each model. We can see that the weekly strategy performs better than the daily strategy and that the week one year ago (week-oya) performs slightly better again. We can see this in both the overall RMSE scores for each model and in the daily scores for each forecast day. One exception is the forecast error for

the first day (Sunday) where it appears that the daily persistence model performs better than the two weekly strategies. We can use the week-oya strategy with an overall RMSE of 465.294 kilowatts as the baseline in performance for more sophisticated models to be considered skillful on this specific framing of the problem.

```
daily: [511.886] 452.9, 596.4, 532.1, 490.5, 534.3, 481.5, 482.0
weekly: [469.389] 567.6, 500.3, 411.2, 466.1, 471.9, 358.3, 482.0
week-oya: [465.294] 550.0, 446.7, 398.6, 487.0, 459.3, 313.5, 555.1
```

Listing 17.25: Sample output from evaluating and comparing naive forecast methods.

A line plot of the daily forecast error is also created. We can see the same observed pattern of the weekly strategies performing better than the daily strategy in general, except in the case of the first day. It is surprising (to me) that the week one-year-ago performs better than using the prior week. I would have expected that the power consumption from last week to be more relevant. Reviewing all strategies on the same plot suggests possible combinations of the strategies that may result in even better performance.
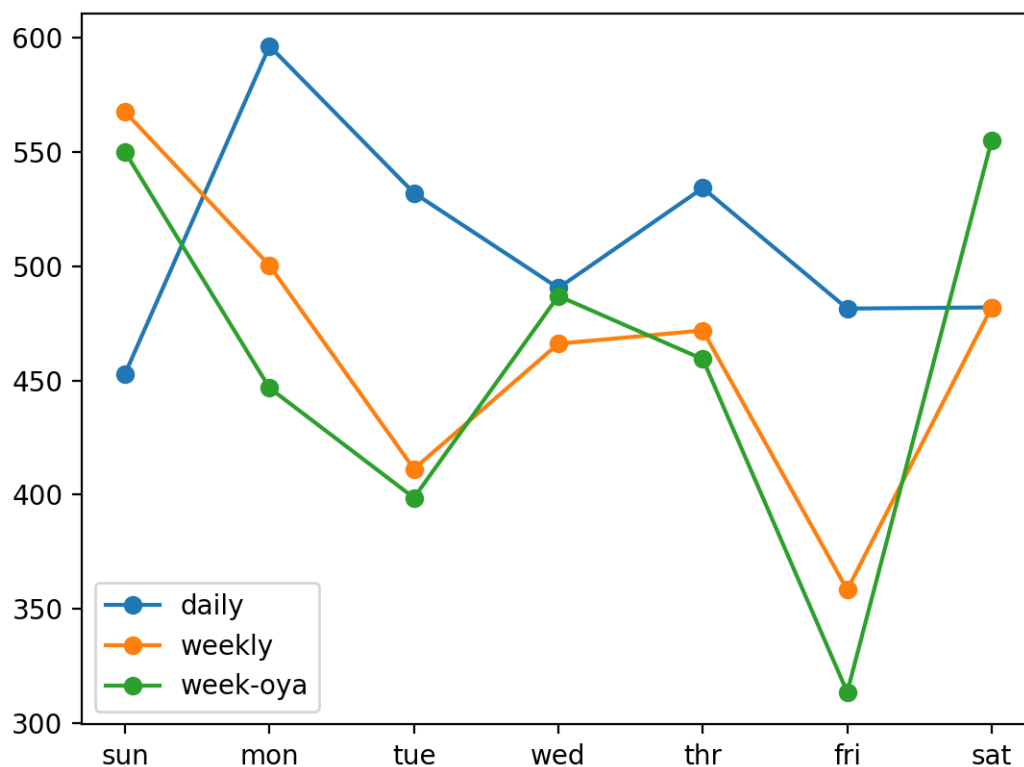

Figure 17.1: Line Plot Comparing Naive Forecast Strategies for Household Power Forecasting.

17.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- `Additional Naive Strategy`. Propose, develop, and evaluate one more naive strategy for forecasting the next week of power consumption.

- `Naive Ensemble Strategy`. Develop an ensemble strategy that combines the predictions from the three proposed naive forecast methods.

- `Optimized Direct Persistence Models`. Test and find the optimal relative prior day (e.g. -1 or -7) to use for each forecast day in a direct persistence model.

If you explore any of these extensions, I'd love to know.

17.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- `pandas.read_csv` API.
  https://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

- `numpy.split` API.
  https://docs.scipy.org/doc/numpy/reference/generated/numpy.split.html

- `pandas.DataFrame.resample` API.
  https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.resample.html

- Resample Offset Aliases.
  http://pandas.pydata.org/pandas-docs/stable/timeseries.html#offset-aliases

- `sklearn.metrics.mean_squared_error` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.mean_squared_error.html

17.8 Summary

In this tutorial, you discovered how to develop a test harness for the household power consumption dataset and evaluate three naive forecast strategies that provide a baseline for more sophisticated algorithms. Specifically, you learned:

- How to load, prepare, and downsample the household power consumption dataset ready for modeling.

- How to develop metrics, dataset split, and walk-forward validation elements for a robust test harness for evaluating forecasting models.

- How to develop and evaluate and compare the performance a suite of naive persistence forecasting methods.

17.8.1 Next

In the next lesson, you will discover how to develop autoregressive models for forecasting the household power usage problem.

Chapter 18

How to Develop ARIMA Models for Multi-step Energy Usage Forecasting

Given the rise of smart electricity meters and the wide adoption of electricity generation technology like solar panels, there is a wealth of electricity usage data available. This data represents a multivariate time series of power-related variables that in turn could be used to model and even forecast future electricity consumption. Autocorrelation models are very simple and can provide a fast and effective way to make skillful one-step and multi-step forecasts for electricity consumption. In this tutorial, you will discover how to develop and evaluate an autoregression model for multi-step forecasting household power consumption. After completing this tutorial, you will know:

- How to create and analyze autocorrelation and partial autocorrelation plots for univariate time series data.

- How to use the findings from autocorrelation plots to configure an autoregression model.

- How to develop and evaluate an autocorrelation model used to make one-week forecasts.

Let's get started.

18.1 Tutorial Overview

This tutorial is divided into five parts; they are:

1. Problem Description

2. Load and Prepare Dataset

3. Model Evaluation

4. Autocorrelation Analysis

5. Develop an Autoregressive Model

18.2 Problem Description

The *Household Power Consumption* dataset is a multivariate time series dataset that describes the electricity consumption for a single household over four years. The data was collected between December 2006 and November 2010 and observations of power consumption within the household were collected every minute. It is a multivariate series comprised of seven variables (besides the date and time); they are:

- `global_active_power`: The total active power consumed by the household (kilowatts).

- `global_reactive_power`: The total reactive power consumed by the household (kilowatts).

- `voltage`: Average voltage (volts).

- `global_intensity`: Average current intensity (amps).

- `sub_metering_1`: Active energy for kitchen (watt-hours of active energy).

- `sub_metering_2`: Active energy for laundry (watt-hours of active energy).

- `sub_metering_3`: Active energy for climate control systems (watt-hours of active energy).

Active and reactive energy refer to the technical details of alternative current. A fourth sub-metering variable can be created by subtracting the sum of three defined sub-metering variables from the total active energy. This dataset was introduced and analyzed in Chapter 16. Refer to that chapter for more details if needed.

18.3 Load and Prepare Dataset

We will use the same framework to load and prepare the data as was used for the naive models. In the interest of brevity, refer to Chapter 17 for the details on how to load and prepare the dataset for modeling.

18.4 Model Evaluation

We will use the same framework to evaluate models as was used for the naive models. In the interest of brevity, refer to Chapter 17 for the details on how to develop a framework for evaluating forecasts for this dataset.

18.5 Autocorrelation Analysis

Statistical correlation summarizes the strength of the relationship between two variables. We can assume the distribution of each variable fits a Gaussian (bell curve) distribution. If this is the case, we can use the Pearson's correlation coefficient to summarize the correlation between the variables. The Pearson's correlation coefficient is a number between -1 and 1 that describes a negative or positive correlation respectively. A value of zero indicates no correlation.

We can calculate the correlation for time series observations with observations with previous time steps, called lags. Because the correlation of the time series observations is calculated with values of the same series at previous times, this is called a serial correlation, or an autocorrelation. A plot of the autocorrelation of a time series by lag is called the AutoCorrelation Function, or the acronym ACF. This plot is sometimes called a correlogram, or an autocorrelation plot. A partial autocorrelation function or PACF is a summary of the relationship between an observation in a time series with observations at prior time steps with the relationships of intervening observations removed.

The autocorrelation for an observation and an observation at a prior time step is comprised of both the direct correlation and indirect correlations. These indirect correlations are a linear function of the correlation of the observation, with observations at intervening time steps. It is these indirect correlations that the partial autocorrelation function seeks to remove. Without going into the math, this is the intuition for the partial autocorrelation. We can calculate auto-correlation and partial autocorrelation plots using the `plot_acf()` and `plot_pacf()` Statsmodels functions respectively.

In order to calculate and plot the autocorrelation, we must convert the data into a univariate time series. Specifically, the observed daily total power consumed. The `to_series()` function below will take the multivariate data divided into weekly windows and will return a single univariate time series.

```python
# convert windows of weekly multivariate data into a series of total power
def to_series(data):
  # extract just the total power from each week
  series = [week[:, 0] for week in data]
  # flatten into a single series
  series = array(series).flatten()
  return series
```

Listing 18.1: Example of converting weekly data to a series.

We can call this function for the prepared training dataset. First, the daily power consumption dataset must be loaded.

```python
# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
```

Listing 18.2: Example of loading the prepared dataset.

The dataset must then be split into train and test sets with the standard week window structure.

```python
# split into train and test
train, test = split_dataset(dataset.values)
```

Listing 18.3: Example of splitting the loaded dataset into train and test sets.

A univariate time series of daily power consumption can then be extracted from the training dataset.

```python
# convert training data into a series
series = to_series(train)
```

Listing 18.4: Example of converting the window-based data into a series.

We can then create a single figure that contains both an ACF and a PACF plot. The number of lag time steps can be specified. We will fix this to be one year of daily observations, or 365 days.

```
# plots
pyplot.figure()
lags = 365
# acf
axis = pyplot.subplot(2, 1, 1)
plot_acf(series, ax=axis, lags=lags)
# pacf
axis = pyplot.subplot(2, 1, 2)
plot_pacf(series, ax=axis, lags=lags)
# show plot
pyplot.show()
```

Listing 18.5: Example of creating ACF and PACF plots.

The complete example is listed below. We would expect that the power consumed tomorrow and in the coming week will be dependent upon the power consumed in the prior days. As such, we would expect to see a strong autocorrelation signal in the ACF and PACF plots.

```
# acf and pacf plots of total power usage
from numpy import split
from numpy import array
from pandas import read_csv
from matplotlib import pyplot
from statsmodels.graphics.tsaplots import plot_acf
from statsmodels.graphics.tsaplots import plot_pacf

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# convert windows of weekly multivariate data into a series of total power
def to_series(data):
  # extract just the total power from each week
  series = [week[:, 0] for week in data]
  # flatten into a single series
  series = array(series).flatten()
  return series

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# convert training data into a series
series = to_series(train)
# plots
pyplot.figure()
```

```
lags = 365
# acf
axis = pyplot.subplot(2, 1, 1)
plot_acf(series, ax=axis, lags=lags)
# pacf
axis = pyplot.subplot(2, 1, 2)
plot_pacf(series, ax=axis, lags=lags)
# show plot
pyplot.show()
```

Listing 18.6: Example of creating ACF and PACF plots from the training dataset.

Running the example creates a single figure with both ACF and PACF plots. The plots are very dense, and hard to read. Nevertheless, we might be able to see a familiar autoregression pattern. We might also see some significant lag observations at one year out. Further investigation may suggest a seasonal autocorrelation component, which would not be a surprising finding.
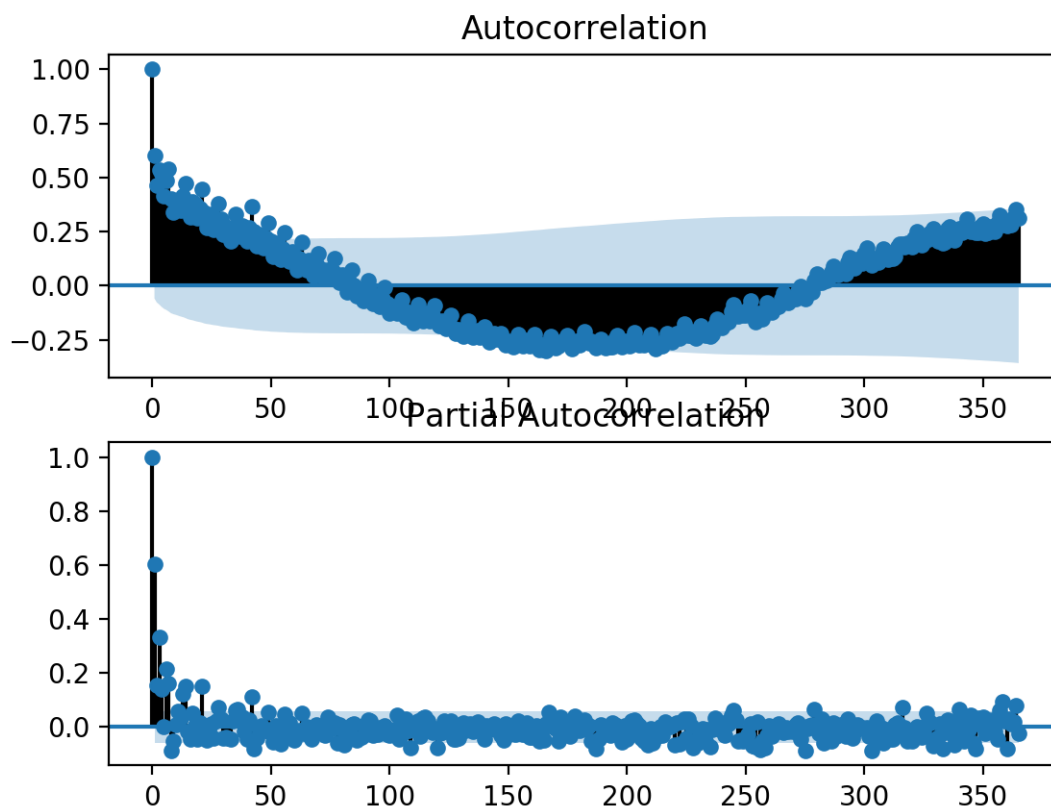

Figure 18.1: ACF and PACF plots for the univariate series of power consumption.

We can zoom in the plot and change the number of lag observations from 365 to 50.

```
lags = 50
```

Listing 18.7: Example of changing the number of lag observations in the plots.

Re-running the code example with this change results is a zoomed-in version of the plots with much less clutter. We can clearly see a familiar autoregression pattern across the two plots. This pattern is comprised of two elements:

- **ACF**: A large number of significant lag observations that slowly degrade as the lag increases.

- **PACF**: A few significant lag observations that abruptly drop as the lag increases.

The ACF plot indicates that there is a strong autocorrelation component, whereas the PACF plot indicates that this component is distinct for the first approximately seven lag observations. This suggests that a good starting model would be an `AR(7)`; that is an autoregression model with seven lag observations used as input.
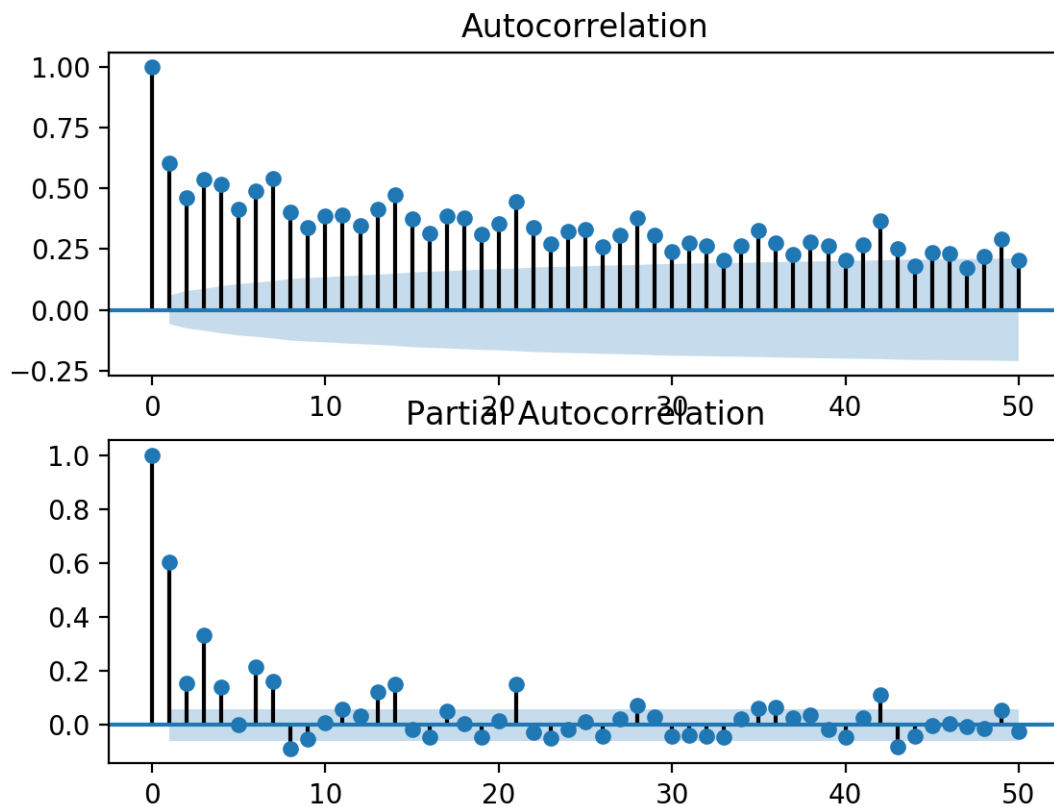

Figure 18.2: Zoomed in ACF and PACF plots for the univariate series of power consumption.

18.6 Develop an Autoregressive Model

We can develop an autoregression model for univariate series of daily power consumption. For more information on autoregressive models see Chapter 5. The Statsmodels library provides multiple ways of developing an AR model, such as using the AR, ARMA, ARIMA, and SARIMAX classes. For more information on developing SARIMAX models with the Statsmodels library,

see Chapter 13. We will use the ARIMA implementation as it allows for easy expandability into differencing and moving average. First, the history data comprised of weeks of prior observations must be converted into a univariate time series of daily power consumption. We can use the `to_series()` function developed in the previous section.

```
# convert history into a univariate series
series = to_series(history)
```

Listing 18.8: Example of converting the window-based history data into a series.

Next, an ARIMA model can be defined by passing arguments to the constructor of the ARIMA class. We will specify an `AR(7)` model, which in ARIMA notation is `ARIMA(7,0,0)`.

```
# define the model
model = ARIMA(series, order=(7,0,0))
```

Listing 18.9: Example of defining an ARIMA model.

Next, the model can be fit on the training data. We will use the defaults and disable all debugging information during the fit by setting `disp=False`.

```
# fit the model
model_fit = model.fit(disp=False)
```

Listing 18.10: Example of fitting an ARIMA model.

Now that the model has been fit, we can make a prediction. A prediction can be made by calling the `predict()` function and passing it either an interval of dates or indices relative to the training data. We will use indices starting with the first time step beyond the training data and extending it six more days, giving a total of a seven day forecast period beyond the training dataset.

```
# make forecast
yhat = model_fit.predict(len(series), len(series)+6)
```

Listing 18.11: Example of making a one-week forecast with an ARIMA model.

We can wrap all of this up into a function below named `arima_forecast()` that takes the history and returns a one week forecast.

```
# arima forecast
def arima_forecast(history):
  # convert history into a univariate series
  series = to_series(history)
  # define the model
  model = ARIMA(series, order=(7,0,0))
  # fit the model
  model_fit = model.fit(disp=False)
  # make forecast
  yhat = model_fit.predict(len(series), len(series)+6)
  return yhat
```

Listing 18.12: Example of a function for fitting and forecasting with an ARIMA model.

This function can be used directly in the test harness described previously. The complete example is listed below.

```python
# arima forecast for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from statsmodels.tsa.arima_model import ARIMA

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores

# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# evaluate a single model
def evaluate_model(model_func, train, test):
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = model_func(history)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
```

```
  predictions = array(predictions)
  # evaluate predictions days for each week
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# convert windows of weekly multivariate data into a series of total power
def to_series(data):
  # extract just the total power from each week
  series = [week[:, 0] for week in data]
  # flatten into a single series
  series = array(series).flatten()
  return series

# arima forecast
def arima_forecast(history):
  # convert history into a univariate series
  series = to_series(history)
  # define the model
  model = ARIMA(series, order=(7,0,0))
  # fit the model
  model_fit = model.fit(disp=False)
  # make forecast
  yhat = model_fit.predict(len(series), len(series)+6)
  return yhat

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# define the names and functions for the models we wish to evaluate
models = dict()
models['arima'] = arima_forecast
# evaluate each model
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
for name, func in models.items():
  # evaluate and get scores
  score, scores = evaluate_model(func, train, test)
  # summarize scores
  summarize_scores(name, score, scores)
  # plot scores
  pyplot.plot(days, scores, marker='o', label=name)
# show plot
pyplot.legend()
pyplot.show()
```

Listing 18.13: Example of evaluating an ARIMA model for multi-step power consumption forecasting.

Running the example first prints the performance of the `AR(7)` model on the test dataset. We can see that the model achieves the overall RMSE of about 381 kilowatts. This model has skill when compared to naive forecast models, such as a model that forecasts the week ahead using observations from the same time one year ago that achieved an overall RMSE of about 465 kilowatts.

```
arima: [381.636] 393.8, 398.9, 357.0, 377.2, 393.9, 306.1, 432.2
```

Listing 18.14: Sample output from evaluating an ARIMA model for multi-step power consumption forecasting.

A line plot of the forecast is also created, showing the RMSE in kilowatts for each of the seven lead times of the forecast. We can see an interesting pattern. We might expect that earlier lead times are easier to forecast than later lead times, as the error at each successive lead time compounds. Instead, we see that Friday (lead time +6) is the easiest to forecast and Saturday (lead time +7) is the most challenging to forecast. We can also see that the remaining lead times all have a similar error in the mid- to high-300 kilowatt range.
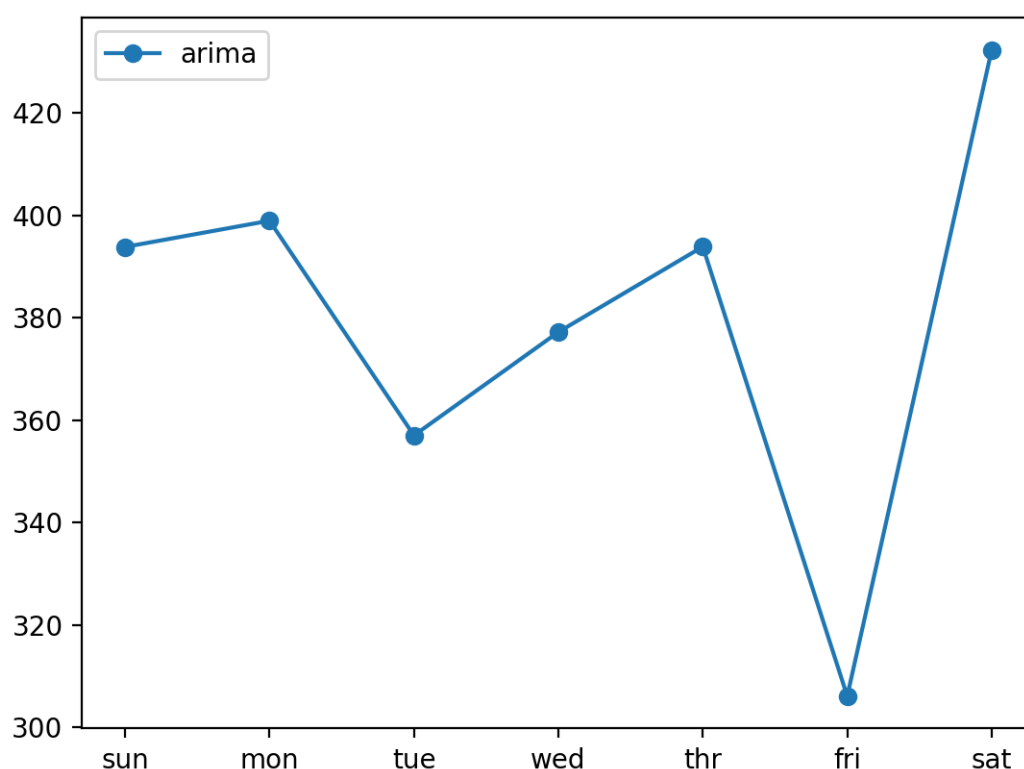

Figure 18.3: Line plot of ARIMA forecast error for each forecasted lead times.

18.7   Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Tune ARIMA**. The parameters of the ARIMA model were not tuned. Explore or search a suite of ARIMA parameters (q, d, p) to see if performance can be further improved.

- **Explore Seasonal AR**. Explore whether the performance of the AR model can be improved by including seasonal autoregression elements. This may require the use of a SARIMA model.

- **Explore ETS**. Explore whether better results may be achieved by using an ETS model instead of an ARIMA model.

- **Explore Data Preparation**. The model was fit on the raw data directly. Explore whether standardization or normalization or even power transforms can further improve the skill of the AR model.

If you explore any of these extensions, I'd love to know.

18.8   Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- `statsmodels.graphics.tsaplots.plot_acf` API.
  http://www.statsmodels.org/dev/generated/statsmodels.graphics.tsaplots.plot_acf.html

- `statsmodels.graphics.tsaplots.plot_pacf` API.
  http://www.statsmodels.org/dev/generated/statsmodels.graphics.tsaplots.plot_pacf.html

- `statsmodels.tsa.arima_model.ARIMA` API.
  http://www.statsmodels.org/dev/generated/statsmodels.tsa.arima_model.ARIMA.html

- `statsmodels.tsa.arima_model.ARIMAResults` API.
  http://www.statsmodels.org/dev/generated/statsmodels.tsa.arima_model.ARIMAResults.html

18.9   Summary

In this tutorial, you discovered how to develop and evaluate an autoregression model for multi-step forecasting household power consumption. Specifically, you learned:

- How to create and analyze autocorrelation and partial autocorrelation plots for univariate time series data.

- How to use the findings from autocorrelation plots to configure an autoregression model.

- How to develop and evaluate an autocorrelation model used to make one-week forecasts.

18.9.1   Next

In the next lesson, you will discover how to develop Convolutional Neural Network models for forecasting the household power usage problem.

Chapter 19

How to Develop CNNs for Multi-step Energy Usage Forecasting

Given the rise of smart electricity meters and the wide adoption of electricity generation technology like solar panels, there is a wealth of electricity usage data available. This data represents a multivariate time series of power-related variables that in turn could be used to model and even forecast future electricity consumption. Unlike other machine learning algorithms, convolutional neural networks are capable of automatically learning features from sequence data, support multiple-variate data, and can directly output a vector for multi-step forecasting. As such, one-dimensional CNNs have been demonstrated to perform well and even achieve state-of-the-art results on challenging sequence prediction problems. In this tutorial, you will discover how to develop 1D convolutional neural networks for multi-step time series forecasting. After completing this tutorial, you will know:

- How to develop a CNN for multi-step time series forecasting model for univariate data.

- How to develop a multi-channel multi-step time series forecasting model for multivariate data.

- How to develop a multi-headed multi-step time series forecasting model for multivariate data.

Let's get started.

19.1   Tutorial Overview

This tutorial is divided into seven parts; they are:

1. Problem Description

2. Load and Prepare Dataset

3. Model Evaluation

4. CNNs for Multi-step Forecasting

5. Univariate CNN Model

367

6. Multi-channel CNN Model

7. Multi-headed CNN Model

19.2 Problem Description

The *Household Power Consumption* dataset is a multivariate time series dataset that describes the electricity consumption for a single household over four years. The data was collected between December 2006 and November 2010 and observations of power consumption within the household were collected every minute. It is a multivariate series comprised of seven variables (besides the date and time); they are:

- `global_active_power`: The total active power consumed by the household (kilowatts).

- `global_reactive_power`: The total reactive power consumed by the household (kilowatts).

- `voltage`: Average voltage (volts).

- `global_intensity`: Average current intensity (amps).

- `sub_metering_1`: Active energy for kitchen (watt-hours of active energy).

- `sub_metering_2`: Active energy for laundry (watt-hours of active energy).

- `sub_metering_3`: Active energy for climate control systems (watt-hours of active energy).

Active and reactive energy refer to the technical details of alternative current. A fourth sub-metering variable can be created by subtracting the sum of three defined sub-metering variables from the total active energy. This dataset was introduced and analyzed in Chapter 16. Refer to that chapter for more details if needed.

19.3 Load and Prepare Dataset

We will use the same framework to load and prepare the data as was used for the naive models. In the interest of brevity, refer to Chapter 17 for the details on how to load and prepare the dataset for modeling.

19.4 Model Evaluation

We will use the same framework to evaluate models as was used for the naive models. In the interest of brevity, refer to Chapter 17 for the details on how to develop a framework for evaluating forecasts for this dataset. We must update the framework for model evaluation, specifically the walk-forward validation method used to fit a CNN model and make a forecast. The development of the new walk-forward validation framework follows in the next subsection.

19.4.1 Walk-Forward Validation

Models will be evaluated using a scheme called walk-forward validation. This is where a model is required to make a one week prediction, then the actual data for that week is made available to the model so that it can be used as the basis for making a prediction on the subsequent week. This is both realistic for how the model may be used in practice and beneficial to the models, allowing them to make use of the best available data. We can demonstrate this below with separation of input data and output/predicted data.

```
Input,                          Predict
[Week1]                         Week2
[Week1 + Week2]                 Week3
[Week1 + Week2 + Week3]         Week4
...
```

Listing 19.1: Example of weekly walk-forward validation.

The walk-forward validation approach to evaluating predictive models on this dataset is provided below, named `evaluate_model()`. The train and test datasets in standard-week format are provided to the function as arguments. An additional argument, `n_input`, is provided that is used to define the number of prior observations that the model will use as input in order to make a prediction. Two new functions are called: one to build a model from the training data called `build_model()` and another that uses the model to make forecasts for each new standard week, called `forecast()`. These will be covered in subsequent sections.

We are working with neural networks and as such they are generally slow to train but fast to evaluate. This means that the preferred usage of the models is to build them once on historical data and to use them to forecast each step of the walk-forward validation. The models are static (i.e. not updated) during their evaluation. This is different to other models that are faster to train, where a model may be re-fit or updated each step of the walk-forward validation as new data is made available. With sufficient resources, it is possible to use neural networks this way, but we will not in this tutorial. The complete `evaluate_model()` function is listed below.

```python
# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores
```

Listing 19.2: Example of a function for walk-forward validation.

Once we have the evaluation for a model, we can summarize the performance. The function below, named `summarize_scores()`, will display the performance of a model as a single line for easy comparison with other models.

```
# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))
```

Listing 19.3: Example of a function for summarizing model performance.

We now have all of the elements to begin evaluating predictive models on the dataset.

19.5   CNNs for Multi-step Forecasting

Convolutional Neural Network models, or CNNs for short, can be used for multi-step time series forecasting. For more details on the use of CNNs for multi-step forecasting, see Chapter 8. CNNs can be used in either a recursive or direct forecast strategy, where the model makes one-step predictions and outputs are fed as inputs for subsequent predictions, and where one model is developed for each time step to be predicted. Alternately, CNNs can be used to predict the entire output sequence as a one-step prediction of the entire vector. This is a general benefit of feedforward neural networks. An important secondary benefit of using CNNs is that they can support multiple 1D inputs in order to make a prediction. This is useful if the multi-step output sequence is a function of more than one input sequence. This can be achieved using two different model configurations.

- **Multiple Input Channels**. This is where each input sequence is read as a separate channel, like the different channels of an image (e.g. red, green and blue).

- **Multiple Input Heads**. This is where each input sequence is read by a different CNN submodel and the internal representations are combined before being interpreted and used to make a prediction.

In this tutorial, we will explore how to develop three different types of CNN models for multi-step time series forecasting; they are:

- A CNN for multi-step time series forecasting with univariate input data.

- A CNN for multi-step time series forecasting with multivariate input data via channels.

- A CNN for multi-step time series forecasting with multivariate input data via submodels.

The models will be developed and demonstrated on the household power prediction problem. A model is considered skillful if it achieves performance better than a naive model, which is an overall RMSE of about 465 kilowatts across a seven day forecast (for more details of the naive model, see Chapter 17). We will not focus on the tuning of these models to achieve optimal performance; instead we will sill stop short at skillful models as compared to a naive forecast. The chosen structures and hyperparameters are chosen with a little trial and error. Given the stochastic nature of the models, it is good practice to evaluate a given model multiple times and report the mean performance on a test dataset. In the interest of brevity and keeping the code simple, we will instead present single-runs of models in this tutorial.

19.6   Univariate CNN Model

In this section, we will develop a convolutional neural network for multi-step time series forecasting using only the univariate sequence of daily power consumption. Specifically, the framing of the problem is: *Given some number of prior days of total daily power consumption, predict the next standard week of daily power consumption.* The number of prior days used as input defines the one-dimensional (1D) subsequence of data that the CNN will read and learn to extract features. Some ideas on the size and nature of this input include:

- All prior days, up to years worth of data.

- The prior seven days.

- The prior two weeks.

- The prior one month.

- The prior one year.

- The prior week and the week to be predicted from one year ago.

There is no right answer; instead, each approach and more can be tested and the performance of the model can be used to choose the nature of the input that results in the best model performance. These choices define a few things about the implementation, such as:

- How the training data must be prepared in order to fit the model.

- How the test data must be prepared in order to evaluate the model.

- How to use the model to make predictions with a final model in the future.

A good starting point would be to use the prior seven days. A 1D CNN model expects data to have the shape of: `[samples, timesteps, features]`. One sample will be comprised of seven time steps with one feature for the seven days of total daily power consumed. The training dataset has 159 weeks of data, so the shape of the training dataset would be: `[159, 7, 1]`.

This is a good start. The data in this format would use the prior standard week to predict the next standard week. A problem is that 159 instances is not a lot for a neural network. A way to create a lot more training data is to change the problem during training to predict the next seven days given the prior seven days, regardless of the standard week. This only impacts the training data, the test problem remains the same: predict the daily power consumption for the next standard week given the prior standard week. This will require a little preparation of the training data. The training data is provided in standard weeks with eight variables, specifically in the shape `[159, 7, 8]`. The first step is to flatten the data so that we have eight time series sequences.

```
# flatten data
data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
```

Listing 19.4: Example of flattened weekly data.

We then need to iterate over the time steps and divide the data into overlapping windows; each iteration moves along one time step and predicts the subsequent seven days. For example:

```
Input,                              Output
[d01, d02, d03, d04, d05, d06, d07], [d08, d09, d10, d11, d12, d13, d14]
[d02, d03, d04, d05, d06, d07, d08], [d09, d10, d11, d12, d13, d14, d15]
...
```

Listing 19.5: Example of overlapping weekly data.

We can do this by keeping track of start and end indexes for the inputs and outputs as we iterate across the length of the flattened data in terms of time steps. We can also do this in a way where the number of inputs and outputs are parameterized (e.g. n_input, n_out) so that you can experiment with different values or adapt it for your own problem. Below is a function named to_supervised() that takes a list of weeks (history) and the number of time steps to use as inputs and outputs and returns the data in the overlapping moving window format.

```python
# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)
```

Listing 19.6: Example of a function for creating overlapping windows of data.

When we run this function on the entire training dataset, we transform 159 samples into 1,099; specifically, the transformed dataset has the shapes X=[1099, 7, 1] and y=[1099, 7]. Next, we can define and fit the CNN model on the training data. This multi-step time series forecasting problem is an autoregression. That means it is likely best modeled where that the next seven days is some function of observations at prior time steps. This and the relatively small amount of data means that a small model is required. We will use a model with one convolution layer with 16 filters and a kernel size of 3. This means that the input sequence of seven days will be read with a convolutional operation three time steps at a time and this operation will be performed 16 times. A pooling layer will reduce these feature maps by $\frac{1}{4}$ their size before the internal representation is flattened to one long vector. This is then interpreted by a fully connected layer before the output layer predicts the next seven days in the sequence.

We will use the mean squared error loss function as it is a good match for our chosen error metric of RMSE. We will use the efficient Adam implementation of stochastic gradient descent and fit the model for 20 epochs with a batch size of 4. The small batch size and the stochastic nature of the algorithm means that the same model will learn a slightly different mapping of

inputs to outputs each time it is trained. This means results may vary when the model is evaluated. You can try running the model multiple times and calculating an average of model performance. The `build_model()` below prepares the training data, defines the model, and fits the model on the training data, returning the fit model ready for making predictions.

```python
# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 20, 4
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # define model
  model = Sequential()
  model.add(Conv1D(filters=16, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(10, activation='relu'))
  model.add(Dense(n_outputs))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model
```

Listing 19.7: Example of a function for fitting a CNN model.

Now that we know how to fit the model, we can look at how the model can be used to make a prediction. Generally, the model expects data to have the same three dimensional shape when making a prediction. In this case, the expected shape of an input pattern is one sample, seven days of one feature for the daily power consumed: `[1, 7, 1]`. Data must have this shape when making predictions for the test set and when a final model is being used to make predictions in the future. If you change the number of input days to 14, then the shape of the training data and the shape of new samples when making predictions must be changed accordingly to have 14 time steps. It is a modeling choice that you must carry forward when using the model.

We are using walk-forward validation to evaluate the model as described in the previous section. This means that we have the observations available for the prior week in order to predict the coming week. These are collected into an array of standard weeks, called history. In order to predict the next standard week, we need to retrieve the last days of observations. As with the training data, we must first flatten the history data to remove the weekly structure so that we end up with eight parallel time series.

```python
# flatten data
data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
```

Listing 19.8: Example of flattened weekly data.

Next, we need to retrieve the last seven days of daily total power consumed (feature number 0). We will parameterize as we did for the training data so that the number of prior days used as input by the model can be modified in the future.

```python
# retrieve last observations for input data
input_x = data[-n_input:, 0]
```

Listing 19.9: Example of retrieving the required input data.

Next, we reshape the input into the expected three-dimensional structure.

```
# reshape into [1, n_input, 1]
input_x = input_x.reshape((1, len(input_x), 1))
```

Listing 19.10: Example of reshaping input data.

We then make a prediction using the fit model and the input data and retrieve the vector of seven days of output.

```
# forecast the next week
yhat = model.predict(input_x, verbose=0)
# we only want the vector forecast
yhat = yhat[0]
```

Listing 19.11: Example of making a single one-week prediction.

The `forecast()` function below implements this and takes as arguments the model fit on the training dataset, the history of data observed so far, and the number of inputs time steps expected by the model.

```
# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [1, n_input, 1]
  input_x = input_x.reshape((1, len(input_x), 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat
```

Listing 19.12: Example of a function for making a multi-step forecast with a CNN model.

That's it; we now have everything we need to make multi-step time series forecasts with a CNN model on the daily total power consumed univariate dataset. We can tie all of this together. The complete example is listed below.

```
# univariate multi-step cnn for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
```

```
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores

# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)

# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
```

```python
  verbose, epochs, batch_size = 0, 20, 4
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # define model
  model = Sequential()
  model.add(Conv1D(filters=16, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(10, activation='relu'))
  model.add(Dense(n_outputs))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [1, n_input, 1]
  input_x = input_x.reshape((1, len(input_x), 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# evaluate model and get scores
```

```
n_input = 7
score, scores = evaluate_model(train, test, n_input)
# summarize scores
summarize_scores('cnn', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='cnn')
pyplot.show()
```

Listing 19.13: Example of evaluating a univariate CNN model for multi-step forecasting.

Running the example fits and evaluates the model, printing the overall RMSE across all seven days, and the per-day RMSE for each lead time. We can see that in this case, the model was skillful as compared to a naive forecast, achieving an overall RMSE of about 404 kilowatts, less than 465 kilowatts achieved by a naive model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
cnn: [404.411] 436.1, 400.6, 346.2, 388.2, 405.5, 326.0, 502.9
```

Listing 19.14: Sample output from evaluating a univariate CNN model for multi-step forecasting.

A plot of the daily RMSE is also created. The plot shows that perhaps Tuesdays and Fridays are easier days to forecast than the other days and that perhaps Saturday at the end of the standard week is the hardest day to forecast.
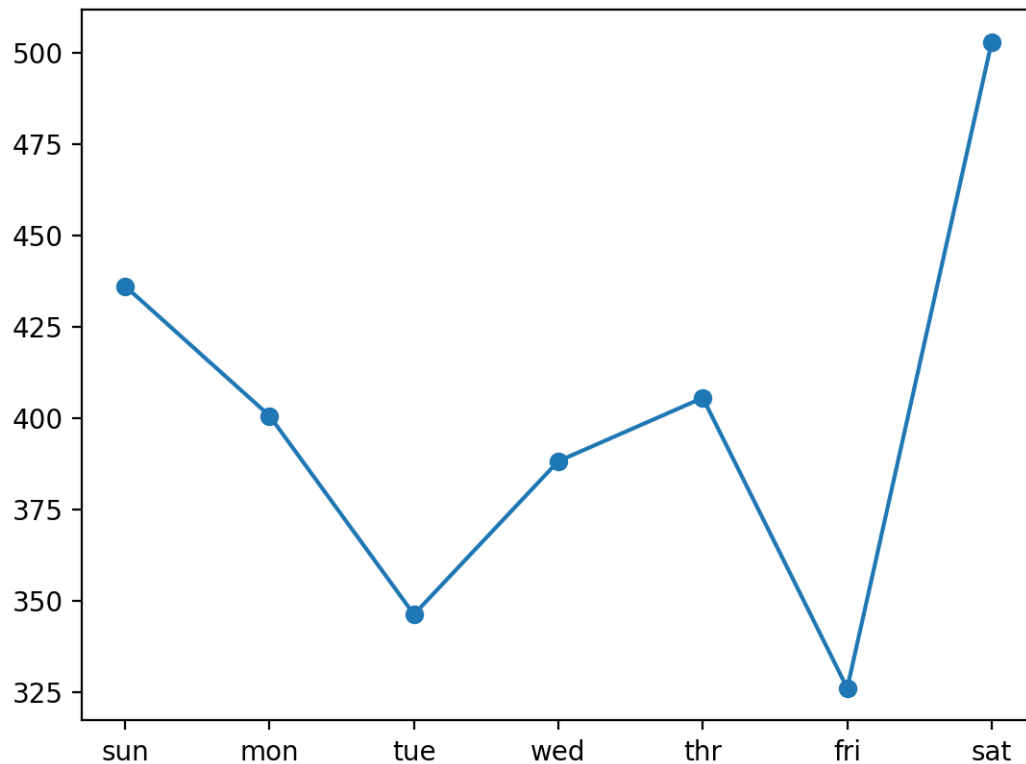
Figure 19.1: Line Plot of RMSE per Day for Univariate CNN with 7-day Inputs.

We can increase the number of prior days to use as input from seven to 14 by changing the `n_input` variable.

```
# evaluate model and get scores
n_input = 14
```

Listing 19.15: Example of changing the size of the input for making a forecast.

Re-running the example with this change first prints a summary of the performance of the model. In this case, we can see a further drop in the overall RMSE, suggesting that further tuning of the input size and perhaps the kernel size of the model may result in better performance.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
cnn: [396.497] 392.2, 412.8, 384.0, 389.0, 387.3, 381.0, 427.1
```

Listing 19.16: Sample output from evaluating the updated CNN model for multi-step forecasting.

Comparing the per-day RMSE scores, we see some are better and some are worse than using seventh inputs. This may suggest a benefit in using the two different sized inputs in some way, such as an ensemble of the two approaches or perhaps a single model (e.g. a multi-headed model) that reads the training data in different ways.
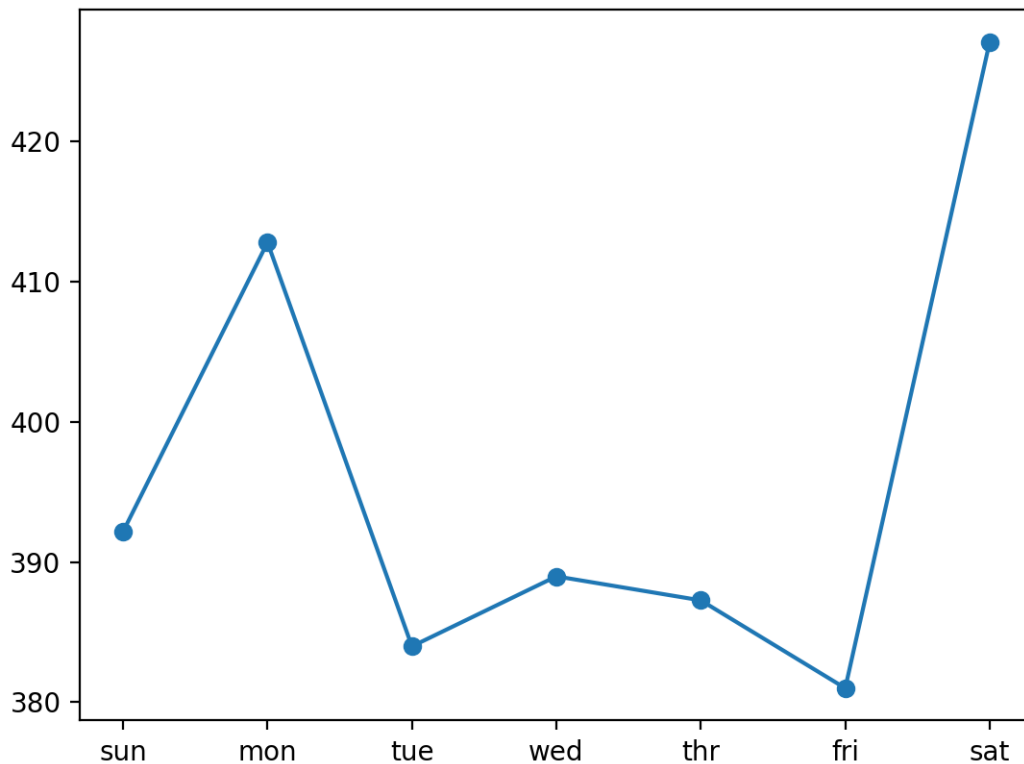
Figure 19.2: Line Plot of RMSE per Day for Univariate CNN with 14-day Inputs.

19.7   Multi-channel CNN Model

In this section, we will update the CNN developed in the previous section to use each of the eight time series variables to predict the next standard week of daily total power consumption. We will do this by providing each one-dimensional time series to the model as a separate channel of input. The CNN will then use a separate kernel and read each input sequence onto a separate set of filter maps, essentially learning features from each input time series variable. This is helpful for those problems where the output sequence is some function of the observations at prior time steps from multiple different features, not just (or including) the feature being forecasted. It is unclear whether this is the case in the power consumption problem, but we can explore it nonetheless. First, we must update the preparation of the training data to include all of the eight features, not just the one total daily power consumed. It requires a single line:

```
# use all variables in input samples
X.append(data[in_start:in_end, :])
```

Listing 19.17: Example of data preparation using all data variables.

The complete `to_supervised()` function with this change is listed below.

```
# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
```

```
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      X.append(data[in_start:in_end, :])
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)
```

Listing 19.18: Example of a function for creating overlapping windows of data with all variables.

We also must update the function used to make forecasts with the fit model to use all eight features from the prior time steps. Again, another small change:

```
# retrieve last observations for input data
input_x = data[-n_input:, :]
# reshape into [1, n_input, n]
input_x = input_x.reshape((1, input_x.shape[0], input_x.shape[1]))
```

Listing 19.19: Example of using all variables as input when making a forecast.

The complete `forecast()` with this change is listed below:

```
# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, :]
  # reshape into [1, n_input, n]
  input_x = input_x.reshape((1, input_x.shape[0], input_x.shape[1]))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat
```

Listing 19.20: Example of a function for making a multi-step forecast with a CNN model and all input variables.

We will use 14 days of prior observations across eight of the input variables as we did in the final section of the prior section that resulted in slightly better performance.

```
# evaluate model and get scores
n_input = 14
```

Listing 19.21: Example of using two weeks of data as input.

Finally, the model used in the previous section does not perform well on this new framing of the problem. The increase in the amount of data requires a larger and more sophisticated model that is trained for longer. With a little trial and error, one model that performs well uses two convolutional layers with 32 filter maps followed by pooling, then another convolutional layer with 16 feature maps and pooling. The fully connected layer that interprets the features is increased to 100 nodes and the model is fit for 70 epochs with a batch size of 16 samples. The updated `build_model()` function that defines and fits the model on the training dataset is listed below.

```python
# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 70, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # define model
  model = Sequential()
  model.add(Conv1D(filters=32, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Conv1D(filters=16, kernel_size=3, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model
```

Listing 19.22: Example of a function for fitting a multi-channel CNN model.

We now have all of the elements required to develop a multi-channel CNN for multivariate input data to make multi-step time series forecasts. The complete example is listed below.

```python
# multichannel multi-step cnn for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
```

```python
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores

# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      X.append(data[in_start:in_end, :])
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)

# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 70, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # define model
  model = Sequential()
  model.add(Conv1D(filters=32, kernel_size=3, activation='relu',
```

```
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=32, kernel_size=3, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Conv1D(filters=16, kernel_size=3, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, :]
  # reshape into [1, n_input, n]
  input_x = input_x.reshape((1, input_x.shape[0], input_x.shape[1]))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# evaluate model and get scores
n_input = 14
score, scores = evaluate_model(train, test, n_input)
```

```
# summarize scores
summarize_scores('cnn', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='cnn')
pyplot.show()
```

Listing 19.23: Example of evaluating a multi-channel CNN model for multi-step forecasting.

Running the example fits and evaluates the model, printing the overall RMSE across all seven days, and the per-day RMSE for each lead time. We can see that in this case, the use of all eight input variables does result in another small drop in the overall RMSE score.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
cnn: [385.711] 422.2, 363.5, 349.8, 393.1, 357.1, 318.8, 474.3
```

Listing 19.24: Sample output from evaluating a multi-channel CNN model for multi-step forecasting.

For the daily RMSE scores, we do see that some are better and some are worse than the univariate CNN from the previous section. The final day, Saturday, remains a challenging day to forecast, and Friday an easy day to forecast. There may be some benefit in designing models to focus specifically on reducing the error of the harder to forecast days. It may be interesting to see if the variance across daily scores could be further reduced with a tuned model or perhaps an ensemble of multiple different models. It may also be interesting to compare the performance for a model that uses seven or even 21 days of input data to see if further gains can be made.
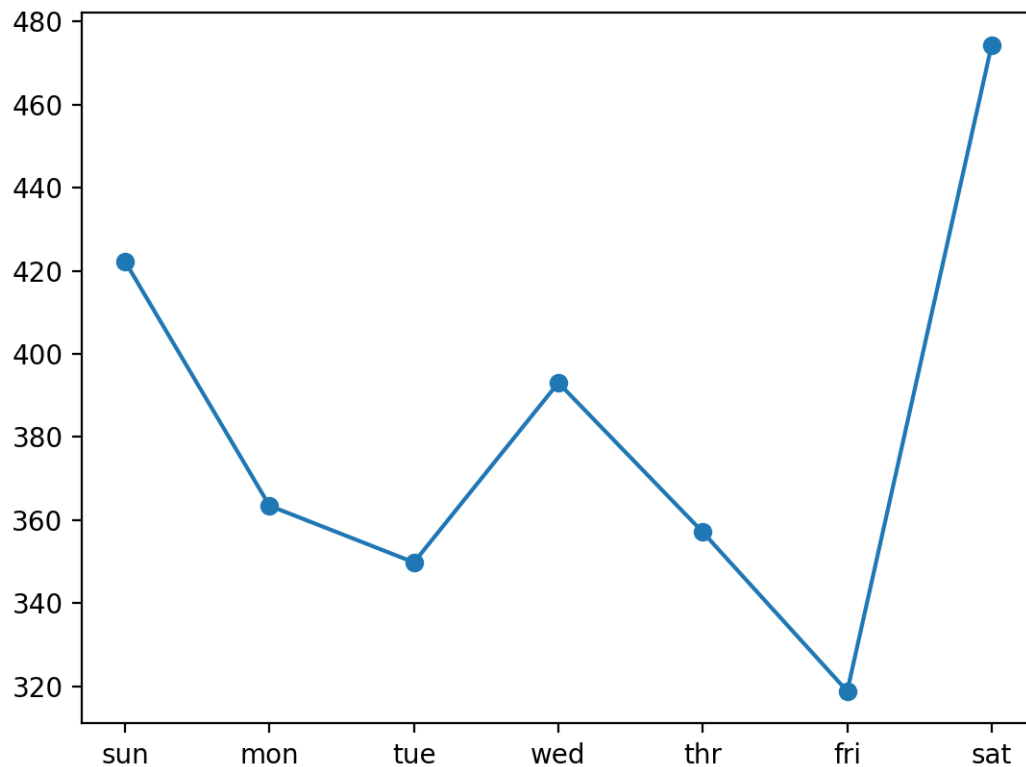
Figure 19.3: Line Plot of RMSE per Day for a Multi-channel CNN with 14-day Inputs.

19.8 Multi-headed CNN Model

We can further extend the CNN model to have a separate sub-CNN model or head for each input variable, which we can refer to as a multi-headed CNN model. This requires a modification to the preparation of the model, and in turn, modification to the preparation of the training and test datasets. Starting with the model, we must define a separate CNN model for each of the eight input variables. The configuration of the model, including the number of layers and their hyperparameters, were also modified to better suit the new approach. The new configuration is not optimal and was found with a little trial and error. The multi-headed model is specified using the more flexible functional API for defining Keras models.

We can loop over each variable and create a submodel that takes a one-dimensional sequence of 14 days of data and outputs a flat vector containing a summary of the learned features from the sequence. Each of these vectors can be merged via concatenation to make one very long vector that is then interpreted by some fully connected layers before a prediction is made. As we build up the submodels, we keep track of the input layers and flatten layers in lists. This is so that we can specify the inputs in the definition of the model object and use the list of flatten layers in the merge layer.

```
# create a channel for each variable
in_layers, out_layers = list(), list()
```

```
for i in range(n_features):
  inputs = Input(shape=(n_timesteps,1))
  conv1 = Conv1D(filters=32, kernel_size=3, activation='relu')(inputs)
  conv2 = Conv1D(filters=32, kernel_size=3, activation='relu')(conv1)
  pool1 = MaxPooling1D(pool_size=2)(conv2)
  flat = Flatten()(pool1)
  # store layers
  in_layers.append(inputs)
  out_layers.append(flat)
# merge heads
merged = concatenate(out_layers)
# interpretation
dense1 = Dense(200, activation='relu')(merged)
dense2 = Dense(100, activation='relu')(dense1)
outputs = Dense(n_outputs)(dense2)
model = Model(inputs=in_layers, outputs=outputs)
# compile model
model.compile(loss='mse', optimizer='adam')
```

Listing 19.25: Example of defining a multi-headed CNN model.

When the model is used, it will require eight arrays as input: one for each of the submodels. This is required when training the model, when evaluating the model, and when making predictions with a final model. We can achieve this by creating a list of 3D arrays, where each 3D array contains [`samples, timesteps, 1`], with one feature. We can prepare the training dataset in this format as follows:

```
# define list of input data
input_data = [train_x[:,:,i].reshape((train_x.shape[0],n_timesteps,1)) for i in
    range(n_features)]
```

Listing 19.26: Example of defining input where each variable is a separate array.

The updated `build_model()` function with these changes is listed below.

```
# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 25, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # create a channel for each variable
  in_layers, out_layers = list(), list()
  for i in range(n_features):
    inputs = Input(shape=(n_timesteps,1))
    conv1 = Conv1D(filters=32, kernel_size=3, activation='relu')(inputs)
    conv2 = Conv1D(filters=32, kernel_size=3, activation='relu')(conv1)
    pool1 = MaxPooling1D(pool_size=2)(conv2)
    flat = Flatten()(pool1)
    # store layers
    in_layers.append(inputs)
    out_layers.append(flat)
  # merge heads
  merged = concatenate(out_layers)
  # interpretation
  dense1 = Dense(200, activation='relu')(merged)
```

```
dense2 = Dense(100, activation='relu')(dense1)
outputs = Dense(n_outputs)(dense2)
model = Model(inputs=in_layers, outputs=outputs)
# compile model
model.compile(loss='mse', optimizer='adam')
# plot the model
plot_model(model, show_shapes=True, to_file='multiheaded_cnn.png')
# fit network
input_data = [train_x[:,:,i].reshape((train_x.shape[0],n_timesteps,1)) for i in
    range(n_features)]
model.fit(input_data, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
return model
```

Listing 19.27: Example of a function to define and fit a multi-headed CNN model.

When the model is built, a diagram of the structure of the model is created and saved to file. The structure of the network looks as follows.

Note: the call to `plot_model()` requires that pygraphviz and pydot are installed. If this is a problem, you can comment out this line.
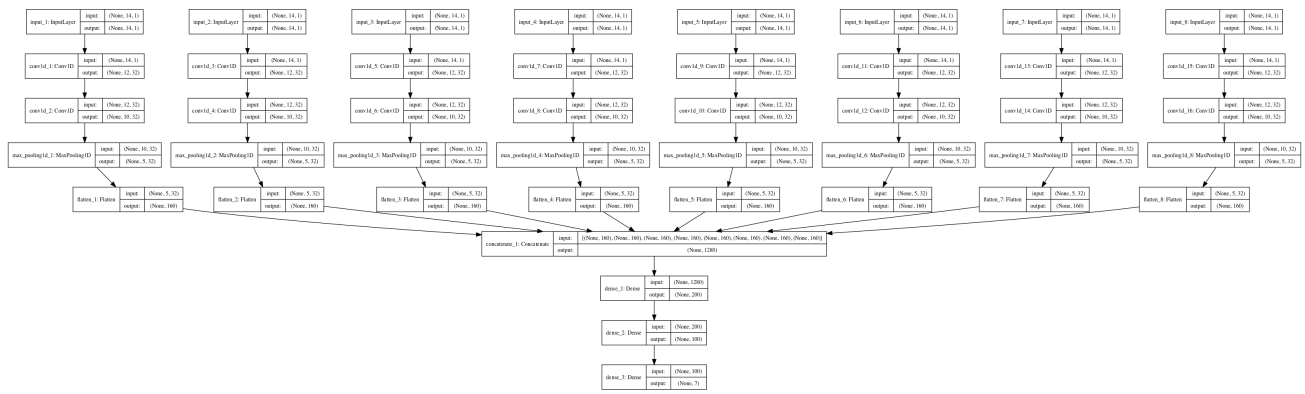

Figure 19.4: Structure of the Multi Headed Convolutional Neural Network.

Next, we can update the preparation of input samples when making a prediction for the test dataset. We must perform the same change, where an input array of [1, 14, 8] must be transformed into a list of eight 3D arrays each with [1, 14, 1].

```
# input for making a prediction
input_x = [input_x[:,i].reshape((1,input_x.shape[0],1)) for i in range(input_x.shape[1])]
```

Listing 19.28: Example of defining input where each variable is a separate array.

The `forecast()` function with this change is listed below.

```
# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
```

```
  input_x = data[-n_input:, :]
  # reshape into n input arrays
  input_x = [input_x[:,i].reshape((1,input_x.shape[0],1)) for i in range(input_x.shape[1])]
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat
```

Listing 19.29: Example of a function for making a forecast with a multi-headed CNN model.

That's it. We can tie all of this together; the complete example is listed below.

```
# multi headed multi-step cnn for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.utils.vis_utils import plot_model
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.models import Model
from keras.layers import Input
from keras.layers.merge import concatenate

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores
```

```python
# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      X.append(data[in_start:in_end, :])
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)

# plot training history
def plot_history(history):
  # plot loss
  pyplot.subplot(2, 1, 1)
  pyplot.plot(history.history['loss'], label='train')
  pyplot.plot(history.history['val_loss'], label='test')
  pyplot.title('loss', y=0, loc='center')
  pyplot.legend()
  # plot rmse
  pyplot.subplot(2, 1, 2)
  pyplot.plot(history.history['rmse'], label='train')
  pyplot.plot(history.history['val_rmse'], label='test')
  pyplot.title('rmse', y=0, loc='center')
  pyplot.legend()
  pyplot.show()

# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 25, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # create a channel for each variable
  in_layers, out_layers = list(), list()
  for _ in range(n_features):
    inputs = Input(shape=(n_timesteps,1))
    conv1 = Conv1D(filters=32, kernel_size=3, activation='relu')(inputs)
    conv2 = Conv1D(filters=32, kernel_size=3, activation='relu')(conv1)
    pool1 = MaxPooling1D(pool_size=2)(conv2)
    flat = Flatten()(pool1)
    # store layers
```

```python
    in_layers.append(inputs)
    out_layers.append(flat)
  # merge heads
  merged = concatenate(out_layers)
  # interpretation
  dense1 = Dense(200, activation='relu')(merged)
  dense2 = Dense(100, activation='relu')(dense1)
  outputs = Dense(n_outputs)(dense2)
  model = Model(inputs=in_layers, outputs=outputs)
  # compile model
  model.compile(loss='mse', optimizer='adam')
  # plot the model
  plot_model(model, show_shapes=True, to_file='multiheaded_cnn.png')
  # fit network
  input_data = [train_x[:,:,i].reshape((train_x.shape[0],n_timesteps,1)) for i in
      range(n_features)]
  model.fit(input_data, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, :]
  # reshape into n input arrays
  input_x = [input_x[:,i].reshape((1,input_x.shape[0],1)) for i in range(input_x.shape[1])]
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
```

```
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# evaluate model and get scores
n_input = 14
score, scores = evaluate_model(train, test, n_input)
# summarize scores
summarize_scores('cnn', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='cnn')
pyplot.show()
```

Listing 19.30: Example of evaluating a multi-headed CNN model for multi-step forecasting.

Running the example fits and evaluates the model, printing the overall RMSE across all seven days, and the per-day RMSE for each lead time. We can see that in this case, the overall RMSE is skillful compared to a naive forecast, but with the chosen configuration may not perform better than the multi-channel model in the previous section.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
cnn: [396.116] 414.5, 385.5, 377.2, 412.1, 371.1, 380.6, 428.1
```

Listing 19.31: Sample output from evaluating a multi-headed CNN model for multi-step forecasting.

We can also see a different, more pronounced profile for the daily RMSE scores where perhaps Mon-Tue and Thu-Fri are easier for the model to predict than the other forecast days. These results may be useful when combined with another forecast model. It may be interesting to explore alternate methods in the architecture for merging the output of each submodel.
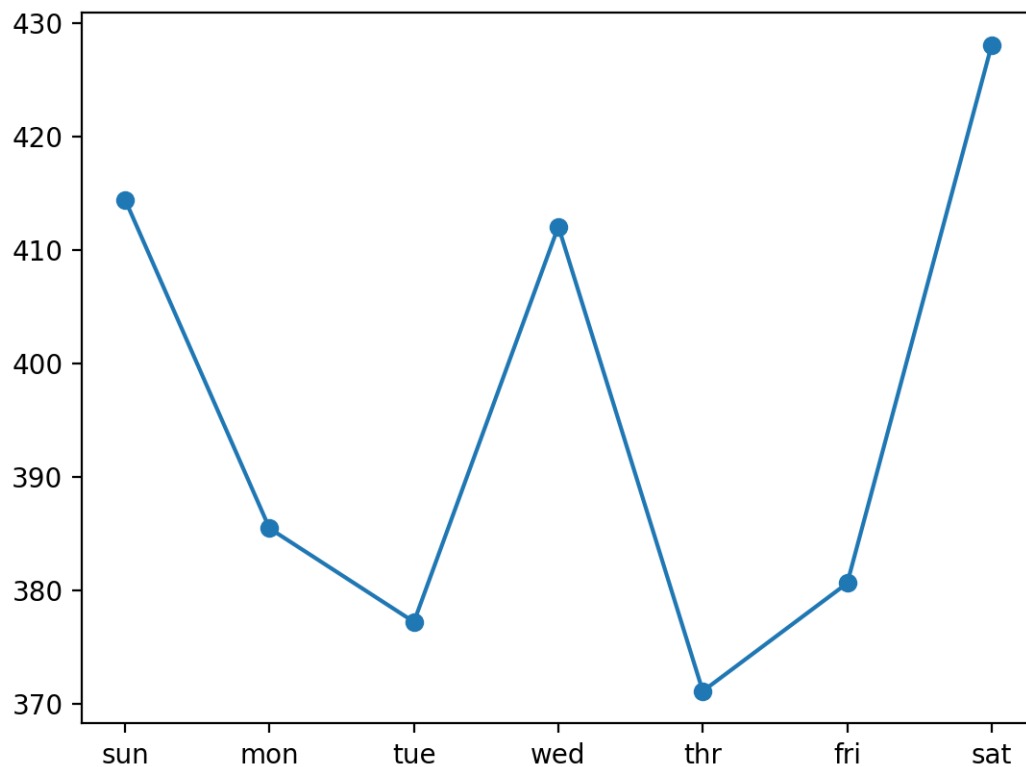
Figure 19.5: Line Plot of RMSE per Day for a Multi-head CNN with 14-day Inputs.

19.9 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Size of Input**. Explore more or fewer numbers of days used as input for the model, such as three days, 21 days, 30 days and more.

- **Model Tuning**. Tune the structure and hyperparameters for a model and further lift model performance on average.

- **Data Scaling**. Explore whether data scaling, such as standardization and normalization, can be used to improve the performance of any of the CNN models.

- **Learning Diagnostics**. Use diagnostics such as learning curves for the train and validation loss and mean squared error to help tune the structure and hyperparameters of a CNN model.

- **Vary Kernel Size**. Combine the multi-channel CNN with the multi-headed CNN and use a different kernel size for each head to see if this configuration can further improve performance.

If you explore any of these extensions, I'd love to know.

19.10 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/

- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

19.11 Summary

In this tutorial, you discovered how to develop 1D convolutional neural networks for multi-step time series forecasting. Specifically, you learned:

- How to develop a CNN for multi-step time series forecasting model for univariate data.

- How to develop a multi-channel multi-step time series forecasting model for multivariate data.

- How to develop a multi-headed multi-step time series forecasting model for multivariate data.

19.11.1 Next

In the next lesson, you will discover how to develop Recurrent Neural Network models for forecasting the household power usage problem.

Chapter 20

How to Develop LSTMs for Multi-step Energy Usage Forecasting

Given the rise of smart electricity meters and the wide adoption of electricity generation technology like solar panels, there is a wealth of electricity usage data available. This data represents a multivariate time series of power-related variables that in turn could be used to model and even forecast future electricity consumption. Unlike other machine learning algorithms, long short-term memory recurrent neural networks are capable of automatically learning features from sequence data, support multiple-variate data, and can output a variable length sequences that can be used for multi-step forecasting. In this tutorial, you will discover how to develop long short-term memory recurrent neural networks for multi-step time series forecasting of household power consumption. After completing this tutorial, you will know:

- How to develop and evaluate Univariate and multivariate Encoder-Decoder LSTMs for multi-step time series forecasting.

- How to develop and evaluate an CNN-LSTM Encoder-Decoder model for multi-step time series forecasting.

- How to develop and evaluate a ConvLSTM Encoder-Decoder model for multi-step time series forecasting.

Let's get started.

20.1 Tutorial Overview

This tutorial is divided into nine parts; they are:

1. Problem Description

2. Load and Prepare Dataset

3. Model Evaluation

4. LSTMs for Multi-step Forecasting

5. Univariate Input and Vector Output

6. Encoder-Decoder LSTM With Univariate Input

7. Encoder-Decoder LSTM With Multivariate Input

8. CNN-LSTM Encoder-Decoder With Univariate Input

9. ConvLSTM Encoder-Decoder With Univariate Input

20.2 Problem Description

The *Household Power Consumption* dataset is a multivariate time series dataset that describes the electricity consumption for a single household over four years. The data was collected between December 2006 and November 2010 and observations of power consumption within the household were collected every minute. It is a multivariate series comprised of seven variables (besides the date and time); they are:

- `global_active_power`: The total active power consumed by the household (kilowatts).

- `global_reactive_power`: The total reactive power consumed by the household (kilowatts).

- `voltage`: Average voltage (volts).

- `global_intensity`: Average current intensity (amps).

- `sub_metering_1`: Active energy for kitchen (watt-hours of active energy).

- `sub_metering_2`: Active energy for laundry (watt-hours of active energy).

- `sub_metering_3`: Active energy for climate control systems (watt-hours of active energy).

Active and reactive energy refer to the technical details of alternative current. A fourth sub-metering variable can be created by subtracting the sum of three defined sub-metering variables from the total active energy. This dataset was introduced and analyzed in Chapter 16. Refer to that chapter for more details if needed.

20.3 Load and Prepare Dataset

We will use the same framework to load and prepare the data as was used for the naive models. In the interest of brevity, refer to Chapter 17 for the details on how to load and prepare the dataset for modeling.

20.4 Model Evaluation

We will use the same framework to evaluate models as was used for the naive models. In the interest of brevity, refer to Chapter 17 for the details on how to develop a framework for evaluating forecasts for this dataset. We must update the framework for model evaluation, specifically the walk-forward validation method used to fit a LSTM model and make a forecast. Refer to Chapter 19 for the details of the development of the new walk-forward validation framework for evaluating deep learning models on this problem.

20.5 LSTMs for Multi-step Forecasting

Recurrent neural networks, or RNNs, are specifically designed to work, learn, and predict sequence data and can be used for multi-step time series forecasting. For more details on the use of LSTMs for multi-step forecasting, see Chapter 9. In this tutorial, we will explore a suite of LSTM architectures for multi-step time series forecasting. Specifically, we will look at how to develop the following models:

- **Vanilla LSTM** model with vector output for multi-step forecasting with univariate input data.

- **Encoder-Decoder LSTM** model for multi-step forecasting with univariate input data.

- **Encoder-Decoder LSTM** model for multi-step forecasting with multivariate input data.

- **CNN-LSTM Encoder-Decoder** model for multi-step forecasting with univariate input data.

- **ConvLSTM Encoder-Decoder** model for multi-step forecasting with univariate input data.

The models will be developed and demonstrated on the household power prediction problem. A model is considered skillful if it achieves performance better than a naive model, which is an overall RMSE of about 465 kilowatts across a seven day forecast (for more details of the naive model, see Chapter 17). We will not focus on the tuning of these models to achieve optimal performance; instead, we will stop short at skillful models as compared to a naive forecast. The chosen structures and hyperparameters are chosen with a little trial and error. The scores should be taken as just an example rather than a study of the optimal model or configuration for the problem.

Given the stochastic nature of the models, it is good practice to evaluate a given model multiple times and report the mean performance on a test dataset. In the interest of brevity and keeping the code simple, we will instead present single-runs of models in this tutorial. We cannot know which approach will be the most effective for a given multi-step forecasting problem. It is a good idea to explore a suite of methods in order to discover what works best on your specific dataset.

20.6 Univariate Input and Vector Output

We will start off by developing a simple or vanilla LSTM model that reads in a sequence of days of total daily power consumption and predicts a vector output of the next standard week of daily power consumption. This will provide the foundation for the more elaborate models developed in subsequent sections. The number of prior days used as input defines the one-dimensional (1D) subsequence of data that the LSTM will read and learn to extract features. Some ideas on the size and nature of this input include:

- All prior days, up to years worth of data.

- The prior seven days.

- The prior two weeks.

- The prior one month.

- The prior one year.

- The prior week and the week to be predicted from one year ago.

There is no right answer; instead, each approach and more can be tested and the performance of the model can be used to choose the nature of the input that results in the best model performance. These choices define a few things:

- How the training data must be prepared in order to fit the model.

- How the test data must be prepared in order to evaluate the model.

- How to use the model to make predictions with a final model in the future.

A good starting point would be to use the prior seven days. An LSTM model expects data to have the shape: `[samples, timesteps, features]`. One sample will be comprised of seven time steps with one feature for the seven days of total daily power consumed. The training dataset has 159 weeks of data, so the shape of the training dataset would be: `[159, 7, 1]`.

This is a good start. The data in this format would use the prior standard week to predict the next standard week. A problem is that 159 instances is not a lot to train a neural network. A way to create a lot more training data is to change the problem during training to predict the next seven days given the prior seven days, regardless of the standard week. This only impacts the training data, and the test problem remains the same: predict the daily power consumption for the next standard week given the prior standard week. This will require a little preparation of the training data. The training data is provided in standard weeks with eight variables, specifically in the shape `[159, 7, 8]`. The first step is to flatten the data so that we have eight time series sequences.

```
# flatten data
data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
```

Listing 20.1: Example of flattened weekly data.

We then need to iterate over the time steps and divide the data into overlapping windows; each iteration moves along one time step and predicts the subsequent seven days. For example:

```
Input,                              Output
[d01, d02, d03, d04, d05, d06, d07], [d08, d09, d10, d11, d12, d13, d14]
[d02, d03, d04, d05, d06, d07, d08], [d09, d10, d11, d12, d13, d14, d15]
...
```

Listing 20.2: Example of overlapping weekly data.

We can do this by keeping track of start and end indexes for the inputs and outputs as we iterate across the length of the flattened data in terms of time steps. We can also do this in a way where the number of inputs and outputs are parameterized (e.g. `n_input`, `n_out`) so that you can experiment with different values or adapt it for your own problem. Below is a function named `to_supervised()` that takes a list of weeks (history) and the number of time steps to use as inputs and outputs and returns the data in the overlapping moving window format.

```
# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)
```

Listing 20.3: Example of a function for creating overlapping windows of data.

When we run this function on the entire training dataset, we transform 159 samples into 1,099; specifically, the transformed dataset has the shapes X=[1099, 7, 1] and y=[1099, 7]. Next, we can define and fit the LSTM model on the training data. This multi-step time series forecasting problem is an autoregression. That means it is likely best modeled where that the next seven days is some function of observations at prior time steps. This and the relatively small amount of data means that a small model is required.

We will develop a model with a single hidden LSTM layer with 200 units. The number of units in the hidden layer is unrelated to the number of time steps in the input sequences. The LSTM layer is followed by a fully connected layer with 200 nodes that will interpret the features learned by the LSTM layer. Finally, an output layer will directly predict a vector with seven elements, one for each day in the output sequence. We will use the mean squared error loss function as it is a good match for our chosen error metric of RMSE. We will use the efficient Adam implementation of stochastic gradient descent and fit the model for 70 epochs with a batch size of 16.

The small batch size and the stochastic nature of the algorithm means that the same model will learn a slightly different mapping of inputs to outputs each time it is trained. This means results may vary when the model is evaluated. You can try running the model multiple times and calculate an average of model performance. The `build_model()` below prepares the training data, defines the model, and fits the model on the training data, returning the fit model ready for making predictions.

```
# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 70, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # define model
```

```
model = Sequential()
model.add(LSTM(200, activation='relu', input_shape=(n_timesteps, n_features)))
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs))
model.compile(loss='mse', optimizer='adam')
# fit network
model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
return model
```

Listing 20.4: Example of a function for fitting an LSTM model.

Now that we know how to fit the model, we can look at how the model can be used to make a prediction. Generally, the model expects data to have the same three dimensional shape when making a prediction. In this case, the expected shape of an input pattern is one sample, seven days of one feature for the daily power consumed: `[1, 7, 1]`. Data must have this shape when making predictions for the test set and when a final model is being used to make predictions in the future. If you change the number if input days to 14, then the shape of the training data and the shape of new samples when making predictions must be changed accordingly to have 14 time steps. It is a modeling choice that you must carry forward when using the model.

We are using walk-forward validation to evaluate the model as described in the previous section. This means that we have the observations available for the prior week in order to predict the coming week. These are collected into an array of standard weeks called history. In order to predict the next standard week, we need to retrieve the last days of observations. As with the training data, we must first flatten the history data to remove the weekly structure so that we end up with eight parallel time series.

```
# flatten data
data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
```

Listing 20.5: Example of flattened weekly data.

Next, we need to retrieve the last seven days of daily total power consumed (feature index 0). We will parameterize this as we did for the training data so that the number of prior days used as input by the model can be modified in the future.

```
# retrieve last observations for input data
input_x = data[-n_input:, 0]
```

Listing 20.6: Example of retrieving the required input data.

Next, we reshape the input into the expected three-dimensional structure.

```
# reshape into [1, n_input, 1]
input_x = input_x.reshape((1, len(input_x), 1))
```

Listing 20.7: Example of reshaping input data.

We then make a prediction using the fit model and the input data and retrieve the vector of seven days of output.

```
# forecast the next week
yhat = model.predict(input_x, verbose=0)
# we only want the vector forecast
yhat = yhat[0]
```

Listing 20.8: Example of making a single one-week prediction.

The `forecast()` function below implements this and takes as arguments the model fit on the training dataset, the history of data observed so far, and the number of input time steps expected by the model.

```python
# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [1, n_input, 1]
  input_x = input_x.reshape((1, len(input_x), 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat
```

Listing 20.9: Example of a function for making a multi-step forecast with an LSTM model.

That's it; we now have everything we need to make multi-step time series forecasts with an LSTM model on the daily total power consumed univariate dataset. We can tie all of this together. The complete example is listed below.

```python
# univariate multi-step lstm for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
```

```python
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores

# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)

# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 70, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # define model
  model = Sequential()
  model.add(LSTM(200, activation='relu', input_shape=(n_timesteps, n_features)))
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
```

```python
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [1, n_input, 1]
  input_x = input_x.reshape((1, len(input_x), 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# evaluate model and get scores
n_input = 7
score, scores = evaluate_model(train, test, n_input)
# summarize scores
summarize_scores('lstm', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='lstm')
pyplot.show()
```

Listing 20.10: Example of evaluating a univariate LSTM model for multi-step forecasting.

Running the example fits and evaluates the model, printing the overall RMSE across all seven days, and the per-day RMSE for each lead time. We can see that in this case, the model was skillful as compared to a naive forecast, achieving an overall RMSE of about 399 kilowatts, less than 465 kilowatts achieved by a naive model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
lstm: [399.456] 419.4, 422.1, 384.5, 395.1, 403.9, 317.7, 441.5
```

Listing 20.11: Sample output from evaluating a univariate LSTM model for multi-step forecasting.

A plot of the daily RMSE is also created. The plot shows that perhaps Tuesdays and Fridays are easier days to forecast than the other days and that perhaps Saturday at the end of the standard week is the hardest day to forecast.
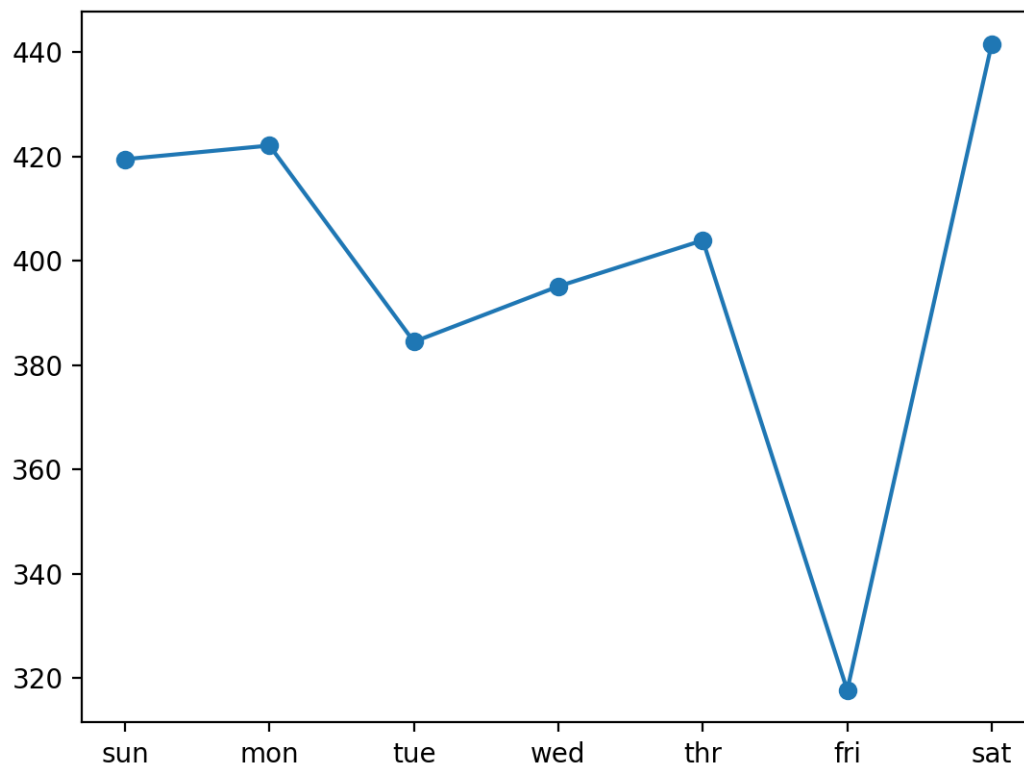

Figure 20.1: Line Plot of RMSE per Day for Univariate LSTM with Vector Output and 7-day Inputs.

We can increase the number of prior days to use as input from seven to 14 by changing the `n_input` variable.

```
# evaluate model and get scores
n_input = 14
```

Listing 20.12: Example of changing the size of the input for making a forecast.

Re-running the example with this change first prints a summary of performance of the model. In this case, we can see a further drop in the overall RMSE to about 370 kilowatts, suggesting that further tuning of the input size and perhaps the number of nodes in the model may result in better performance.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
lstm: [370.028] 387.4, 377.9, 334.0, 371.2, 367.1, 330.4, 415.1
```

Listing 20.13: Sample output from evaluating the updated univariate LSTM model for multi-step forecasting.

Comparing the per-day RMSE scores we see some are better and some are worse than using seven-day inputs. This may suggest benefit in using the two different sized inputs in some way, such as an ensemble of the two approaches or perhaps a single model (e.g. a multi-headed model) that reads the training data in different ways.
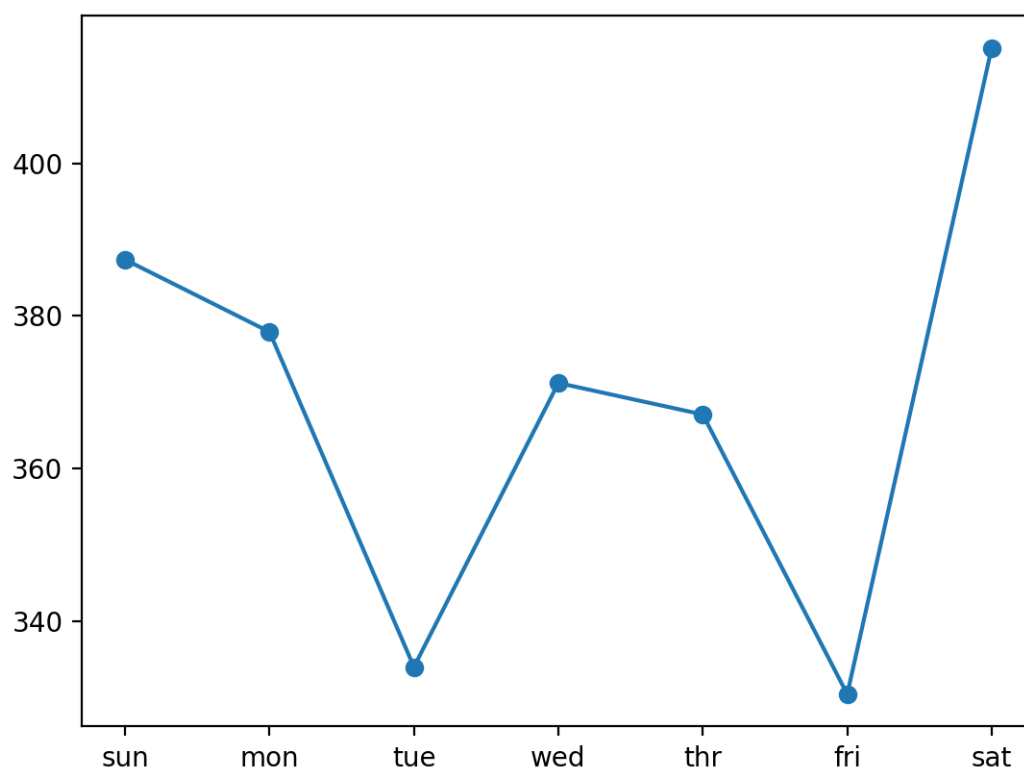

Figure 20.2: Line Plot of RMSE per Day for Univariate LSTM with Vector Output and 14-day Inputs.

20.7 Encoder-Decoder LSTM With Univariate Input

In this section, we can update the vanilla LSTM to use an encoder-decoder model. This means that the model will not output a vector sequence directly. Instead, the model will be comprised of two sub models, the encoder to read and encode the input sequence, and the decoder that

will read the encoded input sequence and make a one-step prediction for each element in the output sequence.

The difference is subtle, as in practice both approaches do in fact predict a sequence output. The important difference is that an LSTM model is used in the decoder, allowing it to both know what was predicted for the prior day in the sequence and accumulate internal state while outputting the sequence. Let's take a closer look at how this model is defined.

As before, we define an LSTM hidden layer with 200 units. This is the decoder model that will read the input sequence and will output a 200 element vector (one output per unit) that captures features from the input sequence. We will use 14 days of total power consumption as input.

```
# define encoder
model.add(LSTM(200, activation='relu', input_shape=(n_timesteps, n_features)))
```

Listing 20.14: Example of defining an encoder model.

We will use a simple encoder-decoder architecture that is easy to implement in Keras, that has a lot of similarity to the architecture of an LSTM autoencoder. First, the internal representation of the input sequence is repeated multiple times, once for each time step in the output sequence. This sequence of vectors will be presented to the LSTM decoder.

```
# repeat encodering
model.add(RepeatVector(7))
```

Listing 20.15: Example of repeating the output of the encoder.

We then define the decoder as an LSTM hidden layer with 200 units. Importantly, the decoder will output the entire sequence, not just the output at the end of the sequence as we did with the encoder. This means that each of the 200 units will output a value for each of the seven days, representing the basis for what to predict for each day in the output sequence.

```
# define decoder model
model.add(LSTM(200, activation='relu', return_sequences=True))
```

Listing 20.16: Example of defining the decoder model.

We will then use a fully connected layer to interpret each time step in the output sequence before the final output layer. Importantly, the output layer predicts a single step in the output sequence, not all seven days at a time. This means that we will use the same layers applied to each step in the output sequence. It means that the same fully connected layer and output layer will be used to process each time step provided by the decoder. To achieve this, we will wrap the interpretation layer and the output layer in a `TimeDistributed` wrapper that allows the wrapped layers to be used for each time step from the decoder.

```
# define output model
model.add(TimeDistributed(Dense(100, activation='relu')))
model.add(TimeDistributed(Dense(1)))
```

Listing 20.17: Example of defining the output model.

This allows the LSTM decoder to figure out the context required for each step in the output sequence and the wrapped dense layers to interpret each time step separately, yet reusing the same weights to perform the interpretation. An alternative would be to flatten all of the structure created by the LSTM decoder and to output the vector directly. You can try this as

an extension to see how it compares. The network therefore outputs a three-dimensional vector with the same structure as the input, with the dimensions `[samples, timesteps, features]`.

There is a single feature, the daily total power consumed, and there are always seven features. A single one-week prediction will therefore have the size: `[1, 7, 1]`. Therefore, when training the model, we must restructure the output data (y) to have the three-dimensional structure instead of the two-dimensional structure of `[samples, features]` used in the previous section.

```python
# reshape output into [samples, timesteps, features]
train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
```

Listing 20.18: Example of reshaping output data for training.

We can tie all of this together into the updated `build_model()` function listed below.

```python
# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 20, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # reshape output into [samples, timesteps, features]
  train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
  # define model
  model = Sequential()
  model.add(LSTM(200, activation='relu', input_shape=(n_timesteps, n_features)))
  model.add(RepeatVector(n_outputs))
  model.add(LSTM(200, activation='relu', return_sequences=True))
  model.add(TimeDistributed(Dense(100, activation='relu')))
  model.add(TimeDistributed(Dense(1)))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model
```

Listing 20.19: Example of a function for defining and fitting an encoder-decoder LSTM model.

The complete example with the encoder-decoder model is listed below.

```python
# univariate multi-step encoder-decoder lstm for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
```

```
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores

# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)

# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 20, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
```

```python
  # reshape output into [samples, timesteps, features]
  train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
  # define model
  model = Sequential()
  model.add(LSTM(200, activation='relu', input_shape=(n_timesteps, n_features)))
  model.add(RepeatVector(n_outputs))
  model.add(LSTM(200, activation='relu', return_sequences=True))
  model.add(TimeDistributed(Dense(100, activation='relu')))
  model.add(TimeDistributed(Dense(1)))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [1, n_input, 1]
  input_x = input_x.reshape((1, len(input_x), 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# evaluate model and get scores
n_input = 14
```

```
score, scores = evaluate_model(train, test, n_input)
# summarize scores
summarize_scores('lstm', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='lstm')
pyplot.show()
```

Listing 20.20: Example of evaluating a univariate Encoder-Decoder LSTM model for multi-step forecasting.

Running the example fits the model and summarizes the performance on the test dataset. We can see that in this case, the model is skillful, achieving an overall RMSE score of about 372 kilowatts.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
lstm: [372.595] 379.5, 399.8, 339.6, 372.2, 370.9, 309.9, 424.8
```

Listing 20.21: Sample output from evaluating a univariate Encoder-Decoder LSTM model for multi-step forecasting.

A line plot of the per-day RMSE is also created showing a similar pattern in error as was seen in the previous section.
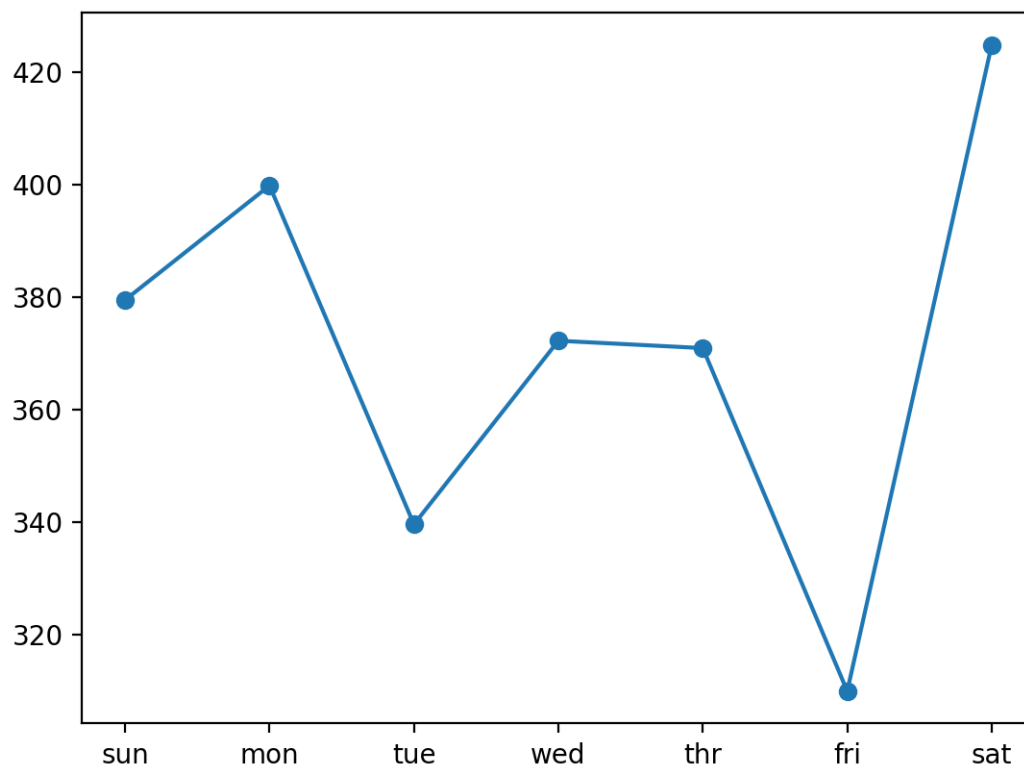
Figure 20.3: Line Plot of RMSE per Day for Univariate Encoder-Decoder LSTM with 14-day Inputs.

20.8 Encoder-Decoder LSTM With Multivariate Input

In this section, we will update the Encoder-Decoder LSTM developed in the previous section to use each of the eight time series variables to predict the next standard week of daily total power consumption. We will do this by providing each one-dimensional time series to the model as a separate sequence of input. The LSTM will in turn create an internal representation of each input sequence that will together be interpreted by the decoder.

Using multivariate inputs is helpful for those problems where the output sequence is some function of the observations at prior time steps from multiple different features, not just (or including) the feature being forecasted. It is unclear whether this is the case in the power consumption problem, but we can explore it nonetheless. First, we must update the preparation of the training data to include all of the eight features, not just the one total daily power consumed. It requires a single line change:

```
# use all variables for input samples
X.append(data[in_start:in_end, :])
```

Listing 20.22: Example of preparing data with all input variables.

The complete `to_supervised()` function with this change is listed below.

```
# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      X.append(data[in_start:in_end, :])
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)
```

Listing 20.23: Example of a function for creating overlapping windows of data with all variables.

We also must update the function used to make forecasts with the fit model to use all eight features from the prior time steps. Again, another small change:

```
# retrieve last observations for input data
input_x = data[-n_input:, :]
# reshape into [1, n_input, n]
input_x = input_x.reshape((1, input_x.shape[0], input_x.shape[1]))
```

Listing 20.24: Example of using all variables as input when making a forecast.

```
# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, :]
  # reshape into [1, n_input, n]
  input_x = input_x.reshape((1, input_x.shape[0], input_x.shape[1]))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat
```

Listing 20.25: Example of a function for making a multi-step forecast with a LSTM model and all input variables.

The same model architecture and configuration is used directly, although we will increase the number of training epochs from 20 to 50 given the 8-fold increase in the amount of input data. The complete example is listed below.

```
# multivariate multi-step encoder-decoder lstm for the power usage dataset
from math import sqrt
from numpy import split
```

```python
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import LSTM
from keras.layers import RepeatVector
from keras.layers import TimeDistributed

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores

# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
```

```
      X.append(data[in_start:in_end, :])
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)

# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 50, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # reshape output into [samples, timesteps, features]
  train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
  # define model
  model = Sequential()
  model.add(LSTM(200, activation='relu', input_shape=(n_timesteps, n_features)))
  model.add(RepeatVector(n_outputs))
  model.add(LSTM(200, activation='relu', return_sequences=True))
  model.add(TimeDistributed(Dense(100, activation='relu')))
  model.add(TimeDistributed(Dense(1)))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, :]
  # reshape into [1, n_input, n]
  input_x = input_x.reshape((1, input_x.shape[0], input_x.shape[1]))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
```

```
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# evaluate model and get scores
n_input = 14
score, scores = evaluate_model(train, test, n_input)
# summarize scores
summarize_scores('lstm', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='lstm')
pyplot.show()
```

Listing 20.26: Example of evaluating a multivariate Encoder-Decoder LSTM model for multi-step forecasting.

Running the example fits the model and summarizes the performance on the test dataset. Experimentation found that this model appears less stable than the univariate case and may be related to the differing scales of the input eight variables. We can see that in this case, the model is skillful, achieving an overall RMSE score of about 376 kilowatts.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
lstm: [376.273] 378.5, 381.5, 328.4, 388.3, 361.2, 308.0, 467.2
```

Listing 20.27: Sample output from evaluating a multivariate Encoder-Decoder LSTM model for multi-step forecasting.
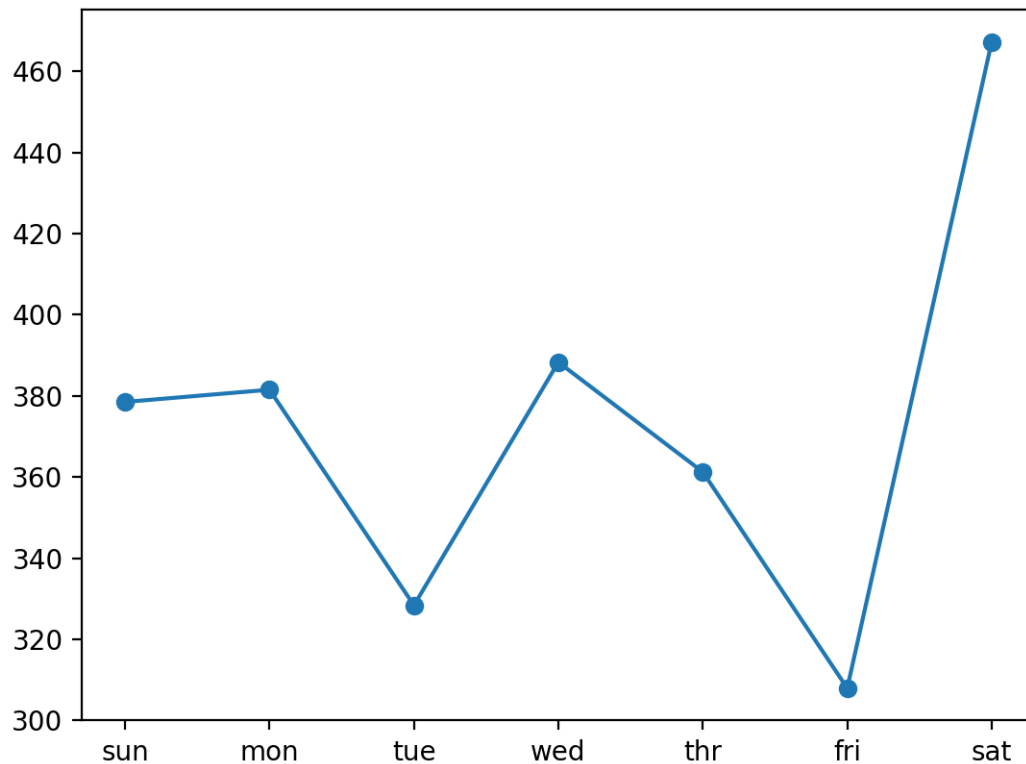
A line plot of the per-day RMSE is also created.

Figure 20.4: Line Plot of RMSE per Day for Multivariate Encoder-Decoder LSTM with 14-day Inputs.

20.9 CNN-LSTM Encoder-Decoder With Univariate Input

A convolutional neural network, or CNN, can be used as the encoder in an encoder-decoder architecture. The CNN does not directly support sequence input; instead, a 1D CNN is capable of reading across sequence input and automatically learning the salient features. These can then be interpreted by an LSTM decoder as per normal. We refer to hybrid models that use a CNN and LSTM as CNN-LSTM models, and in this case we are using them together in an encoder-decoder architecture. The CNN expects the input data to have the same 3D structure as the LSTM model, although multiple features are read as different channels that ultimately have the same effect.

We will simplify the example and focus on the CNN-LSTM with univariate input, but it can just as easily be updated to use multivariate input, which is left as an exercise. As before, we will use input sequences comprised of 14 days of daily total power consumption. We will define a simple but effective CNN architecture for the encoder that is comprised of two convolutional layers followed by a max pooling layer, the results of which are then flattened.

The first convolutional layer reads across the input sequence and projects the results onto

feature maps. The second performs the same operation on the feature maps created by the first layer, attempting to amplify any salient features. We will use 64 feature maps per convolutional layer and read the input sequences with a kernel size of three time steps. The max pooling layer simplifies the feature maps by keeping $\frac{1}{4}$ of the values with the largest (max) signal. The distilled feature maps after the pooling layer are then flattened into one long vector that can then be used as input to the decoding process.

```
# define cnn input model
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
    input_shape=(n_timesteps,n_features)))
model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
```

Listing 20.28: Example of defining the CNN encoder model.

The decoder is the same as was defined in previous sections. The only other change is to set the number of training epochs to 20. The `build_model()` function with these changes is listed below.

```
# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 20, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # reshape output into [samples, timesteps, features]
  train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(RepeatVector(n_outputs))
  model.add(LSTM(200, activation='relu', return_sequences=True))
  model.add(TimeDistributed(Dense(100, activation='relu')))
  model.add(TimeDistributed(Dense(1)))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model
```

Listing 20.29: Example of a function for defining and fitting a CNN Encoder-Decoder LSTM model.

We are now ready to try the encoder-decoder architecture with a CNN encoder. The complete code listing is provided below.

```
# univariate multi-step encoder-decoder cnn-lstm for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
```

```python
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import LSTM
from keras.layers import RepeatVector
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D

# split a univariate dataset into train/test sets
def split_dataset(data):
	# split into standard weeks
	train, test = data[1:-328], data[-328:-6]
	# restructure into windows of weekly data
	train = array(split(train, len(train)/7))
	test = array(split(test, len(test)/7))
	return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
	scores = list()
	# calculate an RMSE score for each day
	for i in range(actual.shape[1]):
		# calculate mse
		mse = mean_squared_error(actual[:, i], predicted[:, i])
		# calculate rmse
		rmse = sqrt(mse)
		# store
		scores.append(rmse)
	# calculate overall RMSE
	s = 0
	for row in range(actual.shape[0]):
		for col in range(actual.shape[1]):
			s += (actual[row, col] - predicted[row, col])**2
	score = sqrt(s / (actual.shape[0] * actual.shape[1]))
	return score, scores

# summarize scores
def summarize_scores(name, score, scores):
	s_scores = ', '.join(['%.1f' % s for s in scores])
	print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
	# flatten data
	data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
	X, y = list(), list()
	in_start = 0
	# step over the entire history one time step at a time
	for _ in range(len(data)):
		# define the end of the input sequence
		in_end = in_start + n_input
		out_end = in_end + n_out
		# ensure we have enough data for this instance
```

```python
    if out_end < len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
      y.append(data[in_end:out_end, 0])
    # move along one time step
    in_start += 1
  return array(X), array(y)

# train the model
def build_model(train, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 20, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # reshape output into [samples, timesteps, features]
  train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
  # define model
  model = Sequential()
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(RepeatVector(n_outputs))
  model.add(LSTM(200, activation='relu', return_sequences=True))
  model.add(TimeDistributed(Dense(100, activation='relu')))
  model.add(TimeDistributed(Dense(1)))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [1, n_input, 1]
  input_x = input_x.reshape((1, len(input_x), 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_input):
  # fit model
  model = build_model(train, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
```

```
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# evaluate model and get scores
n_input = 14
score, scores = evaluate_model(train, test, n_input)
# summarize scores
summarize_scores('lstm', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='lstm')
pyplot.show()
```

Listing 20.30: Example of evaluating a univariate CNN Encoder-Decoder LSTM model for multi-step forecasting.

Running the example fits the model and summarizes the performance on the test dataset. A little experimentation showed that using two convolutional layers made the model more stable than using just a single layer. We can see that in this case the model is skillful, achieving an overall RMSE score of about 372 kilowatts.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
lstm: [372.055] 383.8, 381.6, 339.1, 371.8, 371.8, 319.6, 427.2
```

Listing 20.31: Sample output from evaluating a univariate CNN Encoder-Decoder LSTM model for multi-step forecasting.

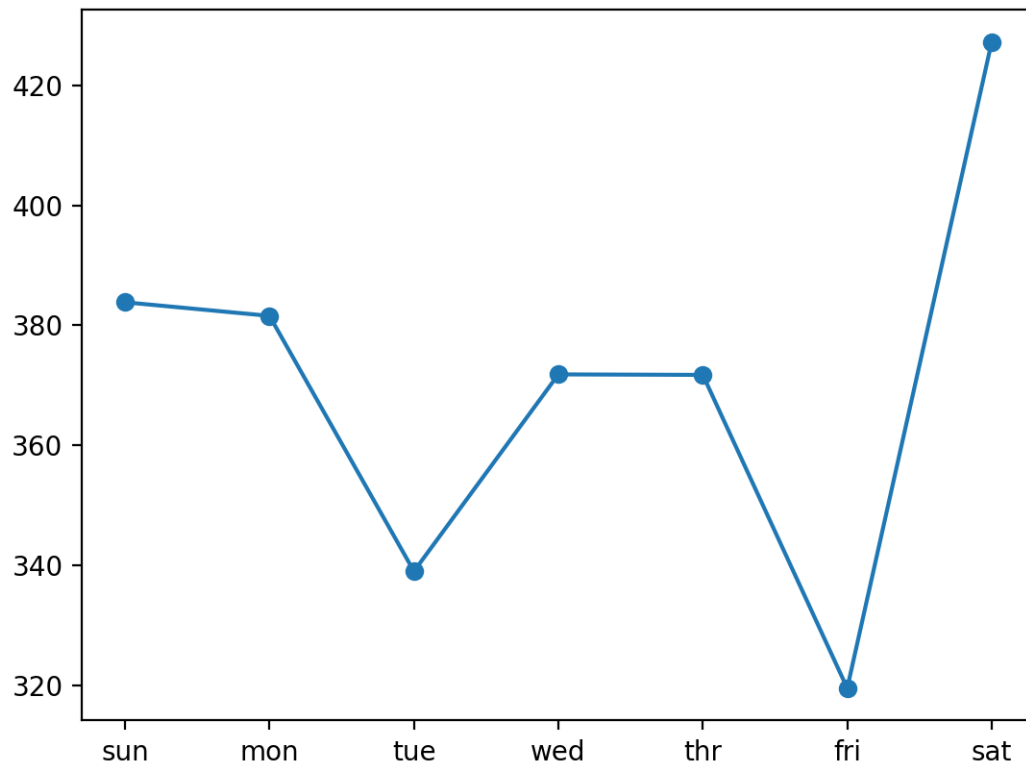A line plot of the per-day RMSE is also created.

Figure 20.5: Line Plot of RMSE per Day for Univariate Encoder-Decoder CNN-LSTM with 14-day Inputs.

20.10 ConvLSTM Encoder-Decoder With Univariate Input

A further extension of the CNN-LSTM approach is to perform the convolutions of the CNN (e.g. how the CNN reads the input sequence data) as part of the LSTM for each time step. This combination is called a Convolutional LSTM, or ConvLSTM for short, and like the CNN-LSTM is also used for spatiotemporal data. Unlike an LSTM that reads the data in directly in order to calculate internal state and state transitions, and unlike the CNN-LSTM that is interpreting the output from CNN models, the ConvLSTM is using convolutions directly as part of reading input into the LSTM units themselves. The Keras library provides the `ConvLSTM2D` class that supports the ConvLSTM model for 2D data. It can be configured for 1D multivariate time series forecasting. The `ConvLSTM2D` class, by default, expects input data to have the shape: `[samples, timesteps, rows, cols, channels]`.

Where each time step of data is defined as an image of (rows × columns) data points. We are working with a one-dimensional sequence of total power consumption, which we can interpret as one row with 14 columns, if we assume that we are using two weeks of data as input. For the ConvLSTM, this would be a single read: that is, the LSTM would read one time step of 14

days and perform a convolution across those time steps.

This is not ideal. Instead, we can split the 14 days into two subsequences with a length of seven days. The ConvLSTM can then read across the two time steps and perform the CNN process on the seven days of data within each. For this chosen framing of the problem, the input for the `ConvLSTM2D` would therefore be: `[n, 2, 1, 7, 1]`. Or:

- **Samples**: n, for the number of examples in the training dataset.

- **Time**: 2, for the two subsequences that we split a window of 14 days into.

- **Rows**: 1, for the one-dimensional shape of each subsequence.

- **Columns**: 7, for the seven days in each subsequence.

- **Channels**: 1, for the single feature that we are working with as input.

You can explore other configurations, such as providing 21 days of input split into three subsequences of seven days, and/or providing all eight features or channels as input. We can now prepare the data for the `ConvLSTM2D` model. First, we must reshape the training dataset into the expected structure of `[samples, timesteps, rows, cols, channels]`.

```
# reshape into subsequences [samples, timesteps, rows, cols, channels]
train_x = train_x.reshape((train_x.shape[0], n_steps, 1, n_length, n_features))
```

Listing 20.32: Example of preparing data for the `ConvLSTM2D` encoder model.

We can then define the encoder as a ConvLSTM hidden layer followed by a flatten layer ready for decoding.

```
# define convlstm input model
model.add(ConvLSTM2D(filters=64, kernel_size=(1,3), activation='relu',
    input_shape=(n_steps, 1, n_length, n_features)))
model.add(Flatten())
```

Listing 20.33: Example of defining the `ConvLSTM2D` encoder model.

We will also parameterize the number of subsequences (`n_steps`) and the length of each subsequence (`n_length`) and pass them as arguments. The rest of the model and training is the same. The `build_model()` function with these changes is listed below.

```
# train the model
def build_model(train, n_steps, n_length, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 20, 16
  n_timesteps, n_features, n_outputs = train_x.shape[1], train_x.shape[2], train_y.shape[1]
  # reshape into subsequences [samples, timesteps, rows, cols, channels]
  train_x = train_x.reshape((train_x.shape[0], n_steps, 1, n_length, n_features))
  # reshape output into [samples, timesteps, features]
  train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
  # define model
  model = Sequential()
  model.add(ConvLSTM2D(filters=64, kernel_size=(1,3), activation='relu',
      input_shape=(n_steps, 1, n_length, n_features)))
```

```
  model.add(Flatten())
  model.add(RepeatVector(n_outputs))
  model.add(LSTM(200, activation='relu', return_sequences=True))
  model.add(TimeDistributed(Dense(100, activation='relu')))
  model.add(TimeDistributed(Dense(1)))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model
```

Listing 20.34: Example of a function for defining and fitting a ConvLSTM Encoder-Decoder LSTM model.

This model expects five-dimensional data as input. Therefore, we must also update the preparation of a single sample in the `forecast()` function when making a prediction.

```
# reshape into [samples, timesteps, rows, cols, channels]
input_x = input_x.reshape((1, n_steps, 1, n_length, 1))
```

Listing 20.35: Example of preparing input data for the `ConvLSTM2D` model when making a prediction.

The `forecast()` function with this change and with the parameterized subsequences is provided below.

```
# make a forecast
def forecast(model, history, n_steps, n_length, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [samples, timesteps, rows, cols, channels]
  input_x = input_x.reshape((1, n_steps, 1, n_length, 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat
```

Listing 20.36: Example of a function for making a forecast with the ConvLSTM Encoder-Decoder LSTM model.

We now have all of the elements for evaluating an encoder-decoder architecture for multi-step time series forecasting where a ConvLSTM is used as the encoder. The complete code example is listed below.

```
# univariate multi-step encoder-decoder convlstm for the power usage dataset
from math import sqrt
from numpy import split
from numpy import array
from pandas import read_csv
from sklearn.metrics import mean_squared_error
from matplotlib import pyplot
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
```

```python
from keras.layers import LSTM
from keras.layers import RepeatVector
from keras.layers import TimeDistributed
from keras.layers import ConvLSTM2D

# split a univariate dataset into train/test sets
def split_dataset(data):
  # split into standard weeks
  train, test = data[1:-328], data[-328:-6]
  # restructure into windows of weekly data
  train = array(split(train, len(train)/7))
  test = array(split(test, len(test)/7))
  return train, test

# evaluate one or more weekly forecasts against expected values
def evaluate_forecasts(actual, predicted):
  scores = list()
  # calculate an RMSE score for each day
  for i in range(actual.shape[1]):
    # calculate mse
    mse = mean_squared_error(actual[:, i], predicted[:, i])
    # calculate rmse
    rmse = sqrt(mse)
    # store
    scores.append(rmse)
  # calculate overall RMSE
  s = 0
  for row in range(actual.shape[0]):
    for col in range(actual.shape[1]):
      s += (actual[row, col] - predicted[row, col])**2
  score = sqrt(s / (actual.shape[0] * actual.shape[1]))
  return score, scores

# summarize scores
def summarize_scores(name, score, scores):
  s_scores = ', '.join(['%.1f' % s for s in scores])
  print('%s: [%.3f] %s' % (name, score, s_scores))

# convert history into inputs and outputs
def to_supervised(train, n_input, n_out=7):
  # flatten data
  data = train.reshape((train.shape[0]*train.shape[1], train.shape[2]))
  X, y = list(), list()
  in_start = 0
  # step over the entire history one time step at a time
  for _ in range(len(data)):
    # define the end of the input sequence
    in_end = in_start + n_input
    out_end = in_end + n_out
    # ensure we have enough data for this instance
    if out_end < len(data):
      x_input = data[in_start:in_end, 0]
      x_input = x_input.reshape((len(x_input), 1))
      X.append(x_input)
      y.append(data[in_end:out_end, 0])
    # move along one time step
```

```
    in_start += 1
  return array(X), array(y)

# train the model
def build_model(train, n_steps, n_length, n_input):
  # prepare data
  train_x, train_y = to_supervised(train, n_input)
  # define parameters
  verbose, epochs, batch_size = 0, 20, 16
  n_features, n_outputs = train_x.shape[2], train_y.shape[1]
  # reshape into subsequences [samples, timesteps, rows, cols, channels]
  train_x = train_x.reshape((train_x.shape[0], n_steps, 1, n_length, n_features))
  # reshape output into [samples, timesteps, features]
  train_y = train_y.reshape((train_y.shape[0], train_y.shape[1], 1))
  # define model
  model = Sequential()
  model.add(ConvLSTM2D(filters=64, kernel_size=(1,3), activation='relu',
      input_shape=(n_steps, 1, n_length, n_features)))
  model.add(Flatten())
  model.add(RepeatVector(n_outputs))
  model.add(LSTM(200, activation='relu', return_sequences=True))
  model.add(TimeDistributed(Dense(100, activation='relu')))
  model.add(TimeDistributed(Dense(1)))
  model.compile(loss='mse', optimizer='adam')
  # fit network
  model.fit(train_x, train_y, epochs=epochs, batch_size=batch_size, verbose=verbose)
  return model

# make a forecast
def forecast(model, history, n_steps, n_length, n_input):
  # flatten data
  data = array(history)
  data = data.reshape((data.shape[0]*data.shape[1], data.shape[2]))
  # retrieve last observations for input data
  input_x = data[-n_input:, 0]
  # reshape into [samples, timesteps, rows, cols, channels]
  input_x = input_x.reshape((1, n_steps, 1, n_length, 1))
  # forecast the next week
  yhat = model.predict(input_x, verbose=0)
  # we only want the vector forecast
  yhat = yhat[0]
  return yhat

# evaluate a single model
def evaluate_model(train, test, n_steps, n_length, n_input):
  # fit model
  model = build_model(train, n_steps, n_length, n_input)
  # history is a list of weekly data
  history = [x for x in train]
  # walk-forward validation over each week
  predictions = list()
  for i in range(len(test)):
    # predict the week
    yhat_sequence = forecast(model, history, n_steps, n_length, n_input)
    # store the predictions
    predictions.append(yhat_sequence)
```

```
    # get real observation and add to history for predicting the next week
    history.append(test[i, :])
  # evaluate predictions days for each week
  predictions = array(predictions)
  score, scores = evaluate_forecasts(test[:, :, 0], predictions)
  return score, scores

# load the new file
dataset = read_csv('household_power_consumption_days.csv', header=0,
    infer_datetime_format=True, parse_dates=['datetime'], index_col=['datetime'])
# split into train and test
train, test = split_dataset(dataset.values)
# define the number of subsequences and the length of subsequences
n_steps, n_length = 2, 7
# define the total days to use as input
n_input = n_length * n_steps
score, scores = evaluate_model(train, test, n_steps, n_length, n_input)
# summarize scores
summarize_scores('lstm', score, scores)
# plot scores
days = ['sun', 'mon', 'tue', 'wed', 'thr', 'fri', 'sat']
pyplot.plot(days, scores, marker='o', label='lstm')
pyplot.show()
```

Listing 20.37: Example of evaluating a univariate ConvLSTM Encoder-Decoder LSTM model
for multi-step forecasting.

Running the example fits the model and summarizes the performance on the test dataset. A
little experimentation showed that using two convolutional layers made the model more stable
than using just a single layer. We can see that in this case the model is skillful, achieving an
overall RMSE score of about 367 kilowatts.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider
running the example a few times.

```
lstm: [367.929] 416.3, 379.7, 334.7, 362.3, 374.7, 284.8, 406.7
```

Listing 20.38: Sample output from evaluating a univariate ConvLSTM Encoder-Decoder LSTM
model for multi-step forecasting.

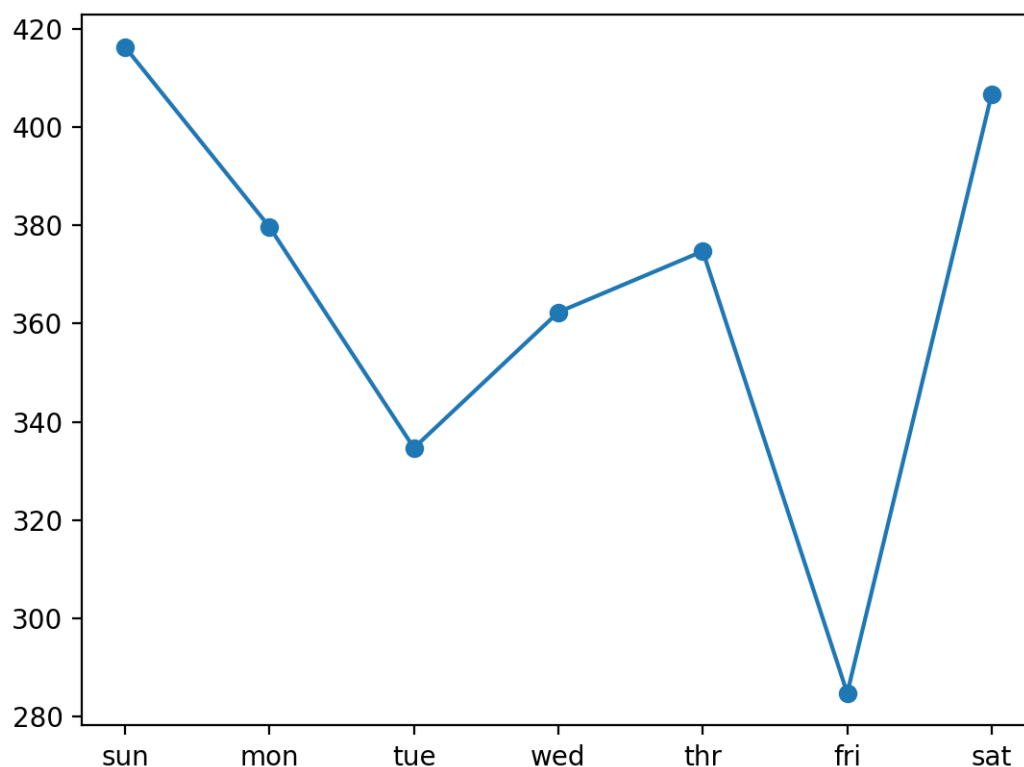A line plot of the per-day RMSE is also created.

Figure 20.6: Line Plot of RMSE per Day for Univariate Encoder-Decoder ConvLSTM with 14-day Inputs.

20.11   Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Size of Input**. Explore more or fewer number of days used as input for the model, such as three days, 21 days, 30 days, and more.

- **Model Tuning**. Tune the structure and hyperparameters for a model and further lift model performance on average.

- **Data Scaling**. Explore whether data scaling, such as standardization and normalization, can be used to improve the performance of any of the LSTM models.

- **Learning Diagnostics**. Use diagnostics such as learning curves for the train and validation loss and mean squared error to help tune the structure and hyperparameters of a LSTM model.

If you explore any of these extensions, I'd love to know.

20.12   Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/

- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

20.13   Summary

In this tutorial, you discovered how to develop long short-term memory recurrent neural networks for multi-step time series forecasting of household power consumption. Specifically, you learned:

- How to develop and evaluate Univariate and multivariate Encoder-Decoder LSTMs for multi-step time series forecasting.

- How to develop and evaluate an CNN-LSTM Encoder-Decoder model for multi-step time series forecasting.

- How to develop and evaluate a ConvLSTM Encoder-Decoder model for multi-step time series forecasting.

20.13.1   Next

This is the final lesson of this part, the next part will focus on how to systematically work through a real-world multivariate time series classification problem to predict human activities from accelerometer data.

Part VI

Time Series Classification

Overview

This part focuses on the real-world time series classification problem of activity recognition from multivariate accelerometer data recorded from a smartphone, and how to develop machine learning and deep learning classification models to address the problem. The tutorials in this part do not seek to demonstrate the best way to solve the problem, instead the dataset provides a context on which each of the specific methods can be demonstrated. As such, the performance of each method on the dataset are not compared directly. After reading the chapters in this part, you will know:

- A review of recent research highlighting deep learning models and their general configuration that are state-of-the-art for human activity recognition (Chapter 21).

- How to load, summarize and plot a standard human activity recognition dataset comprised of accelerometer data recorded from a smartphone (Chapter 22).

- How to develop nonlinear and ensemble machine learning models from accelerometer data with domain-specific engineered features (Chapter 23).

- How to develop and evaluate a suite of Convolutional Neural Network models for human activity recognition from accelerometer data (Chapter 24).

- How to develop and evaluate a suite of Long Short-Term Memory Neural Network models for human activity recognition from accelerometer data (Chapter 25).

Chapter 21

Review of Deep Learning Models for Human Activity Recognition

Human activity recognition, or HAR, is a challenging time series classification task. It involves predicting the movement of a person based on sensor data and traditionally involves deep domain expertise and methods from signal processing to correctly engineer features from the raw data in order to fit a machine learning model. Recently, deep learning methods such as convolutional neural networks and recurrent neural networks have shown capable and even achieve state-of-the-art results by automatically learning features from the raw sensor data. In this tutorial, you will discover the problem of human activity recognition and the deep learning methods that are achieving state-of-the-art performance on this problem. After reading this tutorial, you will know:

- Activity recognition is the problem of predicting the movement of a person, often indoors, based on sensor data, such as an accelerometer in a smartphone.

- Streams of sensor data are often split into subs-sequences called windows, and each window is associated with a broader activity, called a sliding window approach.

- Convolutional neural networks and long short-term memory networks, and perhaps both together, are best suited to learning features from raw sensor data and predicting the associated movement.

Let's get started.

21.1 Overview

This tutorial is divided into five parts; they are:

1. Human Activity Recognition

2. Benefits of Neural Network Modeling

3. Supervised Learning Data Representation

4. Convolutional Neural Network Models

5. Recurrent Neural Network Models

21.2 Human Activity Recognition

Human activity recognition, or HAR for short, is a broad field of study concerned with identifying the specific movement or action of a person based on sensor data. Movements are often typical activities performed indoors, such as walking, talking, standing, and sitting. They may also be more focused activities such as those types of activities performed in a kitchen or on a factory floor. The sensor data may be remotely recorded, such as video, radar, or other wireless methods. Alternately, data may be recorded directly on the subject such as by carrying custom hardware or smartphones that have accelerometers and gyroscopes.

> Sensor-based activity recognition seeks the profound high-level knowledge about human activities from multitudes of low-level sensor readings
>
> — *Deep Learning for Sensor-based Activity Recognition: A Survey*, 2018.

Historically, sensor data for activity recognition was challenging and expensive to collect, requiring custom hardware. Now, smartphones and other personal tracking devices used for fitness and health monitoring are cheap and ubiquitous. As such, sensor data from these devices is cheaper to collect, more common, and therefore is a more commonly studied version of the general activity recognition problem. The problem is to predict the activity given a snapshot of sensor data, typically data from one or a small number of sensor types. Generally, this problem is framed as a univariate or multivariate time series classification task.

It is a challenging problem as there are no obvious or direct ways to relate the recorded sensor data to specific human activities and each subject may perform an activity with significant variation, resulting in variations in the recorded sensor data. The intent is to record sensor data and corresponding activities for specific subjects, fit a model from this data, and generalize the model to classify the activity of new unseen subjects from their sensor data.

21.3 Benefits of Neural Network Modeling

Traditionally, methods from the field of signal processing were used to analyze and distill the collected sensor data. Such methods were for feature engineering, creating domain-specific, sensor-specific, or signal processing-specific features and views of the original data. Statistical and machine learning models were then trained on the processed version of the data. A limitation of this approach is the signal processing and domain expertise required to analyze the raw data and engineer the features required to fit a model. This expertise would be required for each new dataset or sensor modality. In essence, it is expensive and not scalable.

> However, in most daily HAR tasks, those methods may heavily rely on heuristic handcrafted feature extraction, which is usually limited by human domain knowledge. Furthermore, only shallow features can be learned by those approaches, leading to undermined performance for unsupervised and incremental tasks. Due to those limitations, the performances of conventional [pattern recognition] methods are restricted regarding classification accuracy and model generalization.
>
> — *Deep Learning for Sensor-based Activity Recognition: A Survey*, 2018.

Ideally, learning methods could be used that automatically learn the features required to make accurate predictions from the raw data directly. This would allow new problems, new datasets, and new sensor modalities to be adopted quickly and cheaply. Recently, deep neural network models have started delivering on their promises of feature learning and are achieving stat-of-the-art results for human activity recognition. They are capable of performing automatic feature learning from the raw sensor data and out-perform models fit on hand-crafted domain-specific features.

> ... the feature extraction and model building procedures are often performed simultaneously in the deep learning models. The features can be learned automatically through the network instead of being manually designed. Besides, the deep neural network can also extract high-level representation in deep layer, which makes it more suitable for complex activity recognition tasks.

> — *Deep Learning for Sensor-based Activity Recognition: A Survey*, 2018.

There are two main approaches to neural networks that are appropriate for time series classification and that have been demonstrated to perform well on activity recognition using sensor data from commodity smartphones and fitness tracking devices. They are Convolutional Neural Network Models and Recurrent Neural Network Models.

> RNN and LSTM are recommended to recognize short activities that have natural order while CNN is better at inferring long term repetitive activities. The reason is that RNN could make use of the time-order relationship between sensor readings, and CNN is more capable of learning deep features contained in recursive patterns.

> — *Deep Learning for Sensor-based Activity Recognition: A Survey*, 2018.

21.4 Supervised Learning Data Representation

Before we dive into the specific neural networks that can be used for human activity recognition, we need to talk about data preparation. Both types of neural networks suitable for time series classification require that data be prepared in a specific manner in order to fit a model. That is, in a 'supervised learning' way that allows the model to associate signal data with an activity class. A straight-forward data preparation approach that was used both for classical machine learning methods on the hand-crafted features and for neural networks involves dividing the input signal data into windows of signals, where a given window may have one to a few seconds of observation data. This is often called a *sliding window*.

> Human activity recognition aims to infer the actions of one or more persons from a set of observations captured by sensors. Usually, this is performed by following a fixed length sliding window approach for the features extraction where two parameters have to be fixed: the size of the window and the shift.

> — *A Dynamic Sliding Window Approach for Activity Recognition*, 2011.

Each window is also associated with a specific activity. A given window of data may have multiple variables, such as the x, y, and z axes of an accelerometer sensor. Let's make this concrete with an example. We have sensor data for 10 minutes; that may look like:

```
x,      y,      z,      activity
1.1,    2.1,    0.1,    1
1.2,    2.2,    0.2,    1
1.3,    2.3,    0.3,    1
...
```

Listing 21.1: Example of contrived sensor data and associated activity.

If the data is recorded at 8 Hz, that means that there will be eight rows of data for one second of elapsed time performing an activity. We may choose to have one window of data represent one second of data; that means eight rows of data for an 8 Hz sensor. If we have x, y, and z data, that means we would have 3 variables. Therefore, a single window of data would be a 2-dimensional array with eight time steps and three features. One window would represent one sample. One minute of data would represent 480 sensor data points, or 60 windows of eight time steps. The total 10 minutes of data would represent 4,800 data points, or 600 windows of data.

It is convenient to describe the shape of our prepared sensor data in terms of the number of samples or windows, the number of time steps in a window, and the number of features observed at each time step.

```
[samples, timesteps, features]
```

Listing 21.2: Example of the dimensionality of sensor data.

Our example of 10 minutes of accelerometer data recorded at 8 Hz would be summarized as a three-dimensional array with the dimensions:

```
[600, 8, 3]
```

Listing 21.3: Specific example of the dimensionality of a sensor dataset.

There is no best window size, and it really depends on the specific model being used, the nature of the sensor data that was collected, and the activities that are being classified. There is a tension in the size of the window and the size of the model. Larger windows require large models that are slower to train, whereas smaller windows require smaller models that are much easier to fit.

> Intuitively, decreasing the window size allows for a faster activity detection, as well as reduced resources and energy needs. On the contrary, large data windows are normally considered for the recognition of complex activities

> — *Window Size Impact in Human Activity Recognition*, 2014.

Nevertheless, it is common to use one to two seconds of sensor data in order to classify a current fragment of an activity.

> From the results, reduced windows (2 s or less) are demonstrated to provide the most accurate detection performance. In fact, the most precise recognizer is obtained for very short windows (0.25-0.5 s), leading to the perfect recognition of most activities. Contrary to what is often thought, this study demonstrates that large window sizes do not necessarily translate into a better recognition performance.

> — *Window Size Impact in Human Activity Recognition*, 2014.

There is some risk that the splitting of the stream of sensor data into windows may result in windows that miss the transition of one activity to another. As such, it was traditionally common to split data into windows with an overlap such that the first half of the window contained the observations from the last half of the previous window, in the case of a 50% overlap.

> ... an incorrect length may truncate an activity instance. In many cases, errors appear at the beginning or at the end of the activities, when the window overlaps the end of one activity and the beginning of the next one. In other cases, the window length may be too short to provide the best information for the recognition process.

> — *A Dynamic Sliding Window Approach for Activity Recognition*, 2011.

It is unclear whether windows with overlap are required for a given problem. In the adoption of neural network models, the use of overlaps, such as a 50% overlap, will double the size of the training data, which may aid in modeling smaller datasets, but may also lead to models that overfit the training dataset.

> An overlap between adjacent windows is tolerated for certain applications; however, this is less frequently used.

> — *Window Size Impact in Human Activity Recognition*, 2014.

21.5 Convolutional Neural Network Models

Convolutional Neural Network models, or CNNs for short, are a type of deep neural network that were developed for use with image data, e.g. such as handwriting recognition. More recently, CNNs have been used for time series forecasting. For more information on the use of CNNs for time series forecasting, see Chapter 8. CNNs can be applied to human activity recognition data. The CNN model learns to map a given window of signal data to an activity where the model reads across each window of data and prepares an internal representation of the window.

> When applied to time series classification like HAR, CNN has two advantages over other models: local dependency and scale invariance. Local dependency means the nearby signals in HAR are likely to be correlated, while scale invariance refers to the scale-invariant for different paces or frequencies.

> — *Deep Learning for Sensor-based Activity Recognition: A Survey*, 2018.

The first important work using CNNs to HAR was by Ming Zeng, et al in their 2014 paper *Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors*. In the paper, the authors develop a simple CNN model for accelerometer data, where each axis of the accelerometer data is fed into separate convolutional layers, pooling layers, then concatenated before being interpreted by hidden fully connected layers. The figure below taken from the paper clearly shows the topology of the model. It provides a good template for how the CNN may be used for HAR problems and time series classification in general.
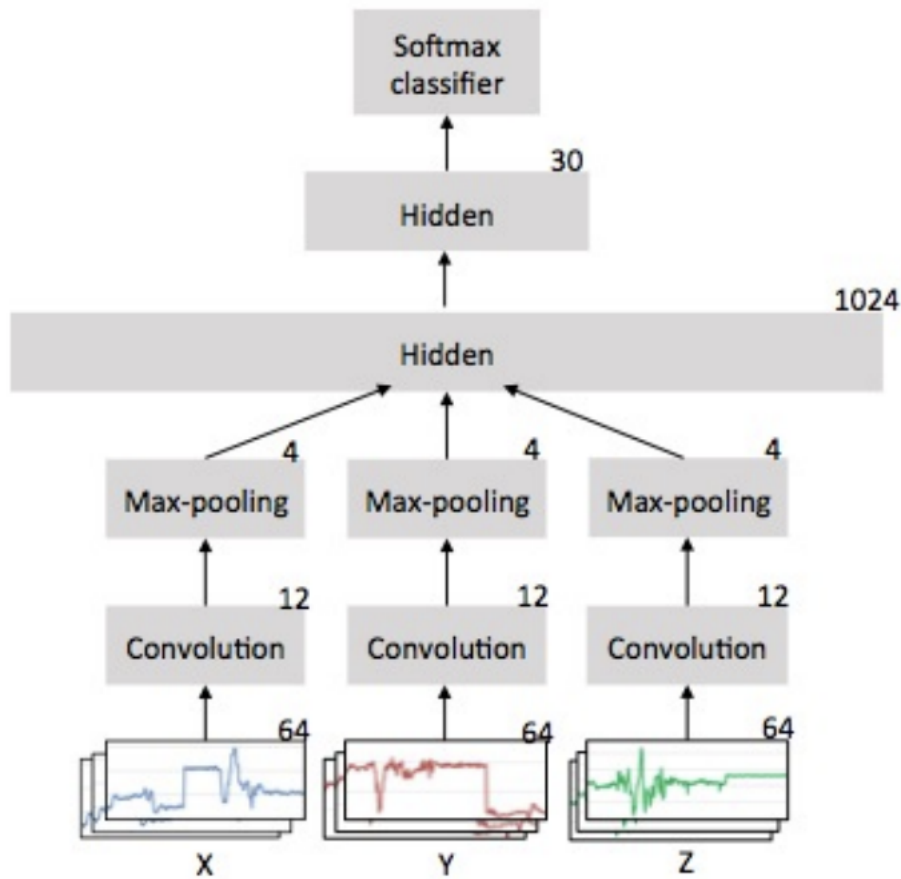
Figure 21.1: Depiction of CNN Model for Accelerometer Data. Taken from *Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors*.

There are many ways to model HAR problems with CNNs. One interesting example was by Heeryon Cho and Sang Min Yoon in their 2018 paper titled *Divide and Conquer-Based 1D CNN Human Activity Recognition Using Test Data Sharpening*. In it, they divide activities into those that involve movement, called *dynamic*, and those where the subject is stationary, called *static*, then develop a CNN model to discriminate between these two main classes. Then, within each class, models are developed to discriminate between activities of that type, such as *walking* for dynamic and *sitting* for static.
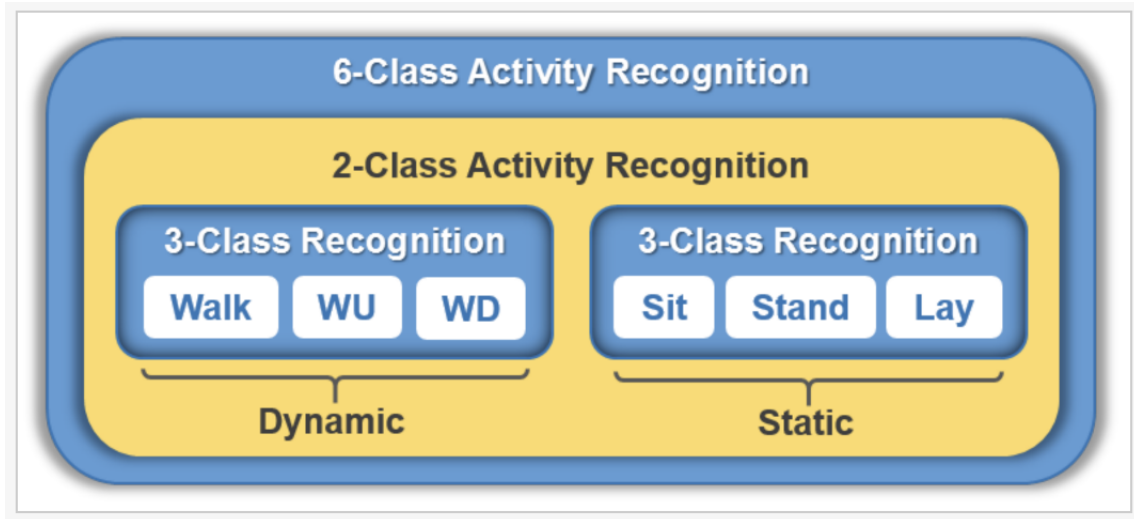
Figure 21.2: Separation of Activities as Dynamic or Static. Taken from *Divide and Conquer-Based 1D CNN Human Activity Recognition Using Test Data Sharpening.*

They refer to this as a two-stage modeling approach.

> Instead of straightforwardly recognizing the individual activities using a single 6-class classifier, we apply a divide and conquer approach and build a two-stage activity recognition process, where abstract activities, i.e., dynamic and static activity, are first recognized using a 2-class or binary classifier, and then individual activities are recognized using two 3-class classifiers.
>
> — *Divide and Conquer-Based 1D CNN Human Activity Recognition Using Test Data Sharpening*, 2018.

Quite large CNN models were developed, which in turn allowed the authors to claim state-of-the-art results on challenging standard human activity recognition datasets. Another interesting approach was proposed by Wenchao Jiang and Zhaozheng Yin in their 2015 paper titled *Human Activity Recognition Using Wearable Sensors by Deep Convolutional Neural Networks*. Instead of using 1D CNNs on the signal data, they instead combine the signal data together to create *images* which are then fed to a 2D CNN and processed as image data with convolutions along the time axis of signals and across signal variables, specifically accelerometer and gyroscope data.

> Firstly, raw signals are stacked row-by-row into a signal image [....]. In the signal image, every signal sequence has the chance to be adjacent to every other sequence, which enables DCNN to extract hidden correlations between neighboring signals. Then, 2D Discrete Fourier Transform (DFT) is applied to the signal image and its magnitude is chosen as our activity image.
>
> — *Human Activity Recognition Using Wearable Sensors by Deep Convolutional Neural Networks*, 2015.

Below is a depiction of the processing of raw sensor data into images, and then from images into an *activity image*, the result of a discrete Fourier transform. Finally, another good paper on the topic is by Charissa Ann Ronao and Sung-Bae Cho in 2016 titled *Human activity recognition with smartphone sensors using deep learning neural networks*. Careful study of the use of CNNs is performed showing that larger kernel sizes of signal data are useful and limited pooling.

> Experiments show that convnets indeed derive relevant and more complex features with every additional layer, although difference of feature complexity level decreases with every additional layer. A wider time span of temporal local correlation can be exploited (1×9 - 1×14) and a low pooling size (1×2 - 1×3) is shown to be beneficial.

— *Human activity recognition with smartphone sensors using deep learning neural networks*,
2016.

Usefully, they also provide the full hyperparameter configuration for the CNN models that may provide a useful starting point on new HAR and other sequence classification problems, summarized below.

Table 1
Experimental setup.

Parameter	Value
The size of input vector	128
The number of input channels	6
The number of feature maps	10–200
Filter size	1×3–1×15
Pooling size	1×3
Activation function	ReLU (rectified linear unit)
Learning rate	0.01
Weight decay	0.00005
Momentum	0.5–0.99
The probability of dropout	0.8
The size of minibatches	128
Maximum epochs	5000

Figure 21.3: Table of CNN Model Hyperparameter Configuration. Taken from *Human activity recognition with smartphone sensors using deep learning neural networks*.

21.6 Recurrent Neural Network Models

Long Short-Term Memory networks or LSTMs for short have proven effective on challenging sequence prediction problems when trained at scale for such tasks as handwriting recognition, language modeling, and machine translation. For more information on the use of LSTMs for time series forecasting, see Chapter 9. LSTMs can be applied to the problem of human

activity recognition. The LSTM learns to map each window of sensor data to an activity, where the observations in the input sequence are read one at a time, where each time step may be comprised of one or more variables (e.g. parallel sequences).

There has been limited application of simple LSTM models to HAR problems. One example is by Abdulmajid Murad and Jae-Young Pyun in their 2017 paper titled *Deep Recurrent Neural Networks for Human Activity Recognition*. Important, in the paper they comment on the limitation of CNNs in their requirement to operate on fixed-sized windows of sensor data, a limitation that LSTMs do not strictly have.

> However, the size of convolutional kernels restricts the captured range of dependencies between data samples. As a result, typical models are unadaptable to a wide range of activity-recognition configurations and require fixed-length input windows.

> — *Deep Recurrent Neural Networks for Human Activity Recognition*, 2017.

They explore the use of LSTMs that both process the sequence data forward (normal) and both directions (Bidirectional LSTM). Interestingly, the LSTM predicts an activity for each input time step of a subsequence of sensor data, which are then aggregated in order to predict an activity for the window.

> There will [be] a score for each time step predicting the type of activity occurring at time t. The prediction for the entire window T is obtained by merging the individual scores into a single prediction

> — *Deep Recurrent Neural Networks for Human Activity Recognition*, 2017.

The figure below taken from the paper provides a depiction of the LSTM model followed by fully connected layers used to interpret the internal representation of the raw sensor data.
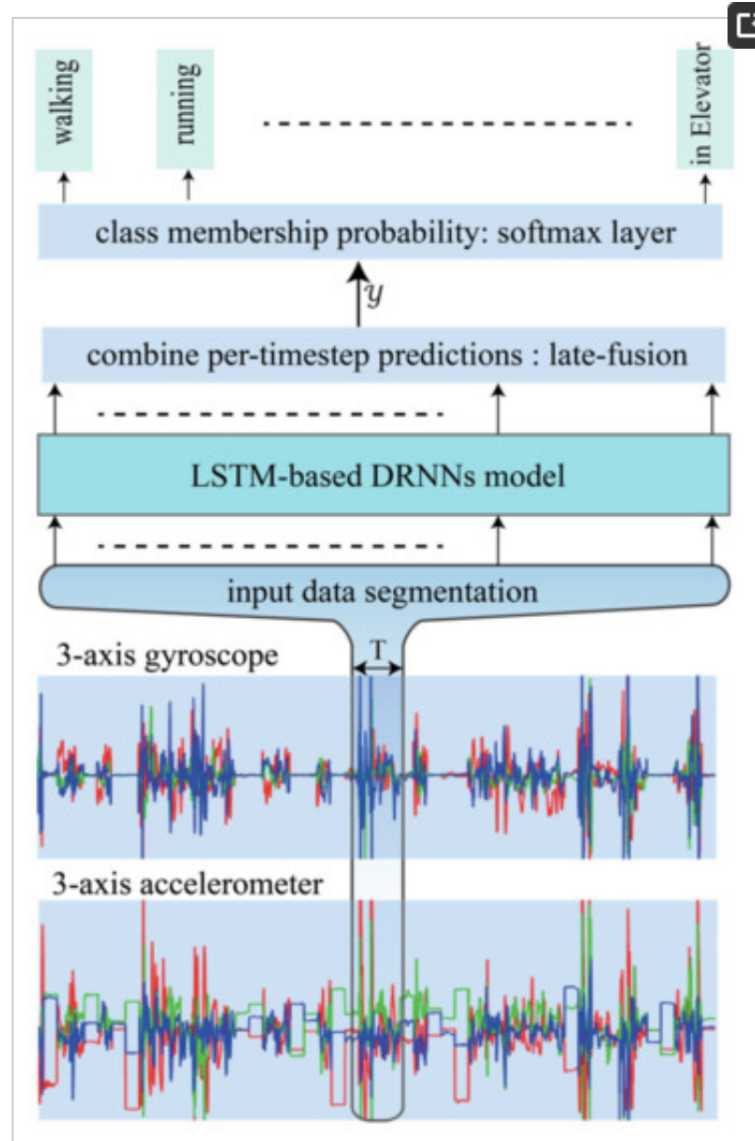
Figure 21.4: Depiction of LSTM RNN for Activity Recognition. Taken from *Deep Recurrent Neural Networks for Human Activity Recognition.*

It may be more common to use an LSTM in conjunction with a CNN on HAR problems, in a CNN-LSTM model or ConvLSTM model. This is where a CNN model is used to extract the features from a subsequence of raw sample data, and output features from the CNN for each subsequence are then interpreted by an LSTM in aggregate. An example of this is in the 2016 paper by Francisco Javier Ordonez and Daniel Roggen titled *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition.*

> We introduce a new DNN framework for wearable activity recognition, which we refer to as DeepConvLSTM. This architecture combines convolutional and recurrent layers. The convolutional layers act as feature extractors and provide abstract representations of the input sensor data in feature maps. The recurrent layers model the temporal dynamics of the activation of the feature maps.

— *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition*, 2016.

A deep network architecture is used with four convolutional layers without any pooling layers, followed by two LSTM layers to interpret the extracted features over multiple time steps. The authors claim that the removal of the pooling layers is a critical part of their model architecture, where the use of pooling layers after the convolutional layers interferes with the convolutional layers' ability to learn to downsample the raw sensor data.

> In the literature, CNN frameworks often include convolutional and pooling layers successively, as a measure to reduce data complexity and introduce translation invariant features. Nevertheless, such an approach is not strictly part of the architecture, and in the time series domain [...] DeepConvLSTM does not include pooling operations because the input of the network is constrained by the sliding window mechanism [...] and this fact limits the possibility of downsampling the data, given that DeepConvLSTM requires a data sequence to be processed by the recurrent layers. However, without the sliding window requirement, a pooling mechanism could be useful to cover different sensor data time scales at deeper layers.

— *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition*, 2016.

The figure below taken from the paper makes the architecture clearer. Note that layers 6 and 7 in the image are in fact LSTM layers.
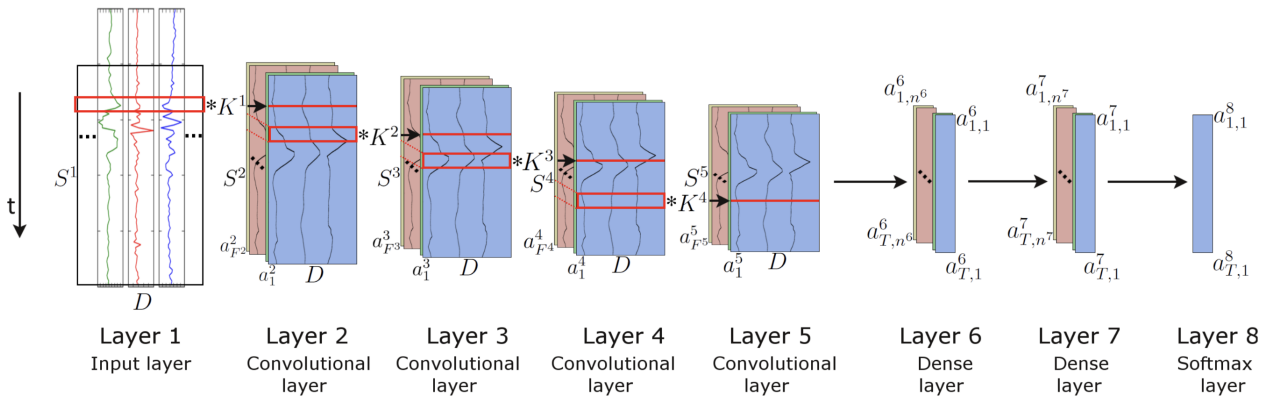

Figure 21.5: Depiction of CNN-LSTM Model for Activity Recognition. Taken from *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition*.

21.7 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Summarize Problem**. Summarize the problem of human activity recognition in 1-3 lines.

- **List Methods**. Create a list of the methods that are known to perform well for human activity recognition.

- **Example Applications**. List 3 examples where models for predicting human activity from sensor data may be useful.

If you explore any of these extensions, I'd love to know.

21.8 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

21.8.1 General

- *Deep Learning for Sensor-based Activity Recognition: A Survey*, 2018.
  https://www.sciencedirect.com/science/article/pii/S016786551830045X

21.8.2 Sliding Windows

- *A Dynamic Sliding Window Approach for Activity Recognition*, 2011.
  https://link.springer.com/chapter/10.1007/978-3-642-22362-4_19

- *Window Size Impact in Human Activity Recognition*, 2014.
  https://www.ncbi.nlm.nih.gov/pmc/articles/PMC4029702/

21.8.3 CNNs

- *Convolutional Neural Networks for Human Activity Recognition using Mobile Sensors*, 2014.
  https://ieeexplore.ieee.org/document/7026300/

- *Divide and Conquer-Based 1D CNN Human Activity Recognition Using Test Data Sharpening*, 2018.
  http://www.mdpi.com/1424-8220/18/4/1055

- *Human Activity Recognition Using Wearable Sensors by Deep Convolutional Neural Networks*, 2015.
  https://dl.acm.org/citation.cfm?id=2806333

- *Human activity recognition with smartphone sensors using deep learning neural networks*, 2016.
  https://www.sciencedirect.com/science/article/pii/S0957417416302056

21.8.4 RNNs

- *Deep Recurrent Neural Networks for Human Activity Recognition*, 2017.
  http://www.mdpi.com/1424-8220/17/11/2556

- *Deep Convolutional and LSTM Recurrent Neural Networks for Multimodal Wearable Activity Recognition*, 2016.
  http://www.mdpi.com/1424-8220/16/1/115/html

21.9 Summary

In this tutorial, you discovered the problem of human activity recognition and the use of deep learning methods that are achieving state-of-the-art performance on this problem. Specifically, you learned:

- Activity recognition is the problem of predicting the movement of a person, often indoors, based on sensor data, such as an accelerometer in a smartphone.

- Streams of sensor data are often split into subs-sequences called windows and each window is associated with a broader activity, called a sliding window approach.

- Convolutional neural networks and long short-term memory networks, and perhaps both together, are best suited to learning features from raw sensor data and predicting the associated movement.

21.9.1 Next

In the next lesson, you will discover how to load and analyze a real-world human activity recognition dataset.

Chapter 22

How to Load and Explore Human Activity Data

Human activity recognition is the problem of classifying sequences of accelerometer data recorded by specialized harnesses or smartphones into known well-defined movements. It is a challenging problem given the large number of observations produced each second, the temporal nature of the observations, and the lack of a clear way to relate accelerometer data to known movements. Classical approaches to the problem involve hand crafting features from the time series data based on fixed-sized windows and training machine learning models, such as ensembles of decision trees. The difficulty is that this feature engineering requires deep expertise in the field. Recently, deep learning methods such as recurrent neural networks and one-dimensional convolutional neural networks, or CNNs, have been shown to provide state-of-the-art results on challenging activity recognition tasks with little or no data feature engineering.

In this tutorial, you will discover the *Activity Recognition Using Smartphones* dataset for time series classification and how to load and explore the dataset in order to make it ready for predictive modeling. This dataset will provided the basis for the remaining tutorials in this part of the book. After completing this tutorial, you will know:

- How to download and load the dataset into memory.

- How to use line plots, histograms, and box plots to better understand the structure of the motion data.

- How to model the problem, including framing, data preparation, modeling, and evaluation.

Let's get started.

22.1   Tutorial Overview

This tutorial is divided into 10 parts; they are:

1. Activity Recognition Using Smartphones Dataset

2. Download the Dataset

3. Load the Dataset

4. Balance of Activity Classes

5. Plot Time Series Per Subject

6. Plot Distribution Per Subject

7. Plot Distribution Per Activity

8. Plot Distribution of Activity Duration

9. Approach to Modeling

22.2 Activity Recognition Using Smartphones Dataset

A standard human activity recognition dataset is the *Activity Recognition Using Smartphones* dataset made available in 2012. It was prepared and made available by Davide Anguita, et al. from the University of Genova, Italy and is described in full in their 2013 paper *A Public Domain Dataset for Human Activity Recognition Using Smartphones*. The dataset was modeled with machine learning algorithms in their 2012 paper titled *Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine*. The dataset was made available and can be downloaded for free from the UCI Machine Learning Repository[1].

Figure 22.1: Activity Recognition Experiment Using Smartphone Sensors.

The data was collected from 30 subjects aged between 19 and 48 years old performing one of 6 standard activities while wearing a waist-mounted smartphone that recorded the movement

---

[1]https://archive.ics.uci.edu/ml/datasets/human+activity+recognition+using+smartphones

data. Video was recorded of each subject performing the activities and the movement data was labeled manually from these videos. Below is an example video of a subject performing the activities while their movement data is being recorded[2]. The six activities performed were as follows:

1. Walking

2. Walking Upstairs

3. Walking Downstairs

4. Sitting

5. Standing

6. Laying

The movement data recorded was the x, y, and z accelerometer data (linear acceleration) and gyroscopic data (angular velocity) from the smartphone, specifically a Samsung Galaxy S II. Observations were recorded at 50 Hz (i.e. 50 data points per second). Each subject performed the sequence of activities twice, once with the device on their left-hand-side and once with the device on their right-hand side.

> A group of 30 volunteers with ages ranging from 19 to 48 years were selected for this task. Each person was instructed to follow a protocol of activities while wearing a waist-mounted Samsung Galaxy S II smartphone. The six selected ADL were standing, sitting, laying down, walking, walking downstairs and upstairs. Each subject performed the protocol twice: on the first trial the smartphone was fixed on the left side of the belt and on the second it was placed by the user himself as preferred

> — *A Public Domain Dataset for Human Activity Recognition Using Smartphones*, 2013.

The raw data is not available. Instead, a pre-processed version of the dataset was made available. The pre-processing steps included:

- Pre-processing accelerometer and gyroscope using noise filters.

- Splitting data into fixed windows of 2.56 seconds (128 data points) with 50% overlap.

- Splitting of accelerometer data into gravitational (total) and body motion components.

> These signals were preprocessed for noise reduction with a median filter and a 3rd order low-pass Butterworth filter with a 20 Hz cutoff frequency. [...] The acceleration signal, which has gravitational and body motion components, was separated using another Butterworth low-pass filter into body acceleration and gravity.

> — *A Public Domain Dataset for Human Activity Recognition Using Smartphones*, 2013.

---

[2]View on YouTube: https://www.youtube.com/watch?v=XOEN9W05_4A

Feature engineering was applied to the window data, and a copy of the data with these engineered features was made available. A number of time and frequency features commonly used in the field of human activity recognition were extracted from each window. The result was a 561 element vector of features. The dataset was split into train (70%) and test (30%) sets based on data for subjects, e.g. 21 subjects for train and nine for test.

This suggests a framing of the problem where a sequence of movement activity is used as input to predict the portion (2.56 seconds) of the current activity being performed, where a model trained on known subjects is used to predict the activity from movement on new subjects. Early experiment results with a support vector machine intended for use on a smartphone (e.g. fixed-point arithmetic) resulted in a predictive accuracy of 89% on the test dataset, achieving similar results as an unmodified SVM implementation.

> This method adapts the standard Support Vector Machine (SVM) and exploits fixed-point arithmetic for computational cost reduction. A comparison with the traditional SVM shows a significant improvement in terms of computational costs while maintaining similar accuracy [...]

— *Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine*, 2012.

Now that we are familiar with the prediction problem, we will look at loading and exploring this dataset.

22.3   Download the Dataset

The data is provided as a single zip file that is about 58 megabytes in size. A direct for downloading the dataset is provided below:

- `HAR_Smartphones.zip` [3]

Download the dataset and unzip all files into a new directory in your current working directory named `HARDataset`. Inspecting the decompressed contents, you will notice a few things:

- There are `train` and `test` folders containing the split portions of the data for modeling (e.g. 70%/30%).

- There is a `README.txt` file that contains a detailed technical description of the dataset and the contents of the unzipped files.

- There is a `features.txt` file that contains a technical description of the engineered features.

The contents of the `train` and `test` folders are similar (e.g. folders and file names), although with differences in the specific data they contain. Inspecting the `train` folder shows a few important elements:

---

[3]https://raw.githubusercontent.com/jbrownlee/Datasets/master/HAR_Smartphones.zip

- An `Inertial Signals` folder that contains the preprocessed data.

- The `X_train.txt` file that contains the engineered features intended for fitting a model.

- The `y_train.txt` that contains the class labels for each observation (1-6).

- The `subject_train.txt` file that contains a mapping of each line in the data files with their subject identifier (1-30).

The number of lines in each file match, indicating that one row is one record in each data file. The `Inertial Signals` directory contains 9 files.

- Gravitational acceleration data files for x, y and z axes: `total_acc_x_train.txt`, `total_acc_y_train.txt`, `total_acc_z_train.txt`.

- Body acceleration data files for x, y and z axes: `body_acc_x_train.txt`, `body_acc_y_train.txt`, `body_acc_z_train.txt`.

- Body gyroscope data files for x, y and z axes: `body_gyro_x_train.txt`, `body_gyro_y_train.txt`, `body_gyro_z_train.txt`.

The structure is mirrored in the `test` directory. We will focus our attention on the data in the `Inertial Signals` as this is most interesting in developing machine learning models that can learn a suitable representation, instead of using the domain-specific feature engineering. Inspecting a datafile shows that columns are separated by whitespace and values appear to be scaled to the range -1, 1. This scaling can be confirmed by a note in the `README.txt` file provided with the dataset. Now that we know what data we have, we can figure out how to load it into memory.

22.4   Load the Dataset

In this section, we will develop some code to load the dataset into memory. First, we need to load a single file. We can use the `read_csv()` Pandas function to load a single data file and specify that the file has no header and to separate columns using white space.

```
# load a csv file as a dataframe
dataframe = read_csv(filepath, header=None, delim_whitespace=True)
```

Listing 22.1: Example of loading a file as a `DataFrame`.

We can wrap this in a function named `load_file()`. The complete example of this function is listed below.

```
# load one file from the har dataset
from pandas import read_csv

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

data = load_file('HARDataset/train/Inertial Signals/total_acc_y_train.txt')
```

```
print(data.shape)
```

Listing 22.2: Example of loading one axis of accelerometer data.

Running the example loads the file `total_acc_y_train.txt`, returns a NumPy array, and prints the shape of the array. We can see that the training data is comprised of 7,352 rows or windows of data, where each window has 128 observations.

```
(7352, 128)
```

Listing 22.3: Example output from loading one axis of accelerometer data.

Next, it would be useful to load a group of files, such as all of the body acceleration data files as a single group. Ideally, when working with multivariate time series data, it is useful to have the data structured in the format: `[samples, timesteps, features]`. This is helpful for analysis and is the expectation of deep learning models such as convolutional neural networks and recurrent neural networks. We can achieve this by calling the above `load_file()` function multiple times, once for each file in a group.

Once we have loaded each file as a NumPy array, we can combine or stack all three arrays together. We can use the `dstack()` NumPy function to ensure that each array is stacked in such a way that the features are separated in the third dimension, as we would prefer. The function `load_group()` implements this behavior for a list of file names and is listed below.

```
# load a list of files, such as x, y, z data for a given variable
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

Listing 22.4: Example of a function for loading a group of files.

We can demonstrate this function by loading all of the total acceleration files. The complete example is listed below.

```
# load group of files from the har dataset
from numpy import dstack
from pandas import read_csv

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files, such as x, y, z data for a given variable
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

```
# load the total acc data
filenames = ['total_acc_x_train.txt', 'total_acc_y_train.txt', 'total_acc_z_train.txt']
total_acc = load_group(filenames, prefix='HARDataset/train/Inertial Signals/')
print(total_acc.shape)
```

Listing 22.5: Example of loading a group of accelerometer data files.

Running the example prints the shape of the returned NumPy array, showing the expected number of samples and time steps with the three features, x, y, and z for the dataset.

```
(7352, 128, 3)
```

Listing 22.6: Example output from loading a group of accelerometer data files.

Finally, we can use the two functions developed so far to load all data for the train and the test dataset. Given the parallel structure in the train and test folders, we can develop a new function that loads all input and output data for a given folder. The function can build a list of all 9 data files to load, load them as one NumPy array with 9 features, then load the data file containing the output class. The load_dataset() function below implements this behavior. It can be called for either the train group or the test group, passed as a string argument.

```
# load a dataset group, such as train or test
def load_dataset(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y
```

Listing 22.7: Example of a function for loading all data files for either train or test.

The complete example is listed below.

```
# load all train and test data from the har dataset
from numpy import dstack
from pandas import read_csv

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files, such as x, y, z data for a given variable
def load_group(filenames, prefix=''):
```

```
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load all train
trainX, trainy = load_dataset('train', 'HARDataset/')
print(trainX.shape, trainy.shape)
# load all test
testX, testy = load_dataset('test', 'HARDataset/')
print(testX.shape, testy.shape)
```

Listing 22.8: Example of loading all accelerometer data files.

Running the example loads the train and test datasets. We can see that the test dataset has 2,947 rows of window data. As expected, we can see that the size of windows in the train and test sets matches and the size of the output (y) in each the train and test case matches the number of samples.

```
(7352, 128, 9) (7352, 1)
(2947, 128, 9) (2947, 1)
```

Listing 22.9: Example output from loading all accelerometer data files.

22.5 Balance of Activity Classes

A good first check of the data is to investigate the balance of each activity. We believe that each of the 30 subjects performed each of the six activities. Confirming this expectation will both check that the data is indeed balanced, making it easier to model, and confirm that we are correctly loading and interpreting the dataset. We can develop a function that summarizes the

breakdown of the output variables, e.g. the y variable. The function `class_breakdown()` below implements this behavior, first wrapping the provided NumPy array in a `DataFrame`, grouping the rows by the class value, and calculating the size of each group (number of rows). The results are then summarized, including the count and the percentage.

```python
# summarize the balance of classes in an output variable column
def class_breakdown(data):
  # convert the numpy array into a dataframe
  df = DataFrame(data)
  # group data by the class value and calculate the number of rows
  counts = df.groupby(0).size()
  # retrieve raw rows
  counts = counts.values
  # summarize
  for i in range(len(counts)):
    percent = counts[i] / len(df) * 100
    print('Class=%d, total=%d, percentage=%.3f' % (i+1, counts[i], percent))
```

Listing 22.10: Example of a function for summarizing the class breakdown.

It may be useful to summarize the breakdown of the classes in the train and test datasets to ensure they have a similar breakdown, then compare the result to the breakdown on the combined dataset. The complete example is listed below.

```python
# summarize class balance from the har dataset
from numpy import vstack
from pandas import read_csv
from pandas import DataFrame

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# summarize the balance of classes in an output variable column
def class_breakdown(data):
  # convert the numpy array into a dataframe
  df = DataFrame(data)
  # group data by the class value and calculate the number of rows
  counts = df.groupby(0).size()
  # retrieve raw rows
  counts = counts.values
  # summarize
  for i in range(len(counts)):
    percent = counts[i] / len(df) * 100
    print('Class=%d, total=%d, percentage=%.3f' % (i+1, counts[i], percent))

# load train file
trainy = load_file('HARDataset/train/y_train.txt')
# summarize class breakdown
print('Train Dataset')
class_breakdown(trainy)

# load test file
testy = load_file('HARDataset/test/y_test.txt')
# summarize class breakdown
```

```
print('Test Dataset')
class_breakdown(testy)

# summarize combined class breakdown
print('Both')
combined = vstack((trainy, testy))
class_breakdown(combined)
```

Listing 22.11: Example of summarizing the breakdown of classes in the dataset.

Running the example first summarizes the breakdown for the training set. We can see a pretty similar distribution of each class hovering between 13% and 19% of the dataset. The result on the test set and on both datasets together look very similar. It is likely safe to work with the dataset assuming the distribution of classes is balanced per train and test set and perhaps per subject.

```
Train Dataset
Class=1, total=1226, percentage=16.676
Class=2, total=1073, percentage=14.595
Class=3, total=986, percentage=13.411
Class=4, total=1286, percentage=17.492
Class=5, total=1374, percentage=18.689
Class=6, total=1407, percentage=19.138

Test Dataset
Class=1, total=496, percentage=16.831
Class=2, total=471, percentage=15.982
Class=3, total=420, percentage=14.252
Class=4, total=491, percentage=16.661
Class=5, total=532, percentage=18.052
Class=6, total=537, percentage=18.222

Both
Class=1, total=1722, percentage=16.720
Class=2, total=1544, percentage=14.992
Class=3, total=1406, percentage=13.652
Class=4, total=1777, percentage=17.254
Class=5, total=1906, percentage=18.507
Class=6, total=1944, percentage=18.876
```

Listing 22.12: Example output from summarizing the breakdown of classes in the dataset.

22.6   Plot Time Series Per Subject

We are working with time series data, therefore an import check is to create a line plot of the raw data. The raw data is comprised of windows of time series data per variable, and the windows do have a 50% overlap. This suggests we may see some repetition in the observations as a line plot unless the overlap is removed. We can start off by loading the training dataset using the functions developed above.

```
# load data
trainX, trainy = load_dataset('train', 'HARDataset/')
```

Listing 22.13: Example of loading the training dataset.

Next, we can load the `subject_train.txt` in the `train` directory that provides a mapping of rows to the subject to which it belongs. We can load this file using the `load_file()` function. Once loaded, we can also use the `unique()` NumPy function to retrieve a list of the unique subjects in the training dataset.

```
# load subject mapping
sub_map = load_file('HARDataset/train/subject_train.txt')
train_subjects = unique(sub_map)
print(train_subjects)
```
Listing 22.14: Example of loading the mapping of observations to subjects.

Next, we need a way to retrieve all of the rows for a single subject, e.g. subject number 1. We can do this by finding all of the row numbers that belong to a given subject and use those row numbers to select the samples from the loaded X and y data from the training dataset. The `data_for_subject()` function below implements this behavior. It will take the loaded training data, the loaded mapping of row number to subjects, and the subject identification number for the subject that we are interested in, and will return the X and y data for only that subject.

```
# get all data for one subject
def data_for_subject(X, y, sub_map, sub_id):
  # get row indexes for the subject id
  ix = [i for i in range(len(sub_map)) if sub_map[i]==sub_id]
  # return the selected samples
  return X[ix, :, :], y[ix]
```
Listing 22.15: Example of a function to get data for a given subject id.

Now that we have data for one subject, we can plot it. The data is comprised of windows with overlap. We can write a function to remove this overlap and squash the windows down for a given variable into one long sequence that can be plotted directly as a line plot. The `to_series()` function below implements this behavior for a given variable, e.g. array of windows.

```
# convert a series of windows to a 1D list
def to_series(windows):
  series = list()
  for window in windows:
    # remove the overlap from the window
    half = int(len(window) / 2) - 1
    for value in window[-half:]:
      series.append(value)
  return series
```
Listing 22.16: Example of a function to convert window data to a series.

Finally, we have enough to plot the data. We can plot each of the nine variables for the subject in turn and a final plot for the activity level. Each series will have the same number of time steps (length of x-axis), therefore, it may be useful to create a subplot for each variable and align all plots vertically so we can compare the movement on each variable. The `plot_subject()` function below implements this behavior for the X and y data for a single subject. The function assumes the same order of the variables (3rd axis) as was loaded in the `load_dataset()` function. A crude title is also added to each plot so we don't get easily confused about what we are looking at.

```
# plot the data for one subject
```

```python
def plot_subject(X, y):
  pyplot.figure()
  # determine the total number of plots
  n, off = X.shape[2] + 1, 0
  # plot total acc
  for i in range(3):
    pyplot.subplot(n, 1, off+1)
    pyplot.plot(to_series(X[:, :, off]))
    pyplot.title('total acc '+str(i), y=0, loc='left', size=7)
    # turn off ticks to remove clutter
    pyplot.yticks([])
    pyplot.xticks([])
    off += 1
  # plot body acc
  for i in range(3):
    pyplot.subplot(n, 1, off+1)
    pyplot.plot(to_series(X[:, :, off]))
    pyplot.title('body acc '+str(i), y=0, loc='left', size=7)
    # turn off ticks to remove clutter
    pyplot.yticks([])
    pyplot.xticks([])
    off += 1
  # plot body gyro
  for i in range(3):
    pyplot.subplot(n, 1, off+1)
    pyplot.plot(to_series(X[:, :, off]))
    pyplot.title('body gyro '+str(i), y=0, loc='left', size=7)
    # turn off ticks to remove clutter
    pyplot.yticks([])
    pyplot.xticks([])
    off += 1
  # plot activities
  pyplot.subplot(n, 1, n)
  pyplot.plot(y)
  pyplot.title('activity', y=0, loc='left', size=7)
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
  pyplot.show()
```

Listing 22.17: Example of a function to plot data for a subject.

The complete example is listed below.

```python
# plot all vars for one subject in the har dataset
from numpy import dstack
from numpy import unique
from pandas import read_csv
from matplotlib import pyplot

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files, such as x, y, z data for a given variable
def load_group(filenames, prefix=''):
```

```python
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# get all data for one subject
def data_for_subject(X, y, sub_map, sub_id):
  # get row indexes for the subject id
  ix = [i for i in range(len(sub_map)) if sub_map[i]==sub_id]
  # return the selected samples
  return X[ix, :, :], y[ix]

# convert a series of windows to a 1D list
def to_series(windows):
  series = list()
  for window in windows:
    # remove the overlap from the window
    half = int(len(window) / 2) - 1
    for value in window[-half:]:
      series.append(value)
  return series

# plot the data for one subject
def plot_subject(X, y):
  pyplot.figure()
  # determine the total number of plots
  n, off = X.shape[2] + 1, 0
  # plot total acc
  for i in range(3):
    pyplot.subplot(n, 1, off+1)
    pyplot.plot(to_series(X[:, :, off]))
    pyplot.title('total acc '+str(i), y=0, loc='left', size=7)
    # turn off ticks to remove clutter
```

```
    pyplot.yticks([])
    pyplot.xticks([])
    off += 1
  # plot body acc
  for i in range(3):
    pyplot.subplot(n, 1, off+1)
    pyplot.plot(to_series(X[:, :, off]))
    pyplot.title('body acc '+str(i), y=0, loc='left', size=7)
    # turn off ticks to remove clutter
    pyplot.yticks([])
    pyplot.xticks([])
    off += 1
  # plot body gyro
  for i in range(3):
    pyplot.subplot(n, 1, off+1)
    pyplot.plot(to_series(X[:, :, off]))
    pyplot.title('body gyro '+str(i), y=0, loc='left', size=7)
    # turn off ticks to remove clutter
    pyplot.yticks([])
    pyplot.xticks([])
    off += 1
  # plot activities
  pyplot.subplot(n, 1, n)
  pyplot.plot(y)
  pyplot.title('activity', y=0, loc='left', size=7)
  # turn off ticks to remove clutter
  pyplot.yticks([])
  pyplot.xticks([])
  pyplot.show()

# load data
trainX, trainy = load_dataset('train', 'HARDataset/')
# load mapping of rows to subjects
sub_map = load_file('HARDataset/train/subject_train.txt')
train_subjects = unique(sub_map)
print(train_subjects)
# get the data for one subject
sub_id = train_subjects[0]
subX, suby = data_for_subject(trainX, trainy, sub_map, sub_id)
print(subX.shape, suby.shape)
# plot data for subject
plot_subject(subX, suby)
```

Listing 22.18: Example of creating line plots accelerometer data for a subject.

Running the example prints the unique subjects in the training dataset, the sample of the data for the first subject, and creates one figure with 10 plots, one for each of the nine input variables and the output class.

```
[ 1 3 5 6 7 8 11 14 15 16 17 19 21 22 23 25 26 27 28 29 30]
(341, 128, 9) (341, 1)
```

Listing 22.19: Example output from preparing data to plot for a subject.

In the plot, we can see periods of large movement corresponding with activities 1, 2, and 3: the walking activities. We can also see much less activity (i.e. a relatively straight line) for

higher numbered activities, 4, 5, and 6 (sitting, standing, and laying). This is good confirmation that we have correctly loaded interpreted the raw dataset. We can see that this subject has performed the same general sequence of activities twice, and some activities are performed more than two times. This suggests that for a given subject, we should not make assumptions about what activities may have been performed or their order.

We can also see some relatively large movement for some stationary activities, such as laying. It is possible that these are outliers or related to activity transitions. It may be possible to smooth or remove these observations as outliers. Finally, we see a lot of commonality across the nine variables. It is very likely that only a subset of these traces are required to develop a predictive model.
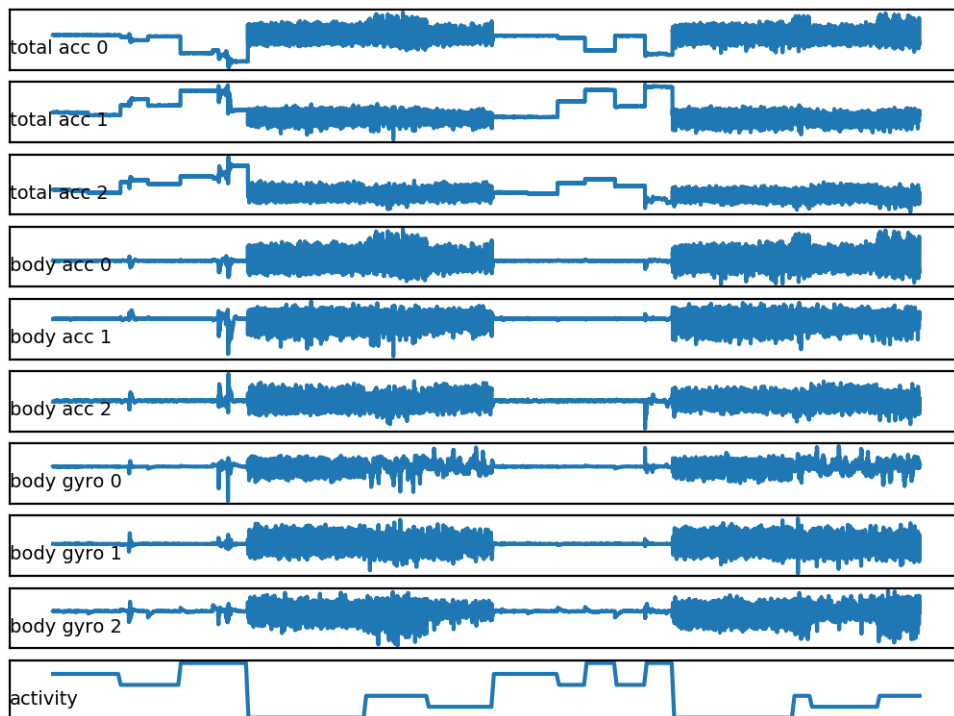
Figure 22.2: Line plot for all variables for a single subject.

We can re-run the example for another subject by making one small change, e.g. choose the identifier of the second subject in the training dataset.

```
# get the data for one subject
sub_id = train_subjects[1]
```

Listing 22.20: Example of changing the subject data to plot.

The plot for the second subject shows similar behavior with no surprises. The double sequence of activities does appear more regular than the first subject.
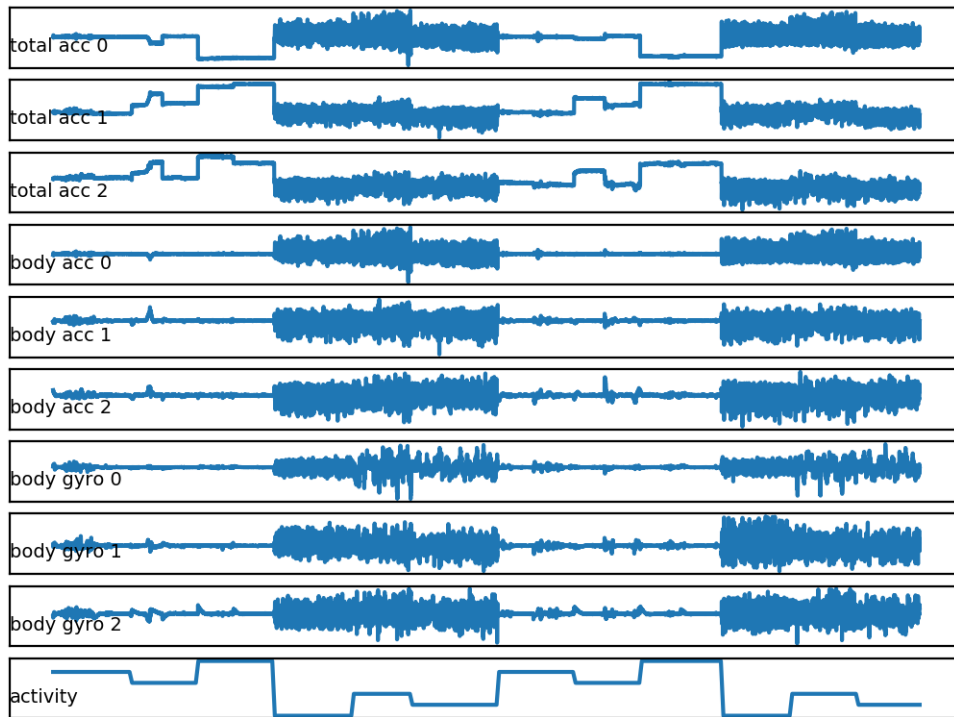
Figure 22.3: Line plot for all variables for a second single subject.

22.7 Plot Distribution Per Subject

As the problem is framed, we are interested in using the movement data from some subjects to predict activities from the movement of other subjects. This suggests that there must be regularity in the movement data across subjects. We know that the data has been scaled between -1 and 1, presumably per subject, suggesting that the amplitude of the detected movements will be similar. We would also expect that the distribution of movement data would be similar across subjects, given that they performed the same actions.

We can check for this by plotting and comparing the histograms of the movement data across subjects. A useful approach would be to create one plot per subject and plot all three axis of a given data (e.g. total acceleration), then repeat this for multiple subjects. The plots can be modified to use the same axis and aligned horizontally so that the distributions for each variable across subjects can be compared. The plot_subject_histograms() function below implements this behavior. The function takes the loaded dataset and mapping of rows to subjects as well as a maximum number of subjects to plot, fixed at 10 by default.

```
# plot histograms for multiple subjects
def plot_subject_histograms(X, y, sub_map, offset, n=10):
  pyplot.figure()
  # get unique subjects
```

```python
  subject_ids = unique(sub_map[:,0])
  # enumerate subjects
  for k in range(n):
    sub_id = subject_ids[k]
    # get data for one subject
    subX, _ = data_for_subject(X, y, sub_map, sub_id)
    # total acc
    for i in range(3):
      ax = pyplot.subplot(n, 1, k+1)
      ax.set_xlim(-1,1)
      ax.hist(to_series(subX[:,:,offset+i]), bins=100)
      pyplot.yticks([])
      pyplot.xticks([-1,0,1])
  pyplot.show()
```

Listing 22.21: Example of a create histograms of data.

The addition of an `offset` argument allows the same function to be called for each of the 3 groups of variables to plot at a time: total acceleration, body acceleration and gyroscopic with offsets 0, 3 and 6 respectively.

```python
# plot total acceleration histograms for subjects
plot_subject_histograms(X, y, sub_map, 0)
# plot body acceleration histograms for subjects
plot_subject_histograms(X, y, sub_map, 3)
# plot gyroscopic histograms for subjects
plot_subject_histograms(X, y, sub_map, 6)
```

Listing 22.22: Example of creating per-subjects histogram plots for each of the 3 groups of variables.

For a given call, a plot is created for each subject and the three variables for one data type are plotted as histograms with 100 bins, to help to make the distribution obvious. Each plot shares the same axis, which is fixed at the bounds of -1 and 1. The complete example is listed below.

```python
# plot histograms for multiple subjects from the har dataset
from numpy import unique
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files, such as x, y, z data for a given variable
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

```python
# load a dataset group, such as train or test
def load_dataset(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# get all data for one subject
def data_for_subject(X, y, sub_map, sub_id):
  # get row indexes for the subject id
  ix = [i for i in range(len(sub_map)) if sub_map[i]==sub_id]
  # return the selected samples
  return X[ix, :, :], y[ix]

# convert a series of windows to a 1D list
def to_series(windows):
  series = list()
  for window in windows:
    # remove the overlap from the window
    half = int(len(window) / 2) - 1
    for value in window[-half:]:
      series.append(value)
  return series

# plot histograms for multiple subjects
def plot_subject_histograms(X, y, sub_map, offset, n=10):
  pyplot.figure()
  # get unique subjects
  subject_ids = unique(sub_map[:,0])
  # enumerate subjects
  for k in range(n):
    sub_id = subject_ids[k]
    # get data for one subject
    subX, _ = data_for_subject(X, y, sub_map, sub_id)
    # total acc
    for i in range(3):
      ax = pyplot.subplot(n, 1, k+1)
      ax.set_xlim(-1,1)
      ax.hist(to_series(subX[:,:,offset+i]), bins=100)
      pyplot.yticks([])
      pyplot.xticks([-1,0,1])
  pyplot.show()
```

```
# load training dataset
X, y = load_dataset('train', 'HARDataset/')
# load mapping of rows to subjects
sub_map = load_file('HARDataset/train/subject_train.txt')
# plot total acceleration histograms for subjects
plot_subject_histograms(X, y, sub_map, 0)
# plot body acceleration histograms for subjects
plot_subject_histograms(X, y, sub_map, 3)
# plot gyroscopic histograms for subjects
plot_subject_histograms(X, y, sub_map, 6)
```

Listing 22.23: Example of plotting histograms of total accelerometer data.

Running the example creates three figures, each with 10 plots with histograms for the three axis. Each of the three axes on a given plot have a different color, specifically x, y, and z are blue, orange, and green respectively. The first figure summarizes the total acceleration. We can see that the distribution for a given axis does appear Gaussian with large separate groups of data. We can see some of the distributions align (e.g. main groups in the middle around 0.0), suggesting there may be some continuity of the movement data across subjects, at least for this data.
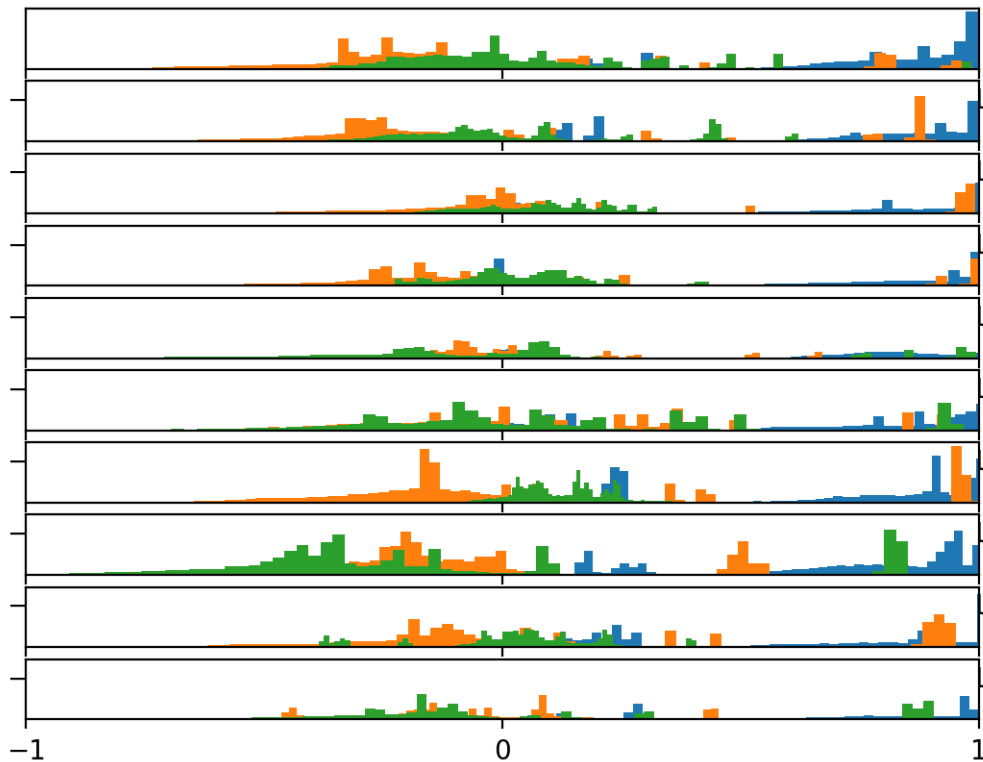

Figure 22.4: Histograms of the total acceleration data for 10 subjects.

The second figure shows histograms for the body acceleration with very different results.

We can see all data clustered around 0.0 across axis within a subject and across subjects. This suggests that perhaps the data was centered (zero mean). This strong consistency across subjects may aid in modeling, and may suggest that the differences across subjects in the total acceleration data may not be as helpful.
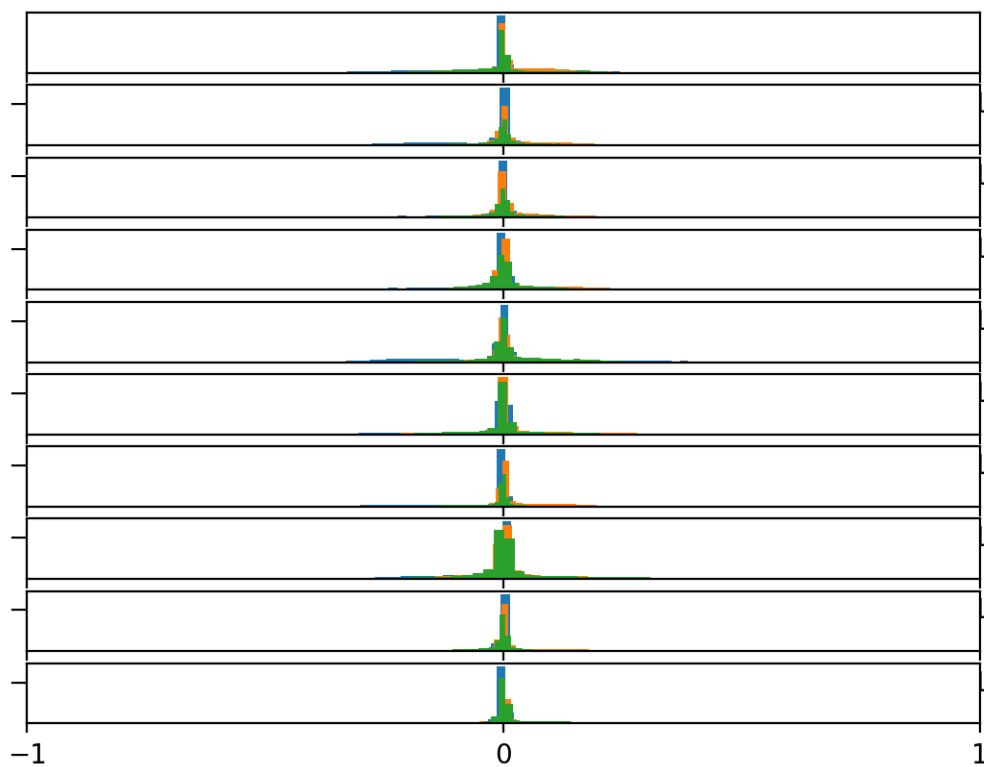

Figure 22.5: Histograms of the body acceleration data for 10 subjects.

Finally, a figure is generated summarizing the distribution the gyroscopic data for the first 10 subjects. We see a high likelihood of a Gaussian distribution for each axis across each subject centered on 0.0. The distributions are a little wider and show fatter tails, but this is an encouraging finding for modeling movement data across subjects.
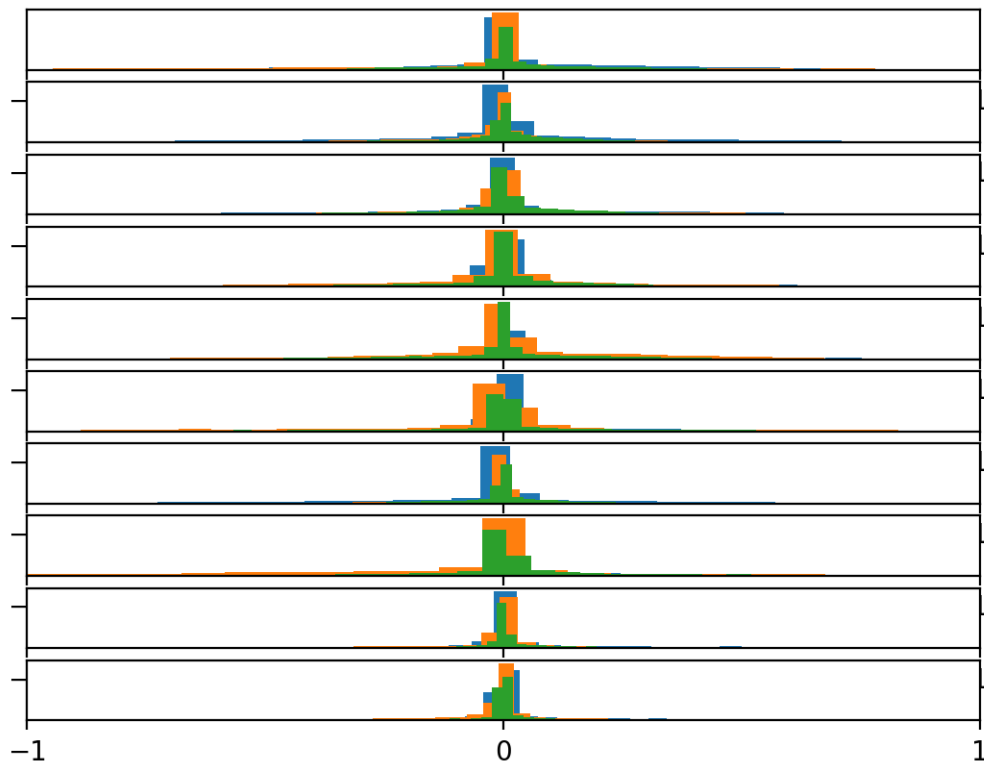
Figure 22.6: Histograms of the body gyroscope data for 10 subjects.

22.8   Plot Distribution Per Activity

We are interested in discriminating between activities based on activity data. The simplest case for this would be to discriminate between activities for a single subject. One way to investigate this would be to review the distribution of movement data for a subject by activity. We would expect to see some difference in the distribution between the movement data for different activities by a single subject.

We can review this by creating a histogram plot per activity, with the three axis of a given data type on each plot. Again, the plots can be arranged horizontally to compare the distribution of each data axis by activity. We would expect to see differences in the distributions across activities down the plots. First, we must group the traces for a subject by activity. The `data_by_activity()` function below implements this behavior.

```python
# group data by activity
def data_by_activity(X, y, activities):
  # group windows by activity
  return {a:X[y[:,0]==a, :, :] for a in activities}
```

Listing 22.24: Example of a function for splitting data by activity.

We can now create plots per activity for a given subject. The `plot_activity_histograms()` function below implements this function for the traces data for a given subject. First, the data is grouped by activity, then one subplot is created for each activity and each axis of the data type is added as a histogram.

```python
# plot histograms for each activity for a subject
def plot_activity_histograms(X, y, offset):
  # get a list of unique activities for the subject
  activity_ids = unique(y[:,0])
  # group windows by activity
  grouped = data_by_activity(X, y, activity_ids)
  # plot per activity, histograms for each axis
  pyplot.figure()
  for k in range(len(activity_ids)):
    act_id = activity_ids[k]
    # total acceleration
    for i in range(3):
      ax = pyplot.subplot(len(activity_ids), 1, k+1)
      ax.set_xlim(-1,1)
      # create histogra,
      pyplot.hist(to_series(grouped[act_id][:,:,offset+i]), bins=100)
      # create title
      pyplot.title('activity '+str(act_id), y=0, loc='left', size=10)
      # simplify axis
      pyplot.yticks([])
      pyplot.xticks([-1,0,1])
  pyplot.show()
```

Listing 22.25: Example of a function for plotting histograms of data by activity.

As in the previous section, the addition of an `offset` argument allows the same function to be called for each of the 3 groups of variables to plot at a time: total acceleration, body acceleration and gyroscopic with offsets 0, 3 and 6 respectively.

```python
# plot total acceleration histograms per activity for a subject
plot_activity_histograms(subX, suby, 0)
# plot body acceleration histograms per activity for a subject
plot_activity_histograms(subX, suby, 3)
# plot gyroscopic histograms per activity for a subject
plot_activity_histograms(subX, suby, 6)
```

Listing 22.26: Example of creating per-activity histogram plots for each of the 3 groups of variables.

The complete example is listed below.

```python
# plot histograms per activity for a subject from the har dataset
from numpy import dstack
from numpy import unique
from pandas import read_csv
from matplotlib import pyplot

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values
```

```python
# load a list of files, such as x, y, z data for a given variable
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# get all data for one subject
def data_for_subject(X, y, sub_map, sub_id):
  # get row indexes for the subject id
  ix = [i for i in range(len(sub_map)) if sub_map[i]==sub_id]
  # return the selected samples
  return X[ix, :, :], y[ix]

# convert a series of windows to a 1D list
def to_series(windows):
  series = list()
  for window in windows:
    # remove the overlap from the window
    half = int(len(window) / 2) - 1
    for value in window[-half:]:
      series.append(value)
  return series

# group data by activity
def data_by_activity(X, y, activities):
  # group windows by activity
  return {a:X[y[:,0]==a, :, :] for a in activities}

# plot histograms for each activity for a subject
def plot_activity_histograms(X, y, offset):
  # get a list of unique activities for the subject
  activity_ids = unique(y[:,0])
```

```
  # group windows by activity
  grouped = data_by_activity(X, y, activity_ids)
  # plot per activity, histograms for each axis
  pyplot.figure()
  for k in range(len(activity_ids)):
    act_id = activity_ids[k]
    # total acceleration
    for i in range(3):
      ax = pyplot.subplot(len(activity_ids), 1, k+1)
      ax.set_xlim(-1,1)
      # create histogra,
      pyplot.hist(to_series(grouped[act_id][:,:,offset+i]), bins=100)
      # create title
      pyplot.title('activity '+str(act_id), y=0, loc='left', size=10)
      # simplify axis
      pyplot.yticks([])
      pyplot.xticks([-1,0,1])
  pyplot.show()

# load data
trainX, trainy = load_dataset('train', 'HARDataset/')
# load mapping of rows to subjects
sub_map = load_file('HARDataset/train/subject_train.txt')
train_subjects = unique(sub_map)
# get the data for one subject
sub_id = train_subjects[0]
subX, suby = data_for_subject(trainX, trainy, sub_map, sub_id)
# plot total acceleration histograms per activity for a subject
plot_activity_histograms(subX, suby, 0)
# plot body acceleration histograms per activity for a subject
plot_activity_histograms(subX, suby, 3)
# plot gyroscopic histograms per activity for a subject
plot_activity_histograms(subX, suby, 6)
```

Listing 22.27: Example of plotting histograms per activity.

Running the example creates three figures, where each figure has with six subplots, one for each activity for the first subject in the train dataset. Each of the x, y, and z axes for the total acceleration data have a blue, orange, and green histogram respectively. In the first figure, can see that each activity has a different data distribution, with a marked difference between the large movement (first three activities) with the stationary activities (last three activities). Data distributions for the first three activities look Gaussian with perhaps differing means and standard deviations. Distributions for the latter activities look multi-modal (i.e. multiple peaks).
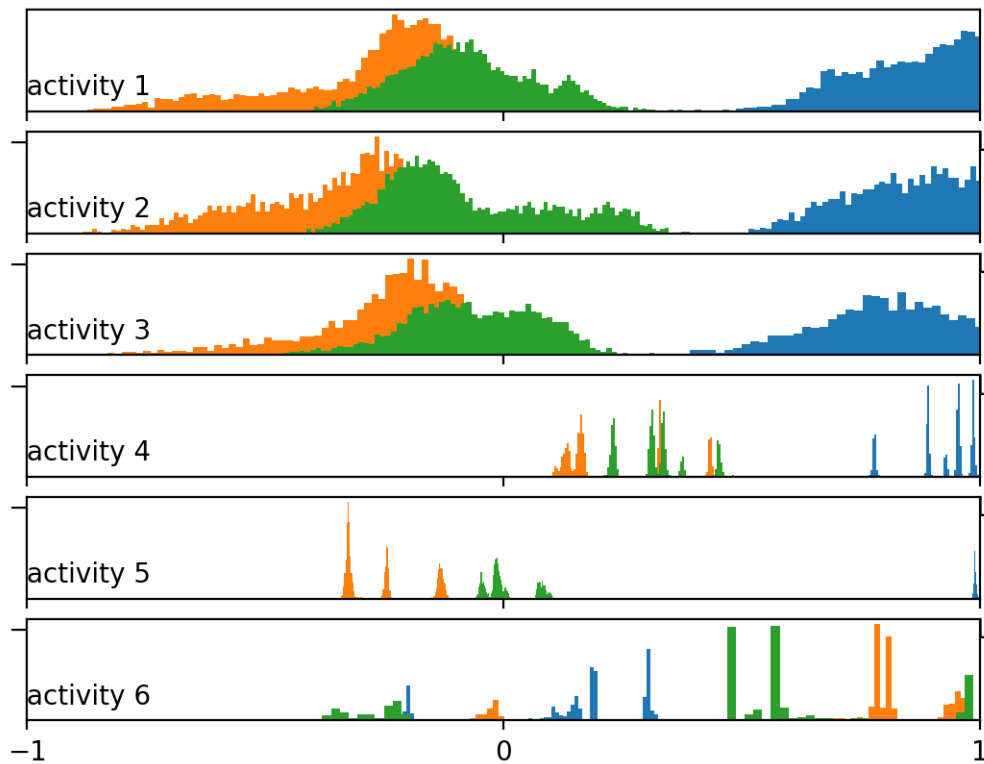
Figure 22.7: Histograms of the total acceleration data by activity.

The second figure summarizes the body acceleration distributions per activity. We can see more similar distributions across the activities amongst the in-motion vs. stationary activities. The data looks bimodal in the case of the in-motion activities and perhaps Gaussian or exponential in the case of the stationary activities. The pattern we see with the total vs. body acceleration distributions by activity mirrors what we see with the same data types across subjects in the previous section. Perhaps the total acceleration data is the key to discriminating the activities.
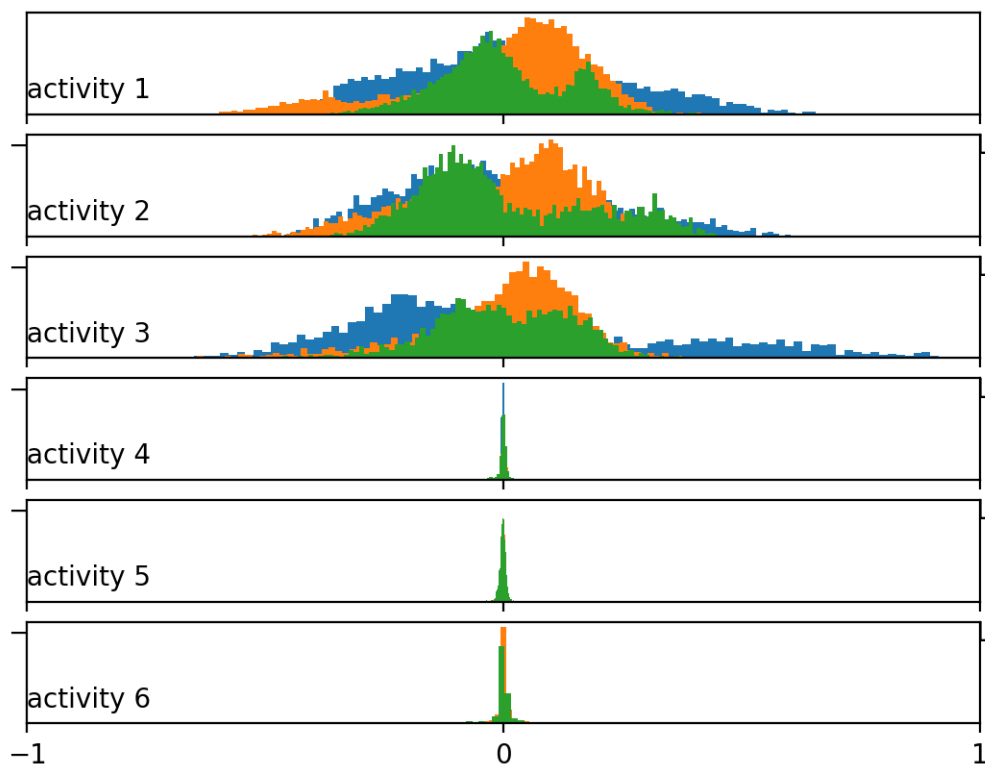
Figure 22.8: Histograms of the body acceleration data by activity.

The final figure summarizes the gyroscopic data per activity for the first subject. We can see plots with the similar pattern as the body acceleration data, although showing perhaps fat-tailed Gaussian-like distributions instead of bimodal distributions for the in-motion activities.
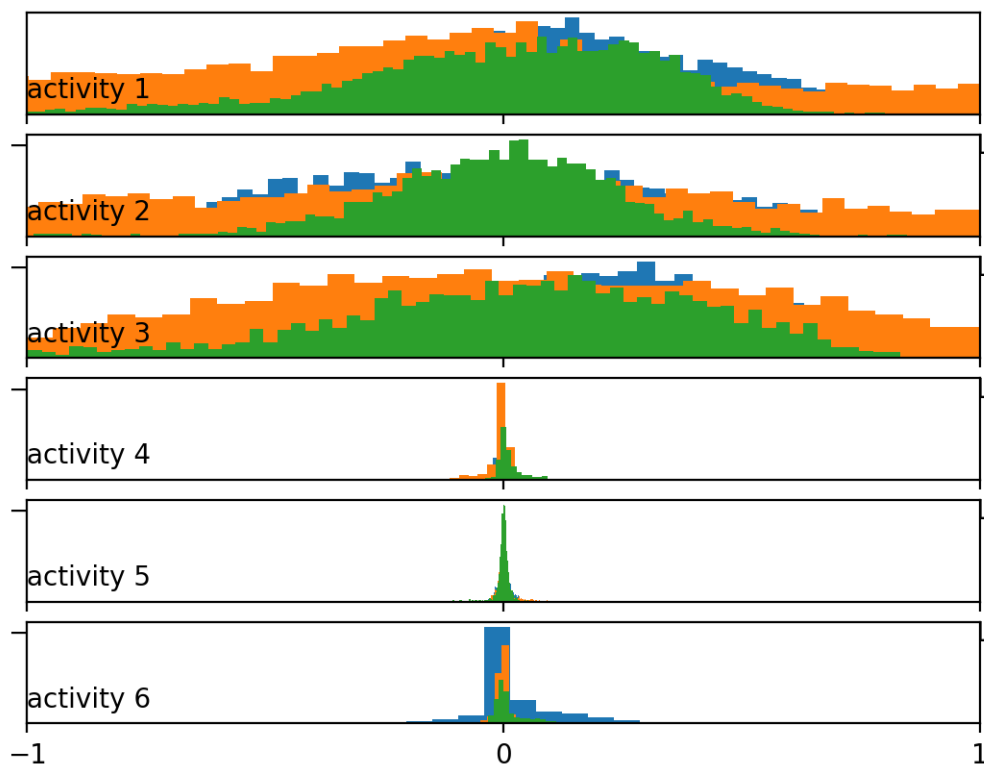
Figure 22.9: Histograms of the body gyroscope data by activity.

All of these plots were created for the first subject, and we would expect to see similar distributions and relationships for the movement data across activities for other subjects.

22.9 Plot Distribution of Activity Duration

A final area to consider is how long a subject spends on each activity. This is closely related to the balance of classes. If the activities (classes) are generally balanced within a dataset, then we expect the balance of activities for a given subject over the course of their trace would also be reasonably well balanced. We can confirm this by calculating how long (in samples or rows) each subject spends on each activity and look at the distribution of durations for each activity.

A handy way to review this data is to summarize the distributions as box plots showing the median (line), the middle 50% (box), the general extent of the data as the interquartile range (the whiskers), and outliers (as dots). The function `plot_activity_durations_by_subject()` below implements this behavior by first splitting the dataset by subject, then the subjects data by activity and counting the rows spent on each activity, before finally creating a box plot per activity of the duration measurements.

```
# plot activity durations by subject
def plot_activity_durations_by_subject(X, y, sub_map):
  # get unique subjects and activities
```

```
  subject_ids = unique(sub_map[:,0])
  activity_ids = unique(y[:,0])
  # enumerate subjects
  activity_windows = {a:list() for a in activity_ids}
  for sub_id in subject_ids:
    # get data for one subject
    _, subj_y = data_for_subject(X, y, sub_map, sub_id)
    # count windows by activity
    for a in activity_ids:
      activity_windows[a].append(len(subj_y[subj_y[:,0]==a]))
  # organize durations into a list of lists
  durations = [activity_windows[a] for a in activity_ids]
  pyplot.boxplot(durations, labels=activity_ids)
  pyplot.show()
```

Listing 22.28: Example of a function for plotting activity duration.

The complete example is listed below.

```
# plot durations of each activity by subject from the har dataset
from numpy import dstack
from numpy import unique
from pandas import read_csv
from matplotlib import pyplot

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files, such as x, y, z data for a given variable
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
```

```
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# get all data for one subject
def data_for_subject(X, y, sub_map, sub_id):
  # get row indexes for the subject id
  ix = [i for i in range(len(sub_map)) if sub_map[i]==sub_id]
  # return the selected samples
  return X[ix, :, :], y[ix]

# convert a series of windows to a 1D list
def to_series(windows):
  series = list()
  for window in windows:
    # remove the overlap from the window
    half = int(len(window) / 2) - 1
    for value in window[-half:]:
      series.append(value)
  return series

# group data by activity
def data_by_activity(X, y, activities):
  # group windows by activity
  return {a:X[y[:,0]==a, :, :] for a in activities}

# plot activity durations by subject
def plot_activity_durations_by_subject(X, y, sub_map):
  # get unique subjects and activities
  subject_ids = unique(sub_map[:,0])
  activity_ids = unique(y[:,0])
  # enumerate subjects
  activity_windows = {a:list() for a in activity_ids}
  for sub_id in subject_ids:
    # get data for one subject
    _, subj_y = data_for_subject(X, y, sub_map, sub_id)
    # count windows by activity
    for a in activity_ids:
      activity_windows[a].append(len(subj_y[subj_y[:,0]==a]))
  # organize durations into a list of lists
  durations = [activity_windows[a] for a in activity_ids]
  pyplot.boxplot(durations, labels=activity_ids)
  pyplot.show()

# load training dataset
X, y = load_dataset('train', 'HARDataset/')
# load mapping of rows to subjects
sub_map = load_file('HARDataset/train/subject_train.txt')
# plot durations
plot_activity_durations_by_subject(X, y, sub_map)
```

Listing 22.29: Example of plotting activity durations.

Running the example creates six box plots, one for each activity. Each box plot summarizes how long (in rows or the number of windows) subjects in the training dataset spent on each activity. We can see that the subjects spent more time on stationary activities (4, 5 and 6) and less time on the in motion activities (1, 2 and 3), with the distribution for 3 being the smallest,

or where time was spent least. The spread across the activities is not large, suggesting little need to trim the longer duration activities or oversampling of the in-motion activities. Although, these approaches remain available if skill of a predictive model on the in-motion activities is generally worse.
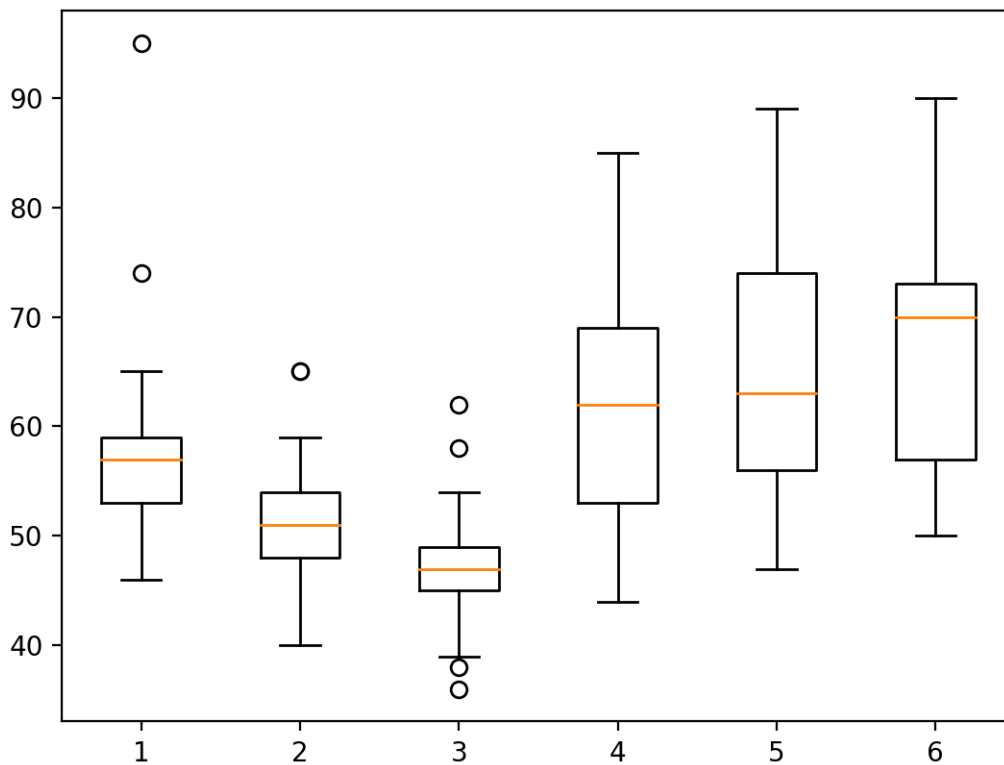

Figure 22.10: Box plot of activity durations per subject on train set.

We can create a similar box plot for the training data with the following additional lines.

```
# load test dataset
X, y = load_dataset('test', 'HARDataset/')
# load mapping of rows to subjects
sub_map = load_file('HARDataset/test/subject_test.txt')
# plot durations
plot_activity_durations_by_subject(X, y, sub_map)
```

Listing 22.30: Example of preparing the test dataset.

Running the updated example shows a similar relationship between activities. This is encouraging, suggesting that indeed the test and training dataset are reasonably representative of the whole dataset.
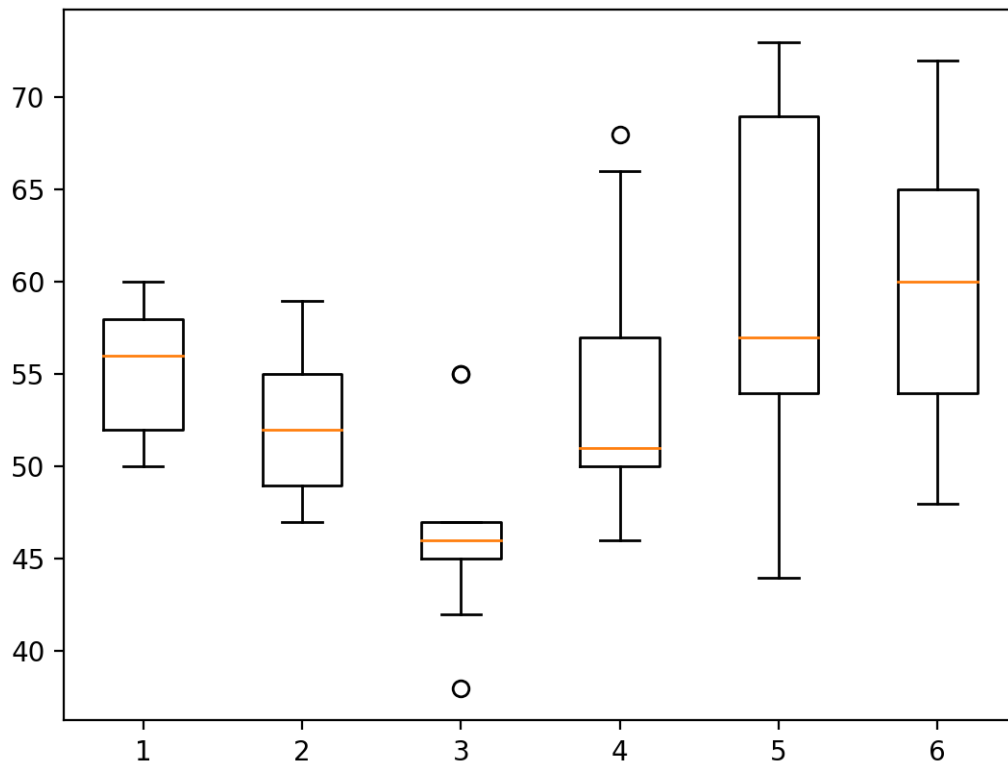
Figure 22.11: Boxplot of activity durations per subject on test set.

Now that we have explored the dataset, we can suggest some ideas for how it may be modeled.

22.10 Approach to Modeling

In this section, we summarize some approaches to modeling the activity recognition dataset. These ideas are divided into the main themes of a project.

22.10.1 Problem Framing

The first important consideration is the framing of the prediction problem. The framing of the problem as described in the original work is the prediction of activity for a new subject given their movement data, based on the movement data and activities of known subjects. We can summarize this as:

- Predict activity given a window of movement data.

This is a reasonable and useful framing of the problem. Some other possible ways to frame the provided data as a prediction problem include the following:

- Predict activity given a time step of movement data.

- Predict activity given multiple windows of movement data.

- Predict the activity sequence given multiple windows of movement data.

- Predict activity given a sequence of movement data for a pre-segmented activity.

- Predict activity cessation or transition given a time step of movement data.

- Predict a stationary or non-stationary activity given a window of movement data.

Some of these framings may be too challenging or too easy. Nevertheless, these framings provide additional ways to explore and understand the dataset.

22.10.2 Data Preparation

Some data preparation may be required prior to using the raw data to train a model. The data already appears to have been scaled to the range `[-1,1]`. Some additional data transforms that could be performed prior to modeling include:

- Normalization across subjects.

- Standardization per subject.

- Standardization across subjects.

- Axis feature selection.

- Data type feature selection.

- Signal outlier detection and removal.

- Removing windows of over-represented activities.

- Oversampling windows of under-represented activities.

- Downsampling signal data to $\frac{1}{4}$, $\frac{1}{2}$, 1, 2 or other fractions of a section.

22.10.3 Predictive Modeling

Generally, the problem is a time series multiclass classification problem. As we have seen, it may also be framed as a binary classification problem and a multi-step time series classification problem. The original paper explored the use of a classical machine learning algorithm on a version of the dataset where features were engineered from each window of data. Specifically, a modified support vector machine.

The results of an SVM on the feature-engineered version of the dataset may provide a baseline in performance on the problem. Expanding from this point, the evaluation of multiple linear, nonlinear, and ensemble machine learning algorithms on this version of the dataset may provide an improved benchmark. The focus of the problem may be on the un-engineered or raw version of the dataset. Here, a progression in model complexity may be explored in order to determine the most suitable model for the problem; some candidate models to explore include:

- Common linear, nonlinear, and ensemble machine learning algorithms.

- Multilayer Perceptron.

- Convolutional neural networks, specifically 1D CNNs.

- Recurrent neural networks, specifically LSTMs.

- Hybrids of CNNs and LSTMs such as the CNN-LSTM and the ConvLSTM.

22.11   Model Evaluation

The evaluation of the model in the original paper involved using a train/test split of the data by subject with a 70% and 30% ratio. Exploration of this pre-defined split of the data suggests that both sets are reasonably representative of the whole dataset. Another alternative methodology may be to use leave-one-out cross-validation, or LOOCV, per subject. In addition to giving the data for each subject the opportunity for being used as the withheld test set, the approach would provide a population of 30 scores that can be averaged and summarized, which may offer a more robust result.

Model performance was presented using classification accuracy and a confusion matrix, both of which are suitable for the multiclass nature of the prediction problem. Specifically, the confusion matrix will aid in determining whether some classes are easier or more challenging to predict than others, such as those for stationary activities versus those activities that involve motion.

22.12   Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **List Intuitions**. Outline the method that you believe may be most effective in making forecasts with on this dataset.

- **Apply Taxonomy**. Use the taxonomy in Chapter 2 to describe the dataset presented in this chapter.

- **Additional Analysis**. Use summary statistics and/or plots to explore one more aspect of the dataset that may provide insight into modeling this problem.

If you explore any of these extensions, I'd love to know.

22.13   Further Reading

This section provides more resources on the topic if you are looking to go deeper.

22.13.1 Papers

- *Deep Learning for Sensor-based Activity Recognition: A Survey*, 2017.
  https://arxiv.org/abs/1707.03502

- *A Public Domain Dataset for Human Activity Recognition Using Smartphones*, 2013.
  https://upcommons.upc.edu/handle/2117/20897

- *Human Activity Recognition on Smartphones using a Multiclass Hardware-Friendly Support Vector Machine*, 2012.
  https://link.springer.com/chapter/10.1007/978-3-642-35395-6_30

22.13.2 API

- `pandas.read_csv` API.
  http://pandas.pydata.org/pandas-docs/stable/generated/pandas.read_csv.html

- `numpy.dstack` API.
  https://www.numpy.org/devdocs/reference/generated/numpy.dstack.html

22.13.3 Articles

- Human Activity Recognition Dataset, UCI Machine Learning Repository.

- Activity recognition, Wikipedia.
  https://en.wikipedia.org/wiki/Activity_recognition

- Activity Recognition Experiment Using Smartphone Sensors, YouTube.
  https://www.youtube.com/watch?v=XOEN9W05_4A

22.14 Summary

In this tutorial, you discovered the Activity Recognition Using Smartphones Dataset for time series classification and how to load and explore the dataset in order to make it ready for predictive modeling. Specifically, you learned:

- How to download and load the dataset into memory.

- How to use line plots, histograms, and box plots to better understand the structure of the motion data.

- How to model the problem including framing, data preparation, modeling, and evaluation.

22.14.1 Next

In the next lesson, you will discover how to develop Machine Learning models for predicting human activities from a stream of smartphone accelerometer data.

Chapter 23

How to Develop ML Models for Human Activity Recognition

Human activity recognition is the problem of classifying sequences of accelerometer data recorded by specialized harnesses or smartphones into known well-defined movements. Classical approaches to the problem involve hand crafting features from the time series data based on fixed-sized windows and training machine learning models, such as ensembles of decision trees. The difficulty is that this feature engineering requires deep expertise in the field.

Recently, deep learning methods such as recurrent neural networks and one-dimensional convolutional neural networks, or CNNs, have been shown to provide state-of-the-art results on challenging activity recognition tasks with little or no data feature engineering, instead using feature learning on raw data. In this tutorial, you will discover how to evaluate a diverse suite of machine learning algorithms on the *Activity Recognition Using Smartphones* dataset. After completing this tutorial, you will know:

- How to load and evaluate nonlinear and ensemble machine learning algorithms on the feature-engineered version of the activity recognition dataset.

- How to load and evaluate machine learning algorithms on the raw signal data for the activity recognition dataset.

- How to define reasonable lower and upper bounds on the expected performance of more sophisticated algorithms capable of feature learning, such as deep learning methods.

Let's get started.

23.1 Tutorial Overview

This tutorial is divided into three parts; they are:

1. Activity Recognition Using Smartphones Dataset

2. Modeling Feature Engineered Data

3. Modeling Raw Data

23.2 Activity Recognition Using Smartphones Dataset

Human Activity Recognition, or HAR for short, is the problem of predicting what a person is doing based on a trace of their movement using sensors. A standard human activity recognition dataset is the *Activity Recognition Using Smartphones* made available in 2012. For more information on this dataset, see Chapter 22. The data is provided as a single zip file that is about 58 megabytes in size. A direct for downloading the dataset is provided below:

- HAR_Smartphones.zip [1]

Download the dataset and unzip all files into a new directory in your current working directory named HARDataset.

23.3 Modeling Feature Engineered Data

In this section, we will develop code to load the feature-engineered version of the dataset and evaluate a suite of nonlinear machine learning algorithms, including SVM used in the original paper. The goal is to achieve at least 89% accuracy on the test dataset. The results of methods using the feature-engineered version of the dataset provide a baseline for any methods developed for the raw data version. This section is divided into five parts; they are:

1. Load Dataset

2. Define Models

3. Evaluate Models

4. Summarize Results

5. Complete Example

23.3.1 Load Dataset

The first step is to load the train and test input (X) and output (y) data. Specifically, the following files:

- HARDataset/train/X_train.txt

- HARDataset/train/y_train.txt

- HARDataset/test/X_test.txt

- HARDataset/test/y_test.txt

The input data is in CSV format where columns are separated via whitespace. Each of these files can be loaded as a NumPy array. The load_file() function below loads a dataset given the file path to the file and returns the loaded data as a NumPy array.

---

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/HAR_Smartphones.zip

```
# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values
```

Listing 23.1: Example of a function for loading a single file.

We can call this function to load the X and y files for a given train or test set group, given the similarity in directory layout and filenames. The `load_dataset_group()` function below will load both of these files for a group and return the X and y elements as NumPy arrays. This function can then be used to load the X and y elements for both the train and test groups.

```
# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  # load input data
  X = load_file(prefix + group + '/X_'+group+'.txt')
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y
```

Listing 23.2: Example of a function for loading a group of files.

Finally, we can load both the train and test dataset and return them as NumPy arrays ready for fitting and evaluating machine learning models.

```
# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  print(trainX.shape, trainy.shape)
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  print(testX.shape, testy.shape)
  # flatten y
  trainy, testy = trainy[:,0], testy[:,0]
  print(trainX.shape, trainy.shape, testX.shape, testy.shape)
  return trainX, trainy, testX, testy
```

Listing 23.3: Example of a function for loading the entire dataset.

We can call this function to load all of the required data; for example:

```
# load dataset
trainX, trainy, testX, testy = load_dataset()
```

Listing 23.4: Example of calling a function to load the dataset.

23.3.2 Define Models

Next, we can define a list of machine learning models to evaluate on this problem. We will evaluate the models using default configurations. We are not looking for optimal configurations of these models at this point, just a general idea of how well sophisticated models with default configurations perform on this problem. We will evaluate a diverse set of nonlinear and ensemble machine learning algorithms, specifically:

Nonlinear Algorithms:

- k-Nearest Neighbors

- Classification and Regression Tree

- Support Vector Machine

- Naive Bayes

Ensemble Algorithms:

- Bagged Decision Trees

- Random Forest

- Extra Trees

- Gradient Boosting Machine

We will define the models and store them in a dictionary that maps the model object to a short name that will help in analyzing the results. The `define_models()` function below defines the eight models that we will evaluate.

```python
# create a dict of standard models to evaluate {name:object}
def define_models(models=dict()):
 # nonlinear models
 models['knn'] = KNeighborsClassifier(n_neighbors=7)
 models['cart'] = DecisionTreeClassifier()
 models['svm'] = SVC()
 models['bayes'] = GaussianNB()
 # ensemble models
 models['bag'] = BaggingClassifier(n_estimators=100)
 models['rf'] = RandomForestClassifier(n_estimators=100)
 models['et'] = ExtraTreesClassifier(n_estimators=100)
 models['gbm'] = GradientBoostingClassifier(n_estimators=100)
 print('Defined %d models' % len(models))
 return models
```

Listing 23.5: Example of a function for defining machine learning models.

This function is quite extensible and you can easily update to define any machine learning models or model configurations you wish.

23.3.3 Evaluate Models

The next step is to evaluate the defined models in the loaded dataset. This step is divided into the evaluation of a single model and the evaluation of all of the models. We will evaluate a single model by first fitting it on the training dataset, making a prediction on the test dataset, and then evaluating the prediction using a metric. In this case we will use classification accuracy that will capture the performance (or error) of a model given the balance observations across the six activities (or classes). The `evaluate_model()` function below implements this behavior, evaluating a given model and returning the classification accuracy as a percentage.

```
# evaluate a single model
def evaluate_model(trainX, trainy, testX, testy, model):
  # fit the model
  model.fit(trainX, trainy)
  # make predictions
  yhat = model.predict(testX)
  # evaluate predictions
  accuracy = accuracy_score(testy, yhat)
  return accuracy * 100.0
```

Listing 23.6: Example of a function for evaluating a machine learning model.

We can now call the `evaluate_model()` function repeatedly for each of the defined model. The `evaluate_models()` function below implements this behavior, taking the dictionary of defined models, and returns a dictionary of model names mapped to their classification accuracy. Because the evaluation of the models may take a few minutes, the function prints the performance of each model after it is evaluated as some verbose feedback.

```
# evaluate a dict of models {name:object}, returns {name:score}
def evaluate_models(trainX, trainy, testX, testy, models):
  results = dict()
  for name, model in models.items():
    # evaluate the model
    results[name] = evaluate_model(trainX, trainy, testX, testy, model)
    # show process
    print('>%s: %.3f' % (name, results[name]))
  return results
```

Listing 23.7: Example of a function for evaluating multiple machine learning models.

23.3.4   Summarize Results

The final step is to summarize the findings. We can sort all of the results by the classification accuracy in descending order because we are interested in maximizing accuracy. The results of the evaluated models can then be printed, clearly showing the relative rank of each of the evaluated models. The `summarize_results()` function below implements this behavior.

```
# print and plot the results
def summarize_results(results, maximize=True):
  # create a list of (name, mean(scores)) tuples
  mean_scores = [(k,v) for k,v in results.items()]
  # sort tuples by mean score
  mean_scores = sorted(mean_scores, key=lambda x: x[1])
  # reverse for descending order (e.g. for accuracy)
  if maximize:
    mean_scores = list(reversed(mean_scores))
  print()
  for name, score in mean_scores:
    print('Name=%s, Score=%.3f' % (name, score))
```

Listing 23.8: Example of a function for summarizing model performance.

23.3.5 Complete Example

We know that we have all of the pieces in place. The complete example of evaluating a suite of eight machine learning models on the feature-engineered version of the dataset is listed below.

```python
# spot check ml algorithms on engineered-features from the har dataset
from pandas import read_csv
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import GradientBoostingClassifier

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  # load input data
  X = load_file(prefix + group + '/X_'+group+'.txt')
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # flatten y
  trainy, testy = trainy[:,0], testy[:,0]
  return trainX, trainy, testX, testy

# create a dict of standard models to evaluate {name:object}
def define_models(models=dict()):
  # nonlinear models
  models['knn'] = KNeighborsClassifier(n_neighbors=7)
  models['cart'] = DecisionTreeClassifier()
  models['svm'] = SVC()
  models['bayes'] = GaussianNB()
  # ensemble models
  models['bag'] = BaggingClassifier(n_estimators=100)
  models['rf'] = RandomForestClassifier(n_estimators=100)
  models['et'] = ExtraTreesClassifier(n_estimators=100)
  models['gbm'] = GradientBoostingClassifier(n_estimators=100)
  print('Defined %d models' % len(models))
  return models

# evaluate a single model
```

```python
def evaluate_model(trainX, trainy, testX, testy, model):
  # fit the model
  model.fit(trainX, trainy)
  # make predictions
  yhat = model.predict(testX)
  # evaluate predictions
  accuracy = accuracy_score(testy, yhat)
  return accuracy * 100.0

# evaluate a dict of models {name:object}, returns {name:score}
def evaluate_models(trainX, trainy, testX, testy, models):
  results = dict()
  for name, model in models.items():
    # evaluate the model
    results[name] = evaluate_model(trainX, trainy, testX, testy, model)
    # show process
    print('>%s: %.3f' % (name, results[name]))
  return results

# print and plot the results
def summarize_results(results, maximize=True):
  # create a list of (name, mean(scores)) tuples
  mean_scores = [(k,v) for k,v in results.items()]
  # sort tuples by mean score
  mean_scores = sorted(mean_scores, key=lambda x: x[1])
  # reverse for descending order (e.g. for accuracy)
  if maximize:
    mean_scores = list(reversed(mean_scores))
  print()
  for name, score in mean_scores:
    print('Name=%s, Score=%.3f' % (name, score))

# load dataset
trainX, trainy, testX, testy = load_dataset()
# get model list
models = define_models()
# evaluate models
results = evaluate_models(trainX, trainy, testX, testy, models)
# summarize results
summarize_results(results)
```

Listing 23.9: Example of evaluating machine learning models for activity recognition.

Running the example first loads the train and test datasets. The eight models are then evaluated in turn, printing the performance for each. Finally, a rank of the models by their performance on the test set is displayed. We can see that both the ExtraTrees ensemble method and the Support Vector Machines nonlinear methods achieve a performance of about 94% accuracy on the test set. This is a great result, exceeding the reported 89% by SVM in the original paper.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Defined 8 models
```

```
>knn: 90.329
>cart: 86.020
>svm: 94.028
>bayes: 77.027
>bag: 89.820
>rf: 92.772
>et: 94.028
>gbm: 93.756

Name=et, Score=94.028
Name=svm, Score=94.028
Name=gbm, Score=93.756
Name=rf, Score=92.772
Name=knn, Score=90.329
Name=bag, Score=89.820
Name=cart, Score=86.020
Name=bayes, Score=77.027
```

Listing 23.10: Example output from evaluating machine learning models for activity recognition.

These results show what is possible given domain expertise in the preparation of the data and the engineering of domain-specific features. As such, these results can be taken as a performance upper-bound of what could be pursued through more advanced methods that may be able to automatically learn features as part of fitting the model, such as deep learning methods. Any such advanced methods would be fit and evaluated on the raw data from which the engineered features were derived. And as such, the performance of machine learning algorithms evaluated on that data directly may provide an expected lower bound on the performance of any more advanced methods. We will explore this in the next section.

23.4 Modeling Raw Data

We can use the same framework for evaluating machine learning models on the raw data. The raw data does require some more work to load. There are three main signal types in the raw data: total acceleration, body acceleration, and body gyroscope. Each has three axes of data. This means that there are a total of nine variables for each time step. Further, each series of data has been partitioned into overlapping windows of 2.65 seconds of data, or 128 time steps. These windows of data correspond to the windows of engineered features (rows) in the previous section.

This means that one row of data has 128×9 or 1,152 elements. This is a little less than double the size of the 561 element vectors in the previous section and it is likely that there is some redundant data. The signals are stored in the `/Inertial Signals/` directory under the train and test subdirectories. Each axis of each signal is stored in a separate file, meaning that each of the train and test datasets have nine input files to load and one output file to load. We can batch the loading of these files into groups given the consistent directory structures and file naming conventions.

First, we can load all data for a given group into a single three-dimensional NumPy array, where the dimensions of the array are `[samples, timesteps, features]`. To make this clearer, there are 128 time steps and nine features, where the number of samples is the number of rows in any given raw signal data file. The `load_group()` function below implements this behavior.

The `dstack()` NumPy function allows us to stack each of the loaded 3D arrays into a single 3D array where the variables are separated on the third dimension (features).

```python
# load a list of files into a 3D array of [samples, timesteps, features]
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

Listing 23.11: Example of a function for loading a group of raw files.

We can use this function to load all input signal data for a given group, such as train or test. The `load_dataset_group()` function below loads all input signal data and the output data for a single group using the consistent naming conventions between the directories.

```python
# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y
```

Listing 23.12: Example of a function for loading multiple groups of raw files.

Finally, we can load each of the train and test datasets. As part of preparing the loaded data, we must flatten the windows and features into one long vector. We can do this with the NumPy reshape function and convert the three dimensions of `[samples, timesteps, features]` into the two dimensions of `[samples, timesteps × features]`. The `load_dataset()` function below implements this behavior and returns the train and test X and y elements ready for fitting and evaluating the defined models.

```python
# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  print(trainX.shape, trainy.shape)
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  print(testX.shape, testy.shape)
```

```
  # flatten X
  trainX = trainX.reshape((trainX.shape[0], trainX.shape[1] * trainX.shape[2]))
  testX = testX.reshape((testX.shape[0], testX.shape[1] * testX.shape[2]))
  # flatten y
  trainy, testy = trainy[:,0], testy[:,0]
  print(trainX.shape, trainy.shape, testX.shape, testy.shape)
  return trainX, trainy, testX, testy
```

Listing 23.13: Example of a function for loading the raw dataset.

Putting this all together, the complete example is listed below.

```
# spot check on raw data from the har dataset
from numpy import dstack
from pandas import read_csv
from sklearn.metrics import accuracy_score
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.naive_bayes import GaussianNB
from sklearn.ensemble import BaggingClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.ensemble import ExtraTreesClassifier
from sklearn.ensemble import GradientBoostingClassifier

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files into a 3D array of [samples, timesteps, features]
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
```

```python
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # flatten X
  trainX = trainX.reshape((trainX.shape[0], trainX.shape[1] * trainX.shape[2]))
  testX = testX.reshape((testX.shape[0], testX.shape[1] * testX.shape[2]))
  # flatten y
  trainy, testy = trainy[:,0], testy[:,0]
  return trainX, trainy, testX, testy

# create a dict of standard models to evaluate {name:object}
def define_models(models=dict()):
  # nonlinear models
  models['knn'] = KNeighborsClassifier(n_neighbors=7)
  models['cart'] = DecisionTreeClassifier()
  models['svm'] = SVC()
  models['bayes'] = GaussianNB()
  # ensemble models
  models['bag'] = BaggingClassifier(n_estimators=100)
  models['rf'] = RandomForestClassifier(n_estimators=100)
  models['et'] = ExtraTreesClassifier(n_estimators=100)
  models['gbm'] = GradientBoostingClassifier(n_estimators=100)
  print('Defined %d models' % len(models))
  return models

# evaluate a single model
def evaluate_model(trainX, trainy, testX, testy, model):
  # fit the model
  model.fit(trainX, trainy)
  # make predictions
  yhat = model.predict(testX)
  # evaluate predictions
  accuracy = accuracy_score(testy, yhat)
  return accuracy * 100.0

# evaluate a dict of models {name:object}, returns {name:score}
def evaluate_models(trainX, trainy, testX, testy, models):
  results = dict()
  for name, model in models.items():
    # evaluate the model
    results[name] = evaluate_model(trainX, trainy, testX, testy, model)
    # show process
    print('>%s: %.3f' % (name, results[name]))
  return results

# print and plot the results
def summarize_results(results, maximize=True):
  # create a list of (name, mean(scores)) tuples
  mean_scores = [(k,v) for k,v in results.items()]
  # sort tuples by mean score
```

```
  mean_scores = sorted(mean_scores, key=lambda x: x[1])
  # reverse for descending order (e.g. for accuracy)
  if maximize:
    mean_scores = list(reversed(mean_scores))
  print()
  for name, score in mean_scores:
    print('Name=%s, Score=%.3f' % (name, score))

# load dataset
trainX, trainy, testX, testy = load_dataset()
# get model list
models = define_models()
# evaluate models
results = evaluate_models(trainX, trainy, testX, testy, models)
# summarize results
summarize_results(results)
```

Listing 23.14: Example of evaluating machine learning models for activity recognition on the raw dataset.

Running the example first loads the dataset. Next the eight defined models are evaluated in turn. The final results suggest that ensembles of decision trees perform the best on the raw data. Gradient Boosting and Extra Trees perform the best with about 87% and 86% accuracy, about seven points below the best performing models on the feature-engineered version of the dataset. It is encouraging that the Extra Trees ensemble method performed well on both datasets; it suggests it and similar tree ensemble methods may be suited to the problem, at least in this simplified framing. We can also see the drop of SVM to about 72% accuracy. The good performance of ensembles of decision trees may suggest the need for feature selection and the ensemble methods ability to select those features that are most relevant to predicting the associated activity.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
Defined 8 models
>knn: 61.893
>cart: 72.141
>svm: 76.960
>bayes: 72.480
>bag: 84.527
>rf: 84.662
>et: 86.902
>gbm: 87.615

Name=gbm, Score=87.615
Name=et, Score=86.902
Name=rf, Score=84.662
Name=bag, Score=84.527
Name=svm, Score=76.960
Name=bayes, Score=72.480
Name=cart, Score=72.141
Name=knn, Score=61.893
```

Listing 23.15: Example output from evaluating machine learning models for activity recognition on the raw dataset.

As noted in the previous section, these results provide a lower-bound on accuracy for any more sophisticated methods that may attempt to learn higher order features automatically (e.g. via feature learning in deep learning methods) from the raw data. In summary, the bounds for such methods extend on this dataset from about 87% accuracy with GBM on the raw data to about 94% with Extra Trees and SVM on the highly processed dataset, [87% to 94%].

23.5 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **More Algorithms**. Only eight machine learning algorithms were evaluated on the problem; try some linear methods and perhaps some more nonlinear and ensemble methods.

- **Algorithm Tuning**. No tuning of the machine learning algorithms was performed; mostly default configurations were used. Pick a method such as SVM, ExtraTrees, or Gradient Boosting and grid search a suite of different hyperparameter configurations to see if you can further lift performance on the problem.

- **Data Scaling**. The data is already scaled to [-1,1], perhaps per subject. Explore whether additional scaling, such as standardization, can result in better performance, perhaps on methods sensitive to such scaling such as kNN.

If you explore any of these extensions, I'd love to know.

23.6 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- scikit-learn: Machine Learning in Python.
  http://scikit-learn.org/stable/

- `sklearn.metrics.accuracy_score` API.
  http://scikit-learn.org/stable/modules/generated/sklearn.metrics.accuracy_score.html

- `sklearn.neighbors` API.
  http://scikit-learn.org/stable/modules/classes.html#module-sklearn.neighbors

- `sklearn.tree` API.
  http://scikit-learn.org/stable/modules/classes.html#module-sklearn.tree

- `sklearn.svm` API.
  http://scikit-learn.org/stable/modules/classes.html#module-sklearn.svm

- `sklearn.naive_bayes` API.
  http://scikit-learn.org/stable/modules/classes.html#module-sklearn.naive_bayes

- `sklearn.ensemble` API.
  http://scikit-learn.org/stable/modules/classes.html#module-sklearn.ensemble

23.7   Summary

In this tutorial, you discovered how to evaluate a diverse suite of machine learning algorithms on the *Activity Recognition Using Smartphones* dataset. Specifically, you learned:

- How to load and evaluate nonlinear and ensemble machine learning algorithms on the feature-engineered version of the activity recognition dataset.

- How to load and evaluate machine learning algorithms on the raw signal data for the activity recognition dataset.

- How to define reasonable lower and upper bounds on the expected performance of more sophisticated algorithms capable of feature learning, such as deep learning methods.

23.7.1   Next

In the next lesson, you will discover how to develop Convolutional Neural Network models for predicting human activities from a stream of smartphone accelerometer data.

Chapter 24

How to Develop CNNs for Human Activity Recognition

Human activity recognition is the problem of classifying sequences of accelerometer data recorded by specialized harnesses or smartphones into known well-defined movements. Classical approaches to the problem involve hand crafting features from the time series data based on fixed-sized windows and training machine learning models, such as ensembles of decision trees. The difficulty is that this feature engineering requires deep expertise in the field. Recently, deep learning methods such as recurrent neural networks and one-dimensional convolutional neural networks, or CNNs, have been shown to provide state-of-the-art results on challenging activity recognition tasks with little or no data feature engineering, instead using feature learning on raw data. In this tutorial, you will discover how to develop one-dimensional convolutional neural networks for time series classification on the problem of human activity recognition. After completing this tutorial, you will know:

- How to load and prepare the data for a standard human activity recognition dataset and develop a single 1D CNN model that achieves excellent performance on the raw data.

- How to further tune the performance of the model, including data transformation, filter maps, and kernel sizes.

- How to develop a sophisticated multi-headed one-dimensional convolutional neural network model that provides an ensemble-like result.

Let's get started.

24.1   Tutorial Overview

This tutorial is divided into four parts; they are:

1. Activity Recognition Using Smartphones Dataset

2. CNN for Activity Recognition

3. Tuned CNN Model

4. Multi-headed CNN Model

491

24.2 Activity Recognition Using Smartphones Dataset

Human Activity Recognition, or HAR for short, is the problem of predicting what a person is doing based on a trace of their movement using sensors. A standard human activity recognition dataset is the *Activity Recognition Using Smartphones* made available in 2012. For more information on this dataset, see Chapter 22. The data is provided as a single zip file that is about 58 megabytes in size. A direct for downloading the dataset is provided below:

- `HAR_Smartphones.zip` [1]

Download the dataset and unzip all files into a new directory in your current working directory named `HARDataset`.

24.3 CNN for Activity Recognition

In this section, we will develop a one-dimensional convolutional neural network model (1D CNN) for the human activity recognition dataset. Convolutional neural network models were developed for image classification problems, where the model learns an internal representation of a two-dimensional input, in a process referred to as feature learning. Although we refer to the model as 1D, it supports multiple dimensions of input as separate channels, like the color channels of an image (red, green and blue). This same process can be harnessed on one-dimensional sequences of data, such as in the case of acceleration and gyroscopic data for human activity recognition. The model learns to extract features from sequences of observations and how to map the internal features to different activity types. For more information on CNNs for time series forecasting, see Chapter 8.

The benefit of using CNNs for sequence classification is that they can learn from the raw time series data directly, and in turn do not require domain expertise to manually engineer input features. The model can learn an internal representation of the time series data and ideally achieve comparable performance to models fit on a version of the dataset with engineered features. This section is divided into 4 parts; they are:

1. Load Data

2. Fit and Evaluate Model

3. Summarize Results

4. Complete Example

Some of the details on loading and preparing the dataset were covered in Chapters 22 and 23. There is some repetition here in order to customize the data preparation and prediction evaluation for deep learning models.

---

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/HAR_Smartphones.zip

24.3.1 Load Data

The first step is to load the raw dataset into memory. There are three main signal types in the raw data: total acceleration, body acceleration, and body gyroscope. Each has three axes of data. This means that there are a total of nine variables for each time step. Further, each series of data has been partitioned into overlapping windows of 2.65 seconds of data, or 128 time steps. These windows of data correspond to the windows of engineered features (rows) in the previous section.

This means that one row of data has (128 × 9), or 1,152, elements. This is a little less than double the size of the 561 element vectors in the previous section and it is likely that there is some redundant data. The signals are stored in the `/Inertial Signals/` directory under the train and test subdirectories. Each axis of each signal is stored in a separate file, meaning that each of the train and test datasets have nine input files to load and one output file to load. We can batch the loading of these files into groups given the consistent directory structures and file naming conventions. The input data is in CSV format where columns are separated by whitespace. Each of these files can be loaded as a NumPy array. The `load_file()` function below loads a dataset given the file path to the file and returns the loaded data as a NumPy array.

```
# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values
```

Listing 24.1: Example of a function for loading a single file.

We can then load all data for a given group (train or test) into a single three-dimensional NumPy array, where the dimensions of the array are `[samples, timesteps, features]`. To make this clearer, there are 128 time steps and nine features, where the number of samples is the number of rows in any given raw signal data file. The `load_group()` function below implements this behavior. The `dstack()` NumPy function allows us to stack each of the loaded 3D arrays into a single 3D array where the variables are separated on the third dimension (features).

```
# load a list of files into a 3D array of [samples, timesteps, features]
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

Listing 24.2: Example of a function for loading a group of files.

We can use this function to load all input signal data for a given group, such as train or test. The `load_dataset_group()` function below loads all input signal data and the output data for a single group using the consistent naming conventions between the train and test directories.

```
# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
```

```
# total acceleration
filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
    'total_acc_z_'+group+'.txt']
# body acceleration
filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
    'body_acc_z_'+group+'.txt']
# body gyroscope
filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
    'body_gyro_z_'+group+'.txt']
# load input data
X = load_group(filenames, filepath)
# load class output
y = load_file(prefix + group + '/y_'+group+'.txt')
return X, y
```

Listing 24.3: Example of a function for loading a dataset group of files.

Finally, we can load each of the train and test datasets. The output data is defined as an integer for the class number. We must one hot encode these class integers so that the data is suitable for fitting a neural network multiclass classification model. We can do this by calling the `to_categorical()` Keras function. The `load_dataset()` function below implements this behavior and returns the train and test X and y elements ready for fitting and evaluating the defined models.

```
# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  print(trainX.shape, trainy.shape)
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  print(testX.shape, testy.shape)
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  print(trainX.shape, trainy.shape, testX.shape, testy.shape)
  return trainX, trainy, testX, testy
```

Listing 24.4: Example of a function for loading the entire dataset.

24.3.2 Fit and Evaluate Model

Now that we have the data loaded into memory ready for modeling, we can define, fit, and evaluate a 1D CNN model. We can define a function named `evaluate_model()` that takes the train and test dataset, fits a model on the training dataset, evaluates it on the test dataset, and returns an estimate of the model's performance. First, we must define the CNN model using the Keras deep learning library. The model requires a three-dimensional input with [`samples, timesteps, features`].

This is exactly how we have loaded the data, where one sample is one window of the time series data, each window has 128 time steps, and a time step has nine variables or features. The

output for the model will be a six-element vector containing the probability of a given window belonging to each of the six activity types. These input and output dimensions are required when fitting the model, and we can extract them from the provided training dataset.

```
# define data shape
n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
```

Listing 24.5: Example of a model shape based on data shape.

The model is defined as a Sequential Keras model, for simplicity. We will define the model as having two 1D CNN layers, followed by a dropout layer for regularization, then a pooling layer. It is common to define CNN layers in groups of two in order to give the model a good chance of learning features from the input data. CNNs learn very quickly, so the dropout layer is intended to help slow down the learning process and hopefully result in a better final model. The pooling layer reduces the learned features to $\frac{1}{4}$ their size, consolidating them to only the most essential elements. After the CNN and pooling, the learned features are flattened to one long vector and pass through a fully connected layer before the output layer used to make a prediction. The fully connected layer ideally provides a buffer between the learned features and the output with the intent of interpreting the learned features before making a prediction.

For this model, we will use a standard configuration of 64 parallel feature maps and a kernel size of 3. The feature maps are the number of times the input is processed or interpreted, whereas the kernel size is the number of input time steps considered as the input sequence is read or processed onto the feature maps. The efficient Adam version of stochastic gradient descent will be used to optimize the network, and the categorical cross entropy loss function will be used given that we are learning a multiclass classification problem. The definition of the model is listed below.

```
# define the CNN model
model = Sequential()
model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
    input_shape=(n_timesteps,n_features)))
model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 24.6: Example of defining a 1D CNN model.

The model is fit for a fixed number of epochs, in this case 10, and a batch size of 32 samples will be used, where 32 windows of data will be exposed to the model before the weights of the model are updated. Once the model is fit, it is evaluated on the test dataset and the accuracy of the fit model on the test dataset is returned. The complete `evaluate_model()` function is listed below.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  verbose, epochs, batch_size = 0, 10, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  model = Sequential()
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
```

```
model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
model.add(Dropout(0.5))
model.add(MaxPooling1D(pool_size=2))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
# fit network
model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
# evaluate model
_, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
return accuracy
```

Listing 24.7: Example of a function for defining, fitting and evaluating a 1D CNN model.

There is nothing special about the network structure or chosen hyperparameters; they are just a starting point for this problem.

24.3.3 Summarize Results

We cannot judge the skill of the model from a single evaluation. The reason for this is that neural networks are stochastic, meaning that a different specific model will result when training the same model configuration on the same data. This is a feature of the network in that it gives the model its adaptive ability, but requires a slightly more complicated evaluation of the model. We will repeat the evaluation of the model multiple times, then summarize the performance of the model across each of those runs. For example, we can call `evaluate_model()` a total of 10 times. This will result in a population of model evaluation scores that must be summarized.

```
# repeat experiment
scores = list()
for r in range(repeats):
  score = evaluate_model(trainX, trainy, testX, testy)
  score = score * 100.0
  print('>#%d: %.3f' % (r+1, score))
  scores.append(score)
```

Listing 24.8: Example of repeating a model evaluation experiment.

We can summarize the sample of scores by calculating and reporting the mean and standard deviation of the performance. The mean gives the average accuracy of the model on the dataset, whereas the standard deviation gives the average variance of the accuracy from the mean. The function `summarize_results()` below summarizes the results of a run.

```
# summarize scores
def summarize_results(scores):
  print(scores)
  m, s = mean(scores), std(scores)
  print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))
```

Listing 24.9: Example of a function for summarizing model performance.

We can bundle up the repeated evaluation, gathering of results, and summarization of results into a main function for the experiment, called `run_experiment()`, listed below. By default, the model is evaluated 10 times before the performance of the model is reported.

```
# run an experiment
def run_experiment(repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
  scores = list()
  for r in range(repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    score = score * 100.0
    print('>#%d: %.3f' % (r+1, score))
    scores.append(score)
  # summarize results
  summarize_results(scores)
```

Listing 24.10: Example of a function for driving the experiment.

24.3.4   Complete Example

Now that we have all of the pieces, we can tie them together into a worked example. The complete code listing is provided below.

```
# cnn model for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
```

```python
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  verbose, epochs, batch_size = 0, 10, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  model = Sequential()
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
  model.add(Dropout(0.5))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy

# summarize scores
def summarize_results(scores):
  print(scores)
  m, s = mean(scores), std(scores)
  print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
```

```python
def run_experiment(repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
  scores = list()
  for r in range(repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    score = score * 100.0
    print('>#%d: %.3f' % (r+1, score))
    scores.append(score)
  # summarize results
  summarize_results(scores)

# run the experiment
run_experiment()
```

Listing 24.11: Example of evaluating a 1D CNN model for activity recognition.

Running the example first loads the dataset. The models are created and evaluated and a debug message is printed for each. Finally, the sample of scores is printed followed by the mean and standard deviation. We can see that the model performed well achieving a classification accuracy of about 90.9% trained on the raw dataset, with a standard deviation of about 1.3. This is a good result, considering that the original paper published a result of 89%, trained on the dataset with heavy domain-specific feature engineering, not the raw dataset.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
>#1: 91.347
>#2: 91.551
>#3: 90.804
>#4: 90.058
>#5: 89.752
>#6: 90.940
>#7: 91.347
>#8: 87.547
>#9: 92.637
>#10: 91.890

[91.34713267729894, 91.55072955548015, 90.80420766881574, 90.05768578215134,
    89.75229046487954, 90.93993892093654, 91.34713267729894, 87.54665761791652,
    92.63657957244655, 91.89005768578215]

Accuracy: 90.787% (+/-1.341)
```

Listing 24.12: Example output from evaluating a 1D CNN model for activity recognition.

Now that we have seen how to load the data and fit a 1D CNN model, we can investigate whether we can further lift the skill of the model with some hyperparameter tuning.

24.4 Tuned CNN Model

In this section, we will tune the model in an effort to further improve performance on the problem. We will look at three main areas:

1. Data Preparation

2. Number of Filters

3. Size of Kernel

24.4.1   Data Preparation

In the previous section, we did not perform any data preparation. We used the data as-is. Each of the main sets of data (body acceleration, body gyroscopic, and total acceleration) have been scaled to the range -1, 1. It is not clear if the data was scaled per-subject or across all subjects. One possible transform that may result in an improvement is to standardize the observations prior to fitting a model.

Standardization refers to shifting the distribution of each variable such that it has a mean of zero and a standard deviation of 1. It really only makes sense if the distribution of each variable is Gaussian. We can quickly check the distribution of each variable by plotting a histogram of each variable in the training dataset. A minor difficulty in this is that the data has been split into windows of 128 time steps, with a 50% overlap. Therefore, in order to get a fair idea of the data distribution, we must first remove the duplicated observations (the overlap), then remove the windowing of the data.

We can do this using NumPy, first slicing the array and only keeping the second half of each window, then flattening the windows into a long vector for each variable. This is quick and dirty and does mean that we lose the data in the first half of the first window.

```
# remove overlap
cut = int(trainX.shape[1] / 2)
longX = trainX[:, -cut:, :]
# flatten windows
longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
```

Listing 24.13: Example of flattening the window data.

The complete example of loading the data, flattening it, and plotting a histogram for each of the nine variables is listed below.

```
# plot distributions for the har dataset
from numpy import dstack
from pandas import read_csv
from keras.utils import to_categorical
from matplotlib import pyplot

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
```

```
    loaded = dstack(loaded)
    return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  return trainX, trainy, testX, testy

# plot a histogram of each variable in the dataset
def plot_variable_distributions(trainX):
  # remove overlap
  cut = int(trainX.shape[1] / 2)
  longX = trainX[:, -cut:, :]
  # flatten windows
  longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
  pyplot.figure()
  for i in range(longX.shape[1]):
    # create figure
    ax = pyplot.subplot(longX.shape[1], 1, i+1)
    ax.set_xlim(-1, 1)
    # create histogram
    pyplot.hist(longX[:, i], bins=100)
    # simplify axis remove clutter
    pyplot.yticks([])
    pyplot.xticks([-1,0,1])
  pyplot.show()
```

```
# load data
trainX, trainy, testX, testy = load_dataset()
# plot histograms
plot_variable_distributions(trainX)
```

Listing 24.14: Example of plotting histograms of all data for each variable.

Running the example creates a figure with nine histogram plots, one for each variable in the training dataset. The order of the plots matches the order in which the data was loaded, specifically:

1. Total Acceleration x

2. Total Acceleration y

3. Total Acceleration z

4. Body Acceleration x

5. Body Acceleration y

6. Body Acceleration z

7. Body Gyroscope x

8. Body Gyroscope y

9. Body Gyroscope z

We can see that each variable has a Gaussian-like distribution, except perhaps the first variable (Total Acceleration x). The distributions of total acceleration data is flatter than the body data, which is more pointed. We could explore using a power transform on the data to make the distributions more Gaussian, although this is left as an exercise.
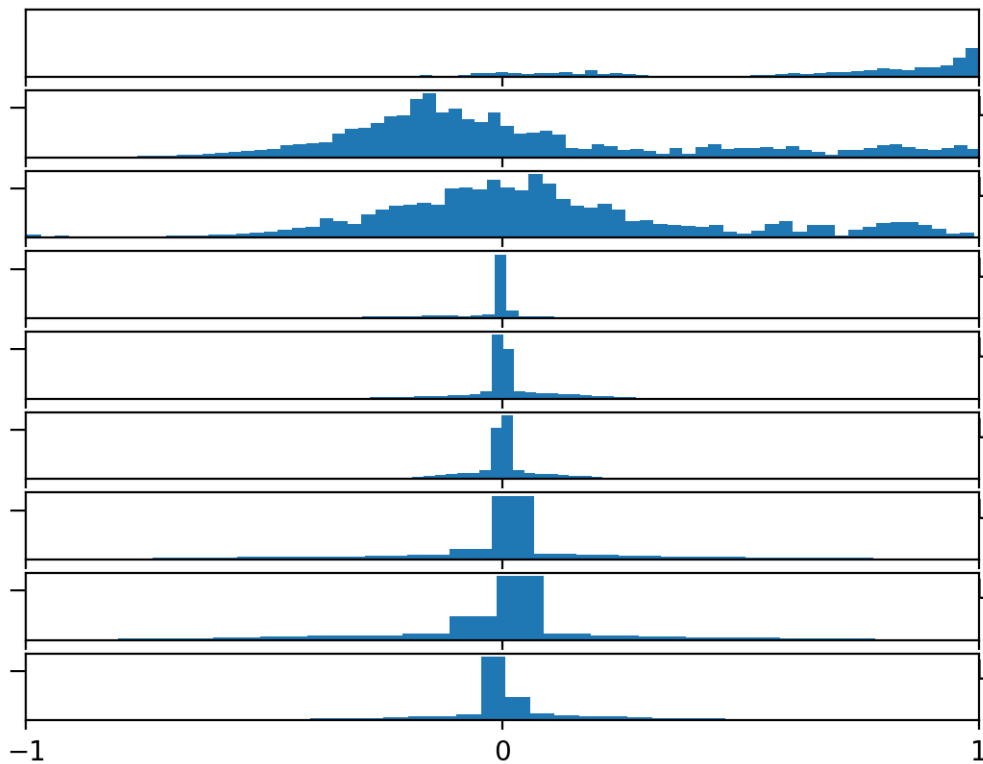
Figure 24.1: Histograms of each variable in the training data set.

The data is sufficiently Gaussian-like to explore whether a standardization transform will help the model extract salient signal from the raw observations. The function below named scale_data() can be used to standardize the data prior to fitting and evaluating the model. The StandardScaler scikit-learn class will be used to perform the transform. It is first fit on the training data (e.g. to find the mean and standard deviation for each variable), then applied to the train and test sets. The standardization is optional, so we can apply the process and compare the results to the same code path without the standardization in a controlled experiment.

```python
# standardize data
def scale_data(trainX, testX, standardize):
  # remove overlap
  cut = int(trainX.shape[1] / 2)
  longX = trainX[:, -cut:, :]
  # flatten windows
  longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
  # flatten train and test
  flatTrainX = trainX.reshape((trainX.shape[0] * trainX.shape[1], trainX.shape[2]))
  flatTestX = testX.reshape((testX.shape[0] * testX.shape[1], testX.shape[2]))
  # standardize
  if standardize:
    s = StandardScaler()
    # fit on training data
    s.fit(longX)
```

```
    # apply to training and test data
    longX = s.transform(longX)
    flatTrainX = s.transform(flatTrainX)
    flatTestX = s.transform(flatTestX)
  # reshape
  flatTrainX = flatTrainX.reshape((trainX.shape))
  flatTestX = flatTestX.reshape((testX.shape))
  return flatTrainX, flatTestX
```

Listing 24.15: Example of a function to standardize the dataset.

We can update the `evaluate_model()` function to take a parameter, then use this parameter to decide whether or not to perform the standardization.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, param):
  verbose, epochs, batch_size = 0, 10, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  # scale data
  trainX, testX = scale_data(trainX, testX, param)
  model = Sequential()
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
  model.add(Dropout(0.5))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy
```

Listing 24.16: Example of a function to evaluate a model with parameterized data standardization.

We can also update the `run_experiment()` to repeat the experiment 10 times for each parameter; in this case, only two parameters will be evaluated `[False, True]` for no standardization and standardization respectively.

```
# run an experiment
def run_experiment(params, repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # test each parameter
  all_scores = list()
  for p in params:
    # repeat experiment
    scores = list()
    for r in range(repeats):
      score = evaluate_model(trainX, trainy, testX, testy, p)
      score = score * 100.0
      print('>p=%d #%d: %.3f' % (p, r+1, score))
      scores.append(score)
```

```
    all_scores.append(scores)
  # summarize results
  summarize_results(all_scores, params)
```

Listing 24.17: Example of a function to run the experiment with parameterization.

This will result in two samples of results that can be compared. We will update the summarize_results() function to summarize the sample of results for each configuration parameter and to create a box plot to compare each sample of results.

```
# summarize scores
def summarize_results(scores, params):
  print(scores, params)
  # summarize mean and standard deviation
  for i in range(len(scores)):
    m, s = mean(scores[i]), std(scores[i])
    print('Param=%d: %.3f%% (+/-%.3f)' % (params[i], m, s))
  # box plot of scores
  pyplot.boxplot(scores, labels=params)
  pyplot.savefig('exp_cnn_standardize.png')
```

Listing 24.18: Example of a function to summarize the results of a parameterized experiment.

These updates will allow us to directly compare the results of a model fit as before and a model fit on the dataset after it has been standardized. It is also a generic change that will allow us to evaluate and compare the results of other sets of parameters in the following sections. The complete code listing is provided below.

```
# cnn model with standardization for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
from sklearn.preprocessing import StandardScaler
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

```python
# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  return trainX, trainy, testX, testy

# standardize data
def scale_data(trainX, testX, standardize):
  # remove overlap
  cut = int(trainX.shape[1] / 2)
  longX = trainX[:, -cut:, :]
  # flatten windows
  longX = longX.reshape((longX.shape[0] * longX.shape[1], longX.shape[2]))
  # flatten train and test
  flatTrainX = trainX.reshape((trainX.shape[0] * trainX.shape[1], trainX.shape[2]))
  flatTestX = testX.reshape((testX.shape[0] * testX.shape[1], testX.shape[2]))
  # standardize
  if standardize:
    s = StandardScaler()
    # fit on training data
    s.fit(longX)
    # apply to training and test data
    longX = s.transform(longX)
    flatTrainX = s.transform(flatTrainX)
    flatTestX = s.transform(flatTestX)
  # reshape
  flatTrainX = flatTrainX.reshape((trainX.shape))
```

```
  flatTestX = flatTestX.reshape((testX.shape))
  return flatTrainX, flatTestX

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, param):
  verbose, epochs, batch_size = 0, 10, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  # scale data
  trainX, testX = scale_data(trainX, testX, param)
  model = Sequential()
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=64, kernel_size=3, activation='relu'))
  model.add(Dropout(0.5))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy

# summarize scores
def summarize_results(scores, params):
  print(scores, params)
  # summarize mean and standard deviation
  for i in range(len(scores)):
    m, s = mean(scores[i]), std(scores[i])
    print('Param=%s: %.3f%% (+/-%.3f)' % (params[i], m, s))
  # boxplot of scores
  pyplot.boxplot(scores, labels=params)
  pyplot.savefig('exp_cnn_standardize.png')

# run an experiment
def run_experiment(params, repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # test each parameter
  all_scores = list()
  for p in params:
    # repeat experiment
    scores = list()
    for r in range(repeats):
      score = evaluate_model(trainX, trainy, testX, testy, p)
      score = score * 100.0
      print('>p=%s #%d: %.3f' % (p, r+1, score))
      scores.append(score)
    all_scores.append(scores)
  # summarize results
  summarize_results(all_scores, params)

# run the experiment
n_params = [False, True]
```

```
run_experiment(n_params)
```

Listing 24.19: Example of evaluating a CNN with and without data standardization.

Running the example may take a while, depending on your hardware. The performance is printed for each evaluated model. At the end of the run, the performance of each of the tested configurations is summarized showing the mean and the standard deviation. We can see that it does look like standardizing the dataset prior to modeling does result in a small lift in performance from about 90.4% accuracy (close to what we saw in the previous section) to about 91.5% accuracy.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
>p=False #1: 91.483
>p=False #2: 91.245
>p=False #3: 90.838
>p=False #4: 89.243
>p=False #5: 90.193
>p=False #6: 90.465
>p=False #7: 90.397
>p=False #8: 90.567
>p=False #9: 88.938
>p=False #10: 91.144
>p=True #1: 92.908
>p=True #2: 90.940
>p=True #3: 92.297
>p=True #4: 91.822
>p=True #5: 92.094
>p=True #6: 91.313
>p=True #7: 91.653
>p=True #8: 89.141
>p=True #9: 91.110
>p=True #10: 91.890

[[91.48286392941975, 91.24533423820834, 90.83814048184594, 89.24329826942655,
    90.19341703427214, 90.46487953851374, 90.39701391245333, 90.56667797760434,
    88.93790295215473, 91.14353579911774], [92.90804207668816, 90.93993892093654,
    92.29725144214456, 91.82219205972176, 92.09365456396336, 91.31319986426874,
    91.65252799457076, 89.14149983033593, 91.10960298608755, 91.89005768578215]] [False,
    True]

Param=False: 90.451% (+/-0.785)
Param=True: 91.517% (+/-0.965)
```

Listing 24.20: Example output from evaluating a CNN with and without data standardization.

A box and whisker plot of the results is also created. This allows the two samples of results to be compared in a nonparametric way, showing the median and the middle 50% of each sample. We can see that the distribution of results with standardization is quite different from the distribution of results without standardization. This is likely a real effect.
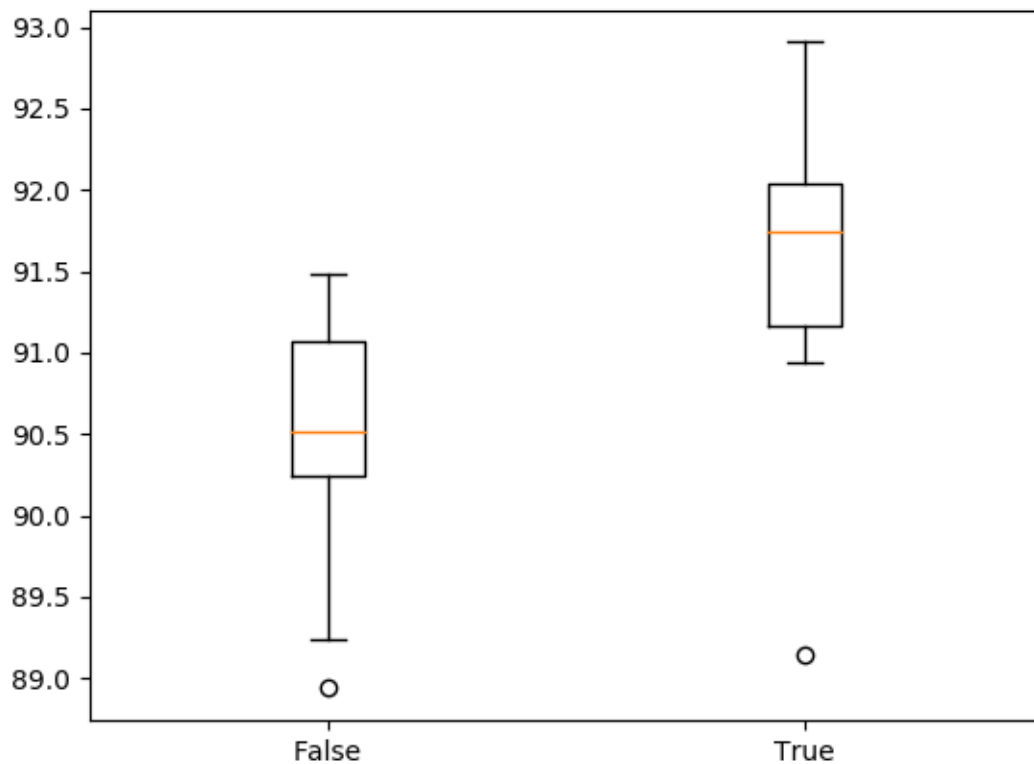
Figure 24.2: Box and whisker plot of 1D CNN with and without standardization.

24.4.2 Number of Filters

Now that we have an experimental framework, we can explore varying other hyperparameters of the model. An important hyperparameter for the CNN is the number of filter maps. We can experiment with a range of different values, from less to many more than the 64 used in the first model that we developed. Specifically, we will try the following numbers of feature maps:

```
# define configuration
n_params = [8, 16, 32, 64, 128, 256]
```

Listing 24.21: Example of a configuration for the number of filter maps.

We can use the same code from the previous section and update the `evaluate_model()` function to use the provided parameter as the number of filters in the `Conv1D` layers. We can also update the `summarize_results()` function to save the box plot as `exp_cnn_filters.png`. The complete code example is listed below.

```
# cnn model with filters for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
```

```python
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
    dataframe = read_csv(filepath, header=None, delim_whitespace=True)
    return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
    loaded = list()
    for name in filenames:
        data = load_file(prefix + name)
        loaded.append(data)
    # stack group so that features are the 3rd dimension
    loaded = dstack(loaded)
    return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
    filepath = prefix + group + '/Inertial Signals/'
    # load all 9 files as a single array
    filenames = list()
    # total acceleration
    filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
        'total_acc_z_'+group+'.txt']
    # body acceleration
    filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
        'body_acc_z_'+group+'.txt']
    # body gyroscope
    filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
        'body_gyro_z_'+group+'.txt']
    # load input data
    X = load_group(filenames, filepath)
    # load class output
    y = load_file(prefix + group + '/y_'+group+'.txt')
    return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
    # load all train
    trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
    # load all test
    testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
    # zero-offset class values
    trainy = trainy - 1
    testy = testy - 1
    # one hot encode y
    trainy = to_categorical(trainy)
    testy = to_categorical(testy)
    return trainX, trainy, testX, testy
```

```python
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, n_filters):
  verbose, epochs, batch_size = 0, 10, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  model = Sequential()
  model.add(Conv1D(filters=n_filters, kernel_size=3, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=n_filters, kernel_size=3, activation='relu'))
  model.add(Dropout(0.5))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy

# summarize scores
def summarize_results(scores, params):
  print(scores, params)
  # summarize mean and standard deviation
  for i in range(len(scores)):
    m, s = mean(scores[i]), std(scores[i])
    print('Param=%d: %.3f%% (+/-%.3f)' % (params[i], m, s))
  # boxplot of scores
  pyplot.boxplot(scores, labels=params)
  pyplot.savefig('exp_cnn_filters.png')

# run an experiment
def run_experiment(params, repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # test each parameter
  all_scores = list()
  for p in params:
    # repeat experiment
    scores = list()
    for r in range(repeats):
      score = evaluate_model(trainX, trainy, testX, testy, p)
      score = score * 100.0
      print('>p=%d #%d: %.3f' % (p, r+1, score))
      scores.append(score)
    all_scores.append(scores)
  # summarize results
  summarize_results(all_scores, params)

# run the experiment
n_params = [8, 16, 32, 64, 128, 256]
run_experiment(n_params)
```

Listing 24.22: Example of evaluating a CNN with different numbers of filter maps.

Running the example repeats the experiment for each of the specified number of filters. At the end of the run, a summary of the results with each number of filters is presented. We can see perhaps a trend of increasing average performance with the increase in the number of filter maps. The variance stays pretty constant, and perhaps 128 feature maps might be a good configuration for the network.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
Param=8: 89.148% (+/-0.790)
Param=16: 90.383% (+/-0.613)
Param=32: 90.356% (+/-1.039)
Param=64: 90.098% (+/-0.615)
Param=128: 91.032% (+/-0.702)
Param=256: 90.706% (+/-0.997)
```

Listing 24.23: Example output from evaluating a CNN with different numbers of filter maps.

A box and whisker plot of the results is also created, allowing the distribution of results with each number of filters to be compared. From the plot, we can see the trend upward in terms of median classification accuracy (orange line on the box) with the increase in the number of feature maps. We do see a dip at 64 feature maps (the default or baseline in our experiments), which is surprising, and perhaps a plateau in accuracy across 32, 128, and 256 filter maps. Perhaps 32 would be a more stable configuration.
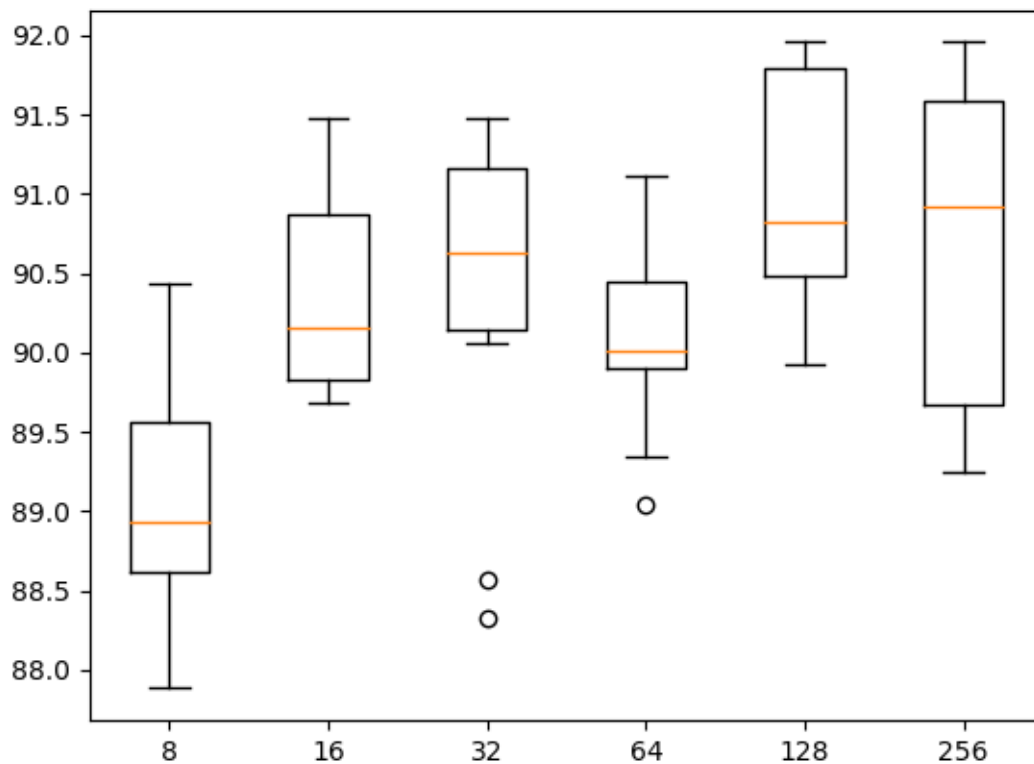
Figure 24.3: Box and whisker plot of 1D CNN with different numbers of filter maps.

24.4.3 Size of Kernel

The size of the kernel is another important hyperparameter of the 1D CNN to tune. The kernel size controls the number of time steps consider in each *read* of the input sequence, that is then projected onto the feature map (via the convolutional process). A large kernel size means a less rigorous reading of the data, but may result in a more generalized snapshot of the input. We can use the same experimental setup and test a suite of different kernel sizes in addition to the default of three time steps. The full list of values is as follows:

```
# define configuration
n_params = [2, 3, 5, 7, 11]
```

Listing 24.24: Example of a configuration for different sized kernels.

The complete code listing is provided below:

```
# cnn model vary kernel size for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from matplotlib import pyplot
from keras.models import Sequential
```

```python
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  return trainX, trainy, testX, testy
```

```python
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy, n_kernel):
  verbose, epochs, batch_size = 0, 15, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  model = Sequential()
  model.add(Conv1D(filters=64, kernel_size=n_kernel, activation='relu',
      input_shape=(n_timesteps,n_features)))
  model.add(Conv1D(filters=64, kernel_size=n_kernel, activation='relu'))
  model.add(Dropout(0.5))
  model.add(MaxPooling1D(pool_size=2))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy

# summarize scores
def summarize_results(scores, params):
  print(scores, params)
  # summarize mean and standard deviation
  for i in range(len(scores)):
    m, s = mean(scores[i]), std(scores[i])
    print('Param=%d: %.3f%% (+/-%.3f)' % (params[i], m, s))
  # boxplot of scores
  pyplot.boxplot(scores, labels=params)
  pyplot.savefig('exp_cnn_kernel.png')

# run an experiment
def run_experiment(params, repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # test each parameter
  all_scores = list()
  for p in params:
    # repeat experiment
    scores = list()
    for r in range(repeats):
      score = evaluate_model(trainX, trainy, testX, testy, p)
      score = score * 100.0
      print('>p=%d #%d: %.3f' % (p, r+1, score))
      scores.append(score)
    all_scores.append(scores)
  # summarize results
  summarize_results(all_scores, params)

# run the experiment
n_params = [2, 3, 5, 7, 11]
run_experiment(n_params)
```

Listing 24.25: Example of evaluating a CNN with different sized kernels.

Running the example tests each kernel size in turn. The results are summarized at the end

of the run. We can see a general increase in model performance with the increase in kernel size. The results suggest a kernel size of 5 might be good with a mean skill of about 91.8%, but perhaps a size of 7 or 11 may also be just as good with a smaller standard deviation.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
...
Param=2: 90.176% (+/-0.724)
Param=3: 90.275% (+/-1.277)
Param=5: 91.853% (+/-1.249)
Param=7: 91.347% (+/-0.852)
Param=11: 91.456% (+/-0.743)
```

Listing 24.26: Example output from evaluating a CNN with different sized kernels.

A box and whisker plot of the results is also created. The results suggest that a larger kernel size does appear to result in better accuracy and that perhaps a kernel size of 7 provides a good balance between good performance and low variance.
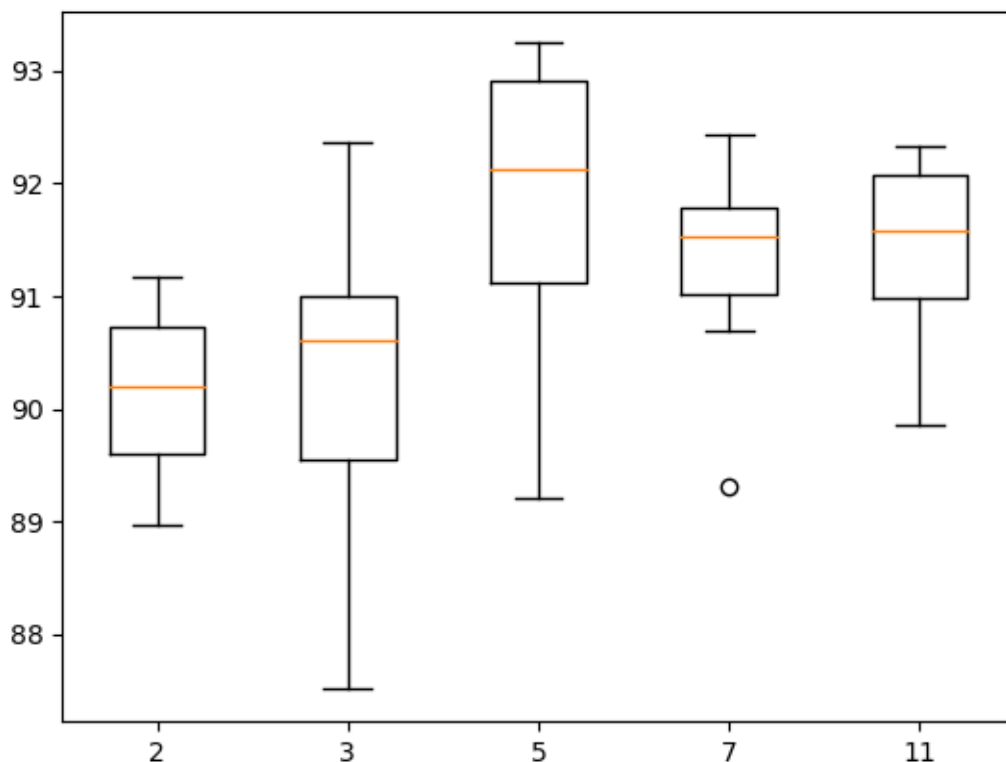

Figure 24.4: Box and whisker plot of 1D CNN with different numbers of kernel sizes.

This is just the beginning of tuning the model, although we have focused on perhaps the more important elements. It might be interesting to explore combinations of some of the above

findings to see if performance can be lifted even further. It may also be interesting to increase the number of repeats from 10 to 30 or more to see if it results in more stable findings.

24.5 Multi-headed CNN Model

Another popular approach with 1D CNNs is to have a multi-headed model, where each head of the model reads the input time steps using a different sized kernel. For example, a three-headed model may have three different kernel sizes of 3, 5, 11, allowing the model to read and interpret the sequence data at three different resolutions. The interpretations from all three heads are then concatenated within the model and interpreted by a fully-connected layer before a prediction is made.

We can implement a multi-headed 1D CNN using the Keras functional API. The updated version of the `evaluate_model()` function is listed below that creates a three-headed 1D CNN model. We can see that each head of the model is the same structure, although the kernel size is varied. The three heads then feed into a single merge layer before being interpreted prior to making a prediction.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  verbose, epochs, batch_size = 0, 10, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  # head 1
  inputs1 = Input(shape=(n_timesteps,n_features))
  conv1 = Conv1D(filters=64, kernel_size=3, activation='relu')(inputs1)
  drop1 = Dropout(0.5)(conv1)
  pool1 = MaxPooling1D(pool_size=2)(drop1)
  flat1 = Flatten()(pool1)
  # head 2
  inputs2 = Input(shape=(n_timesteps,n_features))
  conv2 = Conv1D(filters=64, kernel_size=5, activation='relu')(inputs2)
  drop2 = Dropout(0.5)(conv2)
  pool2 = MaxPooling1D(pool_size=2)(drop2)
  flat2 = Flatten()(pool2)
  # head 3
  inputs3 = Input(shape=(n_timesteps,n_features))
  conv3 = Conv1D(filters=64, kernel_size=11, activation='relu')(inputs3)
  drop3 = Dropout(0.5)(conv3)
  pool3 = MaxPooling1D(pool_size=2)(drop3)
  flat3 = Flatten()(pool3)
  # merge
  merged = concatenate([flat1, flat2, flat3])
  # interpretation
  dense1 = Dense(100, activation='relu')(merged)
  outputs = Dense(n_outputs, activation='softmax')(dense1)
  model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
  # save a plot of the model
  plot_model(model, show_shapes=True, to_file='multiheaded.png')
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit([trainX,trainX,trainX], trainy, epochs=epochs, batch_size=batch_size,
      verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate([testX,testX,testX], testy, batch_size=batch_size, verbose=0)
```

```
    return accuracy
```

Listing 24.27: Example of a function to define a multi-headed CNN model.

When the model is created, a plot of the network architecture is created; provided below, it gives a clear idea of how the constructed model fits together.

Note: the call to `plot_model()` requires that pygraphviz and pydot are installed. If this is a problem, you can comment out this line.
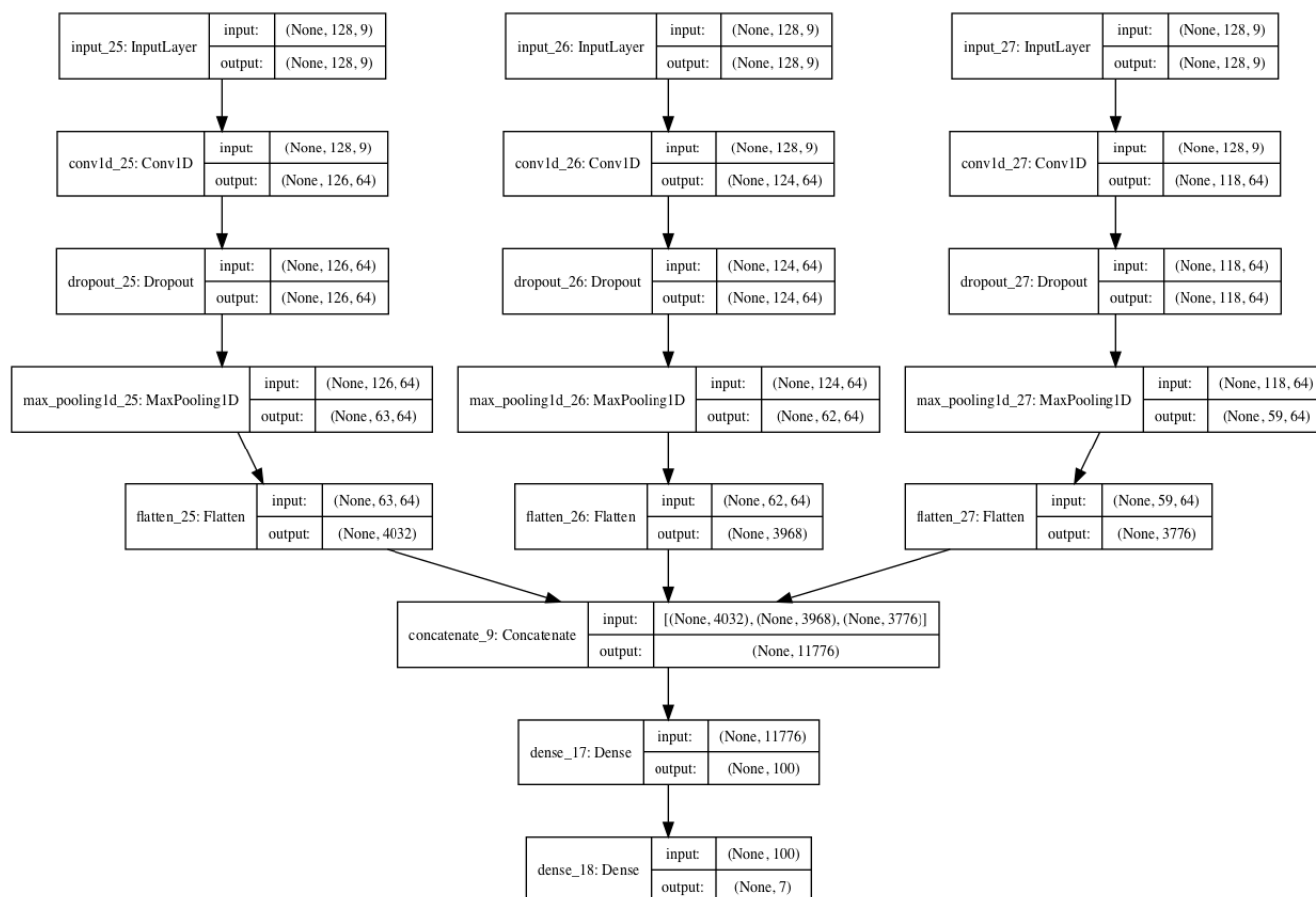

Figure 24.5: Plot of the Multi-headed 1D Convolutional Neural Network.

Other aspects of the model could be varied across the heads, such as the number of filters or even the preparation of the data itself. The complete code example with the multi-headed 1D CNN is listed below.

```
# multi-headed cnn model for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from keras.utils import to_categorical
from keras.utils.vis_utils import plot_model
```

```python
from keras.models import Model
from keras.layers import Input
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.layers.merge import concatenate

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
```

```python
    return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  verbose, epochs, batch_size = 0, 10, 32
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  # head 1
  inputs1 = Input(shape=(n_timesteps,n_features))
  conv1 = Conv1D(filters=64, kernel_size=3, activation='relu')(inputs1)
  drop1 = Dropout(0.5)(conv1)
  pool1 = MaxPooling1D(pool_size=2)(drop1)
  flat1 = Flatten()(pool1)
  # head 2
  inputs2 = Input(shape=(n_timesteps,n_features))
  conv2 = Conv1D(filters=64, kernel_size=5, activation='relu')(inputs2)
  drop2 = Dropout(0.5)(conv2)
  pool2 = MaxPooling1D(pool_size=2)(drop2)
  flat2 = Flatten()(pool2)
  # head 3
  inputs3 = Input(shape=(n_timesteps,n_features))
  conv3 = Conv1D(filters=64, kernel_size=11, activation='relu')(inputs3)
  drop3 = Dropout(0.5)(conv3)
  pool3 = MaxPooling1D(pool_size=2)(drop3)
  flat3 = Flatten()(pool3)
  # merge
  merged = concatenate([flat1, flat2, flat3])
  # interpretation
  dense1 = Dense(100, activation='relu')(merged)
  outputs = Dense(n_outputs, activation='softmax')(dense1)
  model = Model(inputs=[inputs1, inputs2, inputs3], outputs=outputs)
  # save a plot of the model
  plot_model(model, show_shapes=True, to_file='multiheaded.png')
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit([trainX,trainX,trainX], trainy, epochs=epochs, batch_size=batch_size,
      verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate([testX,testX,testX], testy, batch_size=batch_size, verbose=0)
  return accuracy

# summarize scores
def summarize_results(scores):
  print(scores)
  m, s = mean(scores), std(scores)
  print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
def run_experiment(repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
  scores = list()
  for r in range(repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    score = score * 100.0
    print('>#%d: %.3f' % (r+1, score))
```

```
    scores.append(score)
  # summarize results
  summarize_results(scores)

# run the experiment
run_experiment()
```

Listing 24.28: Example of evaluating a multi-headed CNN model.

Running the example prints the performance of the model each repeat of the experiment and then summarizes the estimated score as the mean and standard deviation, as we did in the first case with the simple 1D CNN. We can see that the average performance of the model is about 91.6% classification accuracy with a standard deviation of about 0.8. This example may be used as the basis for exploring a variety of other models that vary different model hyperparameters and even different data preparation schemes across the input heads.

It would not be an apples-to-apples comparison to compare this result with a single-headed CNN given the relative tripling of the resources in this model. Perhaps an apples-to-apples comparison would be a model with the same architecture and the same number of filters across each input head of the model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
>#1: 91.788
>#2: 92.942
>#3: 91.551
>#4: 91.415
>#5: 90.974
>#6: 91.992
>#7: 92.162
>#8: 89.888
>#9: 92.671
>#10: 91.415

[91.78825924669155, 92.94197488971835, 91.55072955548015, 91.41499830335935,
    90.97387173396675, 91.99185612487275, 92.16152019002375, 89.88802171700034,
    92.67051238547675, 91.41499830335935]

Accuracy: 91.680% (+/-0.823)
```

Listing 24.29: Example output from evaluating a multi-headed CNN model.

24.6 Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Number of Variables**. Run experiments to see if all nine variables are required in the CNN or if it can do as well or better with a subset, such as just total acceleration.

- **Date Preparation**. Explore other data preparation schemes such as data normalization and perhaps normalization after standardization.

- **Network Architecture**. Explore other network architectures, such as deeper CNN architectures and deeper fully-connected layers for interpreting the CNN input features.

- **Diagnostics**. Use simple learning curve diagnostics to interpret how the model is learning over the epochs and whether more regularization, different learning rates, or different batch sizes or numbers of epochs may result in a better performing or more stable model.

If you explore any of these extensions, I'd love to know.

24.7 Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/

- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

24.8 Summary

In this tutorial, you discovered how to develop one-dimensional convolutional neural networks for time series classification on the problem of human activity recognition. Specifically, you learned:

- How to load and prepare the data for a standard human activity recognition dataset and develop a single 1D CNN model that achieves excellent performance on the raw data.

- How to further tune the performance of the model, including data transformation, filter maps, and kernel sizes.

- How to develop a sophisticated multi-headed one-dimensional convolutional neural network model that provides an ensemble-like result.

24.8.1 Next

In the next lesson, you will discover how to develop Recurrent Neural Network models for predicting human activities from a stream of smartphone accelerometer data.

Chapter 25

How to Develop LSTMs for Human Activity Recognition

Human activity recognition is the problem of classifying sequences of accelerometer data recorded by specialized harnesses or smartphones into known well-defined movements. Classical approaches to the problem involve hand crafting features from the time series data based on fixed-sized windows and training machine learning models, such as ensembles of decision trees. The difficulty is that this feature engineering requires strong expertise in the field. Recently, deep learning methods such as recurrent neural networks like as LSTMs and variations that make use of one-dimensional convolutional neural networks or CNNs have been shown to provide state-of-the-art results on challenging activity recognition tasks with little or no data feature engineering, instead using feature learning on raw data. In this tutorial, you will discover three recurrent neural network architectures for modeling an activity recognition time series classification problem. After completing this tutorial, you will know:

- How to develop a Long Short-Term Memory Recurrent Neural Network for human activity recognition.

- How to develop a one-dimensional Convolutional Neural Network LSTM, or CNN-LSTM, model.

- How to develop a one-dimensional Convolutional LSTM, or ConvLSTM, model for the same problem.

Let's get started.

25.1 Tutorial Overview

This tutorial is divided into four parts; they are:

1. Activity Recognition Using Smartphones Dataset

2. LSTM Model

3. CNN-LSTM Model

4. ConvLSTM Model

25.2 Activity Recognition Using Smartphones Dataset

Human Activity Recognition, or HAR for short, is the problem of predicting what a person is doing based on a trace of their movement using sensors. A standard human activity recognition dataset is the *Activity Recognition Using Smartphones* made available in 2012. For more information on this dataset, see Chapter 22. The data is provided as a single zip file that is about 58 megabytes in size. A direct for downloading the dataset is provided below:

- HAR_Smartphones.zip [1]

Download the dataset and unzip all files into a new directory in your current working directory named HARDataset.

25.3 LSTM Model

In this section, we will develop a Long Short-Term Memory network model (LSTM) for the human activity recognition dataset. LSTM network models are a type of recurrent neural network that are able to learn and remember over long sequences of input data. They are intended for use with data that is comprised of long sequences of data, up to 200 to 400 time steps. They may be a good fit for this problem. The model can support multiple parallel sequences of input data, such as each axis of the accelerometer and gyroscope data. The model learns to extract features from sequences of observations and how to map the internal features to different activity types.

The benefit of using LSTMs for sequence classification is that they can learn from the raw time series data directly, and in turn do not require domain expertise to manually engineer input features. The model can learn an internal representation of the time series data and ideally achieve comparable performance to models fit on a version of the dataset with engineered features. For more information on LSTMs for time series forecasting, see Chapter 9. This section is divided into four parts; they are:

1. Load Data

2. Fit and Evaluate Model

3. Summarize Results

4. Complete Example

Some of the details on loading and preparing the dataset were covered in Chapters 22 and 23. There is some repetition here in order to customize the data preparation and prediction evaluation for deep learning models.

---

[1] https://raw.githubusercontent.com/jbrownlee/Datasets/master/HAR_Smartphones.zip

25.3.1   Load Data

The first step is to load the raw dataset into memory. There are three main signal types in the raw data: total acceleration, body acceleration, and body gyroscope. Each has three axes of data. This means that there are a total of nine variables for each time step. Further, each series of data has been partitioned into overlapping windows of 2.65 seconds of data, or 128 time steps. These windows of data correspond to the windows of engineered features (rows) in the previous section.

This means that one row of data has (128×9), or 1,152, elements. This is a little less than double the size of the 561 element vectors in the previous section and it is likely that there is some redundant data. The signals are stored in the `/Inertial Signals/` directory under the train and test subdirectories. Each axis of each signal is stored in a separate file, meaning that each of the train and test datasets have nine input files to load and one output file to load. We can batch the loading of these files into groups given the consistent directory structures and file naming conventions. The input data is in CSV format where columns are separated by whitespace. Each of these files can be loaded as a NumPy array. The `load_file()` function below loads a dataset given the file path to the file and returns the loaded data as a NumPy array.

```
# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values
```

Listing 25.1: Example of a function for loading a single file.

We can then load all data for a given group (train or test) into a single three-dimensional NumPy array, where the dimensions of the array are `[samples, timesteps, features]`. To make this clearer, there are 128 time steps and nine features, where the number of samples is the number of rows in any given raw signal data file. The `load_group()` function below implements this behavior. The `dstack()` NumPy function allows us to stack each of the loaded 3D arrays into a single 3D array where the variables are separated on the third dimension (features).

```
# load a list of files into a 3D array of [samples, timesteps, features]
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

Listing 25.2: Example of a function for loading a group of files.

We can use this function to load all input signal data for a given group, such as train or test. The `load_dataset_group()` function below loads all input signal data and the output data for a single group using the consistent naming conventions between the train and test directories.

```
# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
```

```
# total acceleration
filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
    'total_acc_z_'+group+'.txt']
# body acceleration
filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
    'body_acc_z_'+group+'.txt']
# body gyroscope
filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
    'body_gyro_z_'+group+'.txt']
# load input data
X = load_group(filenames, filepath)
# load class output
y = load_file(prefix + group + '/y_'+group+'.txt')
return X, y
```

Listing 25.3: Example of a function for loading a dataset group of files.

Finally, we can load each of the train and test datasets. The output data is defined as an integer for the class number. We must one hot encode these class integers so that the data is suitable for fitting a neural network multiclass classification model. We can do this by calling the to_categorical() Keras function. The load_dataset() function below implements this behavior and returns the train and test X and y elements ready for fitting and evaluating the defined models.

```
# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  print(trainX.shape, trainy.shape)
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  print(testX.shape, testy.shape)
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  print(trainX.shape, trainy.shape, testX.shape, testy.shape)
  return trainX, trainy, testX, testy
```

Listing 25.4: Example of a function for loading the entire dataset.

25.3.2 Fit and Evaluate Model

Now that we have the data loaded into memory ready for modeling, we can define, fit, and evaluate an LSTM model. We can define a function named evaluate_model() that takes the train and test dataset, fits a model on the training dataset, evaluates it on the test dataset, and returns an estimate of the model's performance. First, we must define the LSTM model using the Keras deep learning library. The model requires a three-dimensional input with [samples, timesteps, features].

This is exactly how we have loaded the data, where one sample is one window of the time series data, each window has 128 time steps, and a time step has nine variables or features. The

output for the model will be a six-element vector containing the probability of a given window belonging to each of the six activity types. The input and output dimensions are required when fitting the model, and we can extract them from the provided training dataset.

```
# define data shape
n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
```

Listing 25.5: Example of a model shape based on data shape.

he model is defined as a Sequential Keras model, for simplicity.

We will define the model as having a single LSTM hidden layer. This is followed by a dropout layer intended to reduce overfitting of the model to the training data. Finally, a dense fully connected layer is used to interpret the features extracted by the LSTM hidden layer, before a final output layer is used to make predictions. The efficient Adam version of stochastic gradient descent will be used to optimize the network, and the categorical cross entropy loss function will be used given that we are learning a multiclass classification problem. The definition of the model is listed below.

```
# define the lstm model
model = Sequential()
model.add(LSTM(100, input_shape=(n_timesteps,n_features)))
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
```

Listing 25.6: Example of defining an LSTM model.

The model is fit for a fixed number of epochs, in this case 15, and a batch size of 64 samples will be used, where 64 windows of data will be exposed to the model before the weights of the model are updated. Once the model is fit, it is evaluated on the test dataset and the accuracy of the fit model on the test dataset is returned. Note, it is common to not shuffle sequence data when fitting an LSTM. Here we do shuffle the windows of input data during training (the default). In this problem, we are interested in harnessing the LSTMs ability to learn and extract features across the time steps in a window, not across windows. The complete `evaluate_model()` function is listed below.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  verbose, epochs, batch_size = 0, 15, 64
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  model = Sequential()
  model.add(LSTM(100, input_shape=(n_timesteps,n_features)))
  model.add(Dropout(0.5))
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy
```

Listing 25.7: Example of a function for defining, fitting and evaluating an LSTM model.

There is nothing special about the network structure or chosen hyperparameters, they are just a starting point for this problem.

25.3.3 Summarize Results

We cannot judge the skill of the model from a single evaluation. The reason for this is that neural networks are stochastic, meaning that a different specific model will result when training the same model configuration on the same data. This is a feature of the network in that it gives the model its adaptive ability, but requires a slightly more complicated evaluation of the model. We will repeat the evaluation of the model multiple times, then summarize the performance of the model across each of those runs. For example, we can call `evaluate_model()` a total of 10 times. This will result in a population of model evaluation scores that must be summarized.

```
# repeat experiment
scores = list()
for r in range(repeats):
  score = evaluate_model(trainX, trainy, testX, testy)
  score = score * 100.0
  print('>#%d: %.3f' % (r+1, score))
  scores.append(score)
```

Listing 25.8: Example of repeating a model evaluation experiment.

We can summarize the sample of scores by calculating and reporting the mean and standard deviation of the performance. The mean gives the average accuracy of the model on the dataset, whereas the standard deviation gives the average variance of the accuracy from the mean. The function `summarize_results()` below summarizes the results of a run.

```
# summarize scores
def summarize_results(scores):
  print(scores)
  m, s = mean(scores), std(scores)
  print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))
```

Listing 25.9: Example of a function for summarizing model performance.

We can bundle up the repeated evaluation, gathering of results, and summarization of results into a main function for the experiment, called `run_experiment()`, listed below. By default, the model is evaluated 10 times before the performance of the model is reported.

```
# run an experiment
def run_experiment(repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
  scores = list()
  for r in range(repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    score = score * 100.0
    print('>#%d: %.3f' % (r+1, score))
    scores.append(score)
  # summarize results
  summarize_results(scores)
```

Listing 25.10: Example of a function for driving the experiment.

25.3.4 Complete Example

Now that we have all of the pieces, we can tie them together into a worked example. The complete code listing is provided below.

```python
# lstm model for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from keras.layers import LSTM
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
    dataframe = read_csv(filepath, header=None, delim_whitespace=True)
    return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
    loaded = list()
    for name in filenames:
        data = load_file(prefix + name)
        loaded.append(data)
    # stack group so that features are the 3rd dimension
    loaded = dstack(loaded)
    return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
    filepath = prefix + group + '/Inertial Signals/'
    # load all 9 files as a single array
    filenames = list()
    # total acceleration
    filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
        'total_acc_z_'+group+'.txt']
    # body acceleration
    filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
        'body_acc_z_'+group+'.txt']
    # body gyroscope
    filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
        'body_gyro_z_'+group+'.txt']
    # load input data
    X = load_group(filenames, filepath)
    # load class output
    y = load_file(prefix + group + '/y_'+group+'.txt')
    return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
    # load all train
    trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
    # load all test
```

```
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  verbose, epochs, batch_size = 0, 15, 64
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  model = Sequential()
  model.add(LSTM(100, input_shape=(n_timesteps,n_features)))
  model.add(Dropout(0.5))
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy

# summarize scores
def summarize_results(scores):
  print(scores)
  m, s = mean(scores), std(scores)
  print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
def run_experiment(repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
  scores = list()
  for r in range(repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    score = score * 100.0
    print('>#%d: %.3f' % (r+1, score))
    scores.append(score)
  # summarize results
  summarize_results(scores)

# run the experiment
run_experiment()
```

Listing 25.11: Example of evaluating an LSTM model for activity recognition.

Running the example first loads the dataset. The models are created and evaluated and a debug message is printed for each. Finally, the sample of scores is printed, followed by the mean and standard deviation. We can see that the model performed well, achieving a classification accuracy of about 89.7% trained on the raw dataset, with a standard deviation of about 1.3. This is a good result, considering that the original paper published a result of 89%, trained on

the dataset with heavy domain-specific feature engineering, not the raw dataset.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
>#1: 90.058
>#2: 85.918
>#3: 90.974
>#4: 89.515
>#5: 90.159
>#6: 91.110
>#7: 89.718
>#8: 90.295
>#9: 89.447
>#10: 90.024

[90.05768578215134, 85.91788259246692, 90.97387173396675, 89.51476077366813,
    90.15948422124194, 91.10960298608755, 89.71835765184933, 90.29521547336275,
    89.44689514760775, 90.02375296912113]

Accuracy: 89.722% (+/-1.371)
```

Listing 25.12: Example output from evaluating an LSTM model for activity recognition.

Now that we have seen how to develop an LSTM model for time series classification, let's look at how we can develop a more sophisticated CNN-LSTM model.

25.4 CNN-LSTM Model

The CNN-LSTM architecture involves using Convolutional Neural Network (CNN) layers for feature extraction on input data combined with LSTMs to support sequence prediction. For more information on the use of CNN-LSTM models for time series forecasting, see Chapter 9. The CNN-LSTM model will read subsequences of the main sequence in as blocks, extract features from each block, then allow the LSTM to interpret the features extracted from each block. One approach to implementing this model is to split each window of 128 time steps into subsequences for the CNN model to process. For example, the 128 time steps in each window can be split into four subsequences of 32 time steps.

```
# reshape data into time steps of sub-sequences
n_steps, n_length = 4, 32
trainX = trainX.reshape((trainX.shape[0], n_steps, n_length, n_features))
testX = testX.reshape((testX.shape[0], n_steps, n_length, n_features))
```

Listing 25.13: Example of reshaping the window data into subsequences.

We can then define a CNN model that expects to read in sequences with a length of 32 time steps and nine features. The entire CNN model can be wrapped in a `TimeDistributed` layer to allow the same CNN model to read in each of the four subsequences in the window. The extracted features are then flattened and provided to the LSTM model to read, extracting its own features before a final mapping to an activity is made.

```
# define cnn-lstm model
model = Sequential()
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu'),
    input_shape=(None,n_length,n_features)))
model.add(TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu')))
model.add(TimeDistributed(Dropout(0.5)))
model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
model.add(TimeDistributed(Flatten()))
model.add(LSTM(100))
model.add(Dropout(0.5))
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
```

Listing 25.14: Example of defining a CNN-LSTM model.

It is common to use two consecutive CNN layers followed by dropout and a max pooling layer, and that is the simple structure used in the this CNN-LSTM model. The updated `evaluate_model()` is listed below.

```
# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  # define model
  verbose, epochs, batch_size = 0, 25, 64
  n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
  # reshape data into time steps of sub-sequences
  n_steps, n_length = 4, 32
  trainX = trainX.reshape((trainX.shape[0], n_steps, n_length, n_features))
  testX = testX.reshape((testX.shape[0], n_steps, n_length, n_features))
  # define model
  model = Sequential()
  model.add(TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu'),
      input_shape=(None,n_length,n_features)))
  model.add(TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu')))
  model.add(TimeDistributed(Dropout(0.5)))
  model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(100))
  model.add(Dropout(0.5))
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy
```

Listing 25.15: Example of a function for defining, fitting and evaluating a CNN-LSTM model.

We can evaluate this model as we did the straight LSTM model in the previous section. The complete code listing is provided below.

```
# cnn lstm model for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
```

```python
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import LSTM
from keras.layers import TimeDistributed
from keras.layers.convolutional import Conv1D
from keras.layers.convolutional import MaxPooling1D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded

# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
```

```python
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  # define model
  verbose, epochs, batch_size = 0, 25, 64
  n_features, n_outputs = trainX.shape[2], trainy.shape[1]
  # reshape data into time steps of sub-sequences
  n_steps, n_length = 4, 32
  trainX = trainX.reshape((trainX.shape[0], n_steps, n_length, n_features))
  testX = testX.reshape((testX.shape[0], n_steps, n_length, n_features))
  # define model
  model = Sequential()
  model.add(TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu'),
      input_shape=(None,n_length,n_features)))
  model.add(TimeDistributed(Conv1D(filters=64, kernel_size=3, activation='relu')))
  model.add(TimeDistributed(Dropout(0.5)))
  model.add(TimeDistributed(MaxPooling1D(pool_size=2)))
  model.add(TimeDistributed(Flatten()))
  model.add(LSTM(100))
  model.add(Dropout(0.5))
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
  return accuracy

# summarize scores
def summarize_results(scores):
  print(scores)
  m, s = mean(scores), std(scores)
  print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
def run_experiment(repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
  scores = list()
  for r in range(repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    score = score * 100.0
    print('>#%d: %.3f' % (r+1, score))
    scores.append(score)
  # summarize results
  summarize_results(scores)

# run the experiment
run_experiment()
```

Listing 25.16: Example of evaluating a CNN-LSTM model for activity recognition.

Running the example summarizes the model performance for each of the 10 runs before a final summary of the model's performance on the test set is reported. We can see that the model achieved a performance of about 90.6% with a standard deviation of about 1%.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
>#1: 91.517
>#2: 91.042
>#3: 90.804
>#4: 92.263
>#5: 89.684
>#6: 88.666
>#7: 91.381
>#8: 90.804
>#9: 89.379
>#10: 91.347

[91.51679674244994, 91.04173736002714, 90.80420766881574, 92.26331862911435,
    89.68442483881914, 88.66644044791313, 91.38106549032915, 90.80420766881574,
    89.37902952154734, 91.34713267729894]

Accuracy: 90.689% (+/-1.051)
```

Listing 25.17: Example output from evaluating a CNN-LSTM model for activity recognition.

25.5 ConvLSTM Model

A further extension of the CNN-LSTM idea is to perform the convolutions of the CNN (e.g. how the CNN reads the input sequence data) as part of the LSTM. This combination is called a Convolutional LSTM, or ConvLSTM for short, and like the CNN-LSTM is also used for spatiotemporal data. The `ConvLSTM2D` class, by default, expects input data to have the shape: `[samples, time, rows, cols, channels]`. Where each time step of data is defined as an image of (rows × columns) data points.

In the previous section, we divided a given window of data (128 time steps) into four subsequences of 32 time steps. We can use this same subsequence approach in defining the `ConvLSTM2D` input where the number of time steps is the number of subsequences in the window, the number of rows is 1 as we are working with one-dimensional data, and the number of columns represents the number of time steps in the subsequence, in this case 32. For this chosen framing of the problem, the input for the `ConvLSTM2D` would therefore be:

- **Samples**: n, for the number of windows in the dataset.

- **Time**: 4, for the four subsequences that we split a window of 128 time steps into.

- **Rows**: 1, for the one-dimensional shape of each subsequence.

- **Columns**: 32, for the 32 time steps in an input subsequence.

- **Channels**: 9, for the nine input variables.

We can now prepare the data for the `ConvLSTM2D` model.

```
n_timesteps, n_features, n_outputs = trainX.shape[1], trainX.shape[2], trainy.shape[1]
# reshape into subsequences (samples, timesteps, rows, cols, channels)
n_steps, n_length = 4, 32
trainX = trainX.reshape((trainX.shape[0], n_steps, 1, n_length, n_features))
testX = testX.reshape((testX.shape[0], n_steps, 1, n_length, n_features))
```

Listing 25.18: Example of preparing input data for the ConvLSTM model.

The `ConvLSTM2D` class requires configuration both in terms of the CNN and the LSTM. This includes specifying the number of filters (e.g. 64), the two-dimensional kernel size, in this case (1 row and 3 columns of the subsequence time steps), and the activation function, in this case rectified linear. As with a CNN or LSTM model, the output must be flattened into one long vector before it can be interpreted by a dense layer.

```
# define convlstm model
model = Sequential()
model.add(ConvLSTM2D(filters=64, kernel_size=(1,3), activation='relu',
    input_shape=(n_steps, 1, n_length, n_features)))
model.add(Dropout(0.5))
model.add(Flatten())
model.add(Dense(100, activation='relu'))
model.add(Dense(n_outputs, activation='softmax'))
```

Listing 25.19: Example of defining the ConvLSTM model.

We can then evaluate the model as we did the LSTM and CNN-LSTM models before it. The complete example is listed below.

```
# convlstm model for the har dataset
from numpy import mean
from numpy import std
from numpy import dstack
from pandas import read_csv
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import ConvLSTM2D
from keras.utils import to_categorical

# load a single file as a numpy array
def load_file(filepath):
  dataframe = read_csv(filepath, header=None, delim_whitespace=True)
  return dataframe.values

# load a list of files and return as a 3d numpy array
def load_group(filenames, prefix=''):
  loaded = list()
  for name in filenames:
    data = load_file(prefix + name)
    loaded.append(data)
  # stack group so that features are the 3rd dimension
  loaded = dstack(loaded)
  return loaded
```

```python
# load a dataset group, such as train or test
def load_dataset_group(group, prefix=''):
  filepath = prefix + group + '/Inertial Signals/'
  # load all 9 files as a single array
  filenames = list()
  # total acceleration
  filenames += ['total_acc_x_'+group+'.txt', 'total_acc_y_'+group+'.txt',
      'total_acc_z_'+group+'.txt']
  # body acceleration
  filenames += ['body_acc_x_'+group+'.txt', 'body_acc_y_'+group+'.txt',
      'body_acc_z_'+group+'.txt']
  # body gyroscope
  filenames += ['body_gyro_x_'+group+'.txt', 'body_gyro_y_'+group+'.txt',
      'body_gyro_z_'+group+'.txt']
  # load input data
  X = load_group(filenames, filepath)
  # load class output
  y = load_file(prefix + group + '/y_'+group+'.txt')
  return X, y

# load the dataset, returns train and test X and y elements
def load_dataset(prefix=''):
  # load all train
  trainX, trainy = load_dataset_group('train', prefix + 'HARDataset/')
  # load all test
  testX, testy = load_dataset_group('test', prefix + 'HARDataset/')
  # zero-offset class values
  trainy = trainy - 1
  testy = testy - 1
  # one hot encode y
  trainy = to_categorical(trainy)
  testy = to_categorical(testy)
  return trainX, trainy, testX, testy

# fit and evaluate a model
def evaluate_model(trainX, trainy, testX, testy):
  # define model
  verbose, epochs, batch_size = 0, 25, 64
  n_features, n_outputs = trainX.shape[2], trainy.shape[1]
  # reshape into subsequences (samples, time steps, rows, cols, channels)
  n_steps, n_length = 4, 32
  trainX = trainX.reshape((trainX.shape[0], n_steps, 1, n_length, n_features))
  testX = testX.reshape((testX.shape[0], n_steps, 1, n_length, n_features))
  # define model
  model = Sequential()
  model.add(ConvLSTM2D(filters=64, kernel_size=(1,3), activation='relu',
      input_shape=(n_steps, 1, n_length, n_features)))
  model.add(Dropout(0.5))
  model.add(Flatten())
  model.add(Dense(100, activation='relu'))
  model.add(Dense(n_outputs, activation='softmax'))
  model.compile(loss='categorical_crossentropy', optimizer='adam', metrics=['accuracy'])
  # fit network
  model.fit(trainX, trainy, epochs=epochs, batch_size=batch_size, verbose=verbose)
  # evaluate model
  _, accuracy = model.evaluate(testX, testy, batch_size=batch_size, verbose=0)
```

```
  return accuracy

# summarize scores
def summarize_results(scores):
  print(scores)
  m, s = mean(scores), std(scores)
  print('Accuracy: %.3f%% (+/-%.3f)' % (m, s))

# run an experiment
def run_experiment(repeats=10):
  # load data
  trainX, trainy, testX, testy = load_dataset()
  # repeat experiment
  scores = list()
  for r in range(repeats):
    score = evaluate_model(trainX, trainy, testX, testy)
    score = score * 100.0
    print('>#%d: %.3f' % (r+1, score))
    scores.append(score)
  # summarize results
  summarize_results(scores)

# run the experiment
run_experiment()
```

Listing 25.20: Example of evaluating a ConvLSTM model for activity recognition.

As with the prior experiments, running the model prints the performance of the model each time it is fit and evaluated. A summary of the final model performance is presented at the end of the run. We can see that the model does consistently perform well on the problem achieving an accuracy of about 90%, perhaps with fewer resources than the larger CNN-LSTM model.

Note: Given the stochastic nature of the algorithm, your specific results may vary. Consider running the example a few times.

```
>#1: 90.092
>#2: 91.619
>#3: 92.128
>#4: 90.533
>#5: 89.243
>#6: 90.940
>#7: 92.026
>#8: 91.008
>#9: 90.499
>#10: 89.922

[90.09161859518154, 91.61859518154056, 92.12758737699356, 90.53274516457415,
    89.24329826942655, 90.93993892093654, 92.02578893790296, 91.00780454699695,
    90.49881235154395, 89.92195453003053]

Accuracy: 90.801% (+/-0.886)
```

Listing 25.21: Example output from evaluating a ConvLSTM model for activity recognition.

25.6   Extensions

This section lists some ideas for extending the tutorial that you may wish to explore.

- **Data Preparation**. Consider exploring whether simple data scaling schemes can further lift model performance, such as normalization, standardization, and power transforms.

- **LSTM Variations**. There are variations of the LSTM architecture that may achieve better performance on this problem, such as stacked LSTMs and Bidirectional LSTMs.

- **Hyperparameter Tuning**. Consider exploring tuning of model hyperparameters such as the number of units, training epochs, batch size, and more.

If you explore any of these extensions, I'd love to know.

25.7   Further Reading

This section provides more resources on the topic if you are looking to go deeper.

- Getting started with the Keras Sequential model.
  https://keras.io/getting-started/sequential-model-guide/

- Getting started with the Keras functional API.
  https://keras.io/getting-started/functional-api-guide/

- Keras Sequential Model API.
  https://keras.io/models/sequential/

- Keras Core Layers API.
  https://keras.io/layers/core/

- Keras Convolutional Layers API.
  https://keras.io/layers/convolutional/

- Keras Pooling Layers API.
  https://keras.io/layers/pooling/

- Keras Recurrent Layers API.
  https://keras.io/layers/recurrent/

25.8   Summary

In this tutorial, you discovered three recurrent neural network architectures for modeling an activity recognition time series classification problem. Specifically, you learned:

- How to develop a Long Short-Term Memory Recurrent Neural Network for human activity recognition.

- How to develop a one-dimensional Convolutional Neural Network LSTM, or CNN-LSTM, model.

- How to develop a one-dimensional Convolutional LSTM, or ConvLSTM, model for the same problem.

25.8.1 Next

This is the final lesson of this part, the next part will provide resources that you can use to learn more about deep learning methods and time series forecasting.

Part VII

Appendix

Appendix A

Getting Help

This is just the beginning of your journey with deep learning for time series forecasting with Python. As you start to work on methods or expand your existing knowledge of algorithms you may need help. This chapter points out some of the best sources of help.

A.1   Applied Time Series

There are a wealth of good books on applied time series suitable for the machine learning perspective. Most are focused on time series using the R platform. Nevertheless, even though the code is not directly usable, the concepts, methods and descriptions are still valuable. This section lists some useful books on applied time series that you may consider.

- Rob J. Hyndman and George Athanasopoulos, *Forecasting: principles and practice*, 2013.
  http://amzn.to/2lln93c

- Galit Shmueli and Kenneth Lichtendahl, *Practical Time Series Forecasting with R: A Hands-On Guide*, 2016.
  http://amzn.to/2k3QpuV

- Paul S. P. Cowpertwait and Andrew V. Metcalfem *Introductory Time Series with R*, 2009.
  http://amzn.to/2kIQscS

A.2   Official Keras Destinations

This section lists the official Keras sites that you may find helpful.

- Keras Official Blog.
  https://blog.keras.io/

- Keras API Documentation.
  https://keras.io/

- Keras Source Code Project.
  https://github.com/keras-team/keras

A.3 Where to Get Help with Keras

This section lists the 9 best places I know where you can get help with Keras.

- Keras Users Google Group.
  https://groups.google.com/forum/#!forum/keras-users

- Keras Slack Channel (you must request to join).
  https://keras-slack-autojoin.herokuapp.com/

- Keras on Gitter.
  https://gitter.im/Keras-io/Lobby#

- Keras tag on StackOverflow.
  https://stackoverflow.com/questions/tagged/keras

- Keras tag on CrossValidated.
  https://stats.stackexchange.com/questions/tagged/keras

- Keras tag on DataScience.
  https://datascience.stackexchange.com/questions/tagged/keras

- Keras Topic on Quora.
  https://www.quora.com/topic/Keras

- Keras Github Issues.
  https://github.com/keras-team/keras/issues

- Keras on Twitter.
  https://twitter.com/hashtag/keras

A.4 Time Series Datasets

This section lists some websites where you can download free time series datasets on which to practice.

A.4.1 DataMarket

The DataMarket website hosts many univariate datasets that can be downloaded for free (e.g. Click the *Export* tab on any dataset and choose the data format). All of the univariate datasets demonstrated in this book can also be downloaded from DataMarket. A good place to start is the *Time Series Data Library* created by Rob Hyndman, Professor of Statistics at Monash University, Australia.

- Time Series Data Library, DataMarket.
  https://datamarket.com/data/list/?q=provider:tsdl

A.4.2 UCI Machine Learning Repository

The University of California at Irvine hosts a large number of free machine learning datasets, including many for time series forecasting and time series classification. Some examples of time series datasets on the UCI repository include:

- EEG Eye State Dataset.
  http://archive.ics.uci.edu/ml/datasets/EEG+Eye+State

- Room Occupancy Detection Dataset.
  http://archive.ics.uci.edu/ml/datasets/Occupancy+Detection+

- Beijing PM2.5 Air Pollution Dataset.
  https://archive.ics.uci.edu/ml/datasets/Beijing+PM2.5+Data

A.4.3 Past Kaggle Competitions

Kaggle hosts prediction competitions and the datasets used in old competitions are made freely available. You must create an account with Kaggle in order to access the download the datasets. Some examples of time series datasets on Kaggle include:

- Web Traffic Time Series Forecasting Dataset.
  https://www.kaggle.com/c/web-traffic-time-series-forecasting

- EMC Data Science Global Hackathon (Air Quality Prediction) Dataset.
  https://www.kaggle.com/c/dsg-hackathon

- Rossmann Store Sales Dataset.
  https://www.kaggle.com/c/rossmann-store-sales

If you explore any of these datasets, I'd love to hear about it.

A.5 How to Ask Questions

Knowing where to get help is the first step, but you need to know how to get the most out of these resources. Below are some tips that you can use:

- Boil your question down to the simplest form. E.g. not something broad like *my model does not work* or *how does x work*.

- Search for answers before asking questions.

- Provide complete code and error messages.

- Boil your code down to the smallest possible working example that demonstrates the issue.

These are excellent resources both for posting unique questions, but also for searching through the answers to questions on related topics.

A.6 Contact the Author

You are not alone. If you ever have any questions about time series forecasting or this book, please contact me directly. I will do my best to help.

Jason Brownlee
Jason@MachineLearningMastery.com

Appendix B

How to Setup a Workstation for Python

It can be difficult to install a Python machine learning environment on some platforms. Python itself must be installed first and then there are many packages to install, and it can be confusing for beginners. In this tutorial, you will discover how to setup a Python machine learning development environment using Anaconda.

After completing this tutorial, you will have a working Python environment to begin learning, practicing, and developing machine learning and deep learning software. These instructions are suitable for Windows, macOS, and Linux platforms. I will demonstrate them on macOS, so you may see some mac dialogs and file extensions.

B.1   Overview

In this tutorial, we will cover the following steps:

1. Download Anaconda

2. Install Anaconda

3. Start and Update Anaconda

4. Install Deep Learning Libraries

Note: The specific versions may differ as the software and libraries are updated frequently.

B.2   Download Anaconda

In this step, we will download the Anaconda Python package for your platform. Anaconda is a free and easy-to-use environment for scientific Python.

- 1. Visit the Anaconda homepage.
  https://www.continuum.io/

- 2. Click `Anaconda` from the menu and click `Download` to go to the download page.
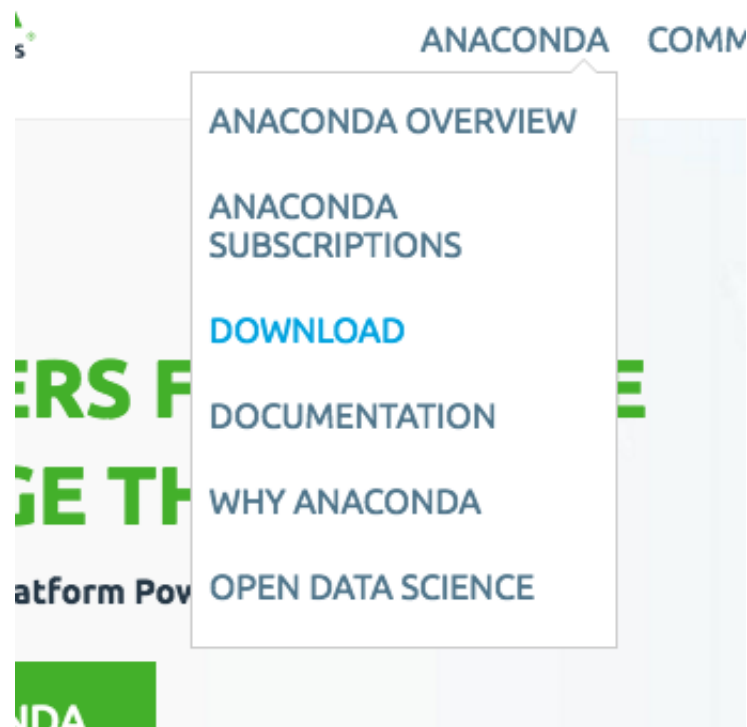 https://www.continuum.io/downloads

Figure B.1: Click Anaconda and Download.

- 3. Choose the download suitable for your platform (Windows, OSX, or Linux):

    - Choose Python 3.6
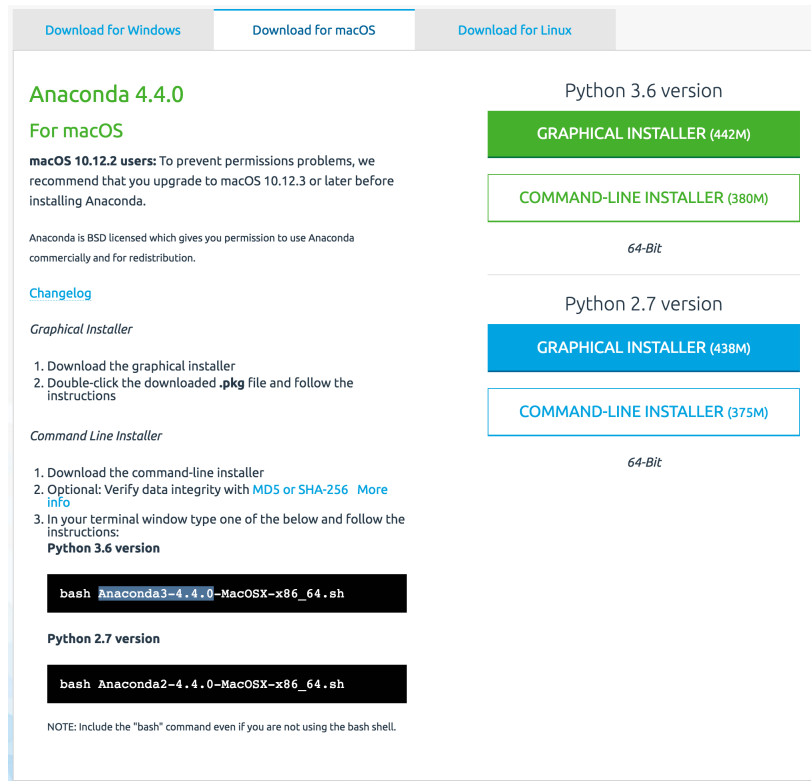   - Choose the Graphical Installer

Figure B.2: Choose Anaconda Download for Your Platform.

This will download the Anaconda Python package to your workstation. I'm on macOS, so I chose the macOS version. The file is about 426 MB. You should have a file with a name like:

```
Anaconda3-4.4.0-MacOSX-x86_64.pkg
```

Listing B.1: Example filename on macOS.

B.3 Install Anaconda

In this step, we will install the Anaconda Python software on your system. This step assumes you have sufficient administrative privileges to install software on your system.

- 1. Double click the downloaded file.
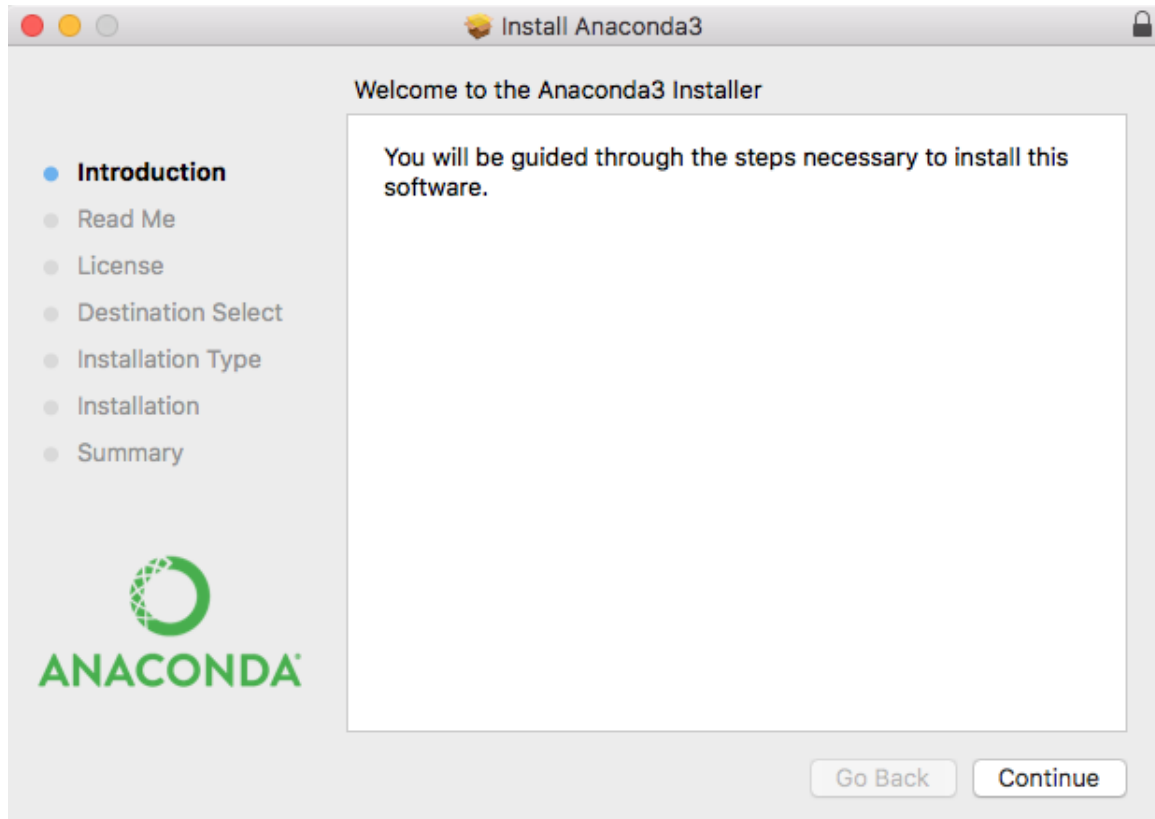
- 2. Follow the installation wizard.

Figure B.3: Anaconda Python Installation Wizard.

Installation is quick and painless. There should be no tricky questions or sticking points.
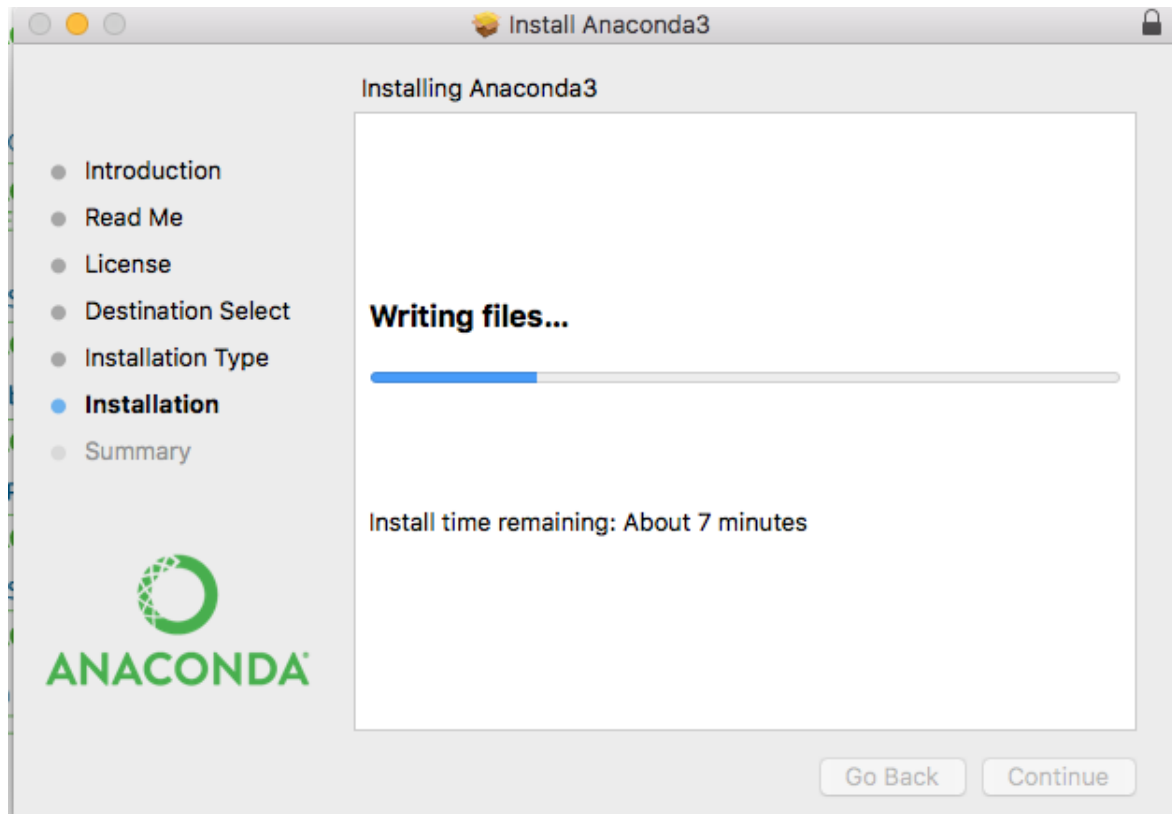
Figure B.4: Anaconda Python Installation Wizard Writing Files.

The installation should take less than 10 minutes and take up a little more than 1 GB of space on your hard drive.

B.4   Start and Update Anaconda

In this step, we will confirm that your Anaconda Python environment is up to date. Anaconda comes with a suite of graphical tools called Anaconda Navigator. You can start Anaconda Navigator by opening it from your application launcher.
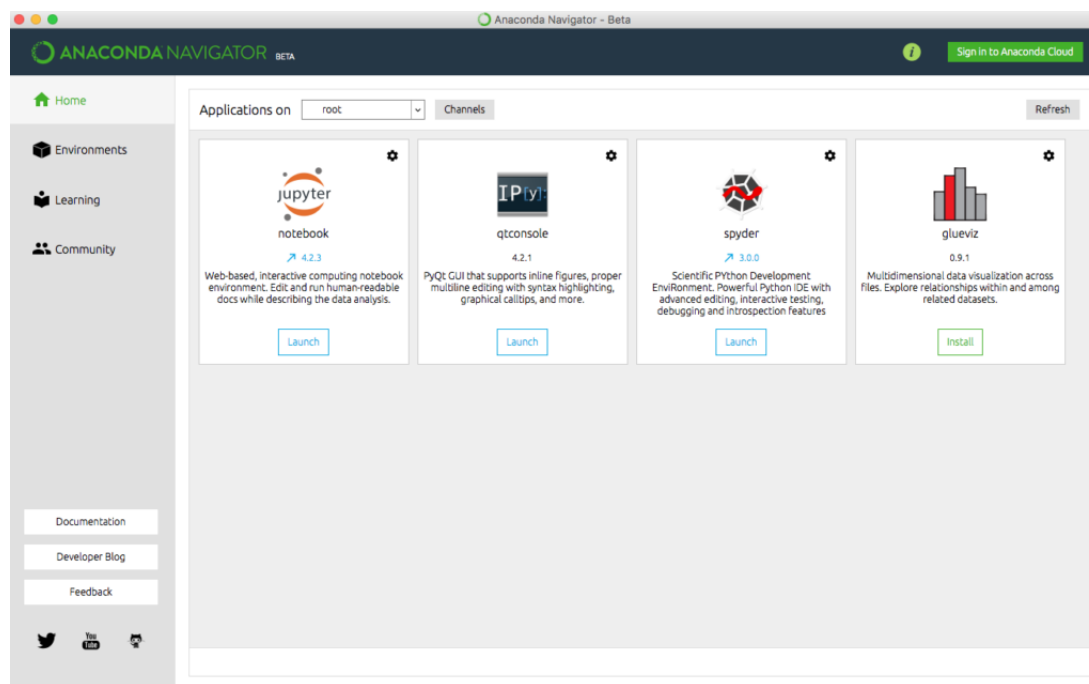
Figure B.5: Anaconda Navigator GUI.

You can use the Anaconda Navigator and graphical development environments later; for now, I recommend starting with the Anaconda command line environment called conda. Conda is fast, simple, it's hard for error messages to hide, and you can quickly confirm your environment is installed and working correctly.

- 1. Open a terminal (command line window).

- 2. Confirm conda is installed correctly, by typing:

```
conda -V
```

Listing B.2: Check the conda version.

You should see the following (or something similar):

```
conda 4.3.21
```

Listing B.3: Example conda version.

- 3. Confirm Python is installed correctly by typing:

```
python -V
```

Listing B.4: Check the Python version.

You should see the following (or something similar):

```
Python 3.6.1 :: Anaconda 4.4.0 (x86_64)
```

Listing B.5: Example Python version.

If the commands do not work or have an error, please check the documentation for help for your platform. See some of the resources in the *Further Reading* section.

- 4. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

Listing B.6: Update conda and anaconda.

You may need to install some packages and confirm the updates.

- 5. Confirm your SciPy environment.

The script below will print the version number of the key SciPy libraries you require for machine learning development, specifically: SciPy, NumPy, Matplotlib, Pandas, Statsmodels, and Scikit-learn. You can type `python` and type the commands in directly. Alternatively, I recommend opening a text editor and copy-pasting the script into your editor.

```python
# check library version numbers
# scipy
import scipy
print('scipy: %s' % scipy.__version__)
# numpy
import numpy
print('numpy: %s' % numpy.__version__)
# matplotlib
import matplotlib
print('matplotlib: %s' % matplotlib.__version__)
# pandas
import pandas
print('pandas: %s' % pandas.__version__)
# statsmodels
import statsmodels
print('statsmodels: %s' % statsmodels.__version__)
# scikit-learn
import sklearn
print('sklearn: %s' % sklearn.__version__)
```

Listing B.7: Code to check that key Python libraries are installed.

Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

Listing B.8: Run the script from the command line.

You should see output like the following:

```
scipy: 1.1.0
numpy: 1.15.0
matplotlib: 2.2.2
pandas: 0.23.1
statsmodels: 0.9.0
sklearn: 0.19.1
```

Listing B.9: Sample output of versions script.

B.5   Install Deep Learning Libraries

In this step, we will install Python libraries used for deep learning, specifically: Theano, TensorFlow, and Keras. Note: I recommend using Keras for deep learning and Keras only requires one of Theano or TensorFlow to be installed. You do not need both. There may be problems installing TensorFlow on some Windows machines.

- 1. Install the Theano deep learning library by typing:

```
conda install theano
```

Listing B.10: Install Theano with conda.

- 2. Install the TensorFlow deep learning library by typing:

```
conda install -c conda-forge tensorflow
```

Listing B.11: Install TensorFlow with conda.

Alternatively, you may choose to install using `pip` and a specific version of TensorFlow for your platform.

- 3. Install Keras by typing:

```
pip install keras
```

Listing B.12: Install Keras with `pip`.

- 4. Confirm your deep learning environment is installed and working correctly.

Create a script that prints the version numbers of each library, as we did before for the SciPy environment.

```
# theano
import theano
print('theano: %s' % theano.__version__)
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__)
# keras
import keras
print('keras: %s' % keras.__version__)
```

Listing B.13: Code to check that key deep learning libraries are installed.

Save the script to a file `deep_versions.py`. Run the script by typing:

```
python deep_versions.py
```

Listing B.14: Run script from the command line.

You should see output like:

```
theano: 1.0.2
tensorflow: 1.9.0
keras: 2.2.0
```

Listing B.15: Sample output of the deep learning versions script.

B.6   Further Reading

This section provides resources if you want to know more about Anaconda.

- Anaconda homepage.
  https://www.continuum.io/

- Anaconda Navigator.
  https://docs.continuum.io/anaconda/navigator.html

- The conda command line tool.
  http://conda.pydata.org/docs/index.html

- Instructions for installing TensorFlow in Anaconda.
  https://www.tensorflow.org/get_started/os_setup#anaconda_installation

B.7   Summary

Congratulations, you now have a working Python development environment for machine learning and deep learning. You can now learn and practice machine learning and deep learning on your workstation.

Part VIII

Conclusions

How Far You Have Come

You made it. Well done. Take a moment and look back at how far you have come. You now know:

- About the promise of neural networks and deep learning methods in general for time series forecasting.

- How to transform time series data in order to train a supervised learning algorithm, such as deep learning methods.

- How to develop baseline forecasts using naive and classical methods by which to determine whether forecasts from deep learning models have skill or not.

- How to develop Multilayer Perceptron, Convolutional Neural Network, Long Short-Term Memory Networks, and hybrid neural network models for time series forecasting.

- How to forecast univariate, multivariate, multi-step, and multivariate multi-step time series forecasting problems in general.

- How to transform sequence data into a three-dimensional structure in order to train convolutional and LSTM neural network models.

- How to grid search deep learning model hyperparameters to ensure that you are getting good performance from a given model.

- How to prepare data and develop deep learning models for forecasting a range of univariate time series problems with different temporal structures.

- How to prepare data and develop deep learning models for multi-step forecasting a real-world household electricity consumption dataset.

- How to prepare data and develop deep learning models for a real-world human activity recognition project.

Don't make light of this. You have come a long way in a short amount of time. You have developed the important and valuable set of skills in developing deep learning models for time series forecasting. You can now confidently:

- Use naive and classical methods like SARIMA and ETS to quickly develop robust baseline models for a range of different time series forecasting problems, the performance of which can be used to challenge whether more elaborate machine learning and deep learning models are adding value.

556

- Transform native time series forecasting data into a form for fitting supervised learning algorithms and confidently tune the amount of lag observations and framing of the prediction problem.

- Develop MLP, CNN, RNN, and hybrid deep learning models quickly for a range of different time series forecasting problems, and confidently evaluate and interpret their performance.

The sky's the limit.