



MACHINE LEARNING
MASTERY

Building Transformer Models

WITH ATTENTION

Implementing a Neural
Machine Translator from
Scratch in Keras



Brief Contents

I	Foundations of Attention	1
1	WhatIsAttention?. A Bird’s Eye	2
2	View of Research on Attention . A Tour of	7
3	Attention-Based Architectures . . The	15
4	Bahdanau Attention Mechanism The	23
5	Luong Attention Mechanism.	28
II	From Recurrent Neural Networks to Transformer	33
6	AnIntroductiontoRecurrentNeuralNetworks.	3
7	UnderstandingSimpleRecurrentNeuralNetworksinKeras	4
8	TheAttentionMechanismfromScratch. Adding a Custom	4
9	Attention Layer to Recurrent Neural Network in Keras	0
10	TheTransformerAttentionMechanism	4
11	TheTransformerModel. TheVisionTransformerModel.	9
12	5
III	Building a Transformer from Scratch	6
1	PositionalEncodinginTransformerModels	86
3	Transformer Positional Encoding Layer in Keras	87
1	Implementing Scaled Dot-Product Attention in Keras.	94
4	ImplementingMulti-HeadAttentioninKeras	104
1	Implementing the Transformer Encoder in Keras	111
5	Implementing the Transformer Decoder in Keras	121
1	Joining the Transformer Encoder and Decoder with Masking.	131
6	TrainingtheTransformerModel.	140
1		151
7		
1		
8		

2	Plotting the Training and Validation Loss Curves for the Transformer Model. .	16
1	Inference with the Transformer Model.	5
2		17
2		6
IV	Application	184
23	A Brief Introduction to BERT.	185

Contents

Preface	ix
Intro duction	x
I Foundations of Attention	1
1 WhatIsAttention?	2
Attention..... Attention in Machine	2
Learning. FurtherReading	4
. Summary.....	6
	6
2 The Concept of Attention	7
Attention in Machine Learning.	7
FurtherReading	8
Summary.....	13
	14
3 A Tour of Attention-Based Architectures	15
The Encoder-Decoder Architecture. . .	15
TheTransformer.	15
GraphNeuralNetworks.	17
Memory-Augmented Neural Networks. .	19
FurtherReading	20
Summary.....	21
The Bahdanau Attention Mechanism	22
Introduction to the Bahdanau Attention.	23
4 TheBahdanauArchitecture	23
FurtherReading	23
Summary.....	24
	27
	27

5	The Luong Attention Mechanism	28
	Introduction to the Luong Attention	2
	The Luong Attention Algorithm	8
	Global Attentional Model	2
	TheLocalAttentionalModel	9
	Comparison to the Bahdanau Attention	2
	FurtherReading	9
	Summary.....	3
		0
		3
II	From Recurrent Neural Networks to Transformer	33
6	AnIntroductiontoRecurrentNeuralNetworks	34
	What Is a Recurrent Neural Network	3
	UnfoldingaRecurrentNeuralNetwork Training a	4
	Recurrent Neural Network.	3
	TypesofRNNs..... DifferentRNNArchitectures.	5
 FurtherReading..... Summary.....	3
		6
		3
	KerasSimpleRNNLayer	7
7	UnderstandingSimpleRecurrentNeuralNetworks in Keras	3
	ConsolidatedCode.....	8
	FurtherReading.....	3
	Summary.....	8
		3
	The Attention Mechanism from Scratch	9
	TheAttentionMechanism.	4
8	The General Attention Mechanism	
	The General Attention Mechanism with NumPy and SciPy	0
	FurtherReading.....	4
	Summary.....	0
		4
	Preparing Dataset for Time Series Forecasting	3
9	TheSimpleRNNNetwork	4
	AddingaCustomAttentionLayertoRecurrentNeuralNetworkinKeras	6
	AddingaCustomAttentionLayertotheNetwork	4
	ConsolidatedCode.....	
	FurtherReading.....	7
	Summary.....	4
		8
		4
	IntroductiontotheTransformerAttention.	
	Scaled Dot-Product Attention	9
10	TheTransformerAttentionMechanism	9
	Multi-HeadAttention	5
		5
		0
		5
		1
		5
		4

FurtherReading.....	Summary.....	7
		1
		7
11 The Transformer Model		1
SumUp: TheTransformerModel		7
ComparisontoRecurrentandConvolutionalLayers.		
FurtherReading.....		2
Summary.....		2
		7
12 The Introduction to the Vision Transformer (ViT)		6
TheViTArchitecture.....		7
TrainingtheViT.....		7
Inductive Bias in Comparison to Convolutional Neural Networks .		7
ComparativePerformanceofViTVariantswithResNets		7
InternalRepresentationofData		7
FurtherReading.....		8
Summary.....		7
		9
III Building a Transformer from Scratch		86
		8
13 Positional Encoding in Transformer Models		87
WhatisPositionalEncoding?		87
PositionalEncodingLayerinTransformers	Coding the	88
Positional Encoding Matrix from Scratch	Understanding	89
the Positional Encoding Matrix.		90
FurtherReading.....	Summary.....	92
		92
		8
		94
14 The Text Vectorization Layer in Keras		94
TheEmbeddingLayer.....		94
SubclassingtheKerasEmbeddingLayer.....		95
PositionalEncodinginTransformers		97
VisualizingtheFinalEmbedding.		99
FurtherReading.....		100
Summary.....		102
		103
		104
15 Recap of the Transformer Architecture		104
Implementing Scaled Dot-Product Attention in Keras		104
Implementing the Scaled Dot-Product Attention from Scratch .		106
TestingOuttheCode.....		108
FurtherReading.....		110
Summary.....		110

16 Implementing Multi-Head Attention in Keras	111
Recap of Multi-Head Attention.	11
Implementing Multi-Head Attention from Scratch	1
Testing Out the Code.	11
Further Reading.	3
Summary.	11
17 Implementing the Transformer Encoder in Keras	8
Recap of the Transformer Encoder	11
Implementing the Transformer Encoder from Scratch	9
Testing Out the Code.	12
Further Reading.	0
Summary.	12
18 Implementing the Transformer Decoder in Keras	1
Recap of the Transformer Decoder	12
Implementing the Transformer Decoder from Scratch	1
Testing Out the Code.	12
Further Reading.	2
Summary.	12
19 Joining the Transformer Encoder and Decoder with Masking	12
Recap of the Transformer Architecture	9
Masking.	13
Joining the Transformer Encoder and Decoder	0
Creating an Instance of the Transformer Model	13
Further Reading.	1
Summary.	13
20 Preparing the Training Dataset	1
Training the Transformer Model	13
Applying a Padding Mask.	3
Training the Transformer Model	13
Further Reading.	6
Summary.	13
21 Preparing the Training, Validation, and Testing Splits of the Dataset	8
Plotting the Training and Validation Loss Curves for the Transformer Model	13
Training the Transformer Model	9
Plotting the Training and Validation Loss Curves.	14
Further Reading.	0
Summary.	14
22 Inferencing the Transformer Model	14
Inference with the Transformer Model	1
Testing Out the Code.	14
Further Reading.	3
	14

Summary	183
IV Application	184
23 A Brief Introduction to BERT	185
From Transformer Model to BERT	18
WhatCanBERTDo?..... Using Pre-Trained BERT	5
Model for Summarization. . . Using Pre-Trained BERT	18
Model for Question-Answering	7
FurtherReading..... Summary.....	18
	8
	18
	9
V Appendix	191
	0
A HowtoSetupPythononYourWorkstation	192
Overview..... DownloadAnaconda	19
..... InstallAnaconda.	2
StartandUpdateAnaconda. Install	19
Deep Learning Libraries. Install	2
Visual Studio Code for Python.	19
FurtherReading	4
Summary.....	19
	5
	19
B HowtoSetupAmazonEC2forDeepLearningonGPUs	8
Overview.....	19
Setup Your AWS Account	8
Launch Your Server Instance.....	20
Login, Configure and Run	1
Build and Run Models on AWS	20
CloseYourEC2Instance	1
Tips and Tricks for Using Keras on AWS	20
FurtherReading	2
Summary.....	20
	2
How Far You Have Come	202
	20
	3
	20
	4
	20
	6
	20
	9
	21
	0
	21

Preface

It is not an easy task to ask a computer to understand human language. In recent years, we have seen significant progress due to the advance in machine learning techniques. In particular, attention mechanisms and transformers.

Take machine translation as an example. In the past we would consider that as a sequence to sequence transformation problem that a recurrent neural network would fit. But instead of a simple linear transformation, using an attention mechanism was proven to work better with longer sentences. Later, it is discovered that attention without recurrent neural network is not only possible, but also better in many situations.

This book is a guide to lead you to fully understand attention and transformer architecture. We start with the first principle: to build a transformer model in Keras, from scratch. We hope by the time you finish the book, you can appreciate the idea of using attention to extract *context* out of a sequence.

Introduction

Welcome to Building Transformer Models with Attention.

A Recurrent Neural Network (RNN) has been considered magical, and someone even called it unreasonably effective.¹ However, it is not almighty. In machine translation, we have seen that an RNN can give sensible output but not always correct and accurate. While it is a neural network, it is not about the network being too simple or that we didn't train it enough. On the contrary, no matter how hard we train an RNN, there is a ceiling that it cannot breakthrough. Researchers noticed that when using an RNN for translation from one language to another, the neural network reads one word at a time but never sees the entire sentence. Therefore, the traditional way of using an RNN means it will lose the *context*.

Attention is how to mitigate this situation. But it is not as simple as the linear transformation that we usually see in neural networks. Furthermore, researchers found that attention is not necessary to use an RNN. Attention itself can be extended to be a neural network as well. If we do that, we translate words one by one to some encoding or translate the encoding to words. This is a transformer.

However, building an effective transformer for the translation of human languages is not trivial. Partially it is due to the high dimensionality of languages, i.e., any language has thousands of words and can carry a tremendous amount of information. It is also due to the complex architecture of the transformer. But once this hurdle is overcome, you will find the capability of a neural network to deal with human language at a new stage. For example, BERT is an extension of the transformer encoder. We saw that it can be used to build a named entity recognition (NER) system effectively. Another example is GPT-2, which is an extension of the transformer decoder. We saw that it can be used to build a natural language generator that produces realistic-looking paragraphs. These two examples are much larger than the original transformer and very slow to train, but undeniably have their root in attention and transformer.

This book is to guide you through creating a transformer, step-by-step. By doing that, you will learn how to transform a word in a language to an embedding vector, how to implement the attention mechanism, to how a transformer is constructed, and eventually, use it to perform a language translation task.

Book Organization

This book is in four parts:

Part 1: Foundation of Attention

In this part you will learn about the theoretical background of attention mechanism. In particular, you will see how attention is defined mathematically and the algorithm to get it. You will also see from a high level how attention, a concept to understand a sequence, is incorporated into a larger neural network architecture as well as its application. This part of the book includes the following chapters:

- ◄ What Is Attention?
- ◄ A Bird's Eye View of Research on Attention
- ◄ A Tour of Attention-Based Architectures
- ◄ The Bahdanau Attention Mechanism
- ◄ The Luong Attention Mechanism

Part 2: From Recurrent Neural Networks to Transformer

Since attention was originally designed for recurrent neural networks, we follow the footstep of its history to start from a traditional recurrent neural network and add an attention layer into it. You may have forgotten how a recurrent neural network is structured, therefore we start from an introduction to the RNN. This part of the book includes the following chapter:

- ◄ An Introduction to Recurrent Neural Networks
- ◄ Understanding Simple Recurrent Neural Networks in Keras
- ◄ The Attention Mechanism from Scratch
- ◄ Adding a Custom Attention Layer to Recurrent Neural Network in Keras
- ◄ The Transformer Attention Mechanism
- ◄ The Transformer Model
- ◄ The Vision Transformer Model

At the end of this part, we introduce how an attention mechanism can stand on its own without an RNN. This is how a transformer is created. While the entire story of attention and transformer is motivated by applying neural networks to natural language processing tasks, the last chapter of this part give you an angle from computer vision to show you that the potential of transformer is not limited to NLP.

Part 3: Building a Transformer from Scratch

Unlike other chapters of this book, you are required to read the chapters of this book in its prescribed sequence. The ten chapters in this part lead you into building a fully working transformer model from scratch. We start from the first step, namely, adding positional

encoding to input sequence, and end with using a trained transformer model for inference. This part of the book includes the following chapters:

- ◄ Positional Encoding in Transformer Models
- ◄ Transformer Positional Encoding Layer in Keras
- ◄ Implementing Scaled Dot-Product Attention in Keras
- ◄ Implementing Multi-Head Attention in Keras
- ◄ Implementing the Transformer Encoder in Keras
- ◄ Implementing the Transformer Decoder in Keras
- ◄ Joining the Transformer Encoder and Decoder with Masking
- ◄ Training the Transformer Model
- ◄ Plotting the Training and Validation Loss Curves for the Transformer Model
- ◄ Inference with the Transformer Model

There are a lot to cover in these chapters because of the complexity in transformer architecture.

Be patient. But you will find it not difficult to have your own transformer created out of basic Keras functions.

Part 4: Application

While you already created your own transformer and by the end of last part, your transformer should be able to do sentence to sentence translation between two languages in a reasonable quality. However, the story of transformer is not stop here. There are larger transformer-based

architectures proposed with pre-trained model weights made public. We will look into one example and see how we can do some amazing projects with the pre-trained model.

There is only one chapter in this part. It is:

- ◄ A Brief Introduction to BERT

In this chapter you will learn about BERT, which is an extension to transformer's encoder, and its simplified model, DistilBERT. You will see how you can do summarization and question-answering with a pre-trained DistilBERT model.

Requirements for This Book

Python and TensorFlow 2.x

This book covers some advanced topic. You do not need to be a Python expert, but you need to know how to install and setup Python and TensorFlow. You need to be able to install libraries if required and you should be able to navigate through the Python development environment comfortably. You may set up your environment on your workstation or laptop. It can be in a VM or a Docker instance that you run, or it may be a server that you can configure in the cloud.

Appendix A and Appendix B of this book gives you step-by-step guidance on how to set up a Python environment on your own computer and on AWS cloud, respectively.

Machine Learning

You do not need to be a machine learning expert, but it would be helpful if you know how to solve a small machine learning problem, especially a natural language processing task. Basic concepts like cross-validation are described briefly, and you are expected to know how to train and use a neural network in TensorFlow and Keras. You may learn about these in another book, *Deep Learning with Python*.

Training a transformer model can take a long time. It is possible to train it using a CPU but GPU can speed it up significantly. You can access GPU hardware easily and cheaply in the cloud and a step-by-step procedure is taught on how to do this in Appendix B.

Your Outcomes from Reading This Book

This book is a guidebook to help you learn the internal of a transformer model and the attention mechanism. Upon finishing the book, you should be able to explain clearly why attention works and how a transformer can handle a sequence such as a paragraph of words. Specifically, you will know:

- ◁ What is attention, especially the Bahdanau attention and Luong attention
- ◁ What is a multi-head attention and how it is used in transformer models
- ◁ How to build encoder and decoder of a transformer
 - ◁ How to combine the encoder and decoder to create a fully working transformer, and how to train it
- ◁ How to use transformer for real-world tasks

From here you can go deeper to investigate other transformer models, for natural language or for computer vision. You also understand how and what a transformer does. Therefore, you can download some pre-trained models and use them for various tasks.

To get the very most from this book, we recommend following each chapter and build upon them. Attempt to improve the results or the implementation. Write up what you tried or learned and share it on your blog, social media or send us an email at jason@MachineLearningMastery.com.

Summary

This book is a bit different from our other books from MachineLearningMastery.com in the sense that there are not a lot of small projects to work with. Instead, this entire book is one big project, to build a transformer model and apply it to NLP. A big project has many small components. By doing this project, you will learn a lot of ideas. Hope this will be eye-opening for you and bring you to a different level of deep learning. We are excited for you. Take your

time, have fun and we're so excited to see where you can take this amazing new technology to.

Next

Let's dive in. Next up is Part I where you will learn the foundation of attention.

Foundations of Attentio

In

What Is Attention?

Attention is becoming increasingly popular in machine learning, but what makes it such an attractive concept? What is the relationship between attention applied in artificial neural networks and its biological counterpart? What components would one expect to form an attention-based system in machine learning?

In this chapter, you will discover an overview of attention and its application in machine learning. After completing this chapter, you will know:

- ◄ A brief overview of how attention can manifest itself in the human brain
 - ◄ The components that make up an attention-based system and how these are inspired by biological attention

Let's get started.

Overview

This chapter is divided into two parts; they are:

- ◄ Attention
- ◄ Attention in Machine Learning

1.1 Attention

Attention is a widely investigated concept that has often been studied in conjunction with arousal, alertness, and engagement with one's surroundings.

“In its most generic form, attention could be described as merely an overall level of alertness.”

Itti et al. (2001) mention that such salient image parts are often characterized by visual attributes, including intensity contrast, oriented edges, corners and junctions, and motion. The human brain attends to these salient visual features at different neuronal stages.

“Neurons at the earliest stages are retuned to simple visual attributes such as intensity contrast, colour opponency, orientation, direction and velocity of motion, or stereo disparity at several spatial scales. Neuronal tuning becomes increasingly more



Figure 1.1: Visual attention is to find the salient image parts. From “Computational Modelling of Visual Attention”

specialized with the progression from low-level to high-level visual areas, such that higher-level visual areas include neurons that respond only to corners or junctions, shape from shading cues or views of specific real-world objects.

Research has also discovered several forms of interaction between memory and attention. — “Computational Modelling of Visual Attention”, 2001

Interestingly, research has also observed that different subjects tend to be attracted to the same salient visual cues.

1.2 Attention in Machine Learning

Implementing the attention mechanism in artificial neural networks does not necessarily track the biological and psychological mechanisms of the human brain. Instead, it is the ability to dynamically highlight and use the *salient* parts of the information at hand — in a similar manner as it does in the human brain — that makes attention such an attractive concept in machine learning.

Think of an attention-based system consisting of three components:

“1. A process that “reads” raw data (such as source word sequences or source sentences), and converts them into distributed representations, with one feature vector associated with each word position.

2. A list of feature vectors storing the output of the reader. This can be understood as a “memory” containing a sequence of facts, which can be retrieved later, not necessarily in the same order, without having to visit all of them.

3. A process that “exploits” the content of the memory to sequentially perform a task, at each time step having the ability to put attention on the content of one memory element (or a few, with a different weight).

— Page 491, *Deep Learning*, 2016

Let's take the encoder-decoder framework as an example since it is within

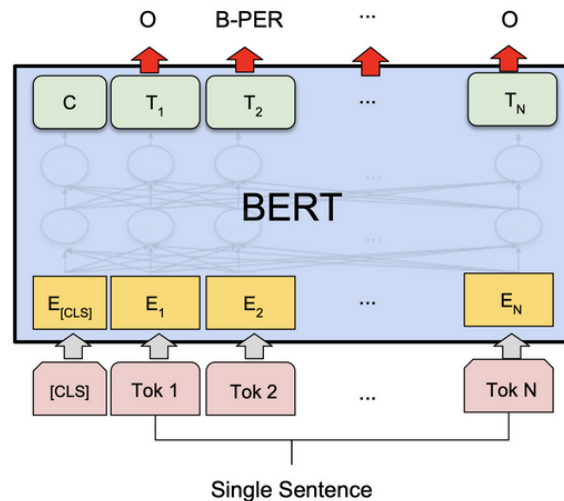


Figure 1.2: Sequence of words fed into encoder will output a vector for every element in the sequence. From “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”

At each time step, the attention mechanism then takes the previous hidden state of the decoder and the list of encoded vectors, using them to generate unnormalized *score* values that indicate how well the elements of the input sequence align with the current output. Since the generated score values need to make relative sense in terms of their importance, they are normalized by passing them through a softmax function to generate the *weights*. Following the softmax normalization, all the weight values will lie in the interval $[0,1]$ and add up to 1, meaning they can be interpreted as probabilities. Finally, the encoded vectors are scaled by the computed weights to generate a *context vector*. This attention process forms the third component of the attention-based system above. It is this context vector that is then fed into the decoder to generate a translated output.

“This type of artificial attention is like biological attention.

— “Attention in Psychology, Neuroscience, and Machine Learning”, 2020

The process implemented by a

with one that does not. In the latter, the encoder would generate a fixed-length vector irrespective of the input’s length or complexity. In the absence of a mechanism that highlights the salient information across the entirety of the input, the decoder would only have access to the limited information that would be encoded within the fixed-length vector. This would potentially result in the decoder missing important information.

The attention mechanism was initially proposed to process sequences of words in machine translation, which have an implied temporal aspect to them. However, it can be generalized to process information that can be static, and not necessarily related in a sequential fashion, such as in the context of image processing. You will see how this generalization can be achieved in other chapters.

1.3 Further Reading

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

Books

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

<https://www.amazon.com/dp/0262035618>

(Online version at <http://www.deeplearningbook.org>).

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing, 2018.

<https://www.amazon.com/dp/1785880365>

Papers

Grace W. Lindsay. “Attention in Psychology, Neuroscience, and Machine Learning”. *Frontiers in Computational Neuroscience*, 14, 2020, p. 29. DOI: [10.3389/fncom.2020.00029](https://doi.org/10.3389/fncom.2020.00029).

<https://www.frontiersin.org/articles/10.3389/fncom.2020.00029/full>

Laurent Itti and Christof Koch. “Computational Modelling of Visual Attention”. *Nature Reviews Neuroscience*, 2, Feb. 2001, pp. 1–11.

<https://authors.library.caltech.edu/40408/1/391.pdf>

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proc. NAACL*. Vol. 1. June 2019, pp. 4171–4186. DOI: .

[10.18653/v1/N19-1423](https://doi.org/10.18653/v1/N19-1423)

<https://arxiv.org/abs/1810.04805>

1.4 Summary

In this chapter, you discovered an overview of attention and its application in machine learning. Specifically, you learned:

- ◁ A brief overview of how attention can manifest itself in the human brain

- ◁ The components that make up an attention-based system and how these are inspired by biological attention

In the next chapter, you will skim through some landmark research on attention in machine learning.

A Bird's Eye View of Research on Attention

Attention is a concept that is scientifically studied across multiple disciplines, including psychology, neuroscience, and, more recently, machine learning. While all disciplines may have produced their own definitions for attention, one core quality they can all agree on is that attention is a mechanism for making both biological and artificial neural systems more flexible.

In this chapter, you will discover an overview of the research advances on attention. After completing this chapter, you will know:

- ◁ The concept of attention that is of significance to different scientific disciplines
 - ◁ How attention is revolutionizing machine learning, specifically in the domains of natural language processing and computer vision

Let's get started.

Overview

This chapter is divided into two parts; they are:

- ◁ The Concept of Attention
- ◁ Attention in Machine Learning

2.1 The Concept of Attention

Research on attention finds its origin in the field of psychology.

“The scientific study of attention

— “Attention in Psychology, Neuroscience, and Machine Learning”, 2020

Observations derived from such

underlying such behavioral patterns. While the different fields of psychology, neuroscience, and, more recently, machine learning have all produced their own definitions of attention, there is one core quality that is of great significance to all:

Attention is the flexible control of limited computational resources.

— “Attention in Psychology, Neuroscience, and Machine Learning”, 2020

With this in mind, the following sections review the role of attention in revolutionizing the field of machine learning.

2.2 Attention in Machine Learning

The concept of attention in machine learning is *very* loosely inspired by the psychological mechanisms of attention in the human brain.

The use of attention mechanisms in artificial neural networks came about — much like the apparent need for attention in the brain — as a means of making neural systems more flexible.

— “Attention in Psychology, Neuroscience, and Machine Learning”, 2020

The idea is to be able to work with an artificial neural network that can perform well on tasks where the input may be of variable length, size, or structure or even handle several different tasks. It is in this spirit that attention mechanisms in machine learning are said to inspire themselves from psychology rather than because they replicate the biology of the human brain.

In the form of attention originally developed for ANNs, attention mechanisms worked within an encoder-decoder framework and in the context of sequence models.

— “Attention in Psychology, Neuroscience, and Machine Learning”, 2020

The task of the encoder is to generate a vector representation of the input, whereas the task of the decoder is to transform this vector representation into an output. The attention mechanism connects the two.

There have been different propositions of neural network architectures that implement attention mechanisms, which are also tied to the specific applications in which they find their use. Natural language processing (NLP) and computer vision are among the most popular applications.

2.2.1 Attention in Natural Language Processing

An early application for attention in NLP was machine translation, where the goal was to translate an input sentence in a source language to an output sentence in a target language. Within this context, the encoder would generate a set of *context* vectors, one for each word in the source sentence. The decoder, on the other hand, would read the context vectors to generate an output sentence in the target language, one word at a time.

“In the traditional encoder-decoder framework without attention, the encoder produces

Representing the input by a fixed-length vector was especially problematic for long sequences or sequences that were complex in structure since the dimensionality of their representation was forced to be the same as for shorter or simpler sequences.

“For example, in some languages, such as Japanese, the last word might be very important to predict the first word, while translating English to French might be easier as the order of the sentences (how the sentence is organized) is more similar

to each other.

— “Attention in Psychology, Neuroscience, and Machine Learning”, 2020

This created a bottleneck whereby the decoder has limited access to the information provided

by the input — that which is available within the fixed-length encoding vector. On the other hand, preserving the length of the input sequence during the encoding process could make it possible for the decoder to utilize its most relevant parts in a flexible manner.

The latter is how the attention mechanism operates.

“Attention helps determine

More recently, Vaswani et al. (2017) proposed an entirely different architecture that has steered the field of machine translation in a new direction. Termed *transformer*, their architecture dispenses with any recurrence and convolutions altogether but implements a *self-attention* mechanism. Words in the source sequence are first encoded in parallel to generate key, query, and value representations. The keys and queries are combined to generate attention weightings that capture how each word relates to the others in the sequence. These attention weightings are then used to scale the values, in order to retain focus on the important words and drown out the irrelevant ones.

“The output is computed as a weighted sum of the values, where the weights assigned to each value is computed by a compatibility function of the query with the corresponding key.”

— “Attention Is All You Need”, 2017

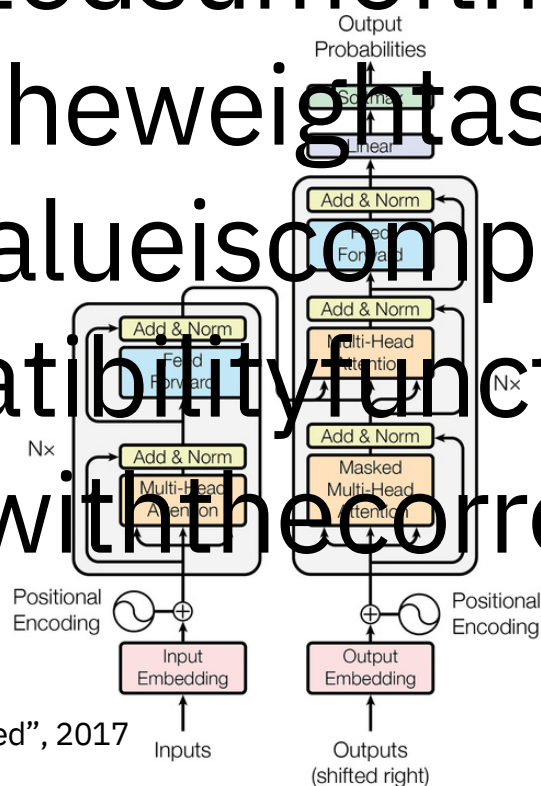


Figure 2.1: The transformer architecture. From “Attention Is All You Need”

At the time, the proposed transformer architecture established a new state-of-the-art on English-to-German and English-to-French translation tasks. It was reportedly also faster to train than architectures based on recurrent or convolutional layers. Subsequently, the method called BERT by Devlin et al. (2019) built on Vaswani et al.’s work by proposing a multilayer bidirectional architecture. The transformer architecture will be covered in detail in Chapters 10 and 11.

As we shall see shortly, the uptake of the transformer architecture was not only rapid in the domain of NLP but also in the computer vision domain.

2.2.2 Attention in Computer Vision

In computer vision, attention has found its way into several applications, such as in the domains of image classification, image segmentation, and image captioning.

For example, if we had to reframe the encoder-decoder model to the task of image captioning, then the encoder could be a Convolutional Neural Network (CNN) that captured the salient visual cues in the images into a vector representation. And the decoder can be an RNN or LSTM that transformed the vector representation into an output.

“Also, as in the neuroscience literature, these attentional processes can be divided into spatial and feature-based attention.



—“Attention in Psychology, Neuroscience, and Machine Learning”, 2020

Figure 2.2: Model for image caption generation. From “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”

Feature attention, in comparison, permits individual feature maps to be attributed their own weight values. One such example, also applied to image captioning, is the encoder-decoder framework of Chen et al. (2017), which incorporates spatial and channel-wise attentions in the same CNN.

Similarly to how the transformer has quickly become the standard architecture for NLP tasks, it has also been recently taken up and adapted by the computer vision community.

In *spatial* attention, different spatial locations are attributed different

The earliest work to do so was proposed by Dosovitskiy et al. (2021), who applied their *Vision Transformer* (ViT) to an image classification task. They argued that the long-standing reliance on CNNs for image classification was not necessary, and the same task could be accomplished by a pure transformer. Dosovitskiy et al. reshape an input image into a sequence of flattened 2D image patches, which they subsequently embed by a trainable linear projection to generate the *patch embeddings*. These patch embeddings, together with their *position embeddings*, to retain positional information, are fed into the encoder part of the transformer architecture, whose output is subsequently fed into a Multilayer Perceptron (MLP) for classification.

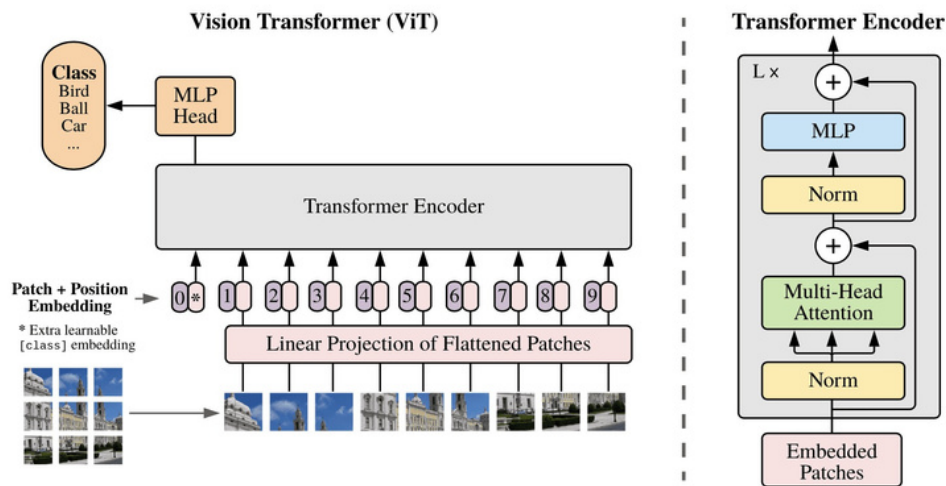


Figure 2.3: The vision transformer architecture. From “An Image is Worth 16x16 Words”

“Inspired by ViT, and the fact that

— “ViViT: A Video Vision Transformer”, 2021

Arnab et al. (2021), subsequent

spatiotemporal information contained within videos for the task of video classification. Their method explores different approaches of extracting the spatiotemporal data, such as by sampling and embedding each frame independently, or by extracting non-overlapping tubelets (an image patch that spans across several image frames, creating a *tube*) and embedding each one in turn. They also investigate different methods of factorising the spatial and temporal dimensions of the input video, for increased efficiency and scalability.

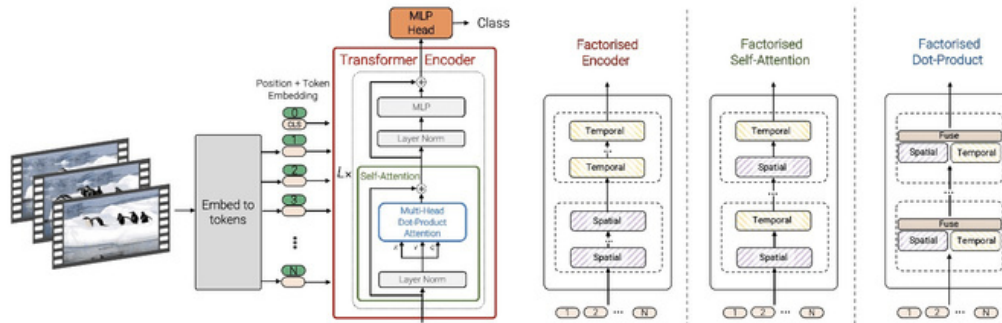


Figure 2.4: The video vision transformer architecture. From “ViViT: A Video Vision Transformer”

In its first application for image classification, the Vision Transformer is already being applied to several other computer vision domains, such as action localization, gaze estimation, and image generation. This surge of interest among computer vision practitioners suggests an exciting near future, where we’ll be seeing more adaptations and applications of the transformer architecture.

2.3 Further Reading

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

Books

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing, 2018.

<https://www.amazon.com/dp/1785880365>

Papers

Grace W. Lindsay. “Attention in Psychology, Neuroscience, and Machine Learning”. *Frontiers in Computational Neuroscience*, 14, 2020, p. 29. DOI: [10.3389/fncom.2020.00029](https://doi.org/10.3389/fncom.2020.00029).

<https://www.frontiersin.org/articles/10.3389/fncom.2020.00029/full>

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *Proc. 3rd International Conference on Learning Representations (ICLR 2015)*. May 2015.

<http://arxiv.org/abs/1409.0473>

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *Proc. NIPS*. Vol. 2. Dec. 2014, pp. 3104–3112.

<https://arxiv.org/abs/1409.3215>

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. In: *Proc. EMNLP*. Sept. 2015, pp. 1412–1421. DOI: [10.18653/v1/D15-1166](https://doi.org/10.18653/v1/D15-1166)

<https://arxiv.org/abs/1508.04025>

- Kelvin Xu et al. “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. July 2015, pp. 2048–2057.
<https://arxiv.org/abs/1502.03044>
- Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.
<https://arxiv.org/pdf/1706.03762.pdf>
- Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding”. In: *Proc. NAACL*. Vol. 1. June 2019, pp. 4171–4186. DOI: [10.18653/v1/N19-1423](https://doi.org/10.18653/v1/N19-1423).
<https://arxiv.org/abs/1810.04805>
- Long Chen, Hanwang Zhang, Jun Xiao, Liqiang Nie, Jian Shao, Wei Liu, and Tat-Seng Chua. “SCA-CNN: Spatial and Channel-wise Attention in Convolutional Networks for Image Captioning”. In: *Proc. IEEE/CVF Computer Vision and Pattern Recognition Conference (CVPR)*. July 2017.
<https://arxiv.org/abs/1611.05594>
- Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *Proc. 9th International Conference on Learning Representations (ICLR)*. May 2021.
<https://arxiv.org/abs/2010.11929>
- Anurag Arnab, Mostafa Dehghani, Georg Heigold, Chen Sun, Mario Lučić, and Cordelia Schmid. “ViViT: A Video Vision Transformer”. In: *Proc. International Conference on Computer Vision (ICCV)*. Oct. 2021.
<https://arxiv.org/abs/2103.15691>
- Yutong Feng et al. “Relation Modeling in Spatio-Temporal Action Localization”, 2021.
<https://arxiv.org/abs/2106.08061>
- Yihua Cheng and Feng Lu. “Gaze Estimation using Transformer”. In: *Proc. 26th International Conference on Pattern Recognition (ICPR)*. Montréal Québec, Aug. 2022.
<https://arxiv.org/abs/2105.14424>
- Kwonjoon Lee, Huiwen Chang, Lu Jiang, Han Zhang, Zhuowen Tu, and Ce Liu. “ViTGAN: Training GANs with Vision Transformers”, 2021.
<https://arxiv.org/abs/2107.04589>

2.4 Summary

In this chapter, you discovered an overview of the research advances on attention. Specifically, you learned:

- ◁ The concept of attention that is of significance to different scientific disciplines
 - ◁ How attention is revolutionizing machine learning, specifically in the domains of natural language processing and computer vision

In the next chapter, you will see how attention can be incorporated in a neural network model.

A Tour of Attention-Based Architectures

As the popularity of attention in machine learning grows, so does the list of neural architectures that incorporate an attention mechanism.

In this chapter, you will discover the salient neural architectures that have been used in conjunction with attention. After completing this chapter, you will better understand how the attention mechanism is incorporated into different neural architectures and for which purpose.

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ◀ The Encoder-Decoder Architecture
- ◀ The Transformer
- ◀ Graph Neural Networks
- ◀ Memory-Augmented Neural Networks

3.1 The Encoder-Decoder Architecture

The encoder-decoder architecture has been extensively applied to sequence-to-sequence (seq2seq) tasks for language processing. Examples of such tasks within the domain of language processing include machine translation and image captioning.

“The earliest use of attention was as part of RNN based encoder-decoder framework to enco

For an RNN-based encoder-decoder architecture *without* attention, unrolling each RNN would produce the following graph:

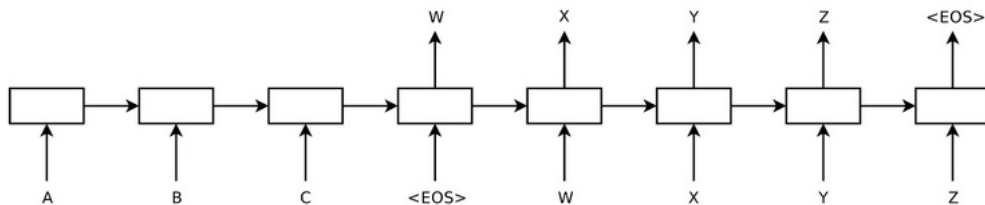


Figure 3.1: Unrolled RNN-based encoder and decoder. From “Sequence to Sequence Learning with Neural Networks”

Here, the encoder reads the input sequence one word at a time, each time updating its internal state. It stops when it encounters the $\langle \text{EOS} \rangle$ symbol, which signals that the *end of sequence* has been reached. The hidden state generated by the encoder essentially contains a vector representation of the input sequence, which the decoder will then process.

The decoder generates the output sequence one word at a time, taking the word at the previous time step ($t - 1$) as input to generate the next word in the output sequence. An $\langle \text{EOS} \rangle$ symbol at the decoding side signals that the decoding process has ended.

As we have previously mentioned, the problem with the encoder-decoder architecture without attention arises when sequences of different lengths and complexities are represented by a fixed-length vector, potentially resulting in the decoder missing important information.

“A potential issue with this encoder-decoder approach is that a neural network needs to be able to compress all the necessary information of a source sentence into a fixed-

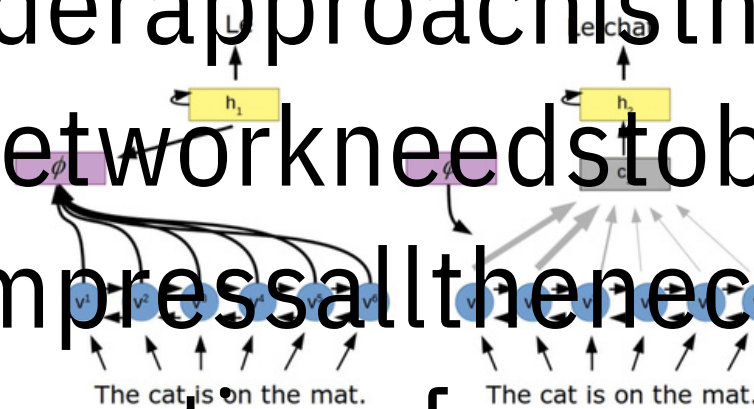


Figure 3.2: Encoder-decoder architecture with attention. From “Attention in Psychology, Neuroscience, and Machine Learning”

Here, the attention mechanism (ϕ) learns a set of attention weights that capture the relationship between the encoded vectors (v) and the hidden state of the decoder (h) to generate a context vector (c) through a weighted sum of all the hidden states of the encoder. In doing so, the decoder would have access to the entire input sequence, with a specific focus on the input information most relevant for generating the output.

3.2 The Transformer

The architecture of the transformer also implements an encoder and decoder. However, as opposed to the architectures reviewed above, it does not rely on the use of recurrent neural networks. For this reason, this chapter will review this architecture and its variants separately.

The transformer architecture dispenses of any recurrence and instead relies solely on a *self-attention* (or intra-attention) mechanism.

“In terms of computational complexity, self-attention layers are faster than recurrent layers when the sequence length is smaller than the representation dimensionality.

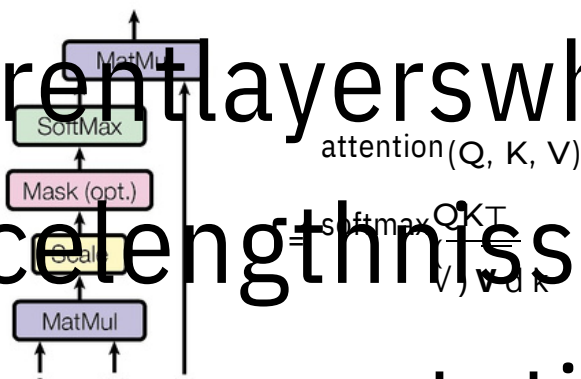


Figure 3.3: Multiplicative attention, in flow chart and in formula. From “Attention Is All You Need”

Intuitively, since all queries, keys and values originate from the same input sequence, the self-attention mechanism captures the relationship between the different elements of the same sequence, highlighting those that are most relevant to one another.

Since the transformer does not rely on RNNs, the positional information of each element in the sequence can be preserved by augmenting the encoder’s representation of each element.

The self-attention mechanism relies on the use of *queries*, *keys*, and *values*, which are

with positional encoding. This means that the transformer architecture may also be applied to tasks where the information may not necessarily be related sequentially, such as for the computer vision tasks of image classification, segmentation, or captioning.

“Transformers can capture global/long ranged dependencies between input and output, support parallel processing, require minimal inductive biases (prior knowledge), demonstrate scalability to large sequences and datasets, and

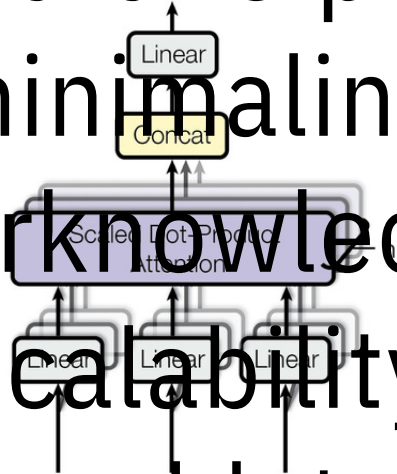


Figure 3.4: Multi-head attention. From “Attention Is All You Need”

Some variants of the transformer architecture that address the limitations of the vanilla model are:

- ◀ **Transformer-XL:** Introduces recurrence so that it can learn longer-term dependency beyond the fixed length of the fragmented sequences that are typically used during training. (Attention: Survey of Attention Models”, 2021)

- ◀ **XLNet:** A bidirectional transformer that builds on Transformer-XL by introducing a permutation-based mechanism, where training is carried out not only on the original order of the elements comprising the input sequence but also over different permutations of the input sequence order.

Furthermore, several can be stacked in parallel in what has been termed

multi-head attention. Each head works in parallel over different linear transformations of the same input, and the outputs of the heads are then concatenated to produce the final attention result. Because multiple attention heads can work independently and in parallel, having a multi-head model can have each head attend to different elements of the sequence.

3.3 Graph Neural Networks

A graph can be defined as a set of *nodes* (or vertices) that are linked through *connections* (or edges).

“ A graph is a versatile data structure that lends itself well to the way data is organized in many real-world scenarios.

— *Advanced Deep Learning with Python*, 2019

For example, take a social network where users can be represented by nodes in a graph and

their relationships with friends by edges. Or a molecule, where the nodes would be the atoms, and the edges would represent the chemical bonds between them.

We can think of an image as a graph, where each pixel is a node, directly connected to its neighboring pixels ...

— *Advanced Deep Learning with Python*, 2019

Of particular interest are the Graph Attention Networks (GAT) that employ a self-attention

mechanism within a graph convolutional network (GCN), where the latter updates the state vectors by performing a convolution over the nodes of the graph. The convolution operation is applied to the central node and the neighboring nodes using a weighted filter to update the representation of the central node. The filter weights in a GCN can be fixed or learnable.

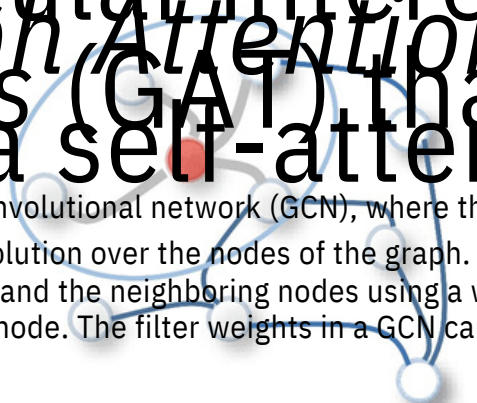


Figure 3.5: Graph convolution over a central node (red) and a neighborhood of nodes.
From “A Comprehensive Survey on Graph Neural Networks”

In comparison, a GAT assigns weights to the neighboring nodes using attention scores.

The computation of these attention scores follows a similar procedure as in the methods for the seq2seq tasks reviewed above: (1) alignment scores are first computed between the feature vectors of two neighboring nodes, from which (2) attention scores are computed by applying a softmax operation, and finally (3) an output feature vector for each node (equivalent to the context vector in a seq2seq task) can be computed by a weighted combination of the feature vectors of all its neighbors.

Multi-head attention can also be applied here in a very similar manner to how it was proposed in the transformer architecture previously seen. Each node in the graph would be assigned multiple heads, and their outputs would be averaged in the final layer.

Once the final output has been produced, this can be used as the input for a subsequent task-specific layer. Tasks that can be solved by graphs can be the classification of individual nodes between different groups (for example, in predicting which of several clubs a person will decide to become a member of). Or they can be the classification of individual edges to determine whether an edge exists between two nodes (for example, to predict whether two persons in a social network might be friends) or even the classification of a full graph (for example, to predict if a molecule is toxic).

3.4 Memory-Augmented Neural Networks

In the encoder-decoder attention-based architectures reviewed so far, the set of vectors that encode the input sequence can be considered external memory, to which the encoder writes and from which the decoder reads. However, a limitation arises because the encoder can only write to this memory, and the decoder can only read.

Memory-Augmented Neural Networks (MANNs) are recent algorithms that aim to address this limitation. The Neural Turing Machine (NTM) is one type of MANN. It consists of a neural network controller that takes an input to produce an output and performs read and write operations to memory.

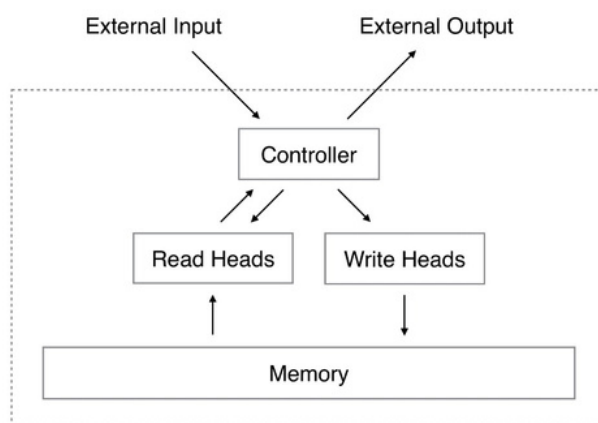


Figure 3.6: Neural Turing machine architecture. From “Neural Turing Machines”

The operation performed by the read head is similar to the attention mechanism employed for seq2seq tasks, where an attention weight indicates the importance of the vector under consideration in forming the output.

“A read head always reads the full memory matrix, but it does

The write head also makes use of an attention vector, together with an erase and add vectors. A memory location is erased based on the values in the attention and erase vectors, and information is written via the add vector.

Examples of applications for MANNs include question-answering and chat bots, where an external memory stores a large database of sequences (or facts) that the neural network taps into. The role of the attention mechanism is crucial in selecting facts from the database that are more relevant than others for the task at hand.

3.5 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing, 2018.

<https://www.amazon.com/dp/1785880365>

Papers

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *Proc. NIPS*. Vol. 2. Dec. 2014, pp. 3104–3112.

<https://arxiv.org/abs/1409.3215>

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *Proc. 3rd International Conference on Learning Representations (ICLR 2015)*. May 2015.

<http://arxiv.org/abs/1409.0473>

Grace W. Lindsay. “Attention in Psychology, Neuroscience, and Machine Learning”. *Frontiers in Computational Neuroscience*, 14, 2020, p. 29. DOI: [10.3389/fncom.2020.00029](https://doi.org/10.3389/fncom.2020.00029).

<https://www.frontiersin.org/articles/10.3389/fncom.2020.00029/full>

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu.

“A Comprehensive Survey on Graph Neural Networks”. *IEEE Transactions on Neural Networks and Learning Systems*, 32(1), 2021, pp. 4–24. DOI: [10.1109/TNNLS.2020.2978386](https://doi.org/10.1109/TNNLS.2020.2978386).

Alex Graves, Greg Wayne, and Ivo Danihelka. “Neural Turing Machines”, 2014.

<https://arxiv.org/abs/1901.00596>

<https://arxiv.org/abs/1410.5401>

3.6 Summary

In this chapter, you discovered the salient neural architectures that have been used in conjunction with attention. Specifically, you gained a better understanding of how the attention mechanism is incorporated into different neural architectures and for which purpose.

In the next chapter, you will look into the first proposed attention mechanism.

The Bahdanau Attention Mechanism

Conventional encoder-decoder architectures for machine translation encoded every source sentence into a fixed-length vector, regardless of its length, from which the decoder would then generate a translation. This made it difficult for the neural network to cope with long sentences, essentially resulting in a performance bottleneck. The Bahdanau attention was proposed to address the performance bottleneck of conventional encoder-decoder architectures, achieving significant improvements over the conventional approach.

In this chapter, you will discover the Bahdanau attention mechanism for neural machine translation. After completing this chapter, you will know:

- ◁ Where the Bahdanau attention derives its name from and the challenge it addresses
- ◁ The role of the different components that form part of the Bahdanau encoder-decoder architecture
- ◁ The operations performed by the Bahdanau attention algorithm

Let's get started.

Overview

This chapter is divided into two parts; they are:

- ◁ Introduction to the Bahdanau Attention
- ◁ The Bahdanau Architecture

4.1 Introduction to the Bahdanau Attention

The Bahdanau attention mechanism inherited its name from the first author of the paper in which it was published. It follows the work of Cho et al. (2014) and Sutskever et al. (2014), who also employed an RNN encoder-decoder framework for neural machine translation, specifically by encoding a variable-length source sentence into a fixed-length vector. The latter would then be decoded into a variable-length target sentence.

Bahdanau et al. (2015) argued that this encoding of a variable-length input into a fixed-length vector *squashes* the information of the source sentence, irrespective of its length, causing

the performance of a basic encoder-decoder model to deteriorate rapidly with an increasing length of the input sentence. The approach they proposed replaces the fixed-length vector with a variable-length one to improve the translation performance of the basic encoder-decoder model.

“The most important distinguishing feature of this approach from the basic encoder-decoder is that it does not attempt to encode a whole input sentence into a single fixed-length vector. Instead, it encodes the input sentence into a sequence of vectors



Figure 4.1: The Bahdanau architecture. From “Neural Machine Translation by Jointly Learning to Align and Translate”

4.2 The Bahdanau Architecture

The main components in use by the Bahdanau encoder-decoder architecture (Figure 4.1) are the following:

◁ **s** and chooses a subset of these vectors adaptively while decoding the translation. $t-1$ is the *hidden decoder state* at the previous time step t

— “Neural Machine Translation by Jointly Learning to Align and Translate”

◁ **c**

t is the *context vector* at time step t . It is uniquely generated at each decoder step to generate a target word y_t .

◁ **h**

i is an *annotation* that captures the information contained in the words forming the entire input sentence $\{x_1, x_2, \dots, x_T\}$ with strong focus around the i -th word out of T total words.

◁ **α**

t,i is a *weight* value assigned to each annotation h_i at the current time step t .

◁ **e**

t,i is an *attention score* generated by an alignment model $a()$ that scores how well

s_{t-1} and h_i match.

These components find their use at different stages of the Bahdanau architecture, which employs a bidirectional RNN as an encoder and an RNN decoder, with an attention mechanism in between.

The Encoder

The role of the encoder is to generate an annotation \mathbf{h}_i (a column vector) for every word x_i in an input sentence of length T words.

For this purpose, Bahdanau et al. employ a bidirectional RNN, which reads the input sentence in the forward direction to produce a forward hidden state \mathbf{h}_i , and then reads the input sentence in the reverse direction to produce a backward hidden state \mathbf{h}_i . The annotation for some particular word x_i concatenates the two states:

$$\mathbf{h}_i = [\mathbf{h}_i^{\rightarrow} \parallel \mathbf{h}_i^{\leftarrow}]$$

The idea behind generating each annotation in this manner was to capture a summary of both the preceding and succeeding words.

“In this way, the annotation \mathbf{h}_i contains the summaries of both the preceding words and the following words.

—“Neural Machine Translation by Jointly Learning to Align and Translate”

$$2.a(\mathbf{s}, \mathbf{h}) = \mathbf{v}^T \tanh(\mathbf{W}_1 \mathbf{h}_i + \mathbf{W}_2 \mathbf{s}_{t-1})$$

Here \mathbf{v} is a weight vector.

The alignment model is parameterized as a feedforward neural network and jointly trained with the remaining system components. Subsequently, a softmax function is applied to each attention score to obtain the corresponding weight value:

$$\alpha_{t,i} = \text{softmax}(e_{t,i})$$

The application of the softmax function essentially normalizes the annotation values to a range between 0 and 1; hence, the resulting weights can be considered probability values. Each probability (or weight) value reflects how important \mathbf{h}_i and \mathbf{s}_{t-1} are in generating the next state \mathbf{s}_t and the next output \mathbf{y}_t .

“Intuitively, this implements a

to encode all information in the source sentence into a fixed-length vector.

– “Neural Machine Translation by Jointly Learning to Align and Translate”

This is finally followed by the c

annotations:

$$\mathbf{c} = \sum_{i=1}^T \alpha_{t,i} \mathbf{h}_i$$

The Bahdanau Attention Algorithm

In summary, the attention algorithm proposed by Bahdanau et al. performs the following operations:

1. The encoder generates a set of annotations \mathbf{h}_i from the input sentence.
2. These annotations are fed to an alignment model and the previous hidden decoder state. The alignment model uses this information to generate the attention scores $e_{t,i}$.
3. A softmax function is applied to the attention scores, effectively normalizing them into weight values $\alpha_{t,i}$ in a range between 0 and 1.
4. Together with the previously computed annotations, the weights are used to generate a context vector \mathbf{c}_t through a weighted sum of the annotations.
5. The context vector is fed to the decoder together with the previous hidden decoder state and the previous output to compute the final output \mathbf{y}_t .
6. Steps 2–6 are repeated until the end of the sequence.

Bahdanau et al. (2015) tested their architecture on the task of English-to-French translation. They reported that their model significantly outperformed the conventional encoder-decoder model, regardless of the sentence length.

There have been several improvements over the Bahdanau attention proposed, such as those of Luong et al. (2015), which we shall review in Chapter 5.

4.3 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Papers

Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. “Sequence to Sequence Learning with Neural Networks”. In: *Proc. NIPS*. Vol. 2. Dec. 2014, pp. 3104–3112.

<https://arxiv.org/abs/1409.3215>

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *Proc. 3rd International Conference on Learning Representations (ICLR 2015)*. May 2015.

<http://arxiv.org/abs/1409.0473>

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. In: *Proc. EMNLP*. Sept. 2015, pp. 1412–1421. DOI:

[10.18653/v1/D15-1166](https://doi.org/10.18653/v1/D15-1166)

<https://arxiv.org/abs/1508.04025>

4.4 Summary

In this chapter, you discovered the Bahdanau attention mechanism for neural machine translation. Specifically, you learned:

- ◁ Where the Bahdanau attention derives its name from and the challenge it addresses
- ◁ The role of the different components that form part of the Bahdanau encoder-decoder architecture
- ◁ The operations performed by the Bahdanau attention algorithm

In the next chapter, you will learn another attention mechanism.

The Luong Attention Mechanism

The Luong attention sought to introduce several improvements over the Bahdanau model for neural machine translation, notably by introducing two new classes of attentional mechanisms: a *global* approach that attends to all source words and a *local* approach that only attends to a selected subset of words in predicting the target sentence.

In this chapter, you will discover the Luong attention mechanism for neural machine translation. After completing this chapter, you will know:

- ◄ The operations performed by the Luong attention algorithm
- ◄ How the global and local attentional models work
- ◄ How the Luong attention compares to the Bahdanau attention

Let's get started.

Overview

This chapter is divided into five parts; they are:

- ◄ Introduction to the Luong Attention
- ◄ The Luong Attention Algorithm
- ◄ The Global Attentional Model
- ◄ The Local Attentional Model
 - ◄ Comparison to the Bahdanau Attention

5.1 Introduction to the Luong Attention

Luong et al. (2015) inspire themselves from previous attention models to propose two attention mechanisms:

“In this work, we design, with simplicity, an effective approach to attention-based neural machine translation.”

The *global* attentional model

source words but aims to simplify it architecturally. The *local* attentional model is inspired by the hard and soft attention models of Xu et al. (2015) and attends to *only a few* of the source positions. The two attentional models share many of the steps in their prediction of the current word but differ mainly in their computation of the context vector. Let's first take a look at the overarching Luong attention algorithm and then delve into the differences between the global and local attentional models afterward.

5.2 The Luong Attention Algorithm

The attention algorithm of Luong et al. performs the following operations:

1. The encoder generates a set of annotations $H = \{\mathbf{h}_i : i = 1, \dots, T\}$ from the input sentence.
2. The current decoder hidden state is computed as: $\mathbf{s}_t = \text{RNND}(\mathbf{s}_{t-1}, \mathbf{y}_{t-1})$. Here, \mathbf{s}_{t-1} denotes the previous hidden decoder state and \mathbf{y}_{t-1} the previous decoder output.
3. An alignment model $a(\cdot)$, uses the annotations and the current decoder hidden state to compute the alignment scores: $e_{t,i} = a(\mathbf{s}_t, \mathbf{h}_i)$.
4. A softmax function is applied to the alignment scores, effectively normalizing them into weight values in a range between 0 and 1: $\alpha_{t,i} = \text{softmax}(e_{t,i})$.
5. Together with the previously computed annotations, these weights are used to generate a context vector through a weighted sum of the annotations: $\mathbf{c}_t = \sum_{i=1}^T \alpha_{t,i} \mathbf{h}_i$.
6. An attentional hidden state is computed based on a weighted concatenation of the context vector and the current decoder hidden state: $\tilde{\mathbf{s}}_t = \tanh(\mathbf{W}_c[\mathbf{c}_t; \mathbf{s}_t])$.
7. The decoder produces a final output by feeding it a weighted attentional hidden state: $\mathbf{y}_t = \text{softmax}(\mathbf{W}_y \tilde{\mathbf{s}}_t)$.
8. Steps 2–7 are repeated until the end of the sequence.

5.3 The Global Attentional Model

The global attentional model considers all the source words in the input sentence when generating the alignment scores and, eventually, when computing the context vector.

“The idea of a global attentional model is to consider all the

hidden states of the encoder

hidden states of the encoder

and \mathbf{h}_i , while the second and third approaches implement *multiplicative* attention (in contrast to Bahdanau's *additive* attention):

$$1. a(\mathbf{st}, \mathbf{h}_i) = \text{vatanh}(\mathbf{W}_a[\mathbf{st}; \mathbf{h}_i])$$

$$2. a(\mathbf{st}, \mathbf{h}_i) = \mathbf{st} \mathbf{h}_i$$

$$3. a(\mathbf{st}, \mathbf{h}_i) = \mathbf{st} \mathbf{W}_a \mathbf{h}_i$$

Here, \mathbf{W}_a is a trainable weight matrix, and similarly, \mathbf{v}_a is a weight vector.

Intuitively, the use of the dot product in *multiplicative* attention can be interpreted as providing a similarity measure between the vectors \mathbf{st} and \mathbf{h}_i under consideration.

“

...

if the vectors are similar (that

5.4 The Local Attentional Model

In attending to all source words, the global attentional model is computationally expensive and could potentially become impractical for translating longer sentences.

The local attentional model seeks to address these limitations by focusing on a smaller subset of the source words to generate each target word. In order to do so, it takes inspiration from the *hard* and *soft* attention models of the image caption generation work of Xu et al. (2015):

• *Soft* attention is equivalent to the global attention approach, where weights are softly placed over all the source image patches. Hence, soft attention considers the source image in its entirety.

• *Hard* attention attends to a single image patch at a time.

The local attentional model of Luong et al. generates a context vector by computing a weighted average over the set of annotations \mathbf{h}_i within a window centered over an aligned position p_t :

$[p_t$

$-D, p_t + D]$

While a value for D is selected empirically, Luong et al. consider two approaches in computing a value for p_t :

1. *Monotonic alignment*: where the source and target sentences are assumed to be monotonically aligned and hence, $p_t = t$.

2. *Predictive alignment*: where a prediction of the aligned position is based upon trainable model parameters \mathbf{w}_p and \mathbf{v}_p , and the source sentence length S :

$p_t = S$

$\cdot \text{sigmoid}(\mathbf{w}_p \mathbf{p} + \mathbf{h}_p \mathbf{v}_p, \mathbf{st})$

A Gaussian distribution is centered around p_t when computing the alignment weights to favor source words nearer to the window center.

hPython, 2019

This time round, the resulting alignment vector a_t has a fixed length of $2D + 1$.

5.5 Comparison to the Bahdanau Attention

The Bahdanau model and the global attention approach of Luong et al. are mostly similar, but there are key differences between the two:

“While our global attention approach is similar in spirit to the model proposed by Bahdanau et al. (2015), there are several key differences which reflect the

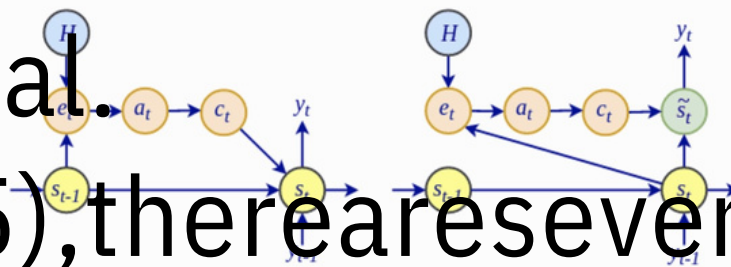


Figure 5.1: The Bahdanau architecture (left) vs. the Luong architecture (right). From *Advanced Deep Learning with Python*

Luong et al. drop the bidirectional encoder used by the Bahdanau model and instead utilize the hidden states at the top LSTM layers for both the encoder and decoder. The global attentional model of Luong et al. investigates the use of multiplicative attention as an alternative to the Bahdanau additive attention.

generalized from the original model.

5.6 Further Reading

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

Books

– “Effective Approaches to Attention-based Neural Machine Translation”

Ivan Vasilyev, *Advanced Deep Learning with Python*, Packt Publishing, 2019.

<https://www.mazhuc.com/cp/7193511x/>

Most notably, the computation of the alignment scores in the Luong global attention model depends on the current decoder hidden state s_t rather than on the previous hidden state s_{t-1} as in the Bahdanau attention. By severing the dependency to previous time steps makes the computation simpler.

Papers

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. In: *Proc. EMNLP*. Sept. 2015, pp. 1412–1421. DOI: [10.18653/v1/D15-1166](https://doi.org/10.18653/v1/D15-1166).

<https://arxiv.org/abs/1508.04025>

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *Proc. 3rd International Conference on Learning Representations (ICLR 2015)*. May 2015.

<http://arxiv.org/abs/1409.0473>

Kelvin Xu et al. “Show, Attend and Tell: Neural Image Caption Generation with Visual Attention”. In: *Proceedings of the 32nd International Conference on Machine Learning*. Vol. 37. July 2015, pp. 2048–2057.

<https://arxiv.org/abs/1502.03044>

5.7 Summary

In this chapter, you discovered the Luong attention mechanism for neural machine translation. Specifically, you learned:

- ◁ The operations performed by the Luong attention algorithm
- ◁ How the global and local attentional models work
- ◁ How the Luong attention compares to the Bahdanau attention

Start from the next chapter, you will learn how attention can learn a sequence better than a recurrent neural network.

From Recurrent Neural Networks to Transformer

An Introduction to Recurrent Neural Networks

When it comes to sequential or time series data, traditional feedforward networks cannot be used for learning and prediction. A mechanism is required that can retain past or historical information to forecast future values. Recurrent neural networks, or RNNs for short, are a variant of the conventional feedforward artificial neural networks that can deal with sequential data and can be trained to hold knowledge about the past.

After completing this chapter, you will know:

- ◄ Recurrent neural networks
- ◄ What is meant by unfolding an RNN
- ◄ How weights are updated in an RNN
- ◄ Various RNN architectures

Let's get started.

Overview

This chapter is divided into five parts; they are:

- ◄ What Is a Recurrent Neural Network
 - ◄ Unfolding a Recurrent Neural Network
- ◄ Training a Recurrent Neural Network
- ◄ Types of RNNs
- ◄ Different RNN Architectures

6.1 WhatIsaRecurrentNeuralNetwork

A recurrent neural network (RNN) is a special type of artificial neural network adapted to work for time series data or data that involves sequences. Ordinary feedforward neural networks are only meant for data points that are independent of each other. However, if we have data in a sequence such that one data point depends upon the previous data point, we need to modify the neural network to incorporate the dependencies between these data points. RNNs

have the concept of *memory* that helps them store the states or information of previous inputs to generate the next output of the sequence.

RNNs have various advantages, such as:

- ◁ Ability to handle sequence data.
- ◁ Ability to handle inputs of varying lengths.
- ◁ Ability to store or 'memorize' historical information.

But their disadvantages are:

- ◁ The computation can be very slow.
- ◁ The network does not take into account future inputs to make decisions.
- ◁ Vanishing gradient problem, where the gradients used to compute the weight update may get very close to zero, preventing the network from learning new weights. The deeper the network, the more pronounced this problem is.

6.2 Unfolding a Recurrent Neural Network

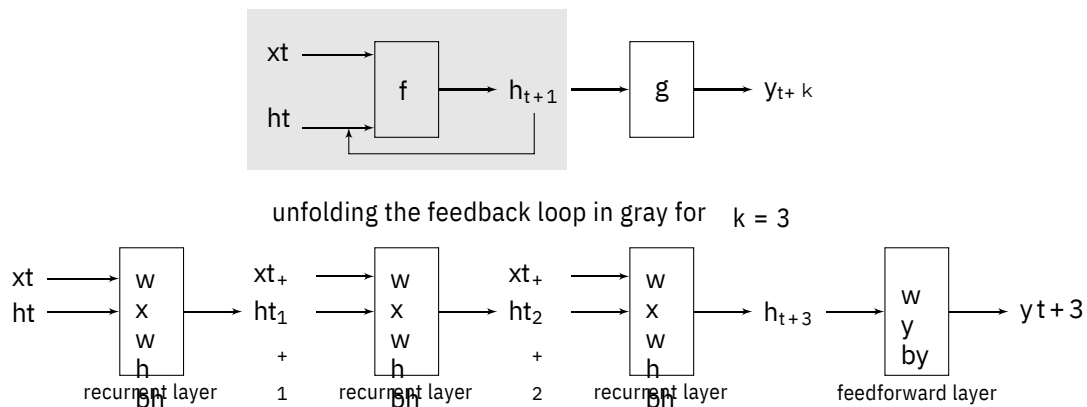


Figure 6.1: Recurrent neural network. Compressed representation (top), unfolded network (bottom).

A simple RNN has a feedback loop, as shown in the first diagram of the above figure. The feedback loop shown in the gray rectangle can be unrolled in three time steps to produce the second network of the above figure. Of course, you can vary the architecture so that the network unrolls k time steps. In the figure, the following notation is used:

◁ x

$\in \mathbb{R}$ is the input at time step t . To keep things simple, we assume that x_t is a scalar value with a single feature. You can extend this idea to a d -dimensional feature vector.

◁ y

$\in \mathbb{R}$ is the output of the network at time step t . We can produce multiple outputs in the network, but for this example, we assume that there is one output.

h
 $\in \mathbb{R}^m$ is a vector that stores the values of the hidden units/states at time t . This is also called the current context. m is the number of hidden units. h_0 vector is initialized to zero.

$\in m \times n$ Rare weights associated with inputs in the recurrent layer

$\in n \times n$

$\in m \times m$ Rare weights associated with hidden units in the recurrent layer

$\in n \times n$

$\in m \times n$ Rare weights associated with hidden units to output units

$\in n$

$\in m$ Bias associated with the recurrent layer

$\in n$

$\in R$ is the bias associated with the feedforward layer

At every time step, we can unfold the network for k time steps to get the output at time step $k + 1$. The unfolded network is very similar to the feedforward neural network. The rectangle in the unfolded network shows an operation taking place. So, for example, with an activation function f :

$$h_{t+1} = f(x_t, h_t, w_x, w_h, b_h) = f(w_x x_t + w_h h_t + b_h)$$

The output y at time t is computed as:

$$y_t = f(h_t, w_y) = f(w_y$$

$$\cdot h_t + b_y)$$

Here,

\cdot is the dot product operator.

Hence, in the feedforward pass of an RNN, the network computes the values of the hidden units and the output after k time steps. The weights associated with the network are shared temporally. Each recurrent layer has two sets of weights: one for the input and the second for the hidden unit. The last feedforward layer, which computes the final output for the k th time step, is just like an ordinary layer of a traditional feedforward network.

We can use any activation function we like in the recurrent neural network. Common choices are:

Sigmoid function: $1 > \sigma(x) =$

6.3 Training a Recurrent Neural Network

ex

The backpropagation algorithm of an artificial neural network is modified to include the unfolding in time to train the weights of the network. This algorithm is based on computing the gradient vector and is called backpropagation in time or BPTT algorithm for short. The pseudo-code for training is given below. The value of k can be selected

by the user for training. In the pseudo-code below, p_t is the target value at time step t :

```

repeat
  Set all  $h$  to zero;
  for  $t=0$  to  $n-k$  do
    Forward propagate the network over the unfolded network for  $k$  time steps to
    compute all  $h$  and  $y$ ;
    Compute the error as  $e = y_{t+k} - p_{t+k}$ ;
  Backpropagate the error across the unfolded network and update the weights;
end
until stopping criterion is met;
```

6.4 Types of RNNs

There are different types of recurrent neural networks with varying architectures. Some examples are:

One-to-One Here, there is a single (x_t, y_t) pair. Traditional neural networks employ a one-to-one architecture.

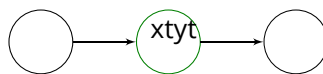


Figure 6.2: One-to-one architecture

One-to-Many
e.g., (y_{t0}, y_{t1}, y_{t2}) employed.

In one-to-many networks, a single input at x_t can produce multiple outputs, (y_{t0}, y_{t1}, y_{t2}) . Music generation is an example area where one-to-many networks are employed.

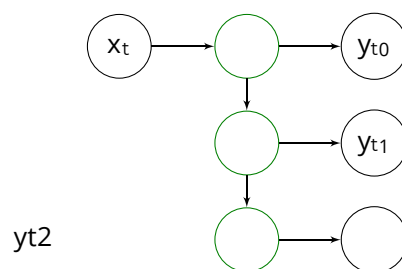


Figure 6.3: One-to-many architecture

Many-to-One In this case, many inputs from different time steps produce a single output. For example, (x_t, x_{t+1}, x_{t+2}) can produce a single output y_t . Such networks are employed in sentiment analysis or emotion detection, where the class label depends upon a sequence of words.

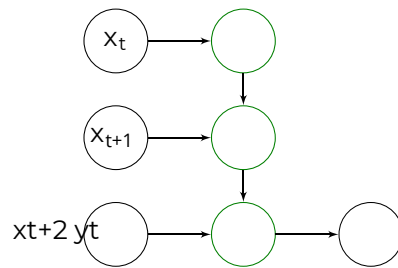


Figure 6.4: Many-to-one architecture

Many-to-Many There are many possibilities for many-to-many. An example is shown above, where two inputs produce three outputs. Many-to-many networks are applied in machine translation, e.g., English to French or vice versa translation systems.

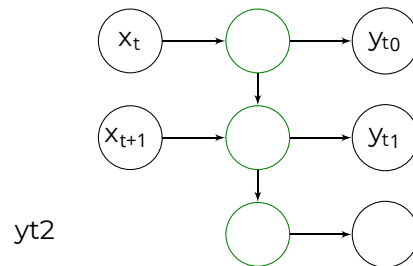


Figure 6.5: Many-to-many architecture

6.5 Different RNN Architectures

There are different variations of RNNs that are being applied practically in machine learning problems:

Bidirectional Recurrent Neural Networks (BRNN) In BRNN, inputs from future time steps are used to improve the accuracy of the network. It is like knowing the first and last words of a sentence to predict the middle words.

Gated Recurrent Units (GRU) These networks are designed to handle the vanishing gradient problem. They have a reset and update gate. These gates determine which information is to be retained for future predictions.

Long Short Term Memory (LSTM) LSTMs were also designed to address the vanishing gradient problem in RNNs. LSTMs use three gates called input, output, and forget gate. Similar to GRU, these gates determine which information to retain.

6.6 Further Reading

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

Books

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

<https://www.amazon.com/dp/0262035618>

(Online version at <http://www.deeplearningbook.org>).

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing, 2018.

<https://www.amazon.com/dp/1785880365>

Articles

Backpropagation through time. Wikipedia.

https://en.wikipedia.org/wiki/Backpropagation_through_time

6.7 Summary

In this chapter, you discovered recurrent neural networks and their various architectures. Specifically, you learned:

- ◁ How a recurrent neural network handles sequential data
- ◁ Unfolding in time in a recurrent neural network
- ◁ What is backpropagation in time
- ◁ Advantages and disadvantages of RNNs
- ◁ Various architectures and variants of RNNs

In the next chapter, we will see how Keras implements a recurrent neural network.

Understanding Simple Recurrent Neural Networks in Keras

This chapter is designed for anyone looking for an understanding of how recurrent neural networks (RNN) work and how to use them via the Keras deep learning library. While the Keras library provides all the methods required for solving problems and building applications, it is also important to gain an insight into how everything works.

In this chapter, the computations taking place in the RNN model are shown step-by-step. Next, a complete end-to-end system for time series prediction is developed. After completing this chapter, you will know:

- ◀ The structure of an RNN
 - ◀ How an RNN computes the output when given an input
 - ◀ How to prepare data for a SimpleRNN in Keras
 - ◀ How to train a SimpleRNN model
- Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◀ Keras SimpleRNN Layer
- ◀ Running the RNN on Sunspots Dataset
- ◀ Consolidated Code

It is assumed that you have a basic understanding of RNNs before you start implementing them. Chapter 6 gives you a quick overview of RNNs. Let's now get right down to the implementation part.

7.1 KerasSimpleRNNLayer

To start the implementation of RNNs, let's add the import section.

```
import math

from pandas import read_csv
import numpy as np
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, SimpleRNN
from sklearn.preprocessing import MinMaxScaler
from sklearn.metrics import mean_squared_error
import matplotlib.pyplot as plt
```

Listing 7.1: Libraries needed

The function below returns a model that includes a SimpleRNN layer and a Dense layer for learning sequential data. The input_shape specifies the parameter (time_steps, features). We'll simplify everything and use univariate data, i.e., one feature only; the time_steps are discussed below.

```
def create_RNN(hidden_units, dense_units, input_shape, activation):
    model = Sequential()
    model.add(SimpleRNN(hidden_units, input_shape=input_shape,
                        activation=activation[0]))
    model.add(Dense(units=dense_units, activation=activation[1]))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model

demo_model = create_RNN(2, 1, (3,1), activation=['linear', 'linear'])
```

Listing 7.2: Create a simple RNN model

The object demo_model is returned with two hidden units created via the SimpleRNN layer and one dense unit created via the Dense layer. The input_shape is set at 3 × 1, and a linear activation function is used in both layers for simplicity. Just to recall, the linear activation function $f(x) = x$ makes no change in the input. If we have m hidden units ($m = 2$ in the code above), then the network looks as follows:

◁ Input: x

$\in \mathbb{R}$

◁ Hidden unit: h

$\in \mathbb{R}^m$

◁ Weights for the input units: w

$\in m \times \mathbb{R}$

$\in m \times m$ ◁ Weights for the hidden units: wh

$\in m$ ◁ Bias for hidden units: bh

eight for the dense layer:

$\in m$ ◁ W_{wy}

◁ Bias for the dense layer: b_y

$\in \mathbb{R}$ ◁ b_y

$w_y = \text{demo_model.get_weights()[3]}$
 $b_y = \text{demo_model.get_weights()[4]}$

Let's look at the above weights.

```
print('wx = ', wx, ' wh = ', wh, ' bh = ', bh, ' wy = ', wy, ' by = ', by)
```

Listing 7.3: Show the weights

```
wx = [[ 0.18662322 -1.2369459 ]] wh = [[ 0.86981213 -0.49338293] [0.
 [ 0.49338293 0.8698122 ]] bh = [ 0.] wy = [[-0.4635998]
 [ 0.6538409]] by = [0.]
```

Output 7.1: Random weights from initialization

INFO-CIRCLE Note: As the weights are randomly initialized, the results showed here will be different from yours. The important thing to remember is that the object being used looks like and how it interacts with others to produce the final output.

Now let's do a simple experiment to see how the layers from a SimpleRNN and Dense layer produce an output. Keep this figure in view.

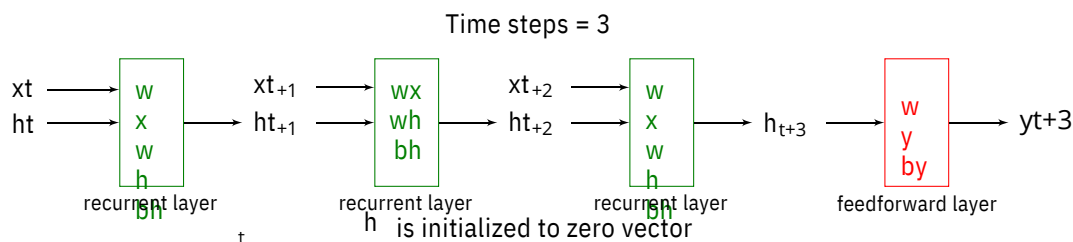


Figure 7.1: Layers of a recurrent neural network

We'll input x for three time steps and let the network generate an output. The values of the hidden units at time steps 1, 2, and 3 will be computed. h_0 is initialized to the zero vector. The output o_3 is computed from h_3 and wy . An activation function is not required as we are using linear units.

```
x = np.array([1, 2, 3])
# Reshape the input to the required sample_size x time_steps x features
x_input = np.reshape(x,(1, 3, 1))
y_pred_model = demo_model.predict(x_input)

m=2
h0 = np.zeros(m)
h1 = np.dot(x[0], wx) + h0 + bh
h2 = np.dot(x[1], wx) + np.dot(h1,wh) + bh
h3 = np.dot(x[2], wx) + np.dot(h2,wh) + bh
o3 = np.dot(h3, wy) + by

print('h1 =', h1,'h2 =', h2,'h3 =', h3)
```

```
print("Prediction from network ", y_pred_model)
print("Prediction from our computation ", o3)
```

x Listing 7.4: Input for three time steps to generate an output

```
h1 = [[ 0.18662322 -1.23694587]] h2 = [[-0.07471441 -3.64187904]] h3 = [[-1.30195881 -6.84172557]] Prediction from
network [[-3.8698118]]
Prediction from our computation [[-3.86981216]]
```

Output 7.2: Output from three time steps

7.2 Running the RNN on Sunspots Dataset

Now that we understand how the `SimpleRNN` and `Dense` layers are put together. Let's run a complete RNN on a simple time series dataset. We'll need to follow these steps:

1. Read the dataset from a given URL
2. Split the data into training and test set
3. Prepare the input to the required Keras format
4. Create an RNN model and train it
5. Make the predictions on training and test sets and print the root mean square error on both sets
6. View the result

Step 1, 2: Reading Data and Splitting into Training and Test

The following function reads the train and test data from a given URL and splits it into a given percentage of train and test data. It returns single-dimensional arrays for train and test data after scaling the data between 0 and 1 using `MinMaxScaler` from `scikit-learn`.

```
# Parameter split_percent defines the ratio of training examples
def get_train_test(url, split_percent=0.8):
    df = read_csv(url, usecols=[1], engine='python')
    data = np.array(df.values.astype('float32'))
    scaler = MinMaxScaler(feature_range=(0, 1))
    data = scaler.fit_transform(data).flatten()
    n = len(data)
    # Point for splitting data into train and test
    split = int(n*split_percent)
    train_data = data[:split]
    test_data = data[split:]
    return train_data, test_data, data

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-sunspots.csv"
train_data, test_data, data = get_train_test(url)
```

Listing 7.5: Get data and split into training and test sets

Step 3: Reshaping Data For Keras

The next step is to prepare the data for Keras model training. The input array should be shaped as: (total_samples, time_steps, features).

There are many ways of preparing time series data for training. We'll create input rows with non-overlapping time steps. An example for time_steps=2 is shown in the figure below. Here, time_steps denotes the number of previous time steps to use for predicting the next value of the time series data.

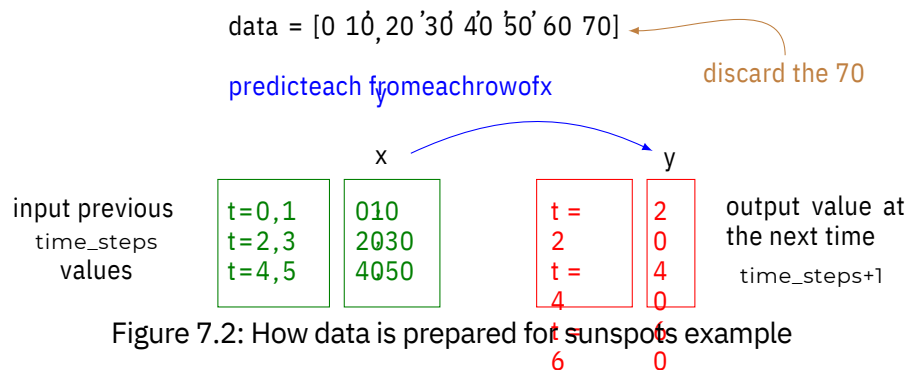


Figure 7.2: How data is prepared for sunspots example

The following function `get_XY()` takes a one-dimensional array as input and converts it to the required input `X` and target `Y` arrays. We'll use `time_steps=12` for the sunspots dataset as the sunspots generally have a cycle of 12 months. You can experiment with other values of `time_steps`.

```
# Prepare the input X and target Y
def get_XY(dat, time_steps):
    # Indices of target array
    Y_ind = np.arange(time_steps, len(dat), time_steps)
    Y = dat[Y_ind]
    # Prepare X
    rows_x = len(Y)
    X = dat[range(time_steps*rows_x)]
    X = np.reshape(X, (rows_x, time_steps, 1))
    return X, Y

time_steps = 12
trainX, trainY = get_XY(train_data, time_steps)
testX, testY = get_XY(test_data, time_steps)
```

Listing 7.6: Create input and target arrays

Step 4: Create RNN Model and Train

For this step, you can reuse your `create_RNN()` function that was defined above.

```
model = create_RNN(hidden_units=3, dense_units=1, input_shape=(time_steps,1),
    activation=['tanh', 'tanh'])
model.fit(trainX, trainY, epochs=20, batch_size=1, verbose=2)
```

Listing 7.7: Create RNN model and train

Step 5: Compute and Print the Root Mean Square Error

The function `print_error()` computes the mean square error between the actual and predicted values.

```
def print_error(trainY, testY, train_predict, test_predict):
    # Error of predictions
    train_rmse = math.sqrt(mean_squared_error(trainY, train_predict))
    test_rmse = math.sqrt(mean_squared_error(testY, test_predict))
    # Print RMSE
    print('Train RMSE: %.3f RMSE' % (train_rmse))
    print('Test RMSE: %.3f RMSE' % (test_rmse))

    # make predictions
    train_predict = model.predict(trainX)
    test_predict = model.predict(testX)
    # Mean square error
    print_error(trainY, testY, train_predict, test_predict)
```

Listing 7.8: Calculate mean square error

```
Train RMSE: 0.058 RMSE
Test RMSE: 0.077 RMSE
```

Output 7.3: Mean square error as calculated

Step 6: View the Result

The following function plots the actual target values and the predicted values. The red line separates the training and test data points.

```
# Plot the result
def plot_result(trainY, testY, train_predict, test_predict):
    actual = np.append(trainY, testY)
    predictions = np.append(train_predict, test_predict)
    rows = len(actual)
    plt.figure(figsize=(15, 6), dpi=80)
    plt.plot(range(rows), actual)
    plt.plot(range(rows), predictions)
    plt.axvline(x=len(trainY), color='r')
    plt.legend(['Actual', 'Predictions'])
    plt.xlabel('Observation number after given time steps')
    plt.ylabel('Sunspots scaled')
    plt.title('Actual and Predicted Values')
    plot_result(trainY, testY, train_predict, test_predict)
```

Listing 7.9: Visualizing the model output

The following plot is generated:

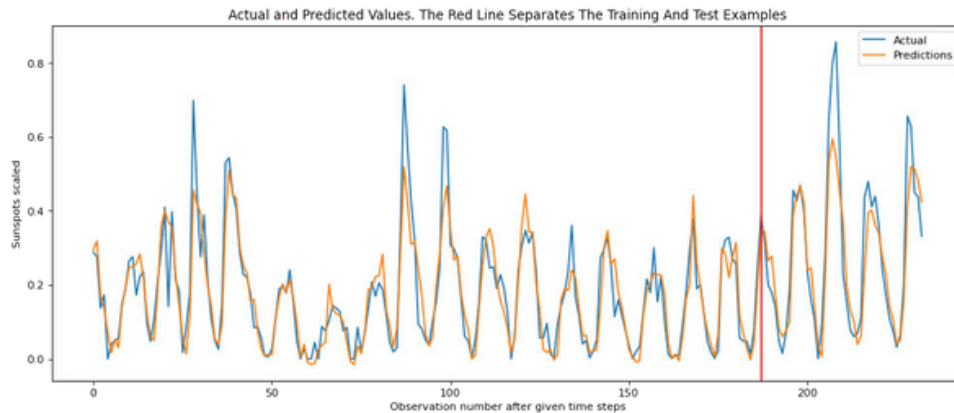


Figure 7.3: Model output as visualized

7.3 Consolidated Code

Given below is the entire code for this chapter. Try this out at your end and experiment with different hidden units and time steps. You can add a second SimpleRNN to the network and see how it behaves. You can also use the scaler object to rescale the data back to its normal range.

```
# Parameter split_percent defines the ratio of training examples
def get_train_test(url, split_percent=0.8):
    df = read_csv(url, usecols=[1], engine='python')
    data = np.array(df.values.astype('float32'))
    scaler = MinMaxScaler(feature_range=(0, 1))
    data = scaler.fit_transform(data).flatten()
    n = len(data)
    # Point for splitting data into train and test
    split = int(n*split_percent)
    train_data = data[range(split)]
    test_data = data[split:]
    return train_data, test_data, data

# Prepare the input X and target Y
def get_XY(dat, time_steps):
    Y_ind = np.arange(time_steps, len(dat), time_steps)
    Y = dat[Y_ind]
    rows_x = len(Y)
    X = dat[range(time_steps*rows_x)]
    X = np.reshape(X, (rows_x, time_steps, 1))
    return X, Y

def create_RNN(hidden_units, dense_units, input_shape, activation):
    model = Sequential()
    model.add(SimpleRNN(hidden_units, input_shape=input_shape, activation=activation[0]))
    model.add(Dense(units=dense_units, activation=activation[1]))
    model.compile(loss='mean_squared_error', optimizer='adam')
    return model
```



```

def print_error(trainY, testY, train_predict, test_predict):
    # Error of predictions
    train_rmse = math.sqrt(mean_squared_error(trainY, train_predict))
    test_rmse = math.sqrt(mean_squared_error(testY, test_predict))
    # Print RMSE
    print('Train RMSE: %.3f RMSE' % (train_rmse))
    print('Test RMSE: %.3f RMSE' % (test_rmse))

# Plot the result
def plot_result(trainY, testY, train_predict, test_predict):
    actual = np.append(trainY, testY)
    predictions = np.append(train_predict, test_predict)
    rows = len(actual)
    plt.figure(figsize=(15, 6), dpi=80)
    plt.plot(range(rows), actual)
    plt.plot(range(rows), predictions)
    plt.axvline(x=len(trainY), color='r')
    plt.legend(['Actual', 'Predictions'])
    plt.xlabel('Observation number after given time steps')
    plt.ylabel('Sunspots scaled')
    plt.title('Actual and Predicted Values')

url = "https://raw.githubusercontent.com/jbrownlee/Datasets/master/monthly-sunspots.csv"
time_steps = 12
train_data, test_data, data = get_train_test(url)
trainX, trainY = get_XY(train_data, time_steps)
testX, testY = get_XY(test_data, time_steps)

# Create model and train
model = create_RNN(hidden_units=3, dense_units=1, input_shape=(time_steps,1),
    activation=['tanh', 'tanh'])
model.fit(trainX, trainY, epochs=20, batch_size=1, verbose=2)

# make predictions
train_predict = model.predict(trainX)
test_predict = model.predict(testX)

# Print error
print_error(trainY, testY, train_predict, test_predict)

#Plot result
plot_result(trainY, testY, train_predict, test_predict)

```

Listing 7.10: A complete RNN model

7.4 Further Reading

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

Books

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

<https://www.amazon.com/dp/0262035618>

(Online version at <http://www.deeplearningbook.org>).

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing, 2018.

<https://www.amazon.com/dp/1785880365>

Articles

Backpropagation through time. Wikipedia.

https://en.wikipedia.org/wiki/Backpropagation_through_time

7.5 Summary

In this chapter, you discovered recurrent neural networks and their various architectures. Specifically, you learned:

- ◁ The structure of RNNs
- ◁ How the RNN computes an output from previous inputs
- ◁ How to implement an end-to-end system for time series forecasting using an RNN

RNN is natural for a sequence but has limited power. In the next chapter, we will see how attention can make an improvement.

The Attention Mechanism from Scratch

The attention mechanism was introduced to improve the performance of the encoder-decoder model for machine translation. The idea behind the attention mechanism was to permit the decoder to utilize the most relevant parts of the input sequence in a flexible manner, by a weighted combination of all the encoded input vectors, with the most relevant vectors being attributed the highest weights.

In this chapter, you will discover the attention mechanism and its implementation. After completing this chapter, you will know:

- ◁ How the attention mechanism uses a weighted sum of all the encoder hidden states to flexibly focus the attention of the decoder on the most relevant parts of the input sequence

- ◁ How the attention mechanism can be generalized for tasks where the information may not necessarily be related in a sequential fashion

- ◁ How to implement the general attention mechanism in Python with NumPy and SciPy

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◁ The Attention Mechanism

- ◁ The General Attention Mechanism

- ◁ The General Attention Mechanism with NumPy and SciPy

8.1 The Attention Mechanism

The attention mechanism was introduced by Bahdanau et al. (2015), to address the bottleneck problem that arises with the use of a fixed-length encoding vector, where the decoder would have limited access to the information provided by the input. This is thought to become especially problematic for long and/or complex sequences, where the dimensionality of their representation would be forced to be the same as for shorter or simpler sequences.

Note that Bahdanau et al.'s *attention mechanism* is divided into the step-by-step computations of the *alignment scores*, the *weights* and the *context vector*:

◁ Alignment scores: The alignment model takes the encoded hidden states \mathbf{h}_i and the previous decoder output \mathbf{s}_{t-1} to compute a score $e_{t,i}$ that indicates how well the elements of the input sequence align with the current output at position t . The alignment model is represented by a function $a(\cdot)$, which can be implemented by a feedforward neural network:

$$e_{t,i} = a(\mathbf{s}_{t-1}, \mathbf{h}_i)$$

◁ Weights: The weights $\alpha_{t,i}$ are computed by applying a softmax operation to the previously computed alignment scores:

$$\alpha_{t,i} = \text{softmax}(e_{t,i})$$

◁ Context vector: A unique context vector \mathbf{c}_t is fed into the decoder at each time step. $\mathbf{c}_t = \sum_i \alpha_{t,i} \mathbf{h}_i$
 \mathbf{c}_t is computed by a weighted sum of all encoder hidden states:

Bahdanau et al. implemented an RNN for both the encoder and decoder.

However, the attention mechanism can be re-formulated into a general form that can be applied to any sequence-to-sequence (abbreviated to seq2seq) task, where the information may not necessarily be related in a sequential fashion.

“ In other words, the database doesn't have to consist of the hidden RNN states at different steps, but could contain any kind of information instead.

— *Advanced Deep Learning with Python*, 2019

8.2

”

The General Attention Mechanism

The general attention mechanism makes use of three main components, namely the *queries* \mathbf{Q} , the *keys* \mathbf{K} , and the *values* \mathbf{V} .

If you had to compare these three components to the attention mechanism as proposed by Bahdanau et al., then the query would be analogous to the previous decoder output \mathbf{s}_{t-1} , while the values would be analogous to the encoded inputs \mathbf{h}_i . In the Bahdanau attention mechanism, the keys and values are the same vector.

In this case, we can think of the vector \mathbf{s}_{t-1} as a query executed against a database of key-value pairs, where the keys are vectors and the hidden states \mathbf{h}_i are the values.

— *Advanced Deep Learning with Python*, 2019

The general attention mechanism

1. Each query vector $\mathbf{q} = \mathbf{s}_{t-1}$ is matched against a database of keys to compute a score value. This matching operation is computed as the dot product of the specific query

under consideration with each key vector, k_i :

$$e_{ij} = \frac{q_i \cdot k_j}{\sum_k q_i \cdot k_k}$$

2. The scores are passed through a softmax operation to generate the weights:

$$\alpha_{q,k} = \text{softmax}(e_{q,k})$$

3. The generalized attention is then computed by a weighted sum of the value vectors v_k , where each value vector is paired with a corresponding key:

$$\text{attention}(q, K, V) = \sum_i \alpha_{q,k_i} v_i$$

Within the context of machine translation, each word in an input sentence would be attributed its own query, key and value vectors. These vectors are generated by multiplying the encoder's representation of the specific word under consideration with three different weight matrices that would have been generated during training.

In essence, when the generalized attention mechanism is presented with a sequence of words, it takes the query vector attributed to some specific word in the sequence and scores it against each key in the database. In doing so, it captures how the word under consideration relates to the others in the sequence. Then it scales the values according to the attention weights (computed from the scores) to retain focus on those words relevant to the query. In doing so, it produces an attention output for the word under consideration.

8.3 The General Attention Mechanism with NumPy and SciPy

This section will explore how to implement the general attention mechanism using the NumPy and SciPy libraries in Python.

For simplicity, you will initially calculate the attention for the first word in a sequence of four. You will then generalize the code to calculate an attention output for all four words in matrix form. Hence, let's start by first defining the word embeddings of the four different words to calculate the attention. In actual practice, these word embeddings would have been generated by an encoder; however, for this particular example, you will define them manually.

```
...
word_1 = array([1, 0, 0])
word_2 = array([0, 1, 0])
word_3 = array([1, 1, 0])
word_4 = array([0, 0, 1])
```

Listing 8.1: Encoder representations of four different words

The next step generates the weight matrices, which we you eventually multiply to the word embeddings to generate the queries, keys, and values. Here, you shall generate these weight matrices randomly; however, in actual practice, these would have been learned during training.

```
...
random.seed(42) # to allow us to reproduce the same attention values
W_Q = random.randint(3, size=(3, 3))
W_K = random.randint(3, size=(3, 3))
W_V = random.randint(3, size=(3, 3))
```

Listing 8.2: Generating the weight matrices

Notice how the number of rows of each of these matrices is equal to the dimensionality of the word embeddings (which in this case is three) to allow us to perform the matrix multiplication. Subsequently, the query, key, and value vectors for each word are generated by multiplying each word embedding by each of the weight matrices.

```
...
query_1 = word_1 @ W_Q
key_1 = word_1 @ W_K
value_1 = word_1 @ W_V

query_2 = word_2 @ W_Q
key_2 = word_2 @ W_K
value_2 = word_2 @ W_V

query_3 = word_3 @ W_Q
key_3 = word_3 @ W_K
value_3 = word_3 @ W_V

query_4 = word_4 @ W_Q
key_4 = word_4 @ W_K
value_4 = word_4 @ W_V
```

Listing 8.3: Generating the queries, keys and values

Considering only the first word for the time being, the next step scores its query vector against all the key vectors using a dot product operation.

```
...
scores = array([dot(query_1, key_1), dot(query_1, key_2), dot(query_1, key_3),
dot(query_1, key_4)])
```

Listing 8.4: Scoring the first query vector against all key vectors

The score values are subsequently passed through a softmax operation to generate the weights. Before doing so, it is common practice to divide the score values by the square root of the dimensionality of the key vectors (in this case, three) to keep the gradients stable.

```
...
weights = softmax(scores / key_1.shape[0] ** 0.5)
```

Listing 8.5: Computing the weights by a softmax operation

Finally, the attention output is calculated by a weighted sum of all four value vectors.

```
...
attention = (weights[0] * value_1) + (weights[1] * value_2) + (weights[2] * value_3) \
+ (weights[3] * value_4)

print(attention)
```

Listing 8.6: Computing the attention by a weighted sum of the value vectors

```
[0.98522025 1.74174051 0.75652026]
```

Output 8.1: Attention as calculated by a weighted sum of the value vectors

For faster processing, the same calculations can be implemented in matrix form to generate an attention output for all four words in one go:

```
from numpy import array
from numpy import random
from numpy import dot
from scipy.special import softmax

# encoder representations of four different words
word_1 = array([1, 0, 0])
word_2 = array([0, 1, 0])
word_3 = array([1, 1, 0])
word_4 = array([0, 0, 1])

# stacking the word embeddings into a single array
words = array([word_1, word_2, word_3, word_4])

# generating the weight matrices
random.seed(42)
W_Q = random.randint(3, size=(3, 3))
W_K = random.randint(3, size=(3, 3))
W_V = random.randint(3, size=(3, 3))

# generating the queries, keys and values
Q = words @ W_Q
K = words @ W_K
V = words @ W_V

# scoring the query vectors against all key vectors
scores = Q @ K.transpose()

# computing the weights by a softmax operation
weights = softmax(scores / K.shape[1] ** 0.5, axis=1)

# computing the attention by a weighted sum of the value vectors
attention = weights @ V

print(attention)
```

Listing 8.7: Attention calculation using matrix operations

```
[[0.98522025 1.74174051 0.75652026]
 [0.909652651 1.409652650.5 ]
 [0.99851226 1.75849334 0.75998108]
 [0.99560386 1.90407309 0.90846923]]
```

Output 8.2: Attention as calculated

8.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing, 2018.

<https://www.amazon.com/dp/1785880365>

Papers

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *Proc. 3rd International Conference on Learning Representations (ICLR 2015)*. May 2015.

<http://arxiv.org/abs/1409.0473>

8.5 Summary

In this chapter, you discovered the attention mechanism and its implementation.

Specifically, you learned:

- ◁ How the attention mechanism uses a weighted sum of all the encoder hidden states to flexibly focus the attention of the decoder to the most relevant parts of the input sequence

- ◁ How the attention mechanism can be generalized for tasks where the information may not necessarily be related in a sequential fashion

- ◁ How to implement the general attention mechanism with NumPy and SciPy

In the next chapter, you will see how to encapsulate the attention algorithm into a layer in Keras.

Adding a Custom Attention Layer to Recurrent Neural Network in Keras

Deep learning networks have gained immense popularity in the past few years. The *attention mechanism* is integrated with deep learning networks to improve their performance. Adding an attention component to the network has shown significant improvement in tasks such as machine translation, image recognition, text summarization, and similar applications.

This chapter shows how to add a custom attention layer to a network built using a recurrent neural network. We'll illustrate an end-to-end application of time series forecasting using a very simple dataset. The chapter is designed for anyone looking for a basic understanding of how to add user-defined layers to a deep learning network and use this simple example to build more complex applications. After completing this chapter, you will know:

- ◀ Which methods are required to create a custom attention layer in Keras

- ◀ How to incorporate the new layer in a network built with SimpleRNN

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ◀ Preparing Dataset for Time Series Forecasting

- ◀ The SimpleRNN Network

- ◀ Adding a Custom Attention Layer to the Network

- ◀ Consolidated Code

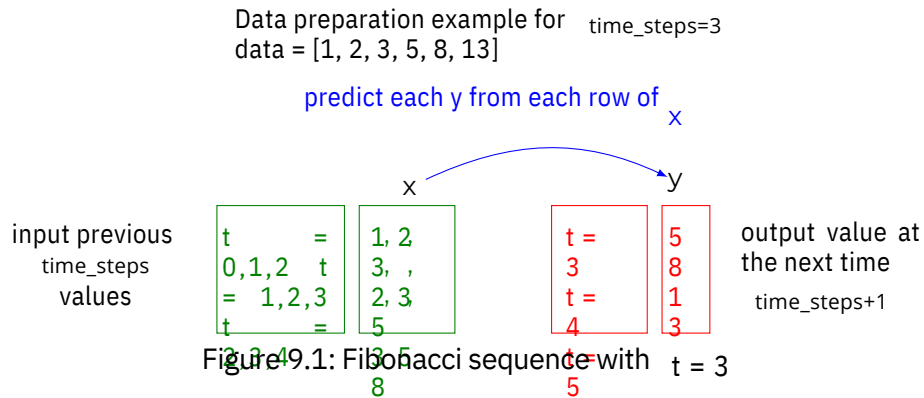
9.1 Preparing Dataset for Time Series Forecasting

The focus of this chapter is to gain a basic understanding of how to build a custom attention layer to a deep learning network. For this purpose, let's use a very simple example of a Fibonacci sequence, where one number is constructed from the previous two numbers. The first 10 numbers of the sequence are shown below:

' ' ' ' ' ' ' ' ' ' 0112358132134 ,...

When given the previous t numbers, can you get a machine to accurately reconstruct the next number? This would mean discarding all the previous inputs except the last two and performing the correct operation on the last two numbers.

For this chapter, you'll construct the training examples from t time steps and use the value at $t+1$ as the target. For example, if $t = 3$, then the training examples and the corresponding target values would look as follows:



9.2 The SimpleRNN Network

In this section, you'll write the basic code to generate the dataset and use a SimpleRNN network to predict the next number of the Fibonacci sequence. Let's first write the import section:

```
from pandas import read_csv
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense, SimpleRNN
from tensorflow.keras.layers import Layer
from tensorflow.keras.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
import tensorflow.keras.backend as K
import numpy as np
```

Listing 9.1: Libraries required

Preparing the Dataset

The following function generates a sequence of n Fibonacci numbers (not counting the starting two values). If `scale_data` is set to `True`, then it would also use the `MinMaxScaler` from `scikit-learn` to scale the values between 0 and 1. Let's see its output for $n = 10$.

```
def get_fib_seq(n, scale_data=True):
    # Get the Fibonacci sequence
    seq = np.zeros(n)
    fib_n1 = 0.0
    fib_n = 1.0
```

```

for i in range(n):
    seq[i] = fib_n1 + fib_n
    fib_n1 = fib_n
    fib_n = seq[i]
    scaler = []
    if scale_data:
        scaler = MinMaxScaler(feature_range=(0, 1))
    seq = np.reshape(seq, (n, 1))
    seq = scaler.fit_transform(seq).flatten()
    return seq, scaler

fib_seq, _ = get_fib_seq(10, False)
print(fib_seq)

```

Listing 9.2: Generating Fibonacci numbers

```
[1. 2. 3. 5. 8. 13. 21. 34. 55. 89.]
```

Output 9.1: First 10 Fibonacci numbers

Next, we need a function `get_fib_XY()` that reformats the sequence into input features and target values to be used by the Keras input layer. When given `time_steps` as a parameter, `get_fib_XY()` constructs each row of the dataset with `time_steps` number of columns. This function not only constructs the training set and test set from the Fibonacci sequence but also shuffles the training examples and reshapes them to the required TensorFlow format, i.e., (`total_samples`, `time_steps`, `features`). Also, the function returns the scaler object that scales the values if `scale_data` is set to `True`.

Let's generate a small training set to see what it looks like. We have set `time_steps=3` and `total_fib_numbers=12`, with approximately 70% of the examples going toward the test points. Note the training and test examples have been shuffled by the `permutation()` function.

```

def get_fib_XY(total_fib_numbers, time_steps, train_percent, scale_data=True):
    dat, scaler = get_fib_seq(total_fib_numbers, scale_data)
    Y_ind = np.arange(time_steps, len(dat), 1)
    Y = dat[Y_ind]
    rows_x = len(Y)
    X = dat[0:rows_x]
    for i in range(time_steps-1):
        temp = dat[i+1:rows_x+i+1]
        X = np.column_stack((X, temp))
    # random permutation with fixed seed
    rand = np.random.RandomState(seed=13)
    idx = rand.permutation(rows_x)
    split = int(train_percent*rows_x)
    train_ind = idx[0:split]
    test_ind = idx[split:]
    trainX = X[train_ind]
    trainY = Y[train_ind]
    testX = X[test_ind]
    testY = Y[test_ind]
    trainX = np.reshape(trainX, (len(trainX), time_steps, 1))
    testX = np.reshape(testX, (len(testX), time_steps, 1))

```

```

return trainX, trainY, testX, testY, scaler

trainX, trainY, testX, testY, scaler = get_fib_XY(12, 3, 0.7, False)
print('trainX = ', trainX)
print('trainY = ', trainY)

```

Listing 9.3: Splitting a sequence into training examples and target values

```

trainX = [[ 8.]
[13.]
[21.]]

[[ 5.]
[8.]
[13.]]

[[ 2.]
[3.]
[ 5.]]

[[13.]
[21.]
[34.]]

[[21.]
[34.]
[55.]]

[[34.]
[55.]
[89.]]]
trainY = [ 34. 21.      8.  55.  89. 144.]

```

Output 9.2: A few examples of input features and target values

Setting Up the Network

Now let's set up a small network with two layers. The first one is the the SimpleRNN layer, and second one is the Dense layer. Below is a summary of the model.

```

# Set up parameters
time_steps = 20
hidden_units = 2
epochs = 30

# Create a traditional RNN network
def create_RNN(hidden_units, dense_units, input_shape, activation):
    model = Sequential()
    model.add(SimpleRNN(hidden_units, input_shape=input_shape, activation=activation[0]))
    model.add(Dense(units=dense_units, activation=activation[1]))
    model.compile(loss='mse', optimizer='adam')
    return model

```

```
model_RNN = create_RNN(hidden_units=hidden_units, dense_units=1,
                        input_shape=(time_steps,1), activation=['tanh', 'tanh'])
model_RNN.summary()
```

Listing 9.4: Creating an RNN model

```
Model: "sequential_1"
-----
Layer(type) OutputShape Param#
-----
simple_rnn_3(SimpleRNN)(None,2) 8
-----
dense_3(Dense) (None,1) 3
-----
Total params: 11
Trainable params: 11
Non-trainable params: 0
```

Output 9.3: The RNN model

Train the Network and Evaluate

The next step is to add code that generates a dataset, trains the network, and evaluates it. This time around, we'll scale the data between 0 and 1. We don't need to pass the `scale_data` parameter as its default value is `True`.

```
# Generate the dataset
trainX, trainY, testX, testY, scaler = get_fib_XY(1200, time_steps, 0.7)

model_RNN.fit(trainX, trainY, epochs=epochs, batch_size=1, verbose=2)

# Evalute model
train_mse = model_RNN.evaluate(trainX, trainY)
test_mse = model_RNN.evaluate(testX, testY)

# Print error
print("Train set MSE = ", train_mse)
print("Test set MSE = ", test_mse)
```

Listing 9.5: Training and evaluating the model

As output, you'll see the progress of the training and the following values for the mean square error:

```
Train set MSE = 5.631405292660929e-
05 Test set MSE = 2.623497312015388e-
05
```

Output 9.4: Mean squared error as evaluated

9.3 Adding a Custom Attention Layer to the Network

In Keras, it is easy to create a custom layer that implements attention by subclassing the Layer class. The Keras guide lists clear steps for creating a new layer via subclassing. You'll use those guidelines here. All the weights and biases corresponding to a single layer are encapsulated by this class. You need to write the `__init__` method as well as override the following methods:

◀

`build()`: The Keras guide recommends adding weights in this method once the size of the inputs is known. This method “lazily” creates weights. The built-in function `add_weight()` can be used to add the weights and biases of the attention layer.

◀

`call()`: The `call()` method implements the mapping of inputs to outputs. It should implement the forward pass during training.

The Call Method for the Attention Layer

The call method of the attention layer has to compute the alignment scores, weights, and context. You can go through the details of these parameters in Chapter 8. You'll implement the Bahdanau attention in your `call()` method. The good thing about inheriting a layer from the Keras Layer class and adding the weights via the `add_weights()` method is that weights are automatically tuned. Keras does an equivalent of “reverse engineering” of the operations/computations of the `call()` method and calculates the gradients during training. It is important to specify `trainable=True` when adding the weights. You can also add a `train_step()` method to your custom layer and specify your own method for weight training if needed.

The code below implements the custom attention layer.

```
class attention(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def build(self, input_shape):
        self.W = self.add_weight(name='attention_weight', shape=(input_shape[-1],1),
            initializer='random_normal', trainable=True)
        self.b = self.add_weight(name='attention_bias', shape=(input_shape[1],1),
            initializer='zeros', trainable=True)
        super().build(input_shape)

    def call(self, x):
        # Alignment scores. Pass them through tanh function
        e = K.tanh(K.dot(x,self.W)+self.b)
        # Remove dimension of size 1
        e = K.squeeze(e, axis=-1)
        # Compute the weights
        alpha = K.softmax(e)
        # Reshape to tensorFlow format
        alpha = K.expand_dims(alpha, axis=-1)
        # Compute the context vector
        context = x * alpha
```

```
context = K.sum(context, axis=1)
return context
```

Listing 9.6: Add attention layer to the deep learning network

RNN with Attention Layer

Let's now add an attention layer to the RNN network you created earlier.

The attention layer expects a sequence as input. To use it *after* the SimpleRNN layer, the latter should return a sequence. The function `create_RNN_with_attention()` now specifies an RNN layer, an attention layer, and a Dense layer in the network. Make sure to set `return_sequences=True` when specifying the SimpleRNN. This will return the output of the hidden units for all the previous time steps, i.e., as a sequence.

Let's look at a summary of the model with attention.

```
def create_RNN_with_attention(hidden_units, dense_units, input_shape, activation):
    x = Input(shape=input_shape)
    RNN_layer = SimpleRNN(hidden_units, return_sequences=True, activation=activation)(x)
    attention_layer = attention()(RNN_layer)
    outputs = Dense(dense_units, trainable=True, activation=activation)(attention_layer)
    model = Model(x, outputs)
    model.compile(loss='mse', optimizer='adam')
    return model

model_attention = create_RNN_with_attention(hidden_units=hidden_units, dense_units=1,
                                             input_shape=(time_steps,1), activation='tanh')
model_attention.summary()
```

Listing 9.7: Create a RNN model with attention layer

```
Model: "model_1"
-----
Layer(type) OutputShape Param#
-----
input_2(InputLayer) [(None,20,1)] 0
-----
simple_rnn_2(SimpleRNN) (None,20,2) 8
-----
attention_1(attention) (None,2) 22
-----
dense_2(Dense) (None,1) 3
=====
Total params: 33
Trainable params: 33
Non-trainable params: 0
-----
```

Output 9.5: A RNN model with an attention layer

Train and Evaluate the Deep Learning Network with Attention

It's time to train and test your model and see how it performs in predicting the next Fibonacci number of a sequence.

```
model_attention.fit(trainX, trainY, epochs=epochs, batch_size=1, verbose=2)

# Evaluate model
train_mse_attn = model_attention.evaluate(trainX, trainY)
test_mse_attn = model_attention.evaluate(testX, testY)

# Print error
print("Train set MSE with attention = ", train_mse_attn)
print("Test set MSE with attention = ", test_mse_attn)
```

Listing 9.8: Training and evaluate the model

You'll see the training progress as output and the following:

```
Train set MSE with attention = 5.3511179430643097e-05
Test set MSE with attention = 9.053358553501312e-06
```

Output 9.6: Mean squared error as calculated

You can see that even for this simple example, the mean square error on the test set is lower with the attention layer. You can achieve better results with hyper-parameter tuning and model selection. Try this out on more complex problems and by adding more layers to the network. You can also use the scaler object to scale the numbers back to their original values.

You can take this example one step further by using LSTM instead of SimpleRNN, or you can build a network via convolution and pooling layers. You can also change this to an encoder-decoder network if you like.

9.4 Consolidated Code

The entire code for this chapter is pasted below if you would like to try it. Note that your outputs would be different from the ones given in this chapter because of the stochastic nature of this algorithm.

```
from pandas import read_csv
from sklearn.preprocessing import MinMaxScaler
from tensorflow.keras import Model
from tensorflow.keras.layers import Input, Dense, SimpleRNN
from tensorflow.keras.layers import Layer
from tensorflow.keras.metrics import mean_squared_error
from tensorflow.keras.models import Sequential
import numpy as np
import tensorflow.keras.backend as K

# Prepare data
def get_fib_seq(n, scale_data=True):
```



```

# Get the Fibonacci sequence
seq = np.zeros(n)
fib_n1 = 0.0
fib_n = 1.0
for i in range(n):
    seq[i] = fib_n1 + fib_n
    fib_n1 = fib_n
    fib_n = seq[i]
    scaler = []
if scale_data:
    scaler = MinMaxScaler(feature_range=(0, 1))
    seq = np.reshape(seq, (n, 1))
    seq = scaler.fit_transform(seq).flatten()
return seq, scaler

def get_fib_XY(total_fib_numbers, time_steps, train_percent, scale_data=True):
    dat, scaler = get_fib_seq(total_fib_numbers, scale_data)
    Y_ind = np.arange(time_steps, len(dat), 1)
    Y = dat[Y_ind]
    rows_x = len(Y)
    X = dat[0:rows_x]
    for i in range(time_steps-1):
        temp = dat[i+1:rows_x+i+1]
        X = np.column_stack((X, temp))
    # random permutation with fixed seed
    rand = np.random.RandomState(seed=13)
    idx = rand.permutation(rows_x)
    split = int(train_percent*rows_x)
    train_ind = idx[0:split]
    test_ind = idx[split:]
    trainX = X[train_ind]
    trainY = Y[train_ind]
    testX = X[test_ind]
    testY = Y[test_ind]
    trainX = np.reshape(trainX, (len(trainX), time_steps, 1))
    testX = np.reshape(testX, (len(testX), time_steps, 1))
    return trainX, trainY, testX, testY, scaler

# Set up parameters
time_steps = 20
hidden_units = 2
epochs = 30

# Create a traditional RNN network
def create_RNN(hidden_units, dense_units, input_shape, activation):
    model = Sequential()
    model.add(SimpleRNN(hidden_units, input_shape=input_shape, activation=activation[0]))
    model.add(Dense(units=dense_units, activation=activation[1]))
    model.compile(loss='mse', optimizer='adam')
    return model

model_RNN = create_RNN(hidden_units=hidden_units, dense_units=1,
input_shape=(time_steps,1), activation=['tanh', 'tanh'])

```

```

# Generate the dataset for the network
trainX, trainY, testX, testY, scaler = get_fib_XY(1200, time_steps, 0.7)
# Train the network
model_RNN.fit(trainX, trainY, epochs=epochs, batch_size=1, verbose=2)

# Evaluate model
train_mse = model_RNN.evaluate(trainX, trainY)
test_mse = model_RNN.evaluate(testX, testY)

# Print error
print("Train set MSE = ", train_mse)
print("Test set MSE = ", test_mse)

# Add attention layer to the deep learning network
class attention(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def build(self, input_shape):
        self.W=self.add_weight(name='attention_weight', shape=(input_shape[-1],1),
            initializer='random_normal', trainable=True)
        self.b=self.add_weight(name='attention_bias', shape=(input_shape[1],1),
            initializer='zeros', trainable=True)
        super().build(input_shape)

    def call(self,x):
        # Alignment scores. Pass them through tanh function
        e = K.tanh(K.dot(x,self.W)+self.b)
        # Remove dimension of size 1
        e = K.squeeze(e, axis=-1)
        # Compute the weights
        alpha = K.softmax(e)
        # Reshape to tensorflow format
        alpha = K.expand_dims(alpha, axis=-1)
        # Compute the context vector
        context = x * alpha
        context = K.sum(context, axis=1)
        return context

def create_RNN_with_attention(hidden_units, dense_units, input_shape, activation):
    x = Input(shape=input_shape)
    RNN_layer = SimpleRNN(hidden_units, return_sequences=True, activation=activation)(x)
    attention_layer = attention()(RNN_layer)
    outputs = Dense(dense_units, trainable=True, activation=activation)(attention_layer)
    model = Model(x,outputs)
    model.compile(loss='mse', optimizer='adam')
    return model

# Create the model with attention, train and evaluate
model_attention = create_RNN_with_attention(hidden_units=hidden_units, dense_units=1,
input_shape=(time_steps,1), activation='tanh')
model_attention.summary()
model_attention.fit(trainX, trainY, epochs=epochs, batch_size=1, verbose=2)

```

```
# Evaluate model
train_mse_attn = model_attention.evaluate(trainX, trainY)
test_mse_attn = model_attention.evaluate(testX, testY)

# Print error
print("Train set MSE with attention = ", train_mse_attn)
print("Test set MSE with attention = ", test_mse_attn)
```

Listing 9.9: Complete code for creating a RNN model with attention

9.5 Further Reading

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

Books

Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

<https://www.amazon.com/dp/0262035618>

(Online version at <http://www.deeplearningbook.org>).

Wei Di, Anurag Bhardwaj, and Jianing Wei. *Deep Learning Essentials*. Packt Publishing, 2018.

<https://www.amazon.com/dp/1785880365>

Papers

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *Proc. 3rd International Conference on Learning Representations (ICLR 2015)*. May 2015.

<http://arxiv.org/abs/1409.0473>

Articles

François Chollet. *Making new layers and models via subclassing*. Keras developer guides, 2020.

https://keras.io/guides/making_new_layers_and_models_via_subclassing/

9.6 Summary

In this chapter, you discovered how to add a custom attention layer to a deep learning network using Keras. Specifically, you learned:

- ◀ How to override the Keras

- Layer class

- ◀ The method

- `build()` is required to add weights to the attention layer

- ◀ The

- `call()` method is required for specifying the mapping of inputs to outputs of the attention layer

- ◀ How to add a custom attention layer to the deep learning network built using SimpleRNN

In the next chapter, you will see how a transformer model brings attention to the next level.

The Transformer Attention Mechanism

10

Before the introduction of the transformer model, the use of attention for neural machine translation was implemented by RNN-based encoder-decoder architectures. The transformer model revolutionized the implementation of attention by dispensing with recurrence and convolutions and, alternatively, relying solely on a self-attention mechanism. We will first be focusing on the transformer attention mechanism in this chapter and subsequently review the transformer model in a separate one.

In this chapter, you will discover the transformer attention mechanism for neural machine translation. After completing this chapter, you will know:

- ◄ How the transformer attention differed from its predecessors
- ◄ How the transformer computes a scaled-dot product attention
- ◄ How the transformer computes multi-head attention

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◄ Introduction to the Transformer Attention
- ◄ Scaled Dot-Product Attention
- ◄ Multi-Head Attention

10.1 Introduction to the Transformer Attention

Thus far, you have familiarized yourself with using an attention mechanism in conjunction with an RNN-based encoder-decoder architecture. Two of the most popular models that implement attention in this manner have been those proposed by Bahdanau et al. (2015) and Luong et al. (2015).

The transformer architecture revolutionized the use of attention by dispensing with recurrence and convolutions, on which the formers had extensively relied.

“ ... the transformer is the first transduction model relying entirely on self-attention to compute representations of its input and output without using sequence-aligned RNNs or convolution. ”

— “Attention Is All You Need”, 2017

The paper “Attention Is All You Need” explained that the transformer model, alternatively, relies solely on the use of self-attention, where the representation of a sequence (or sentence) is computed by relating different words in the same sequence.

“ Self-attention, sometimes called intra-attention, is an attention mechanism relating different positions of a single sequence in order to compute a representation of the sequence. ”

— “Attention Is All You Need”, 2017

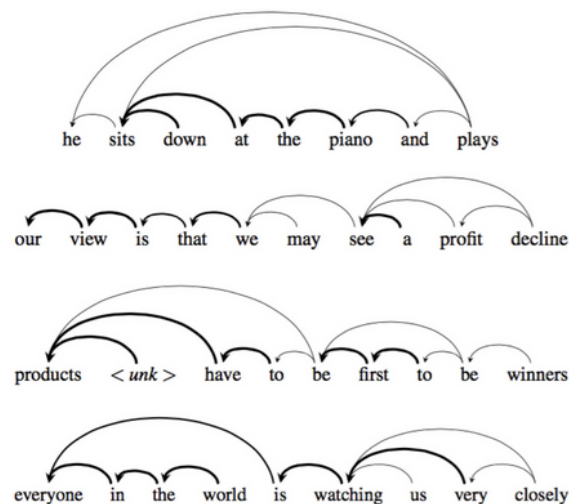


Figure 10.1: Intra-attention mechanism illustrated. Bold lines indicate higher attention scores. Arrows denote which word is being focused when attention is computed. From “Long Short-Term Memory-Networks for Machine

Reading”

The main components used by the transformer attention are the following:

◁ **q** and **k** denoting vectors of dimension d , containing the queries and keys, respectively

◁ **v** denoting a vector of dimension d , containing the values

◁ **Q**, **K**, and **V** denoting matrices packing together sets of queries, keys, and values, respectively

W_Q, **W_K**, and **W_V** denoting projection matrices that are used in generating different subspace representations of the query, key, and value matrices

W_O denoting a projection matrix for the multi-head output

In essence, the attention function can be considered a mapping between a query and a set of key-value pairs to an output.

“The output is computed as a weighted sum of the values, where the weights are computed from the query and key matrices as follows: $\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V$ ”

— “Attention Is All You Need”, 2017

Vaswani et al. propose a *scaled dot-product attention*. Within the context of neural machine translation, the query, keys, and values that

are used as inputs to these attention mechanisms are different projections of the same input sentence.

Intuitively, therefore, the proposed attention mechanisms implement self-attention by capturing the relationships between the different elements (in this case, the words) of the same sentence.

10.2 Scaled Dot-Product Attention

The transformer implements a scaled dot-product attention, which follows the procedure of the general attention mechanism that you had previously seen.

As the name suggests, the scaled dot-product attention first computes *a dot product* for each query \mathbf{q} with all of the keys \mathbf{k} . It subsequently divides each result by $\sqrt{d_k}$ (where d_k is the dimension of the vectors \mathbf{q} and \mathbf{k}) and proceeds to apply a softmax function. In doing so, it obtains the weights that are used to *scale* the values \mathbf{v} .

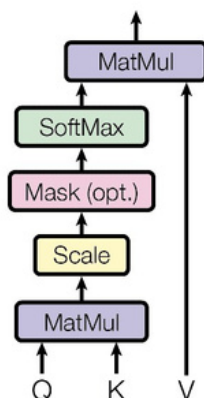


Figure 10.2: Scaled dot-product attention. From “Attention Is All You Need”

In practice, the computations performed by the scaled dot-product attention can be efficiently applied to the entire set of queries simultaneously. In order to do so, the matrices — \mathbf{Q} , \mathbf{K} , and \mathbf{V} — are supplied as inputs to the attention function:

attention \mathbf{Q}^T (softmax(
 $\frac{\mathbf{QK}^T}{\sqrt{d_k}})$) \mathbf{V}

Vaswani et al. explain that their scaled dot-product attention is identical

to the multiplicative attention of Luong et al. (2015), except for the added scaling factor of $1/\sqrt{d_k}$. This scaling factor was introduced to counteract the effect of having the dot products grow large in magnitude for

large values of d_k , where the application of the softmax function would then return extremely small gradients that would lead to the infamous vanishing gradients problem. The scaling factor, therefore, serves to pull the results generated by the dot product multiplication down, preventing this problem.

Vaswani et al. further explain that their choice of opting for multiplicative attention instead of the additive attention of Bahdanau et al. (2015) was based on the computational efficiency associated with the former.

“...dot-product attention is much faster and more space-efficient in practice since it can be implemented using highly optimized matrix multiplication code.

—“Attention Is All You Need”

3. And follow the scaling process by applying a softmax operation in order to obtain a set of weights:

2017, Therefore, the step-by-step procedure for computing the scaled dot product attention is the following:

1. Compute the alignment scores by multiplying the set of queries packed in matrix \mathbf{Q} with the keys in the matrix \mathbf{K} . If the matrix \mathbf{Q} is of size m

4. Finally, apply the resulting weights to the values in matrix V , of size $n \times d_v$:

$$\text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V = \frac{1}{n} \begin{pmatrix} v_{11} & v_{12} & \dots & v_{1d_v} \\ v_{21} & v_{22} & \dots & v_{2d_v} \\ \vdots & \vdots & \ddots & \vdots \\ v_{n1} & v_{n2} & \dots & v_{nd_v} \end{pmatrix}$$

10.3 Multi-Head Attention

Building on their single attention function that takes matrices Q , K , and V as input, as you have just reviewed, Vaswani et al. also propose a multi-head attention mechanism.

Their multi-head attention mechanism linearly projects the queries, keys, and values h times, using a different learned projection each time. The single attention mechanism is then applied to each of these h projections in parallel to produce h outputs, which, in turn, are concatenated and projected again to produce a final result.

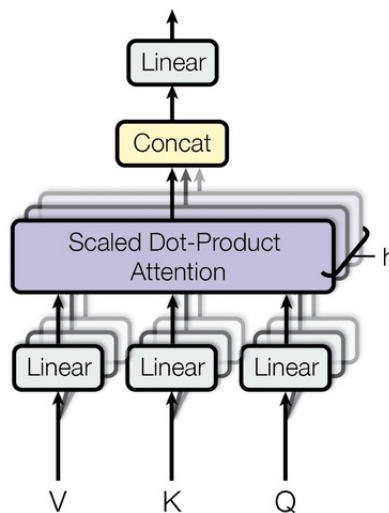


Figure 10.3: Multi-head attention. From “Attention Is All You Need”

The idea behind multi-head attention is to allow the attention function to extract information from different representation subspaces, which would otherwise be impossible with a single attention head.

The multi-head attention function can be represented as follows:

$$\text{multihead}(Q, K, V) = \text{concat}(\text{head}_1, \dots, \text{head}_h)W$$

Here, each head_i , $i = 1, \dots, h$, implements a single attention function characterized by its own learned projection matrices:

$$\text{head}_i \text{attention}(Q, K, V) = (QW_i, KW_i, VW_i)$$

The step-by-step procedure for computing multi-head attention is, therefore, the following:

1. Compute the linearly projected versions of the queries, keys, and values through multiplication with the respective weight matrices QW_K , QW_V , and QW_O , one for each head i .
2. Apply the single attention function for each head by (1) multiplying the queries and keys matrices, (2) applying the scaling and softmax operations, and (3) weighting the values matrix to generate an output for each head.
3. Concatenate the outputs of the heads head $_i$, $i = 1, \dots, h$.
4. Apply a linear projection to the concatenated output through multiplication with the weight matrix WO to generate the final result.

10.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Papers

Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. “Neural Machine Translation by Jointly Learning to Align and Translate”. In: *Proc. 3rd International Conference on Learning Representations (ICLR 2015)*. May 2015.

<http://arxiv.org/abs/1409.0473>

Minh-Thang Luong, Hieu Pham, and Christopher D. Manning. “Effective Approaches to Attention-based Neural Machine Translation”. In: *Proc. EMNLP*. Sept. 2015, pp. 1412–1421. DOI: [10.18653/v1/D15-1166](https://doi.org/10.18653/v1/D15-1166).

<https://arxiv.org/abs/1508.04025>

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

10.5 Summary

In this chapter, you discovered the transformer attention mechanism for neural machine translation. Specifically, you learned:

- ◁ How the transformer attention differed from its predecessors
- ◁ How the transformer computes a scaled-dot product attention
- ◁ How the transformer computes multi-head attention

In the next chapter, you will take a deeper look into the transformer model architecture.

The Transformer Model

We have already familiarized ourselves with the concept of self-attention as implemented by the transformer attention mechanism for neural machine translation. We will now be shifting our focus on the details of the transformer architecture itself to discover how self-attention can be implemented without relying on the use of recurrence and convolutions.

In this chapter, you will discover the network architecture of the transformer model. After completing this chapter, you will know:

- ◁ How the transformer architecture implements an encoder-decoder structure without recurrence and convolutions
- ◁ How the transformer encoder and decoder work
- ◁ How the transformer self-attention compares to the use of recurrent and convolutional layers

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◁ The Transformer Architecture
- ◁ Sum Up: The Transformer Model
 - ◁ Comparison to Recurrent and Convolutional Layers

11.1 The Transformer Architecture

The transformer architecture follows an encoder-decoder structure but does not rely on recurrence and convolutions in order to generate an output.

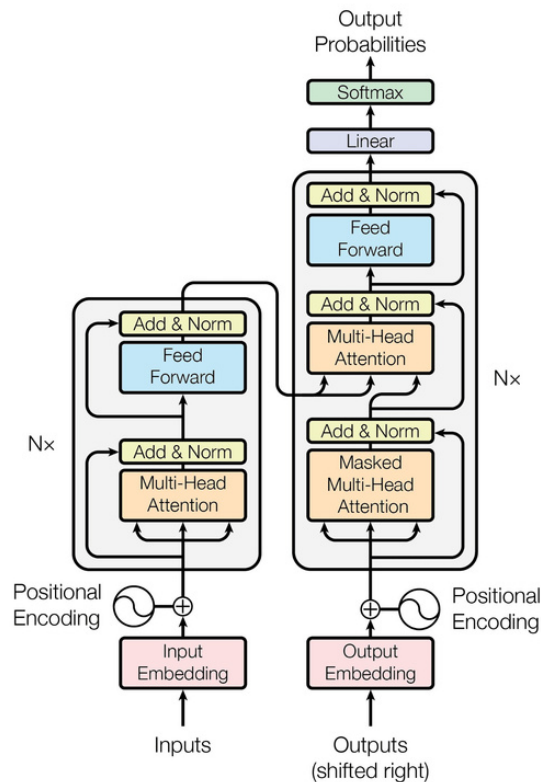


Figure 11.1: The encoder-decoder structure of the transformer architecture.
 “Attention Is All You Need”

From

In a nutshell, the task of the encoder, on the left half of the transformer architecture, is to map an input sequence to a sequence of continuous representations, which is then fed into a decoder. The decoder, on the right half of the architecture, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence.

“At each step the model is aut

o-

regressive, consuming the pre-
 viously generated symbols
 as additional input when ge

The Encoder

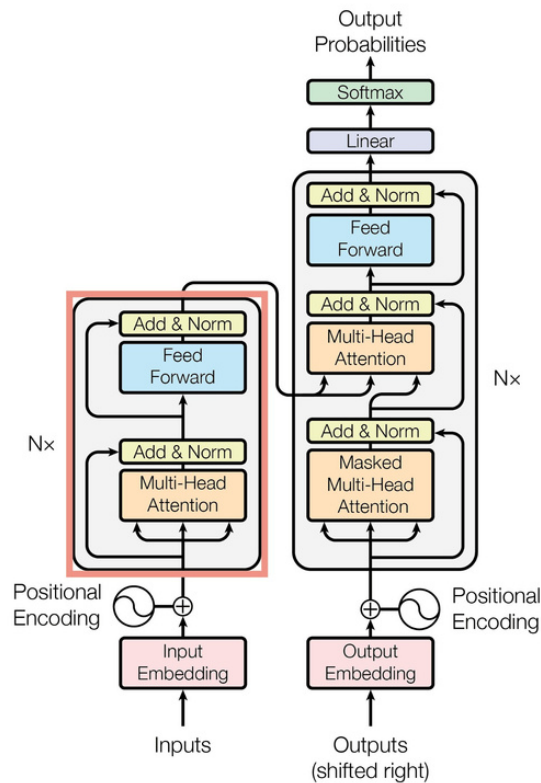


Figure 11.2: The encoder block of the transformer architecture. From “Attention Is All You Need”

The encoder consists of a stack of $N = 6$ identical layers, where each layer is composed of two sublayers:

1. The first sublayer implements a multi-head self-attention mechanism. You have seen that the multi-head mechanism implements h heads that receive a (different) linearly projected version of the queries, keys, and values, each to produce h outputs in parallel that are then used to generate a final result.

2. The second sublayer is a fully connected feedforward network consisting of two linear transformations with Rectified Linear Unit (ReLU) activation in between:

$$\text{FFN}(x) = \text{ReLU}(\mathbf{W}_1 x + b_1) \mathbf{W}_2 + b_2$$

The six layers of the transformer encoder apply the same linear transformations to all the words in the input sequence, but *each* layer employs different weight (\mathbf{W}_1 , \mathbf{W}_2) and bias (b_1 , b_2) parameters to do so.

Furthermore, each of these two sublayers has a residual connection around it. Each sublayer is also succeeded by a normalization layer, $\text{layernorm}(\cdot)$, which normalizes the sum computed between the sublayer input x and the output generated by the sublayer itself, $\text{sublayer}(x)$:

$$\text{layernorm}(x + \text{sublayer}(x))$$

An important consideration to keep in mind is that the transformer architecture cannot inherently capture any information about the relative positions of the words in the sequence since it does not make use of recurrence. This information has to be injected by introducing *positional encodings* to the input embeddings.

The positional encoding vectors are of the same dimension as the input embeddings and are generated using sine and cosine functions of different frequencies. Then, they are simply summed to the input embeddings in order to *inject* the positional information.

The Decoder

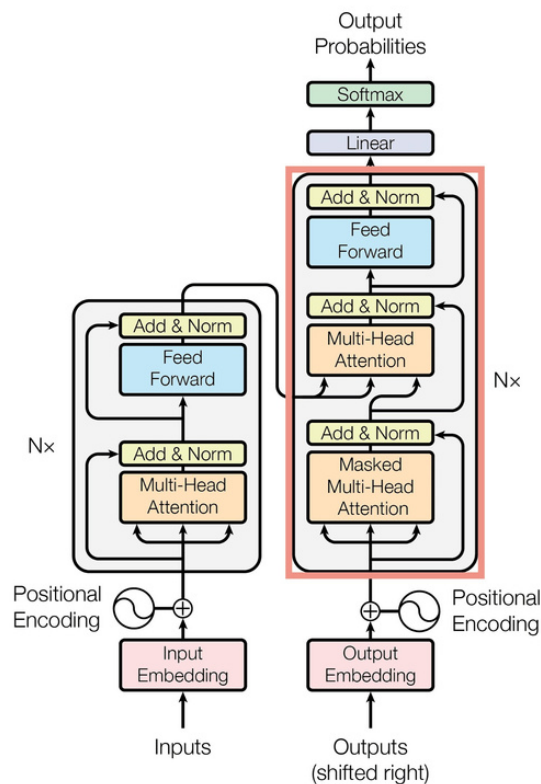


Figure 11.3: The decoder block of the transformer architecture. From “Attention Is All You Need”

The decoder shares several similarities with the encoder. The decoder also consists of a stack of $N = 6$ identical layers that are each composed of three sublayers:

1. The first sublayer receives the previous output of the decoder stack, augments it with positional information, and implements multi-head self-attention over it. While the encoder is designed to attend to all words in the input sequence *regardless* of their position in the sequence, the decoder is modified to attend *only* to the preceding words. Hence, the prediction for a word at position i can only depend on the known outputs for the words that come before it in the sequence. In the multi-head attention mechanism (which implements multiple, single attention functions in parallel), this is achieved by introducing a mask over the values produced by the scaled multiplication

of matrices \mathbf{Q} and \mathbf{K} . This masking is implemented by suppressing the matrix values that would otherwise correspond to illegal connections:

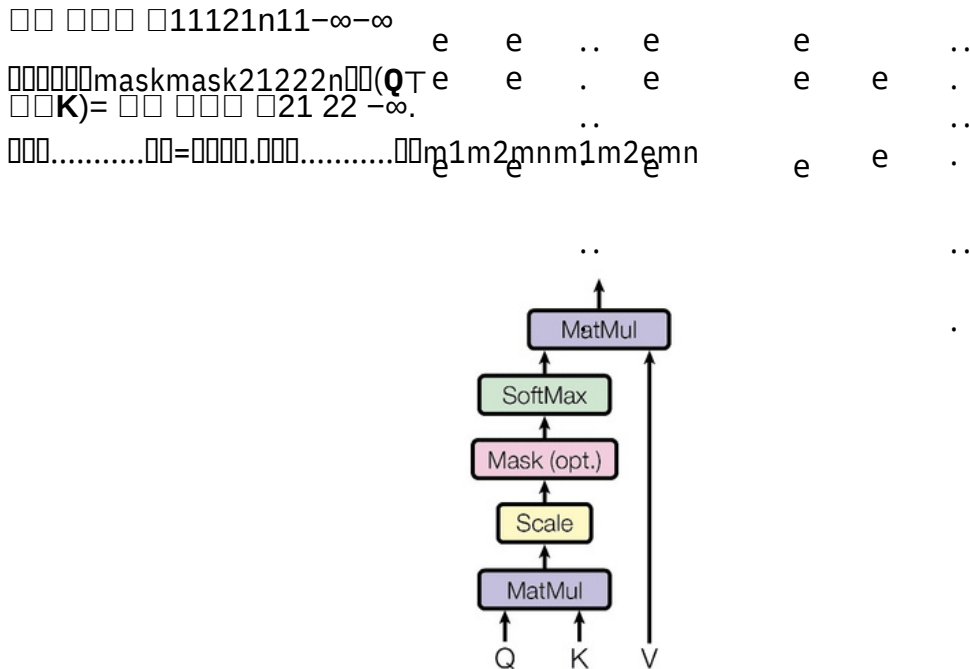


Figure 11.4: The multi-head attention in the decoder implements several masked, single attention functions. From “Attention Is All You Need”

“The masking makes the decoder

2. The second layer implement

implemented in the first sublayer of the encoder. On the decoder side, this multi-head mechanism receives the queries from the previous decoder sublayer and the keys and values from the output of the encoder. This allows the decoder to attend to all the words in the input sequence.

3. The third layer implements a fully connected feedforward network, similar to the one implemented in the second sublayer of the encoder.

Furthermore, the three sublayers on the decoder side also have residual connections around them and are succeeded by a normalization layer. Positional encodings are also added to the input embeddings of the decoder in the same manner as previously explained for the encoder.

11.2 Sum Up: The Transformer Model

The transformer model runs as follows:

1. Each word forming an input sequence is transformed into a d_{model} dimensional embedding vector.

2. Each embedding vector representing an input word is augmented by summing it (element-wise) to a positional encoding vector of the same dmodel length, hence introducing positional information into the input.
3. The augmented embedding vectors are fed into the encoder block consisting of the two sublayers explained above. Since the encoder attends to all words in the input sequence, irrespective if they precede or succeed the word under consideration, then the transformer encoder is *bidirectional*.
4. The decoder receives as input its own predicted output word at timestep $t - 1$.
5. The input to the decoder is also augmented by positional encoding in the same manner done on the encoder side.
6. The augmented decoder input is fed into the three sublayers comprising the decoder block explained above. Masking is applied in the first sublayer in order to stop the decoder from attending to succeeding words. At the second sublayer, the decoder also receives the output of the encoder, which now allows the decoder to attend to all the words in the input sequence.
7. The output of the decoder finally passes through a fully connected layer, followed by a softmax layer, to generate a prediction for the next word of the output sequence.

11.3 Comparison to Recurrent and Convolutional Layers

Vaswani et al. (2017) explain that their motivation for abandoning the use of recurrence and convolutions was based on several factors:

1. Self-attention layers were found to be faster than recurrent layers for shorter sequence lengths and can be restricted to consider only a neighborhood in the input sequence for very long sequence lengths.
2. The number of sequential operations required by a recurrent layer is based on the sequence length, whereas this number remains constant for a self-attention layer.
3. In convolutional neural networks, the kernel width directly affects the long-term dependencies that can be established between pairs of input and output positions. Tracking long-term dependencies would require using large kernels or stacks of convolutional layers that could increase the computational cost.

11.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

11.5 Summary

In this chapter, you discovered the network architecture of the transformer model. Specifically, you learned:

- ◁ How the transformer architecture implements an encoder-decoder structure without recurrence and convolutions
- ◁ How the transformer encoder and decoder work
- ◁ How the transformer self-attention compares to recurrent and convolutional layers

Before we wrap up this part, we will digress a bit from text sequences to see, in the next chapter, how transformer can also be applied to images.

The Vision Transformer Model

12

With the transformer architecture revolutionizing the implementation of attention, and achieving very promising results in the natural language processing domain, it was only a matter of time before we could see its application in the computer vision domain too. This was eventually achieved with the implementation of the Vision Transformer (ViT).

In this chapter, you will discover the architecture of the Vision Transformer model, and its application to the task of image classification. After completing this chapter, you will know:

- ◄ How the ViT works in the context of image classification
- ◄ What the training process of the ViT entails
 - ◄ How the ViT compares to convolutional neural networks in terms of inductive bias
- ◄ How the ViT fares against ResNets on different datasets
- ◄ How the data is processed internally for the ViT to achieve its performance

Let's get started.

Overview

This chapter is divided into six parts; they are:

- ◄ Introduction to the Vision Transformer (ViT)
- ◄ The ViT Architecture
- ◄ Training the ViT
 - ◄ Inductive Bias in Comparison to Convolutional Neural Networks
- ◄ Comparative Performance of ViT Variants with ResNets
- ◄ Internal Representation of Data

12.1 Introduction to the Vision Transformer (ViT)

You have seen how the emergence of the transformer architecture of Vaswani et al. (2017) has revolutionized the use of attention, without relying on recurrence and convolutions as earlier attention models had previously done. In their work, Vaswani et al. had applied their model to the specific problem of natural language processing (NLP).

“In computer vision, however, convolutional architectures remain dominant...”

Recall that the standard transformer model received a one-dimensional sequence of word embeddings as input, since it was originally meant for NLP. In contrast, when applied to the task of image classification in computer vision, the input data to the transformer model is provided in the form of two-dimensional images.

For the purpose of structuring the input image data in a manner that resembles how the input is structured in the NLP domain (in the sense of having a sequence of individual words), the input image, of height H , width W , and C number of channels, is *cut up* into smaller two-dimensional patches. This results into $N = HW/P^2$ number of patches, where each patch has a resolution of $P \times P$ pixels.

Before feeding the data into the transformer, the following operations are applied:

1. Each image patch is flattened into a vector of length $2p \times P$, where $n = 1, \dots, N$.
2. A sequence of embedded image patches is generated by mapping the flattened patches to D dimensions, with a trainable linear projection E .

3. A learnable class embedding x_{class} is prepended to the sequence of embedded image patches. The value of x_{class} represents the classification output y .

4. The patch embeddings are finally augmented with one-dimensional positional embeddings E_{pos} hence introducing positional information into the input, which is also learned during training.

The sequence of embedding vectors that results from the aforementioned operations is the following:

$$z_0 = [x_{\text{class}}; x_1 E; \dots; x_N E] + E_{\text{pos}}$$

Dosovitskiy et al. make use of the encoder part of the transformer architecture of Vaswani et al. In order to perform classification, they feed z_0 at the input of the transformer encoder, which consists of a stack of L identical layers. Then, they proceed to take the value of x_{class} at the L th layer of the encoder output, and feed it into a classification head.

“The classification head is simple The multilayer perceptron (MLP)

Error Linear Unit (GELU) nonlinearity.

In summary, the ViT employs the encoder part of the original transformer architecture. The input to the encoder is a sequence of embedded image patches (including a learnable class embedding prepended to the sequence), which is also augmented with positional information. A classification head attached to the output of the encoder receives the value of the learnable class embedding, to generate a classification output based on its state. All of this is illustrated by the figure below:

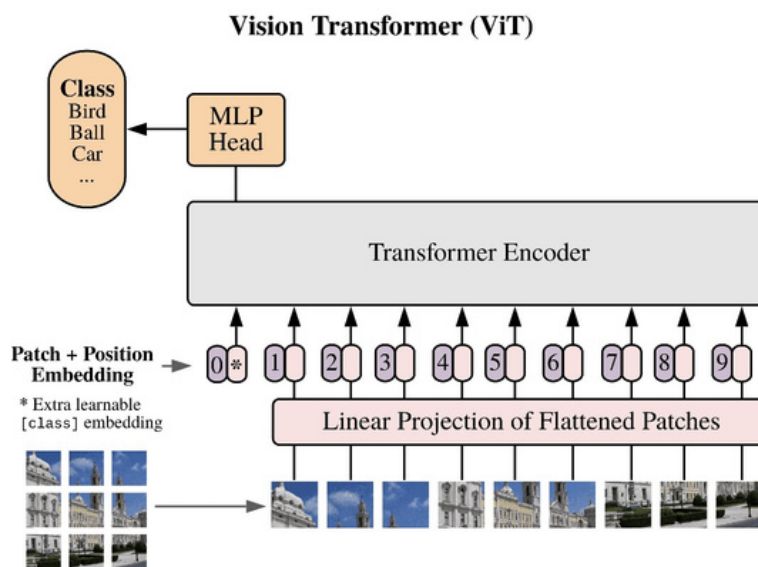


Figure 12.1: The architecture of the vision transformer (ViT). From “An Image is Worth 16x16Words”

Compare to a convolutional layer, ViT does not generate separate feature maps for the entire image. Instead, each patch of the image will be converted into an embedding which various features are represented by a vector. Dosovitskiy et al. also pointed out that, the original image can, alternatively, be fed into a convolutional neural network (CNN) before being passed on to the transformer encoder. The sequence of image patches would then be obtained from the feature maps of the CNN, while the ensuing process of embedding the feature map patches, prepending a class token, and augmenting with positional information remains the same.

12.3 Training the ViT

The ViT is pre-trained on larger datasets (such as ImageNet, ImageNet-21k and JFT-300M) and fine-tuned to a smaller number of classes. During pre-training, the classification head in use that attached to the encoder output is implemented by a MLP with one hidden layer and GELU nonlinearity, as has been described earlier.

During fine-tuning, the MLP is replaced by a single (zero-initialized) feedforward layer of size $D \times K$ where K is the number of classes corresponding to the task at hand. Fine-tuning

is carried out on images that are of higher resolution than those used during pre-training, but the patch size into which the input images are cut is kept the same at all stages of training. This results in an input sequence of larger length at the fine-tuning stage, in comparison to that used during pre-training.

The implication of having a lengthier input sequence is that fine-tuning requires more position embeddings than pre-training. To circumvent this problem, Dosovitskiy et al. interpolate the pre-training position embeddings in two-dimensions according to their location in the original image, to obtain a longer sequence that matches the number of image patches in use during fine-tuning.

12.4 Inductive Bias in Comparison to Convolutional Neural Networks

Inductive bias refers to any assumptions that a model makes to generalise the training data and learn the target function.

“In CNNs, locality, two-dimensional neighborhood structure, and translation equivariance are baked into each layer throughout the whole model.

—“An Image is Worth 16x16

Words” 2021

12.5 Comparative Performance of ViT Variants with ResNets

Dosovitskiy et al. pitted three ViT models of increasing size, against two modified ResNets of different sizes. Their experiments yield several interesting findings:

◀ Experiment 1: Fine-tuning and testing on ImageNet:

- When pre-trained on the smallest dataset (ImageNet), the two larger ViT models underperformed in comparison to their smaller counterpart. The performance of all ViT models remains, generally, below that of the ResNets.
- When pre-trained on a larger dataset (ImageNet-21k), the three ViT models performed similarly to one another, as well as to the ResNets.
- When pre-trained on the largest dataset (JFT-300M), the performance of the larger ViT models overtakes the performance of the smaller ViT and the ResNets.

◀ Experiment 2: Training on random subsets of different sizes of the JFT-300M dataset, and testing on ImageNet, to further investigate the effect of the dataset size:

- On smaller subsets of the dataset, the ViT model overfits more than the ResNet models, and underperform considerably.
- On the larger subset of the dataset, the performance of the larger ViT model surpasses the performance of the ResNets.

“This result reinforces the intuition that the convolutional

12.6 Internal Representation of Data

In analysing the internal representation of the image data in the ViT, Dosovitskiy et al. find the following:

◀ The learned embedding filters that are initially applied to the image patches at the first layer of the ViT, resemble basis functions that can extract the low-level features within each patch:

smaller datasets, but for large
 rones, learning the relevant
 patterns directly from data is
 sufficient, even beneficial.

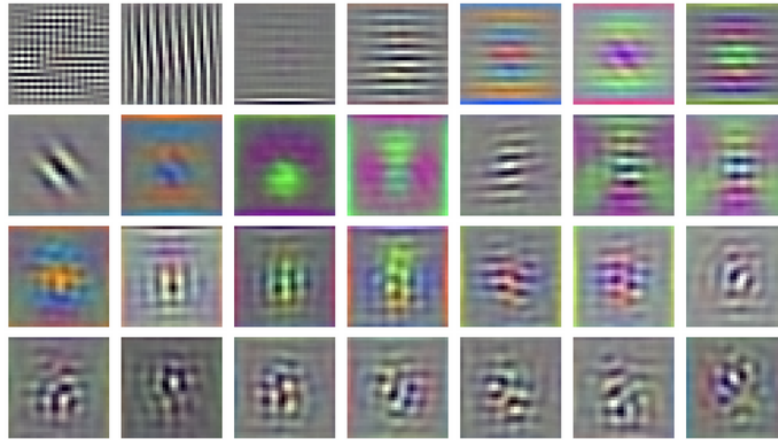


Figure 12.2: Learned Embedding Filters. From “An Image is Worth 16x16 Words”

- Image patches that are spatially close to one another in the original image, are characterised by learned positional embeddings that are similar:

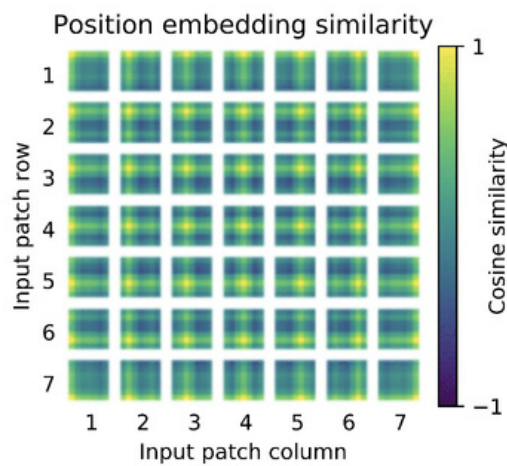


Figure 12.3: Learned Positional Embeddings. From “An Image is Worth 16x16 Words”

- Several self-attention heads at the lowest layers of the model already attend to most of the image information (based on their attention weights), demonstrating the capability of the self-attention mechanism in integrating the information across the entire image:

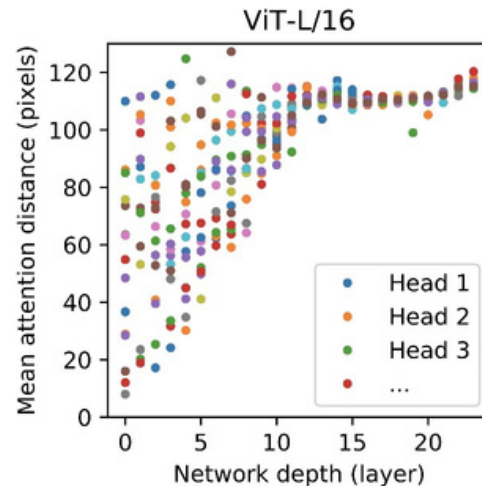


Figure 12.4: Size of Image Area Attended by Different Self-Attention Heads. From “An Image is Worth 16x16 Words”

12.7 Further Reading

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

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

Alexey Dosovitskiy et al. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”. In: *Proc. 9th International Conference on Learning Representations (ICLR)*.

May 2021.

<https://arxiv.org/abs/2010.11929>

12.8 Summary

In this chapter, you discovered the architecture of the vision transformer model, and its application to the task of image classification.

Specifically, you learned:

- ◁ How the ViT works in the context of image classification
- ◁ What the training process of the ViT entails
 - ◁ How the ViT compares to convolutional neural networks in terms of inductive bias
- ◁ How the ViT fares against ResNets on different datasets
- ◁ How the data is processed internally for the ViT to achieve its performance

Start from next chapter, you will build a transformer model in Keras step-by-step.

Building a Transformer from Scratch

Positional Encoding in Transformer Models

13

In languages, the order of the words and their position in a sentence really matters. The meaning of the entire sentence can change if the words are re-ordered. When implementing NLP solutions, recurrent neural networks have an inbuilt mechanism that deals with the order of sequences. The transformer model, however, does not use recurrence or convolution and treats each data point as independent of the other. Hence, positional information is added to the model explicitly to retain the information regarding the order of words in a sentence. Positional encoding is the scheme through which the knowledge of order of objects in a sequence is maintained.

For this chapter, we'll simplify the notations used in "Attention Is All You Need" by Vaswani et al. (2017). After completing this chapter, you will know:

- ◀ What is positional encoding, and why it's important
- ◀ Positional encoding in transformers
- ◀ Code and visualize a positional encoding matrix in Python using NumPy

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ◀ What is Positional Encoding?
- ◀ Positional Encoding Layer in Transformers
 - ◀ Coding the Positional Encoding Matrix from Scratch
- ◀ Understanding the Positional Encoding Matrix

13.1 What is Positional Encoding?

Positional encoding describes the location or position of an entity in a sequence so that each position is assigned a unique representation. There are many reasons why a single number, such as the index value, is not used to represent an item's position in transformer models.

For long sequences, the indices can grow large in magnitude. If you normalize the index value to lie between 0 and 1, it can create problems for variable length sequences as they would be normalized differently.

Transformers use a smart positional encoding scheme, where each position/index is mapped to a vector. Hence, the output of the positional encoding layer is a matrix, where each row of the matrix represents an encoded object of the sequence summed with its positional information. An example of the matrix that encodes only the positional information is shown in the figure below.

Sequence		Index of token		Positional encoding matrix			
I	!!	0	!!	P_{00}	P_{01}	.	P_d
am	!!	1	!!	P_{10}	P_{11}	.	0_d
a		2		P_{20}	P_{21}	.	P_d
Rob ot		3		P_{30}	P_{31}	.	1_d

Figure 13.1: Positional encoding matrix for the sequence “I am a robot”

13.2 Positional Encoding Layer in Transformers

This is a quick recap of sine functions; and you can work equivalently with cosine functions. The function’s range is [-1,+1]. The frequency of this waveform is the number of cycles completed in one second. The wavelength is the distance over which the waveform repeats itself. The wavelength and frequency for different waveforms are shown below:

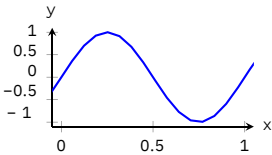
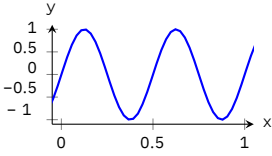
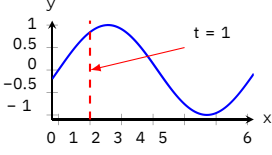
Equation	Graph	Frequency	Wavelength
$\sin(2\pi t)$		1	1
$\sin(2 \times 2\pi t)$		2	$\frac{1}{2}$
$\sin(t)$		$\frac{1}{2\pi}$	2π
$\sin(ct)$	(Depends on c)	$\frac{c}{2\pi}$	$\frac{2\pi}{c}$

Figure 13.2: Different waveforms of sine functions

Let’s dive straight into this. Suppose you have an input sequence of length L and we require the position of the kth object within this sequence. The positional encoding is given by sine

and cosine functions of varying frequencies:

$$P(k, i) = \sin\left(\frac{(k \cdot 2^i)^{\frac{1}{d}}}{i/d}\right)$$

$$P(k, i+1) = \cos\left(\frac{(k \cdot 2^{i+1})^{\frac{1}{d}}}{(i+1)/d}\right)$$

Here:

◁ k: Position of an object in the input sequence, $0 \leq k < L$

◁ d: Dimension of the output embedding space

◁ P(k, i): Position function for mapping a position k in the input sequence to index (k, i) of the positional matrix

◁ n: User-defined scalar, set to 10000 by the authors of “Attention Is All You Need”

◁ i: Used for mapping to column indices $0 \leq i < d/2$, with a single value of i maps to

both sine and cosine functions

In the above expression, you can see that even positions correspond to a sine function and odd positions correspond to cosine functions.

Example

To understand the above expression, let's take an example of the phrase “I am a robot,” with $n = 100$ and $d = 4$. The following table shows the positional encoding matrix for this phrase. In fact the positional encoding matrix would be the same for any four-letter phrase with $n = 100$ and $d = 4$.

Sequence		Index of token, k	Positional encoding matrix with $d = 4, n = 100$ $i = 0 \quad i = 0 \quad i = 1 \quad i = 1$			
I	!	0	!	$P_{00} = \sin(0/1) \quad P_{01} = \cos(0/1) \quad P_{02} = \sin(0/10) \quad P_{03} = \cos(0/10)$		
				$= 0 \quad = 1 \quad = 0 \quad = 1$		
am	!!	1	!!	$P_{10} = \sin(1/1) \quad P_{11} = \cos(1/1) \quad P_{12} = \sin(1/10) \quad P_{13} = \cos(1/10)$		
				$= 0.84 \quad = 0.54 \quad = 0.10 \quad = 1.00$		
a	!	2	!	$P_{20} = \sin(2/1) \quad P_{21} = \cos(2/1) \quad P_{22} = \sin(2/10) \quad P_{23} = \cos(2/10)$		
				$= 0.91 \quad = -0.42 \quad = 0.20 \quad = 0.98$		
Rob ot		3		$P_{30} = \sin(3/1) \quad P_{31} = \cos(3/1) \quad P_{32} = \sin(3/10) \quad P_{33} = \cos(3/10)$		
				$= 0.14 \quad = -0.30 \quad = 0.30 \quad = 0.94$		

Figure 13.3: Positional encoding matrix for the sequence “I am a robot”

13.3 Coding the Positional Encoding Matrix from Scratch

Here is a short Python code to implement positional encoding using NumPy. The code is simplified to make the understanding of positional encoding easier.

```
import numpy as np
import matplotlib.pyplot as plt

def getPositionEncoding(seq_len, d, n=10000):
    P = np.zeros((seq_len, d))
    for k in range(seq_len):
        for i in np.arange(int(d/2)):
            denominator = np.power(n, 2*i/d)
            P[k, 2*i] = np.sin(k/denominator)
            P[k, 2*i+1] = np.cos(k/denominator)
    return P

P = getPositionEncoding(seq_len=4, d=4, n=100)
print(P)
```

Listing 13.1: Computing the positional encoding

```
[[0. 1.          0.          1.]
 [ 0.84147098  0.54030231  0.09983342  0.99500417]
 [ 0.90929743 -0.41614684  0.19866933  0.98006658]
 [ 0.14112001 -0.9899925   0.29552021  0.9553649]]
```

Output 13.1: Positional encoding as computed

13.4 Understanding the Positional Encoding Matrix

To understand the positional encoding, let's start by looking at the sine wave for different positions with $n=10000$ and $d=512$. Python

```
import numpy as np
import matplotlib.pyplot as plt

def plotSinusoid(k, d=512, n=10000):
    x = np.arange(0, 100, 1)
    denominator = np.power(n, 2*x/d)
    y = np.sin(k/denominator)
    plt.plot(x, y)
    plt.title('k = ' + str(k))

fig = plt.figure(figsize=(15, 4))
for i in range(4):
    plt.subplot(141 + i)
    plotSinusoid(i*4)
plt.show()
```

Listing 13.2: Plotting the positional encoding

The following figure is the output of the above code:

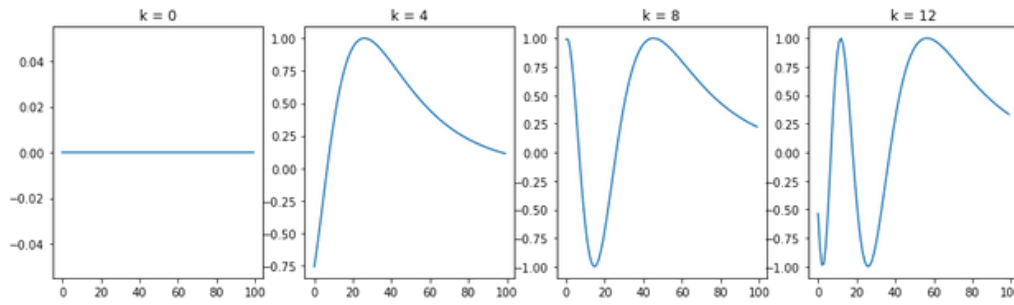


Figure 13.4: Sine wave for different position indices

You can see that each position k corresponds to a different sinusoid, which encodes a single position into a vector. If you look closely at the positional encoding function, you can see that the wavelength for a fixed i is given by:

$$\lambda = 2\pi n^{2i/d_i}$$

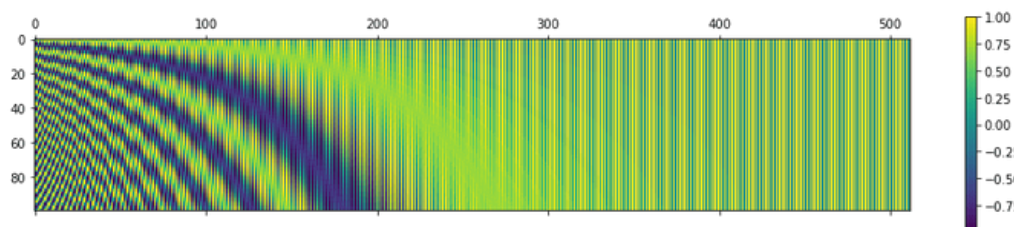
Hence, the wavelengths of the sinusoids form a geometric progression and vary from 2π to $2\pi n$. The scheme for positional encoding has a number of advantages.

1. The sine and cosine functions have values in $[-1, 1]$, which keeps the values of the positional encoding matrix in a normalized range.
2. As the sinusoid for each position is different, you have a unique way of encoding each position.
3. You have a way of measuring or quantifying the similarity between different positions, hence enabling us to encode the relative positions of words.

Let's visualize the positional matrix on bigger values. Use Python's `matshow()` method from the `matplotlib` library. Setting $n = 10000$ as done in the original paper, you get the following:

```
P = getPositionEncoding(seq_len=100, d=512, n=10000)
cax = plt.matshow(P)
plt.gcf().colorbar(cax)
```

Listing 13.3: Visualizing the positional matrix

Figure 13.5: The positional encoding matrix for $n = 10000$, $d = 512$, sequence length=100

The positional encoding layer sums the positional vector with the word encoding and outputs this matrix for the subsequent layers. The entire process is shown below.

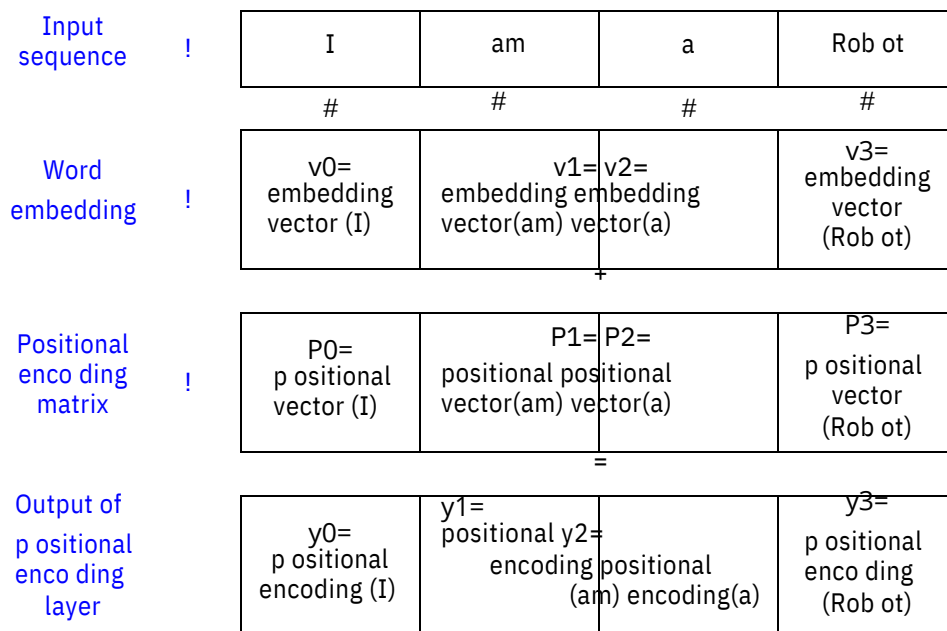


Figure 13.6: The positional encoding layer in the transformer

13.5 Further Reading

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

Books

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

Articles

Neural machine translation with a Transformer and Keras. TensorFlow Tutorials.

<https://www.tensorflow.org/text/tutorials/transformer>

13.6 Summary

In this chapter, you discovered positional encoding in transformers. Specifically, you learned:

- ◀ What is positional encoding, and why it is needed.
- ◀ How to implement positional encoding in Python using NumPy

◀ How to visualize the positional encoding matrix

In the next chapter, you will see how to apply positional encoding to text sequences.

Transformer Positional Encoding Layer in Keras

14

In Chapter 13, we discussed the positional encoding layer of the transformer model. We also showed how you could implement this layer and its functions yourself in Python.

In this chapter, you'll implement the positional encoding layer in Keras and TensorFlow. You can then use this layer in a complete transformer model. After completing this chapter, you will know:

- ◀ Text vectorization in Keras
- ◀ Embedding layer in Keras
 - ◀ How to subclass the embedding layer and write your own positional encoding layer.

Let's get started.

Overview

This chapter is divided into five parts; they are:

- ◀ The Text Vectorization Layer
- ◀ The Embedding Layer
 - ◀ Subclassing the Keras Embedding Layer
- ◀ Positional Encoding in Transformers
- ◀ Visualizing the Final Embedding

14.1 The Text Vectorization Layer

First, let's write the section to import all the required libraries:

```
import tensorflow as tf
from tensorflow import convert_to_tensor, string
from tensorflow.keras.layers import TextVectorization, Embedding, Layer
from tensorflow.data import Dataset
```



```
import numpy as np
import matplotlib.pyplot as plt
```

Listing 14.1: Libraries used

We'll start with a set of English phrases that are already preprocessed and cleaned. The text vectorization layer creates a dictionary of words and replaces each word with its corresponding index in the dictionary. Let's see how you can map these two sentences using the text vectorization layer:

1. *I am a robot*

2. *you too robot*

Note the text has already been converted to lowercase with all the punctuation marks and noise in the text removed. Next, convert these two phrases to vectors of a fixed length 5. The TextVectorization layer of Keras requires a maximum vocabulary size and the required length of output sequence for initialization. The output of the layer is a tensor of shape (number of sentences, output sequence length).

The following code snippet uses the `adapt` method to generate a vocabulary. It next creates a vectorized representation of the text.

```
output_sequence_length = 5
vocab_size = 10
sentences = [["I am a robot"], ["you too robot"]]
sentence_data = Dataset.from_tensor_slices(sentences)
# Create the TextVectorization layer
vectorize_layer = TextVectorization(output_sequence_length=output_sequence_length,
max_tokens=vocab_size)
# Train the layer to create a dictionary
vectorize_layer.adapt(sentence_data)
# Convert all sentences to tensors
word_tensors = convert_to_tensor(sentences, dtype=tf.string)
# Use the word tensors to get vectorized phrases
vectorized_words = vectorize_layer(word_tensors)
print("Vocabulary: ", vectorize_layer.get_vocabulary())
print("Vectorized words: ", vectorized_words)
```

Listing 14.2: Vectorize sentences

```
Vocabulary: ['', '[UNK]', 'robot', 'you', 'too', 'i', 'am', 'a'] Vectorized words:
tf.Tensor(
[[56720]
[3 4 2 0 0]], shape=(2, 5), dtype=int64)
```

Output 14.1: Vocabulary and vectorized words

14.2 The Embedding Layer

The Keras Embedding layer converts integers to dense vectors. This layer maps these integers to random numbers, which are later tuned during the training phase. However, you also have the option to set the mapping to some predefined weight values (shown later). To initialize

this layer, you need to specify the maximum value of an integer to map, along with the length of the output sequence.

The Word Embeddings

Let's see how the layer converts the vectorized_text to tensors.

```
output_length = 6
word_embedding_layer = Embedding(vocab_size, output_length)
embedded_words = word_embedding_layer(vectorized_words)
print(embedded_words)
```

Listing 14.3: Word embeddings

The output has been annotated with some comments, as shown below. Note that you will see a different output every time you run this code because the weights have been initialized randomly.

```
tf.Tensor(
[[[-0.01774162  0.01500502  0.04420716  0.01313546  0.00590974  0.04650447] ← I
 [-0.01465289  0.02831229 -0.03984444  0.04775101 -0.02865747 -0.0456514 ] ← am
 [ 0.03936621  0.03636128 -0.00989999 -0.02402222  0.04513135  0.00486727] ← a
 [-0.04708033 -0.02452066 -0.01620088 -0.00827144  0.00542567  0.00282087] ← robot
 [ 0.01176172 -0.00363815 -0.00991338  0.04611918  0.04991369 -0.02621592]]

[[[-0.04853251  0.02937819 -0.0437277  -0.03330444  0.00035688 -0.00555205] ← you
 [ 0.02585179 -0.03164214  0.03716465 -0.01983647 -0.03591436  0.00649483] ← too
 [-0.04708033 -0.02452066 -0.01620088 -0.00827144  0.00542567  0.00282087] ← robot
 [ 0.01176172 -0.00363815 -0.00991338  0.04611918  0.04991369 -0.02621592]
 [ 0.01176172 -0.00363815 -0.00991338  0.04611918  0.04991369 -0.02621592]]], shape=(2, 5, 6), dtype=float32)
```

Figure 14.1: Word embeddings. This output will be different every time you run the code because of the random numbers involved.

The Position Embeddings

You also need the embeddings for the corresponding positions. The maximum positions correspond to the output sequence length of the `TextVectorization` layer.

```
position_embedding_layer = Embedding(output_sequence_length,
output_length) position_indices = tf.range(output_sequence_length)
embedded_indices = position_embedding_layer(position_indices)
print(embedded_indices)
```

Listing 14.4: Position embeddings

The output is shown below:

```
tf.Tensor(
[[ 0.04834441  0.01801536 -0.04533539  0.0317518  0.00080795  0.04751191] ← Index 0
 [ 0.01603537  0.02398075 -0.04093184  0.03083241 -0.01965805  0.02442647] ← Index 1
 [-0.02292482 -0.02454183  0.04441524  0.04543227 -0.01491278 -0.03743547] ← Index 2
 [-0.0070316  0.03314704 -0.04718336 -0.03553554 -0.01260666 -0.01135062] ← Index 3
 [ 0.00522767 -0.00304329  0.02468617 -0.04479165 -0.04502536 -0.04907389]], ← Index 4
      shape=(5, 6), dtype=float32)
```

Figure 14.2: Position indices embedding

The Output of Positional Encoding Layer in Transformers

In a transformer model, the final output is the sum of both the word embeddings and the position embeddings. Hence, when you set up both embedding layers, you need to make sure that the output_length is the same for both.

```
final_output_embedding = embedded_words +
embedded_indices       print("Final output: ",
final_output_embedding)
```

Listing 14.5: Final output embeddings

The output is shown below, annotated with comments. Again, this will be different from your run of the code because of the random weight initialization.

```
Final output: tf.Tensor(
[[[ 0.03060279  0.03302038 -0.00112823  0.04488726  0.00671769 ← I
    0.09401638]
 [ 0.00138248  0.05229304 -0.08077629  0.07858343 -0.04831553 ← am
 -0.02122493]
 [ 0.0164414  0.01181945  0.03451525  0.02141005  0.03021857 ← a
 -0.03256821]
 [-0.05411192  0.00862638 -0.06338423 -0.04380698 -0.00718099 ← robot
 -0.00852975]
 [ 0.0169894 -0.00668144  0.01477278  0.00132753  0.00488833
 -0.07528981]]
 [[-0.0001881  0.04739355 -0.08906309 -0.00155264  0.00116483 ← you
    0.04195986]
 [ 0.04188716 -0.00766139 -0.00376719  0.01099594 -0.05557241 ← too
    0.0309213 ]
 [-0.07000514 -0.04906249  0.02821436  0.03716083 -0.00948711 ← robot
 -0.0346146 ]
 [ 0.00473013  0.02950889 -0.05709675  0.01058364  0.03730704
 -0.03756654]
 [ 0.0169894 -0.00668144  0.01477278  0.00132753  0.00488833
 -0.07528981]]], shape=(2, 5, 6), dtype=float32)
```

Figure 14.3: The final output after adding word embedding and position embedding

14.3 Subclassing the Keras Embedding Layer

When implementing a transformer model, you'll have to write your own position encoding layer. This is quite simple, as the basic functionality is already provided for you. This Keras example shows how you can subclass the Embedding layer to implement your own functionality. You can add more methods to it as you require.

```

class PositionEmbeddingLayer(Layer):
    def __init__(self, seq_length, vocab_size, output_dim, **kwargs):
        super().__init__(**kwargs)
        self.word_embedding_layer = Embedding(
            input_dim=vocab_size, output_dim=output_dim
        )
        self.position_embedding_layer = Embedding(
            input_dim=seq_length, output_dim=output_dim
        )

    def call(self, inputs):
        position_indices = tf.range(tf.shape(inputs)[-1])
        embedded_words = self.word_embedding_layer(inputs)
        embedded_indices = self.position_embedding_layer(position_indices)
        return embedded_words + embedded_indices

```

Listing 14.6: Creating a position embedding layer in Keras

Let's run this layer.

```

my_embedding_layer = PositionEmbeddingLayer(output_sequence_length,
                                             vocab_size, output_length)
embedded_layer_output = my_embedding_layer(vectorized_words)
print("Output from my_embedded_layer: ", embedded_layer_output)

```

Listing 14.7: Using the position embedding layer

```

Output from my_embedded_layer: tf.Tensor(
[[[ 0.06798736 -0.02821309 0.00571618 0.00314623 -0.03060734
  0.01111387]
  [-0.06097465 0.03966043 -0.05164248 0.06578685 0.03638128
 -0.03397174]
  [ 0.06715029 -0.02453769 0.02205854 0.01110986 0.02345785
  0.05879898]
  [-0.04625867 0.07500569 -0.05690887 -0.07615659 0.01962536
  0.00035865]
  [ 0.01423577 -0.03938593 -0.08625181 0.04841495 0.06951572
  0.08811047]]

  [[0.0163899 0.06895607-0.01131684 0.01810524-0.05857501
  0.01811318]
  [ 0.01915303 -0.0163289 -0.04133433 0.06810946 0.03736673
  0.04218033]
  [ 0.00795418 -0.00143972 -0.01627307 -0.00300788 -0.02759011
  0.09251165]
  [0.0028762 0.04526488-0.05222676-0.02007698 0.07879823
  0.00541583]
  [ 0.01423577 -0.03938593 -0.08625181 0.04841495 0.06951572
  0.08811047]]], shape=(2, 5, 6), dtype=float32)

```

Output 14.2: Output of the position embedding layer

14.4 Positional Encoding in Transformers

Note the above class creates an embedding layer that has trainable weights. Hence, the weights are initialized randomly and tuned in to the training phase. The authors of “Attention Is All You Need” have specified a positional encoding scheme, as shown below. You can read the full details in Chapter 13.

$$P_{k,i} = \sin\left(\frac{k}{n^{i/d}}\right)$$

$$P_{k,i+1} = \cos\left(\frac{k}{n^{i/d}}\right)$$

If you want to use the same positional encoding scheme, you can specify your own embedding matrix, as discussed in Chapter 13, which shows how to create your own embeddings in NumPy. When specifying the Embedding layer, you need to provide the positional encoding matrix as weights along with trainable=False. Let’s create another positional embedding class that does exactly this:

```
class PositionEmbeddingFixedWeights(Layer):
    def __init__(self, seq_length, vocab_size, output_dim, **kwargs):
        super().__init__(**kwargs)
        word_embedding_matrix = self.get_position_encoding(vocab_size, output_dim)
        pos_embedding_matrix = self.get_position_encoding(seq_length, output_dim)
        self.word_embedding_layer = Embedding(
            input_dim=vocab_size, output_dim=output_dim,
            weights=[word_embedding_matrix],
            trainable=False
        )
        self.position_embedding_layer = Embedding(
            input_dim=seq_length, output_dim=output_dim,
            weights=[pos_embedding_matrix],
            trainable=False
        )

    def get_position_encoding(self, seq_len, d, n=10000):
        P = np.zeros((seq_len, d))
        for k in range(seq_len):
            for i in range(int(d/2)):
                denominator = np.power(n, 2*i/d)
                P[k, 2*i] = np.sin(k/denominator)
                P[k, 2*i+1] = np.cos(k/denominator)
        return P

    def call(self, inputs):
        position_indices = tf.range(tf.shape(inputs)[-1])
        embedded_words = self.word_embedding_layer(inputs)
        embedded_indices = self.position_embedding_layer(position_indices)
        return embedded_words + embedded_indices
```

Listing 14.8: A different implementation of the positional embedding layer

Next, we set up everything to run this layer.

```
attnisallyouneed_embedding = PositionEmbeddingFixedWeights(output_sequence_length,
vocab_size, output_length)
attnisallyouneed_output = attnisallyouneed_embedding(vectorized_words)
print("Output from my_embedded_layer: ", attnisallyouneed_output)
```

Listing 14.9: Using the positional embedding layer

```
Output from my_embedded_layer: tf.Tensor(
[[[-0.9589243 1.2836622 0.23000172 1.9731903          0.01077196
1.9999421 ]
[0.56205547 1.5004725 0.3213085 1.9603932          0.0150806
1.9999142 ]
[1.566284 0.3377554 0.41192317 1.9433732          8
1.999877 ]
[1.0504174 -1.4061394 0.2314966 1.9860148          0.01938933
1.9999698 ]
[-0.7568025 0.3463564 0.18459873 1.982814          0.01077211
1.9999628 ]]
[[-0.7568025 0.3463564 0.18459873 1.982814          0.0086176
1.9999628 ]]
[[[0.14112 0.0100075 0.1387981 1.9903207          3
1.9999791 ]
[ 0.08466846 -0.11334133 0.23099795 1.9817369          0.0064632
1.9999605 ]
[1.8185948 -0.8322937 0.185397 1.9913884          6
1.9999814 ]
[0.14112 0.0100075 0.1387981 1.9903207          0.0107720
1.9999791 ]
[-0.7568025 0.3463564 0.18459873 1.982814          7
1.9999628 ]], shape=(2, 5, 6), dtype=float32)
0.00861771
0.0064632
6
0.0086176
```

Output 14.3: Output of the positional embedding layer

14.5 Visualizing the Final Embedding

In order to visualize the embeddings, let's take two bigger sentences: one technical and the other one just a quote. We'll set up the TextVectorization layer along with the positional encoding layer and see what the final output looks like.

```
technical_phrase = "to understand machine learning algorithms you need" +\
" to understand concepts such as gradient of a function "+\
"Hessians of a matrix and optimization etc"
wise_phrase = "patrick henry said give me liberty or give me death" +\
"when he addressed the second virginia convention in march"

total_vocabulary = 200
seq_length = 20
final_output_len = 50
phrase_vectorization_layer = TextVectorization(output_sequence_length=seq_length,
max_tokens=total_vocabulary)
# Learn the dictionary
phrase_vectorization_layer.adapt([technical_phrase, wise_phrase])
```

```
# Convert all sentences to tensors
phrase_tensors = convert_to_tensor([technical_phrase, wise_phrase],
dtype=tf.string)
# Use the word tensors to get vectorized phrases
vectorized_phrases = phrase_vectorization_layer(phrase_tensors)

random_weights_embedding_layer = PositionEmbeddingLayer(seq_length,
                                                         total_vocabulary,
                                                         final_output_len)
fixed_weights_embedding_layer = PositionEmbeddingFixedWeights(seq_length,
                                                             total_vocabulary,
                                                             final_output_len)

random_embedding = random_weights_embedding_layer(vectorized_phrases)
fixed_embedding = fixed_weights_embedding_layer(vectorized_phrases)
```

Listing 14.10: Test out the embedding

Now let's see what the random embeddings look like for both phrases.

```
fig = plt.figure(figsize=(15, 5))
title = ["Tech Phrase", "Wise Phrase"]
for i in range(2):
    ax = plt.subplot(1, 2, 1+i)
    matrix = tf.reshape(random_embedding[i, :, :], (seq_length, final_output_len))
    cax = ax.matshow(matrix)
    plt.gcf().colorbar(cax)
    plt.title(title[i], y=1.2)
fig.suptitle("Random Embedding")
plt.show()
```

Listing 14.11: Visualizing the random embedding

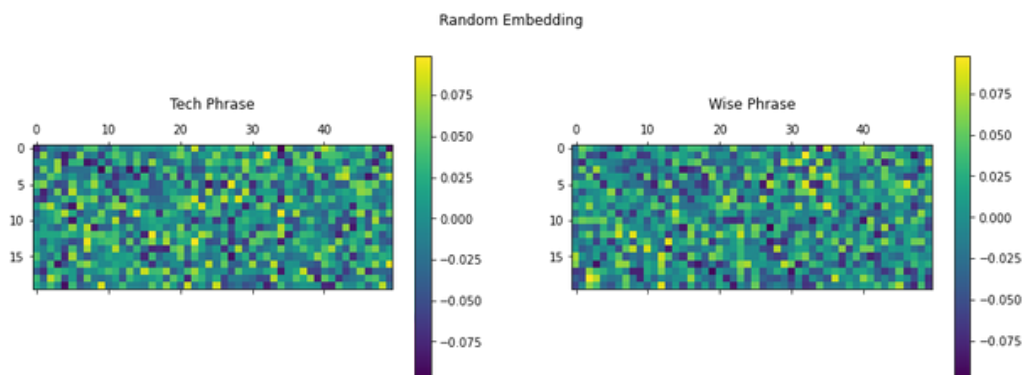


Figure 14.4: Random embeddings

The embedding from the fixed weights layer are visualized below.

```
fig = plt.figure(figsize=(15, 5))
title = ["Tech Phrase", "Wise Phrase"]
for i in range(2):
    ax = plt.subplot(1, 2, 1+i)
    matrix = tf.reshape(fixed_embedding[i, :, :], (seq_length, final_output_len))
```

```

cax = ax.matshow(matrix)
plt.gcf().colorbar(cax)
plt.title(title[i], y=1.2)
fig.suptitle("Fixed Weight Embedding from Attention is All You Need")
plt.show()

```

Listing 14.12: Visualizing the embedding with sinusoidal positional encoding

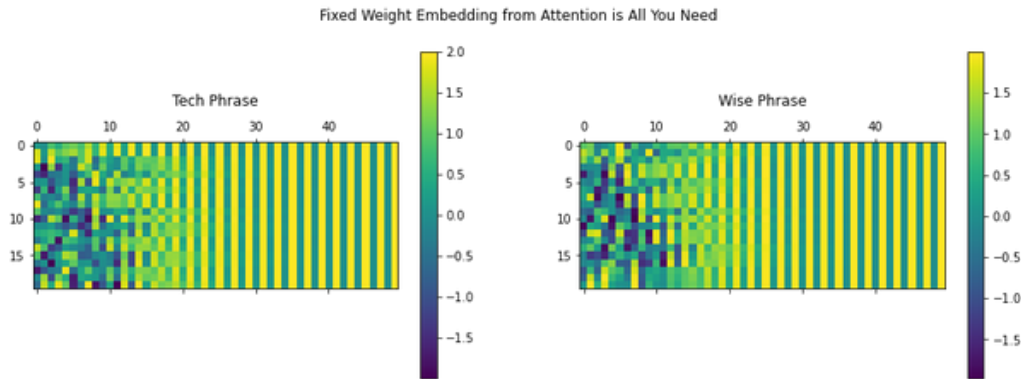


Figure 14.5: Embedding using sinusoidal positional encoding

You can see that the embedding layer initialized using the default parameter outputs random values. On the other hand, the fixed weights generated using sinusoids create a unique signature for every phrase with information on each word position encoded within it. You can experiment with tunable or fixed-weight implementations for your particular application.

14.6 Further Reading

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

Books

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

Articles

Neural machine translation with a Transformer and Keras. TensorFlow Tutorials.

<https://www.tensorflow.org/text/tutorials/transformer>

François Chollet. *English-to-Spanish translation with a sequence-to-sequence Transformer*. Keras code examples, 2021.

https://keras.io/examples/nlp/neural_machine_translation_with_transformer/

14.7 Summary

In this chapter, you discovered the implementation of positional encoding layer in Keras. Specifically, you learned:

- ◀ Text vectorization layer in Keras
- ◀ Positional encoding layer in Keras
- ◀ Creating your own class for positional encoding
- ◀ Setting your own weights for the positional encoding layer in Keras

In the next chapter, you will implement an attention layer in Keras.

Implementing Scaled Dot-Product Attention in Keras

15

Having familiarized ourselves with the theory behind the transformer model and its attention mechanism, we'll start our journey of implementing a complete transformer model by first seeing how to implement the scaled-dot product attention. The scaled dot-product attention is an integral part of the multi-head attention, which, in turn, is an important component of both the transformer encoder and decoder. Our end goal will be to apply the complete transformer model to natural language processing (NLP).

In this chapter, you will discover how to implement scaled dot-product attention from scratch in TensorFlow and Keras. After completing this chapter, you will know:

- ◀ The operations that form part of the scaled dot-product attention mechanism
- ◀ How to implement the scaled dot-product attention mechanism from scratch

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◀ Recap of the Transformer Architecture
 - ◀ Implementing the Scaled Dot-Product Attention from Scratch
- ◀ Testing Out the Code

15.1 Recap of the Transformer Architecture

Recall having seen that the transformer architecture follows an encoder-decoder structure. The encoder, on the left-hand side, is tasked with mapping an input sequence to a sequence of continuous representations; the decoder, on the right-hand side, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence.

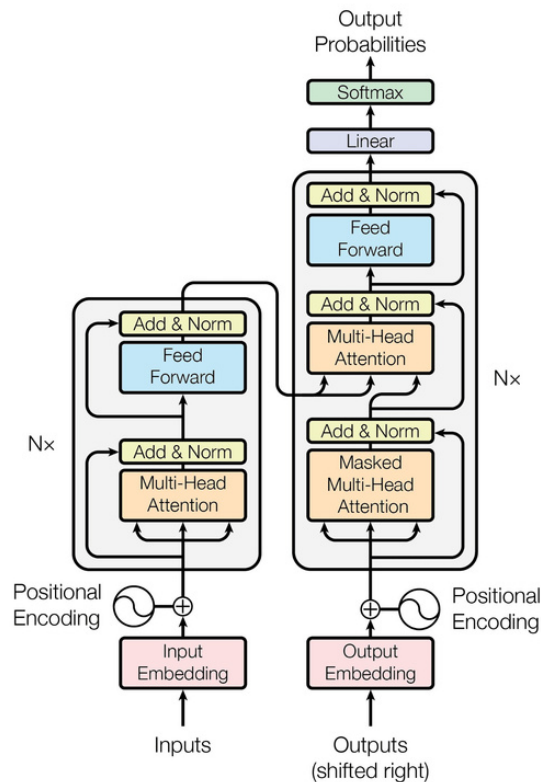


Figure 15.1: The encoder-decoder structure of the transformer architecture. From “Attention Is All You Need”

In generating an output sequence, the transformer does not rely on recurrence and convolutions.

You have seen that the decoder part of the transformer shares many similarities in its architecture with the encoder. One of the core components that both the encoder and decoder share within their multi-head attention blocks is the *scaled dot-product attention*.

First, recall the queries, keys, and values as the important components you will work with. In the encoder stage, they each carry the same input sequence after this has been embedded and augmented by positional information. Similarly, on the decoder side, the queries, keys, and values fed into the first attention block represent the same target sequence after this would have also been embedded and augmented by positional information. The second attention block of the decoder receives the encoder output in the form of keys and values and the normalized output of the first attention block as the queries. The dimensionality of the queries and keys is denoted by d_k , whereas the dimensionality of the values is denoted by d_v .

The scaled dot-product attention receives these queries, keys, and values as inputs and first computes the dot-product of the queries with the keys. The result is subsequently scaled by the square root of d_k , producing the attention scores. They are then fed into a softmax function, obtaining a set of attention weights. Finally, the attention weights are used to scale the values through a weighted multiplication operation. This entire process can be explained mathematically as follows, where \mathbf{Q} , \mathbf{K} , and \mathbf{V} denote the queries, keys, and values, respectively:

$$\text{attention}(\mathbf{Q}, \mathbf{K}, \mathbf{V}) = \text{softmax}\left(\frac{\mathbf{Q}\mathbf{K}^T}{\sqrt{d_k}}\right)\mathbf{V}$$

Each multi-head attention block in the transformer model implements a scaled dot-product attention operation as shown below:

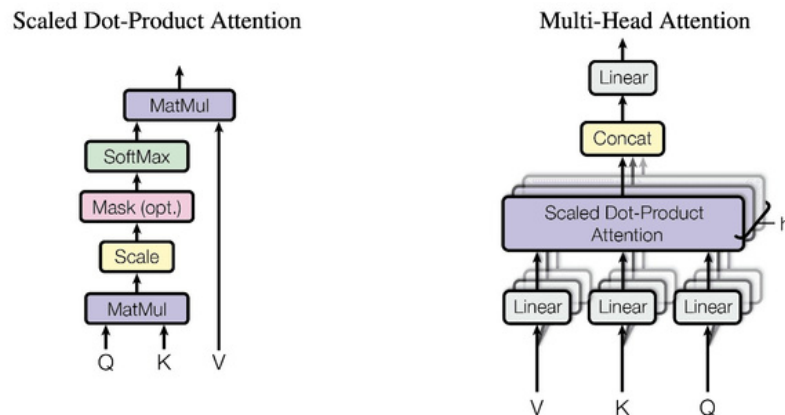


Figure 15.2: Scaled dot-product attention and multi-head attention. From “Attention Is All You Need”

You may note that the scaled dot-product attention can also apply a mask to the attention scores before feeding them into the softmax function.

Since the word embeddings are zero-padded to a specific sequence length, a *padding mask* needs to be introduced in order to prevent the zero tokens from being processed along with the input in both the encoder and decoder stages. Furthermore, a *look-ahead mask* is also required to prevent the decoder from attending to succeeding words, such that the prediction for a particular word can only depend on known outputs for the words that come before it.

These look-ahead and padding masks are applied inside the scaled dot-product attention set to $-\infty$ all the values in the input to the softmax function that should not be considered. For each of these large negative inputs, the softmax function will, in turn, produce an output value that is close to zero, effectively masking them out. The use of these masks will become clearer when you progress to the implementation of the encoder and decoder blocks in Chapters 17 and 18.

For the time being, let’s see how to implement the scaled dot-product attention from scratch in TensorFlow and Keras.

15.2 Implementing the Scaled Dot-Product Attention from Scratch

For this purpose, you will create a class called `DotProductAttention` that inherits from the `Layer` base class in Keras. In it, you will create the class method `call()` that takes as input arguments the queries, keys, and values, as well as the dimensionality `dk`, and a mask (that defaults to `None`):

```

class DotProductAttention(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

        def call(self, queries, keys, values, d_k, mask=None):
            ...

```

Listing 15.1: Building a dot-product attention layer

The first step is to perform a dot-product operation between the queries and the keys, transposing the latter. The result will be scaled through a division by the square root of d_k . You will add the following line of code to the `call()` class method:

```

...
scores = matmul(queries, keys, transpose_b=True) / sqrt(d_k)
...

```

Listing 15.2: Calculating dot-product scores using `matmul()` function

Next, you will check whether the mask argument has been set to a value that is not the default `None`. The mask will contain either 0 values to indicate that the corresponding token in the input sequence should be considered in the computations or a 1 to indicate otherwise. The mask will be multiplied by $-1e9$ to set the 1 values to large negative numbers (remember having mentioned this in the previous section), subsequently applied to the attention scores:

```

...
if mask is not None:
    scores += -1e9 * mask
...

```

Listing 15.3: Applying mask to scores

The attention scores will then be passed through a softmax function to generate the attention weights:

```

...
weights = softmax(scores) ...

```

Listing 15.4: Calculating the weight using softmax

The final step weights the values with the computed attention weights through another dot-product operation:

```

...
return matmul(weights, values)

```

Listing 15.5: Apply dot-product to calculate weighted values

The complete code listing is as follows:

```

from tensorflow import matmul, math, cast, float32
from tensorflow.keras.layers import Layer
from keras.backend import softmax

# Implementing the Scaled-Dot Product Attention
class DotProductAttention(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def call(self, queries, keys, values, d_k, mask=None):
        # Scoring the queries against the keys after transposing the latter, and scaling
        scores = matmul(queries, keys, transpose_b=True) / math.sqrt(cast(d_k, float32))

        # Apply mask to the attention scores
        if mask is not None:
            scores += -1e9 * mask

        # Computing the weights by a softmax operation
        weights = softmax(scores)

        # Computing the attention by a weighted sum of the value vectors
        return matmul(weights, values)

```

Listing 15.6: Complete code to implement dot-product attention layer

15.3 Testing Out the Code

We will be working with the parameter values specified in the paper, “Attention Is All You Need”, 2017:

```

d_k = 64 # Dimensionality of the linearly projected queries and keys d_v = 64
# Dimensionality of the linearly projected values
batch_size = 64 # Batch size from the training process
...

```

Listing 15.7: Parameters used from “Attention Is All You Need”

As for the sequence length and the queries, keys, and values, you will be working with dummy data for the time being until you arrive at the stage of training the complete transformer model, at which point you will use actual sentences. Similarly, for the mask, leave it set to its default value for the time being:

```

...
input_seq_length = 5 # Maximum length of the input sequence

queries = random.random((batch_size, input_seq_length, d_k))
keys = random.random((batch_size, input_seq_length, d_k))
values = random.random((batch_size, input_seq_length, d_v))
...

```

Listing 15.8: Generate random input

In the complete transformer model, values for the sequence length and the queries, keys, and values will be obtained through a process of word tokenization and embedding. You will be covering this in Chapter 20

Returning to the testing procedure, the next step is to create a new instance of the `DotProductAttention` class, assigning its output to the attention variable:

```
...
attention                                =
DotProductAttention() ...
```

Listing 15.9: Create an attention layer

Since the `DotProductAttention` class inherits from the `Layer` base class, the `call()` method of the former will be automatically invoked by the magic `__call()` method of the latter. The final step is to feed in the input arguments and print the result:

```
...
print(attention(queries, keys, values, d_k))
```

Listing 15.10: Printing output from the attention layer

Tying everything together produces the following code listing:

```
from numpy import random

input_seq_length = 5 # Maximum length of the input sequence
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
batch_size = 64 # Batch size from the training process

queries = random.random((batch_size, input_seq_length, d_k))
keys = random.random((batch_size, input_seq_length, d_k))
values = random.random((batch_size, input_seq_length, d_v))

attention = DotProductAttention()
print(attention(queries, keys, values, d_k))
```

Listing 15.11: Complete code to test out the attention layer

Running this code produces an output of shape (*batch size, sequence length, values dimensionality*). Note that you will likely see a different output due to the random initialization of the queries, keys, and values.

```
tf.Tensor(
[[[0.60413814 0.52436507 0.46551135 ... 0.5260341 0.33879933 0.43999898]
[0.60433316 0.52383804 0.465411 ... 0.5262608 0.33915892 0.43782598]
[0.623216030.5349194 0.46824688...0.531323 0.344320830.43554053]
[0.60013235 0.54162943 0.47391182 ... 0.53600514 0.33722004 0.4192218 ]
[0.6295709 0.53511244 0.46552944 ... 0.5317217 0.3462567 0.43129003]]
...

[[0.202910570.184639020.641182 ...0.4706118 0.4194418 0.39908117]
```

```
[0.19932748 0.18717204 0.64831126 ... 0.48373622 0.3995132 0.37968236]
[0.20611541 0.18079443 0.6374859 ... 0.48258874 0.41704425 0.4016996 ]
[0.19703123 0.18210654 0.6400498 ... 0.47037745 0.4257752 0.3962079 ]
[0.19237372 0.18474475 0.64944196 ... 0.49497223 0.38804317 0.36352912]]],
shape=(64, 5, 64), dtype=float32)
```

Output 15.1: Output of the attention layer

15.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

15.5 Summary

In this chapter, you discovered how to implement scaled dot-product attention from scratch in TensorFlow and Keras. Specifically, you learned:

- ◀ The operations that form part of the scaled dot-product attention mechanism
- ◀ How to implement the scaled dot-product attention mechanism from scratch

In the next chapter, you will extend the above into multi-head attention.

Implementing Multi-Head Attention in Keras

16

We have already familiarized ourselves with the theory behind the transformer model and its attention mechanism. We have already started our journey of implementing a complete model by seeing how to implement the scaled-dot product attention. We shall now progress one step further into our journey by encapsulating the scaled-dot product attention into a multi-head attention mechanism, which is a core component. Our end goal remains to apply the complete model to Natural Language Processing (NLP).

In this chapter, you will discover how to implement multi-head attention from scratch in TensorFlow and Keras. After completing this chapter, you will know:

- ◀ The layers that form part of the multi-head attention mechanism.
- ◀ How to implement the multi-head attention mechanism from scratch.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◀ Recap of Multi-Head Attention
 - ◀ Implementing Multi-Head Attention From Scratch
- ◀ Testing Out the Code

16.1 Recap of Multi-Head Attention

Recall from Section 15.1 that the transformer architecture follows an encoder-decoder structure. The encoder, on the left-hand side, is tasked with mapping an input sequence to a sequence of continuous representations; the decoder, on the right-hand side, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence. In generating an output sequence, the transformer does not rely on recurrence and convolutions.

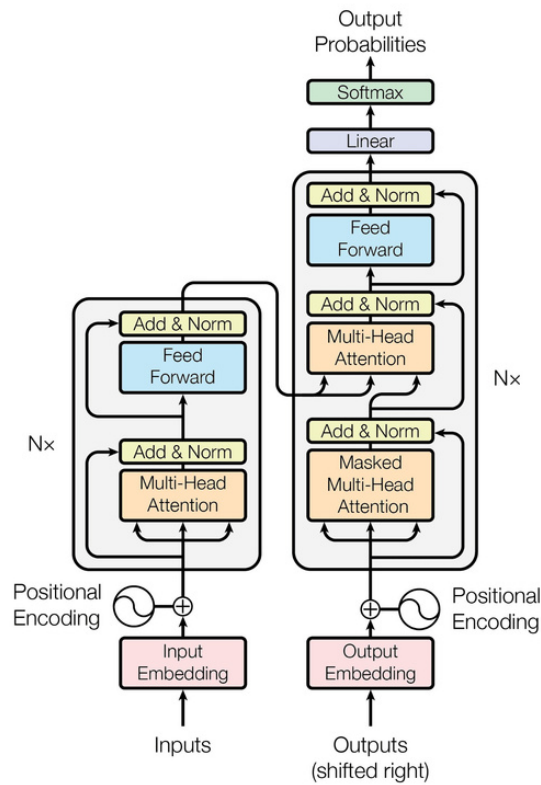


Figure 16.1: The encoder-decoder structure of the transformer architecture.
 “Attention Is All You Need”

From

You have seen that the decoder part of the transformer shares many similarities in its architecture with the encoder. One of the core mechanisms that both the encoder and decoder share is the *multi-head attention* mechanism. Each multi-head attention block is made up of four consecutive levels:

- ◁ On the first level, three linear (dense) layers that each receive the queries, keys, or values

- ◁ On the second level, a scaled dot-product attention function. The operations performed on both the first and second levels are repeated h times and performed in parallel, according to the number of heads composing the multi-head attention block

- ◁ On the third level, a concatenation operation that joins the outputs of the different heads

- ◁ On the fourth level, a final linear (dense) layer that produces the output

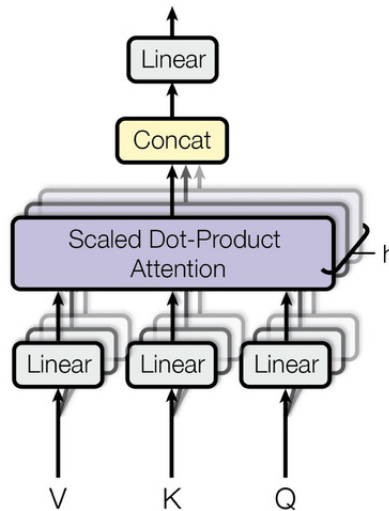


Figure 16.2: Multi-head attention. From “Attention Is All You Need”

Recall as well the important components that will serve as building blocks for your implementation of the multi-head attention:

◁ The *queries*, *keys*, and *values*: These are the inputs to each multi-head attention block. In the encoder stage, they each carry the same input sequence after this has been embedded and augmented by positional information. Similarly, on the decoder side, the queries, keys, and values fed into the first attention block represent the same target sequence after this would have also been embedded and augmented by positional information. The second attention block of the decoder receives the encoder output in the form of keys and values, and the normalized output of the first decoder attention block as the queries. The dimensionality of the queries and keys is denoted by d_k , whereas the dimensionality of the values is denoted by d_v .

◁ The *projection matrices*: When applied to the queries, keys, and values, these projection matrices generate different subspace representations of each. Each attention *head* then works on one of these projected versions of the queries, keys, and values. An additional projection matrix is also applied to the output of the multi-head attention block after the outputs of each individual head would have been concatenated together. The projection matrices are learned during training.

Let’s now see how to implement the multi-head attention from scratch in TensorFlow and Keras.

16.2 Implementing Multi-Head Attention from Scratch

Let’s start by creating the class, `MultiHeadAttention`, which inherits from the `Layer` base class in Keras and initialize several instance attributes that you shall be working with (attribute descriptions may be found in the comments):

```

class MultiHeadAttention(Layer):
    def __init__(self, h, d_k, d_v, d_model, **kwargs):
        super().__init__(**kwargs)
        self.attention = DotProductAttention() # Scaled dot product attention
        self.heads = h # Number of attention heads to use
            self.d_k = d_k # Dimensionality of the linearly projected queries and keys
        self.d_v = d_v # Dimensionality of the linearly projected values
        self.W_q = Dense(d_k) # Learned projection matrix for the queries
        self.W_k = Dense(d_k) # Learned projection matrix for the keys
        self.W_v = Dense(d_v) # Learned projection matrix for the values
            self.W_o = Dense(d_model) # Learned projection matrix for the multi-head output
    ...

```

Listing 16.1: Attributes for a multi-head attention layer

Here note that an instance of the DotProductAttention class that was implemented earlier has been created, and its output was assigned to the variable attention. Recall that we had implemented the DotProductAttention class as follows:

```

from tensorflow import matmul, math, cast, float32
from tensorflow.keras.layers import Layer
from tensorflow.keras.backend import softmax

# Implementing the Scaled-Dot Product Attention
class DotProductAttention(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def call(self, queries, keys, values, d_k, mask=None):
        # Scoring the queries against the keys after transposing the latter, and scaling
        scores = matmul(queries, keys, transpose_b=True) / math.sqrt(cast(d_k, float32))

        # Apply mask to the attention scores
        if mask is not None:
            scores += -1e9 * mask

        # Computing the weights by a softmax operation
        weights = softmax(scores)

        # Computing the attention by a weighted sum of the value vectors
        return matmul(weights, values)

```

Listing 16.2: The DotProductAttention class from Chapter 15

Next, you will be reshaping the *linearly projected* queries, keys, and values in such a manner as to allow the attention heads to be computed in parallel.

The queries, keys, and values will be fed as input into the multi-head attention block having a shape of (*batch size, sequence length, model dimensionality*), where the *batch size* is a hyperparameter of the training process, the *sequence length* defines the maximum length of the input/output phrases, and the *model dimensionality* is the dimensionality of the outputs produced by all sub-layers of the model. They are then passed through the respective dense layer to be linearly projected to a shape of (*batch size, sequence length, queries/keys/values dimensionality*).

The linearly projected queries, keys, and values will be rearranged into (*batch size, number of heads, sequence length, depth*), by first reshaping them into (*batch size, sequence length, number of heads, depth*) and then transposing the second and third dimensions. For this purpose, you will create the class method `reshape_tensor` as follows:

```
def reshape_tensor(self, x, heads, flag):
    if flag:
        # Tensor shape after reshaping and transposing:
        # (batch_size, heads, seq_length, -1)
        x = reshape(x, shape=(shape(x)[0], shape(x)[1], heads, -1))
        x = transpose(x, perm=(0, 2, 1, 3))
    else:
        # Reverting the reshaping and transposing operations:
        # (batch_size, seq_length, d_model)
        x = transpose(x, perm=(0, 2, 1, 3))
        x = reshape(x, shape=(shape(x)[0], shape(x)[1], -1))
    return x
```

Listing 16.3: A method in our multi-head attention class to convert tensors into a correct shape

The `reshape_tensor` method receives the linearly projected queries, keys, or values as input (while setting the flag to True) to be rearranged as previously explained. Once the multi-head attention output has been generated, this is also fed into the same function (this time setting the flag to False) to perform a reverse operation, effectively concatenating the results of all heads together. Hence, the next step is to feed the linearly projected queries, keys, and values into the `reshape_tensor` method to be rearranged, then feed them into the scaled dot-product attention function. In order to do so, let's create another class method call, as follows:

```
def call(self, queries, keys, values, mask=None):
    # Rearrange the queries to be able to compute all heads in parallel
    q_resaped = self.reshape_tensor(self.W_q(queries), self.heads, True)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)

    # Rearrange the keys to be able to compute all heads in parallel
    k_resaped = self.reshape_tensor(self.W_k(keys), self.heads, True)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)

    # Rearrange the values to be able to compute all heads in parallel
    v_resaped = self.reshape_tensor(self.W_v(values), self.heads, True)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)

    # Compute the multi-head attention output using the reshaped queries, keys,
    # and values
    o_resaped = self.attention(q_resaped, k_resaped, v_resaped, self.d_k, mask)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)
    ...
```

Listing 16.4: The `call` method for our multi-head attention class

Note that the `reshape_tensor` method can also receive a mask (whose value defaults to None) as input, in addition to the queries, keys, and values.

Recall that the transformer model introduces a *look-ahead mask* to prevent the decoder from attending to succeeding words, such that the prediction for a particular word can only depend on known outputs for the words that come before it. Furthermore, since the word embeddings are zero-padded to a specific sequence length, a *padding mask* also needs to be introduced to prevent the zero values from being processed along with the input. These look-ahead and padding masks can be passed on to the scaled-dot product attention through the mask argument.

Once you have generated the multi-head attention output from all the attention heads, the final steps are to concatenate back all outputs together into a tensor of shape (*batch size*, *sequence length*, *values dimensionality*) and passing the result through one final dense layer. For this purpose, you will add the next two lines of code to the call method.

```
...
# Rearrange back the output into concatenated form
output = self.reshape_tensor(o_resaped, self.heads, False)
# Resulting tensor shape: (batch_size, input_seq_length, d_v)

# Apply one final linear projection to the output to generate the multi-head #
attention. Resulting tensor shape: (batch_size, input_seq_length, d_model) return
self.W_o(output)
```

Listing 16.5: Returning a tensor in the `call()` method

Putting everything together, you have the following implementation of the multi-head attention:

```
from tensorflow import math, matmul, reshape, shape, transpose, cast, float32
from tensorflow.keras.layers import Dense, Layer
from tensorflow.keras.backend import softmax

# Implementing the Scaled-Dot Product Attention
class DotProductAttention(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def call(self, queries, keys, values, d_k, mask=None):
        # Scoring the queries against the keys after transposing the latter, and scaling
        scores = matmul(queries, keys, transpose_b=True) / math.sqrt(cast(d_k, float32))

        # Apply mask to the attention scores
        if mask is not None:
            scores += -1e9 * mask

        # Computing the weights by a softmax operation
        weights = softmax(scores)

        # Computing the attention by a weighted sum of the value vectors
        return matmul(weights, values)

# Implementing the Multi-Head Attention
class MultiHeadAttention(Layer):
```

```

def __init__(self, h, d_k, d_v, d_model, **kwargs):
    super().__init__(**kwargs)
    self.attention = DotProductAttention() # Scaled dot product attention
    self.heads = h # Number of attention heads to use
        self.d_k = d_k # Dimensionality of the linearly projected queries and keys
    self.d_v = d_v # Dimensionality of the linearly projected values
    self.d_model = d_model # Dimensionality of the model
    self.W_q = Dense(d_k) # Learned projection matrix for the queries
    self.W_k = Dense(d_k) # Learned projection matrix for the keys
    self.W_v = Dense(d_v) # Learned projection matrix for the values
        self.W_o = Dense(d_model) # Learned projection matrix for the multi-head output

def reshape_tensor(self, x, heads, flag):
    if flag:
        # Tensor shape after reshaping and transposing:
        # (batch_size, heads, seq_length, -1)
        x = reshape(x, shape=(shape(x)[0], shape(x)[1], heads, -1))
        x = transpose(x, perm=(0, 2, 1, 3))
    else:
        # Reverting the reshaping and transposing operations:
        # (batch_size, seq_length, d_k)
        x = transpose(x, perm=(0, 2, 1, 3))
        x = reshape(x, shape=(shape(x)[0], shape(x)[1], self.d_k))
    return x

def call(self, queries, keys, values, mask=None):
    # Rearrange the queries to be able to compute all heads in parallel
    q_resaped = self.reshape_tensor(self.W_q(queries), self.heads, True)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)

    # Rearrange the keys to be able to compute all heads in parallel
    k_resaped = self.reshape_tensor(self.W_k(keys), self.heads, True)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)

    # Rearrange the values to be able to compute all heads in parallel
    v_resaped = self.reshape_tensor(self.W_v(values), self.heads, True)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)

    # Compute the multi-head attention output using the reshaped queries,
    # keys, and values
    o_resaped = self.attention(q_resaped, k_resaped, v_resaped, self.d_k, mask)
    # Resulting tensor shape: (batch_size, heads, input_seq_length, -1)

    # Rearrange back the output into concatenated form
    output = self.reshape_tensor(o_resaped, self.heads, False)
    # Resulting tensor shape: (batch_size, input_seq_length, d_v)

    # Apply one final linear projection to the output to generate the multi-head
    # attention. Resulting tensor shape: (batch_size, input_seq_length, d_model)
    return self.W_o(output)

```

Listing 16.6: Implementation of the multi-head attention layer

16.3 Testing Out the Code

You will be working with the parameter values specified in the paper “Attention Is All You Need”, 2017:

```
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of the model sub-layers' outputs
batch_size = 64 # Batch size from the training process
...
```

Listing 16.7: Parameters from “Attention Is All You Need”

As for the sequence length and the queries, keys, and values, you will be working with dummy data for the time being until you arrive at the stage of training the complete transformer model in a later chapter, at which point you will be using actual sentences:

```
...
input_seq_length = 5 # Maximum length of the input sequence

queries = random.random((batch_size, input_seq_length, d_k))
keys = random.random((batch_size, input_seq_length, d_k))
values = random.random((batch_size, input_seq_length, d_v))
...
```

Listing 16.8: Generating random “sentences” for testing

In the complete transformer model, values for the sequence length and the queries, keys, and values will be obtained through a process of word tokenization and embedding. We will be covering this in a separate chapter.

Returning to the testing procedure, the next step is to create a new instance of the `MultiHeadAttention` class, assigning its output to the `multihead_attention` variable:

```
...
multihead_attention = MultiHeadAttention(h, d_k, d_v,
d_model) ...
```

Listing 16.9: Creating a multi-head attention layer

Since the `MultiHeadAttention` class inherits from the `Layer` base class, the `call()` method of the former will be automatically invoked by the magic `__call__` method of the latter. The final step is to pass in the input arguments and print the result:

```
...
print(multihead_attention(queries, keys, values))
```

Listing 16.10: Printing the output from the multi-head attention layer

Tying everything together produces the following code listing:


```

from numpy import random

input_seq_length = 5 # Maximum length of the input sequence
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys d_v = 64
# Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of the model sub-layers' outputs batch_size =
64 # Batch size from the training process

queries = random.random((batch_size, input_seq_length, d_k))
keys = random.random((batch_size, input_seq_length, d_k))
values = random.random((batch_size, input_seq_length, d_v))

multihead_attention = MultiHeadAttention(h, d_k, d_v, d_model)
print(multihead_attention(queries, keys, values))

```

Listing 16.11: Testing out the multi-head attention

Running this code produces an output of shape (*batch size, sequence length, model dimensionality*). Note that you will likely see a different output due to the random initialization of the queries, keys, and values and the parameter values of the dense layers.

```

tf.Tensor(
[[[-0.02185373 0.32784638 0.15958631 ... -0.0353895      0.6645204
 -0.2588266 ]
 [ -0.02272229 0.32292002 0.16208754 ... -0.03644213  0.66478664
 -0.26139447]
 [ -0.01876744 0.32900316 0.16190802 ... -0.03548665  0.6645842
 -0.26155376]
 [ -0.02193783 0.32687354 0.15801215 ... -0.03232524  0.6642926
 -0.25795174]
 [-0.02224652 0.32437912 0.1596448 ... -0.0340827
 -0.26065096]]
...

[[[0.05414441 0.27019292 0.1845745 ... 0.0809482      0.63738805
 -0.34231138]
 [ 0.05546578 0.27191412 0.18483458 ... 0.08379208  0.6366671
 -0.34372014]
 [ 0.05190979 0.27185103 0.18378328 ... 0.08341806  0.63851804
 -0.3422392 ]
 [ 0.05437043 0.27318984 0.18792395 ... 0.08043509  0.6391771
 -0.34357914]
 [ 0.05406848 0.27073097 0.18579456 ... 0.08388947
 -0.34230167]]], shape=(64, 5, 512), dtype=float32)

```

Output 16.1: Output of the multi-head attention

16.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

16.5 Summary

In this chapter, you discovered how to implement multi-head attention from scratch in TensorFlow and Keras. Specifically, you learned:

- ◀ The layers that form part of the multi-head attention mechanism
- ◀ How to implement the multi-head attention mechanism from scratch

In the next chapter, you will use the multi-head attention to build the encoder part of the transformer.

Implementing the Transformer Encoder in Keras

17

Having seen how to implement the scaled dot-product attention and integrate it within the multi-head attention of the transformer model, let's progress one step further toward implementing a complete transformer model by applying its encoder. Our end goal remains to apply the complete model to natural language processing (NLP).

In this chapter, you will discover how to implement the transformer encoder from scratch in TensorFlow and Keras. After completing this chapter, you will know:

- ◀ The layers that form part of the transformer encoder.
- ◀ How to implement the transformer encoder from scratch.

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◀ Recap of the Transformer Encoder
 - ◀ Implementing the Transformer Encoder from Scratch
- ◀ Testing Out the Code

17.1 Recap of the Transformer Encoder

Recall from Section 15.1 that the transformer architecture follows an encoder-decoder structure. The encoder, on the left-hand side, is tasked with mapping an input sequence to a sequence of continuous representations.

The transformer encoder consists of a stack of N identical layers, where each layer further consists of two main sub-layers:

- ◀ The first sub-layer comprises a multi-head attention mechanism that receives the queries, keys, and values as inputs.
- ◀ A second sub-layer comprises a fully-connected feedforward network.

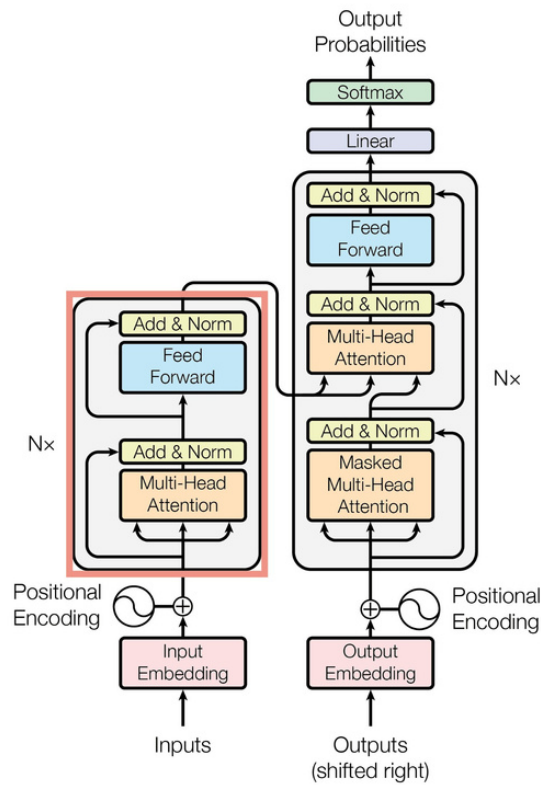


Figure 17.1: The encoder block of the transformer architecture. From “Attention Is All You Need”

Following each of these two sub-layers is layer normalization, into which the sub-layer input (through a residual connection) and output are fed. The output of each layer normalization step is the following:

$$\text{LayerNorm}(\text{Sublayer Input} + \text{Sublayer Output})$$

In order to facilitate such an operation, which involves an addition between the sublayer input and output. Vaswani et al. designed all sub-layers and embedding layers in the model to produce outputs of dimension $d_{\text{model}} = 512$.

Also, recall the queries, keys, and values as the inputs to the transformer encoder. Here, the queries, keys, and values carry the same input sequence after this has been embedded and augmented by positional information, where the queries and keys are of dimensionality d_k and the dimensionality of the values is d_v . Furthermore, Vaswani et al. also introduce regularization into the model by applying a dropout to the output of each sub-layer (before the layer normalization step), as well as to the positional encodings before these are fed into the encoder.

17.2 Implementing the Transformer Encoder from Scratch

Let’s now see how to implement the transformer encoder from scratch in TensorFlow and Keras.

The Feedforward Network and Layer Normalization

Let's begin by creating classes for the *Feed Forward* and *Add & Norm* layers that are shown in the diagram above.

Vaswani et al. tell us that the fully connected feedforward network consists of two linear transformations with a ReLU activation in between. The first linear transformation produces an output of dimensionality $d_{ff} = 2048$, while the second linear transformation produces an output of dimensionality $d_{model} = 512$. For this purpose, let's first create the class `FeedForward` that inherits from the `Layer` base class in Keras and initialize the `Dense` layers and the `ReLU` activation:

```
class FeedForward(Layer):
    def __init__(self, d_ff, d_model, **kwargs):
        super().__init__(**kwargs)
        self.fully_connected1 = Dense(d_ff) # First fully connected layer
        self.fully_connected2 = Dense(d_model) # Second fully connected layer
        self.activation = ReLU() # ReLU activation layer
    ...
```

Listing 17.1: A layer class for the feedforward part

We will add to it the class method `call()` that receives an input and passes it through the two fully connected layers with `ReLU` activation, returning an output of dimensionality equal to 512:

```
...
def call(self, x):
    # The input is passed into the two fully-connected layers, with a ReLU in between
    x_fc1 = self.fully_connected1(x)

    return self.fully_connected2(self.activation(x_fc1))
```

Listing 17.2: Implementing the `call()` method

The next step is to create another class that also inherits from the base class in Keras and initialize a `Layer normalization` layer:

```
class AddNormalization(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.layer_norm = LayerNormalization() # Layer normalization layer
    ...
```

Listing 17.3: Defining the normalization layer

In it, include the following class method that sums its sub-layer's input and output, which it receives as inputs, and applies layer normalization to the result:

```
...
def call(self, x, sublayer_x):
    # The sublayer input and output need to be of the same shape to be summed
    add = x + sublayer_x
```

```
# Apply layer normalization to the sum
return self.layer_norm(add)
```

Listing 17.4: The `call()` method in the normalization layer

The Encoder Layer

Next, you will implement the encoder layer, which the transformer encoder will replicate identically N times. For this purpose, let's create the class `EncoderLayer` and initialize all the sub-layers that it consists of:

```
class EncoderLayer(Layer):
def __init__(self, h, d_k, d_v, d_model, d_ff, rate, **kwargs):
super().__init__(**kwargs)
    self.multihead_attention = MultiHeadAttention(h, d_k, d_v, d_model)
self.dropout1 = Dropout(rate)
self.add_norm1 = AddNormalization()
self.feed_forward = FeedForward(d_ff, d_model)
self.dropout2 = Dropout(rate)
self.add_norm2 = AddNormalization()
...
```

Listing 17.5: Defining the encoder layer

Here, you may notice that you have initialized instances of the `FeedForward` and `AddNormalization` classes, which you just created in the previous section, and assigned their output to the respective variables, `feed_forward`, `add_norm1`, and `add_norm2`. The `Dropout` layer is self-explanatory, where `rate` defines the frequency at which the input units are set to 0. You created the `MultiHeadAttention` class in Chapter 16, and if you saved the code into a separate Python script, then do not forget to import it. Here assumed it is saved in a Python script named `multihead_attention.py` and thus you need to include the line of code “`from multihead_attention import MultiHeadAttention`”.

Let's now proceed to create the class method `call()` that implements all of the encoder sub-layers:

```
...
def call(self, x, padding_mask, training):
# Multi-head attention layer
multihead_output = self.multihead_attention(x, x, x, padding_mask)
# Expected output shape = (batch_size, sequence_length, d_model)

# Add in a dropout layer
multihead_output = self.dropout1(multihead_output, training=training)

# Followed by an Add & Norm layer
addnorm_output = self.add_norm1(x, multihead_output)
# Expected output shape = (batch_size, sequence_length, d_model)

# Followed by a fully connected layer
```

```

feedforward_output = self.feed_forward(addnorm_output)
# Expected output shape = (batch_size, sequence_length, d_model)

# Add in another dropout layer
feedforward_output = self.dropout2(feedforward_output, training=training)

# Followed by another Add & Norm layer
return self.add_norm2(addnorm_output, feedforward_output)

```

Listing 17.6: Connecting all sublayers in the `call()` method

In addition to the input data, the `call()` method can also receive a padding mask. As a brief reminder of what was said in Chapter 15, the *padding* mask is necessary to suppress the zero padding in the input sequence from being processed along with the actual input values. The same class method can receive a training flag which, when set to `True`, will only apply the Dropout layers during training.

The Transformer Encoder

The last step is to create a class for the transformer encoder, which should be named `Encoder`:

```

class Encoder(Layer):
    def __init__(self, vocab_size, sequence_length, h, d_k, d_v, d_model, d_ff, n, rate,
**kwargs):
    super().__init__(**kwargs)
    self.pos_encoding = PositionEmbeddingFixedWeights(sequence_length, vocab_size,
d_model)
    self.dropout = Dropout(rate)
    self.encoder_layer = [EncoderLayer(h, d_k, d_v, d_model, d_ff, rate)
for _ in range(n)]
...

```

Listing 17.7: Creating the encoder layer

The transformer encoder receives an input sequence after this would have undergone a process of word embedding and positional encoding. In order to compute the positional encoding, let's make use of the `PositionEmbeddingFixedWeights` class described by Chapter 14.

As you have similarly done in the previous sections, here, you will also create a class method `call()` that applies word embedding and positional encoding to the input sequence and feeds the result to `N` encoder layers:

```

...
def call(self, input_sentence, padding_mask, training):
# Generate the positional encoding
pos_encoding_output = self.pos_encoding(input_sentence)
# Expected output shape = (batch_size, sequence_length, d_model)

# Add in a dropout layer
x = self.dropout(pos_encoding_output, training=training)

# Pass on the positional encoded values to each encoder layer

```

```

    for i, layer in enumerate(self.encoder_layer):
        x = layer(x, padding_mask, training)

    return x

```

Listing 17.8: Implementing the `call()` method in the encoder layer

The code listing for the full transformer encoder is the following:

```

from tensorflow.keras.layers import LayerNormalization, Layer, Dense, ReLU, Dropout
from multihead_attention import MultiHeadAttention
from positional_encoding import PositionEmbeddingFixedWeights

# Implementing the Add & Norm Layer
class AddNormalization(Layer):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.layer_norm = LayerNormalization() # Layer normalization layer

    def call(self, x, sublayer_x):
        # The sublayer input and output need to be of the same shape to be summed
        add = x + sublayer_x

        # Apply layer normalization to the sum
        return self.layer_norm(add)

# Implementing the Feed-Forward Layer
class FeedForward(Layer):
    def __init__(self, d_ff, d_model, **kwargs):
        super().__init__(**kwargs)
        self.fully_connected1 = Dense(d_ff) # First fully connected layer
        self.fully_connected2 = Dense(d_model) # Second fully connected layer
        self.activation = ReLU() # ReLU activation layer

    def call(self, x):
        # The input is passed into the two fully-connected layers, with a ReLU in between
        x_fc1 = self.fully_connected1(x)

        return self.fully_connected2(self.activation(x_fc1))

# Implementing the Encoder Layer
class EncoderLayer(Layer):
    def __init__(self, h, d_k, d_v, d_model, d_ff, rate, **kwargs):
        super().__init__(**kwargs)
        self.multihead_attention = MultiHeadAttention(h, d_k, d_v, d_model)
        self.dropout1 = Dropout(rate)
        self.add_norm1 = AddNormalization()
        self.feed_forward = FeedForward(d_ff, d_model)
        self.dropout2 = Dropout(rate)
        self.add_norm2 = AddNormalization()

    def call(self, x, padding_mask, training):
        # Multi-head attention layer
        multihead_output = self.multihead_attention(x, x, x, padding_mask)

```



```

# Expected output shape = (batch_size, sequence_length, d_model)

# Add in a dropout layer
multihead_output = self.dropout1(multihead_output, training=training)

# Followed by an Add & Norm layer
addnorm_output = self.add_norm1(x, multihead_output)
# Expected output shape = (batch_size, sequence_length, d_model)

# Followed by a fully connected layer
feedforward_output = self.feed_forward(addnorm_output)
# Expected output shape = (batch_size, sequence_length, d_model)

# Add in another dropout layer
feedforward_output = self.dropout2(feedforward_output, training=training)

# Followed by another Add & Norm layer
return self.add_norm2(addnorm_output, feedforward_output)

# Implementing the Encoder
class Encoder(Layer):
    def __init__(self, vocab_size, sequence_length, h, d_k, d_v, d_model, d_ff, n, rate,
**kwargs):
        super().__init__(**kwargs)
        self.pos_encoding = PositionEmbeddingFixedWeights(sequence_length, vocab_size,
d_model)
        self.dropout = Dropout(rate)
        self.encoder_layer = [EncoderLayer(h, d_k, d_v, d_model, d_ff, rate)
for _ in range(n)]

    def call(self, input_sentence, padding_mask, training):
        # Generate the positional encoding
        pos_encoding_output = self.pos_encoding(input_sentence)
        # Expected output shape = (batch_size, sequence_length, d_model)

        # Add in a dropout layer
        x = self.dropout(pos_encoding_output, training=training)

        # Pass on the positional encoded values to each encoder layer
        for i, layer in enumerate(self.encoder_layer):
            x = layer(x, padding_mask, training)

        return x

```

Listing 17.9: Complete code for the transformer encoder

17.3 Testing Out the Code

You will work with the parameter values specified in the paper “Attention Is All You Need”, 2017:

```

h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_ff = 2048 # Dimensionality of the inner fully connected layer
d_model = 512 # Dimensionality of the model sub-layers' outputs
n = 6 # Number of layers in the encoder stack

batch_size = 64 # Batch size from the training process
dropout_rate = 0.1 # Frequency of dropping the input units in the dropout layers ...

```

Listing 17.10: Parameters from “Attention Is All You Need”

As for the input sequence, you will work with dummy data for the time being until you arrive at the stage of training the complete transformer model in Chapter 20, at which point you will be using actual sentences:

```

...
enc_vocab_size = 20 # Vocabulary size for the encoder
input_seq_length = 5 # Maximum length of the input sequence

input_seq = random.random((batch_size, input_seq_length))
...

```

Listing 17.11: Generating random input for testing

Next, you will create a new instance of the Encoder class, assigning its output to the encoder variable, subsequently feeding in the input arguments, and printing the result. You will set the padding mask argument to None for the time being, but you will return to this when you implement the complete transformer model:

```

...
encoder = Encoder(enc_vocab_size, input_seq_length, h, d_k, d_v, d_model, d_ff, n,
                  dropout_rate)
print(encoder(input_seq, None, True))

```

Listing 17.12: Running the encoder layer

Tying everything together produces the following code listing:

```

from numpy import random

enc_vocab_size = 20 # Vocabulary size for the encoder
input_seq_length = 5 # Maximum length of the input sequence
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_ff = 2048 # Dimensionality of the inner fully connected layer
d_model = 512 # Dimensionality of the model sub-layers' outputs
n = 6 # Number of layers in the encoder stack

batch_size = 64 # Batch size from the training process
dropout_rate = 0.1 # Frequency of dropping the input units in the dropout layers

```

```
input_seq = random.random((batch_size, input_seq_length))

encoder = Encoder(enc_vocab_size, input_seq_length, h, d_k, d_v, d_model, d_ff, n,
                  dropout_rate)
print(encoder(input_seq, None, True))
```

Listing 17.13: Complete test code for the encoder layer

Running this code produces an output of shape (*batch size, sequence length, model dimensionality*). Note that you will likely see a different output due to the random initialization of the input sequence and the parameter values of the Dense layers.

```
tf.Tensor(
[[[-0.4214715 -1.1246173 -0.8444572 ... 1.6388322          -0.1890367
 1.0173352 ]
 [ 0.21662089 -0.61147404 -1.0946581 ... 1.4627445 -0.6000164
 -0.64127874]
 [ 0.46674493 -1.4155326 -0.5686513 ... 1.1790234 -0.94788337
 0.1331717 ]
 [-0.30638126 -1.9047263 -1.8556844 ... 0.9130118 -0.47863355
 0.00976158]
 [-0.22600567 -0.9702025 -0.91090447... 1.7457147 -0.139926
 -0.07021569]]
...

 [[-0.48047638 -1.1034104 -0.16164204... 1.5588069  0.08743562
 -0.08847156]
 [-0.61683714 -0.8403657 -1.0450369 ... 2.3587787 -0.76091915
 -0.02891812]
 [-0.34268388 -0.65042275 -0.6715749 ... 2.8530657 -0.33631966
 0.5215888 ]
 [-0.6288677 -1.0030932 -0.9749813 ... 2.1386387  0.0640307
 -0.69504136]
 [-1.33254 -1.2524267 -0.230098 ... 2.515467 -0.04207756
 -0.3395423 ]], shape=(64, 5, 512), dtype=float32)
```

Output 17.1: Output from the encoder layer

17.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

17.5 Summary

In this chapter, you discovered how to implement the transformer encoder from scratch in TensorFlow and Keras. Specifically, you learned:

- ◁ The layers that form part of the transformer encoder
- ◁ How to implement the transformer encoder from scratch

In the next chapter, you will implement the decoder part of the transformer.

Implementing the Transformer Decoder in Keras

18

There are many similarities between the transformer encoder and decoder, such as their implementation of multi-head attention, layer normalization and a fully connected feedforward network as their final sub-layer. Having implemented the transformer encoder, we will now go ahead and apply our knowledge in implementing the transformer decoder as a further step toward implementing the complete transformer model. Your end goal remains to apply the complete model to natural language processing (NLP).

In this chapter, you will discover how to implement the transformer decoder from scratch in TensorFlow and Keras. After completing this chapter, you will know:

- ◁ The layers that form part of the transformer decoder
- ◁ How to implement the transformer decoder from scratch

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◁ Recap of the Transformer Decoder
 - ◁ Implementing the Transformer Decoder From Scratch
- ◁ Testing Out the Code

18.1 Recap of the Transformer Decoder

Recall from Section 15.1 that the transformer architecture follows an encoder-decoder structure. The decoder, on the right-hand side, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence. In generating an output sequence, the transformer does not rely on recurrence and convolutions.

You have seen that the decoder part of the transformer shares many similarities in its architecture with the encoder. This chapter will explore these similarities.

Similar to the transformer encoder, the transformer decoder also consists of a stack of N identical layers. The transformer decoder, however, implements an additional multi-head attention block for a total of three main sub-layers:

- ◁ The first sub-layer comprises a multi-head attention mechanism that receives the queries, keys, and values as inputs
- ◁ The second sub-layer comprises a second multi-head attention mechanism
- ◁ The third sub-layer comprises a fully-connected feedforward network

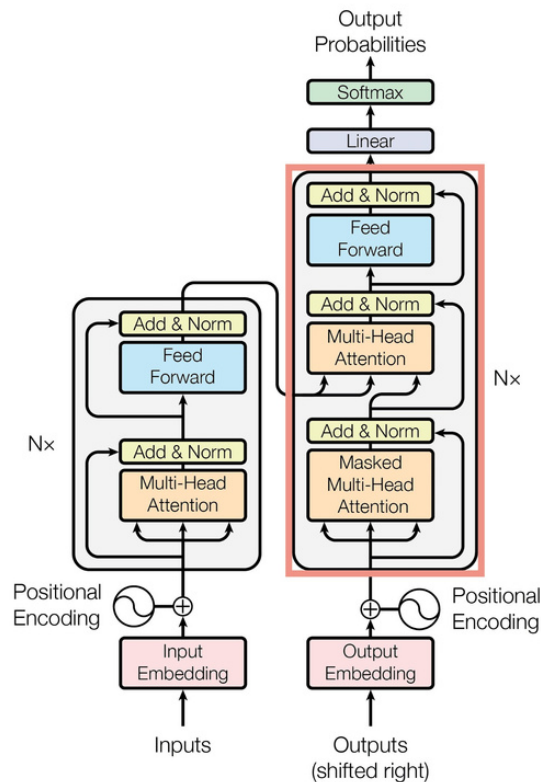


Figure 18.1: The decoder block of the transformer architecture. From “Attention Is All You Need”

Each one of these three sub-layers is also followed by layer normalization, where the input to the layer normalization step is its corresponding sub-layer input (through a residual connection) and output.

On the decoder side, the queries, keys, and values that are fed into the first multi-head attention block also represent the same input sequence. However, this time round it is the *target* sequence that is embedded and augmented with positional information before being supplied to the decoder. On the other hand, the second multi-head attention block receives the encoder output in the form of keys and values and the normalized output of the first decoder attention block as the queries. In both cases, the dimensionality of the queries and keys remains equal to d_k , whereas the dimensionality of the values remains equal to d_v .

Vaswani et al. introduce regularization into the model on the decoder side, too, by applying dropout to the output of each sub-layer (before the layer normalization step), as well as to the

positional encodings before these are fed into the decoder. Let's now see how to implement the transformer decoder from scratch in TensorFlow and Keras.

18.2 Implementing the Transformer Decoder from Scratch

Since you have already implemented the required sub-layers when you covered the implementation of the transformer encoder, you will create a class for the decoder layer that makes use of these sub-layers straight away:

```
from multihead_attention import MultiHeadAttention
from encoder import AddNormalization, FeedForward

class DecoderLayer(Layer):
    def __init__(self, h, d_k, d_v, d_model, d_ff, rate, **kwargs):
        super().__init__(**kwargs)
        self.multihead_attention1 = MultiHeadAttention(h, d_k, d_v, d_model)
        self.dropout1 = Dropout(rate)
        self.add_norm1 = AddNormalization()
        self.multihead_attention2 = MultiHeadAttention(h, d_k, d_v, d_model)
        self.dropout2 = Dropout(rate)
        self.add_norm2 = AddNormalization()
        self.feed_forward = FeedForward(d_ff, d_model)
        self.dropout3 = Dropout(rate)
        self.add_norm3 = AddNormalization()
    ...
```

Listing 18.1: Defining the decoder layer

Notice here that since the code for the different sub-layers had been saved into several Python scripts (namely, `multihead_attention.py` and `encoder.py`), it was necessary to import them to be able to use the required classes. As you did for the transformer encoder, you will now create the class method `call()` that implements all the decoder sub-layers:

```
...
def call(self, x, encoder_output, lookahead_mask, padding_mask, training):
    # Multi-head attention layer
    multihead_output1 = self.multihead_attention1(x, x, x, lookahead_mask)
    # Expected output shape = (batch_size, sequence_length, d_model)

    # Add in a dropout layer
    multihead_output1 = self.dropout1(multihead_output1, training=training)

    # Followed by an Add & Norm layer
    addnorm_output1 = self.add_norm1(x, multihead_output1)
    # Expected output shape = (batch_size, sequence_length, d_model)

    # Followed by another multi-head attention layer
    multihead_output2 = self.multihead_attention2(addnorm_output1, encoder_output,
encoder_output, padding_mask)

    # Add in another dropout layer
```

✱

```

multihead_output2 = self.dropout2(multihead_output2, training=training)

# Followed by another Add & Norm layer
addnorm_output2 = self.add_norm1(addnorm_output1, multihead_output2)

# Followed by a fully connected layer
feedforward_output = self.feed_forward(addnorm_output2)
# Expected output shape = (batch_size, sequence_length, d_model)

# Add in another dropout layer
feedforward_output = self.dropout3(feedforward_output, training=training)

# Followed by another Add & Norm layer
return self.add_norm3(addnorm_output2, feedforward_output)

```

Listing 18.2: Implementing the `call()` method in the decoder layer

The multi-head attention sub-layers can also receive a padding mask or a look-ahead mask. As a brief reminder of what was said in Chapter 15, the *padding* mask is necessary to suppress the zero padding in the input sequence from being processed along with the actual input values. The *look-ahead* mask prevents the decoder from attending to succeeding words, such that the prediction for a particular word can only depend on known outputs for the words that come before it.

The same `call()` class method can also receive a training flag to only apply the Dropout layers during training when the flag's value is set to True.

The Transformer Decoder

The transformer decoder takes the decoder layer you have just implemented and replicates it identically N times. You will create the following `Decoder()` class to implement the transformer decoder:

```

from positional_encoding import PositionEmbeddingFixedWeights

class Decoder(Layer):
    def __init__(self, vocab_size, sequence_length, h, d_k, d_v, d_model, d_ff, n, rate
    **kwargs):
        super().__init__(**kwargs)
        self.pos_encoding = PositionEmbeddingFixedWeights(sequence_length, vocab_size,
        d_model)
        self.dropout = Dropout(rate)
        self.decoder_layer = [DecoderLayer(h, d_k, d_v, d_model, d_ff, rate)
        for _ in range(n)
        ...

```

Listing 18.3: Defining the decoder class

As in the transformer encoder, the input to the first multi-head attention block on the decoder side receives the input sequence after this would have undergone a process of word embedding and positional encoding. For this purpose, an instance of the `PositionEmbeddingFixedWeights` class (covered in Chapter 14) is initialized and its output assigned to the `pos_encoding` variable.

The final step is to create a class method, `call()`, that applies word embedding and positional encoding to the input sequence and feeds the result, together with the encoder output, to N decoder layers:

```
...
def call(self, output_target, encoder_output, lookahead_mask, padding_mask, training):
    # Generate the positional encoding
    pos_encoding_output = self.pos_encoding(output_target)
    # Expected output shape = (number of sentences, sequence_length, d_model)

    # Add in a dropout layer
    x = self.dropout(pos_encoding_output, training=training)

    # Pass on the positional encoded values to each encoder layer
    for i, layer in enumerate(self.decoder_layer):
        x = layer(x, encoder_output, lookahead_mask, padding_mask, training)

    return x
```

Listing 18.4: Implementing the `call()` method for the decoder class

The code listing for the full transformer decoder is the following:

```
from tensorflow.keras.layers import Layer, Dropout
from multihead_attention import MultiHeadAttention
from positional_encoding import PositionEmbeddingFixedWeights
from encoder import AddNormalization, FeedForward

# Implementing the Decoder Layer
class DecoderLayer(Layer):
    def __init__(self, h, d_k, d_v, d_model, d_ff, rate, **kwargs):
        super().__init__(**kwargs)
        self.multihead_attention1 = MultiHeadAttention(h, d_k, d_v, d_model)
        self.dropout1 = Dropout(rate)
        self.add_norm1 = AddNormalization()
        self.multihead_attention2 = MultiHeadAttention(h, d_k, d_v, d_model)
        self.dropout2 = Dropout(rate)
        self.add_norm2 = AddNormalization()
        self.feed_forward = FeedForward(d_ff, d_model)
        self.dropout3 = Dropout(rate)
        self.add_norm3 = AddNormalization()

    def call(self, x, encoder_output, lookahead_mask, padding_mask, training):
        # Multi-head attention layer
        multihead_output1 = self.multihead_attention1(x, x, x, lookahead_mask)
        # Expected output shape = (batch_size, sequence_length, d_model)

        # Add in a dropout layer
        multihead_output1 = self.dropout1(multihead_output1, training=training)

        # Followed by an Add & Norm layer
        addnorm_output1 = self.add_norm1(x, multihead_output1)
        # Expected output shape = (batch_size, sequence_length, d_model)
```

```

# Followed by another multi-head attention layer
multihead_output2 = self.multihead_attention2(addnorm_output1, encoder_output,
encoder_output, padding_mask)

# Add in another dropout layer
multihead_output2 = self.dropout2(multihead_output2, training=training)

# Followed by another Add & Norm layer
addnorm_output2 = self.add_norm1(addnorm_output1, multihead_output2)

# Followed by a fully connected layer
feedforward_output = self.feed_forward(addnorm_output2)
# Expected output shape = (batch_size, sequence_length, d_model)

# Add in another dropout layer
feedforward_output = self.dropout3(feedforward_output, training=training)

# Followed by another Add & Norm layer
return self.add_norm3(addnorm_output2, feedforward_output)

# Implementing the Decoder
class Decoder(Layer):
def __init__(self, vocab_size, sequence_length, h, d_k, d_v, d_model, d_ff, n, rate,
**kwargs):
super().__init__(**kwargs)
self.pos_encoding = PositionEmbeddingFixedWeights(sequence_length, vocab_size,
d_model)
self.dropout = Dropout(rate)
self.decoder_layer = [DecoderLayer(h, d_k, d_v, d_model, d_ff, rate)
for _ in range(n)]

    def call(self, output_target, encoder_output, lookahead_mask, padding_mask, training):
# Generate the positional encoding
pos_encoding_output = self.pos_encoding(output_target)
# Expected output shape = (number of sentences, sequence_length, d_model)

# Add in a dropout layer
x = self.dropout(pos_encoding_output, training=training)

# Pass on the positional encoded values to each encoder layer
for i, layer in enumerate(self.decoder_layer):
x = layer(x, encoder_output, lookahead_mask, padding_mask, training)

return x

```

Listing 18.5: Complete code for the transformer decoder

18.3 Testing Out the Code

You will work with the parameter values specified in the paper “Attention Is All You Need”, 2017:

```

h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_ff = 2048 # Dimensionality of the inner fully connected layer
d_model = 512 # Dimensionality of the model sub-layers' outputs
n = 6 # Number of layers in the encoder stack

batch_size = 64 # Batch size from the training process
dropout_rate = 0.1 # Frequency of dropping the input units in the dropout layers ...

```

Listing 18.6: Parameters from “Attention Is All You Need”

As for the input sequence, you will work with dummy data for the time being until you arrive at the stage of training the complete transformer model in a separate chapter, at which point you will use actual sentences:

```

...
dec_vocab_size = 20 # Vocabulary size for the decoder input_seq_length
= 5 # Maximum length of the input sequence

input_seq = random.random((batch_size, input_seq_length))
enc_output = random.random((batch_size, input_seq_length,
d_model)) ...

```

Listing 18.7: Creating random input to test the decoder

Next, you will create a new instance of the Decoder class, assigning its output to the decoder variable, subsequently passing in the input arguments, and printing the result. You will set the padding and look-ahead masks to None for the time being, but you will return to these when you implement the complete transformer model:

```

...
decoder = Decoder(dec_vocab_size, input_seq_length, h, d_k, d_v, d_model, d_ff, n,
dropout_rate)
print(decoder(input_seq, enc_output, None, True))

```

Listing 18.8: Running the decoder

Tying everything together produces the following code listing:

```

from numpy import random

dec_vocab_size = 20 # Vocabulary size for the decoder input_seq_length = 5
# Maximum length of the input sequence
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys d_v =
64 # Dimensionality of the linearly projected values
d_ff = 2048 # Dimensionality of the inner fully connected layer d_model =
512 # Dimensionality of the model sub-layers' outputs
n = 6 # Number of layers in the decoder stack

batch_size = 64 # Batch size from the training process

```

```
dropout_rate = 0.1 # Frequency of dropping the input units in the dropout layers

input_seq = random.random((batch_size, input_seq_length))
enc_output = random.random((batch_size, input_seq_length, d_model))

decoder = Decoder(dec_vocab_size, input_seq_length, h, d_k, d_v, d_model, d_ff, n,
                  dropout_rate)
print(decoder(input_seq, enc_output, None, True))
```

Listing 18.9: Complete code to test the decoder

Running this code produces an output of shape (*batch size, sequence length, model dimensionality*). Note that you will likely see a different output due to the random initialization of the input sequence and the parameter values of the Dense layers.

```
tf.Tensor(
[[[-0.04132953-1.7236308 0.5391184 ...-0.76394725          1.4969798
 0.37682498]
 [ 0.05501875 -1.7523409 0.58404493 ... -0.70776534   1.4498456
 0.32555297]
 [ 0.04983566 -1.8431275 0.55850077 ... -0.68202156   1.4222856
 0.32104644]
 [-0.05684051-1.8862512 0.4771412 ...-0.7101341          1.431343
 0.39346313]
 [-0.15625843-1.7992781 0.40803364...-0.75190556   1.4602519
 0.53546077]]
...

[[[-0.58847624-1.646842 0.5973466...-0.47778523   1.2060764
 0.34091905]
 [-0.48688865-1.6809179 0.6493542 ...-0.41274604   1.188649
 0.27100053]
 [-0.49568555-1.8002801 0.61536175...-0.38540334   1.2023914
 0.24383534]
 [-0.59913146-1.8598882 0.5098136 ...-0.3984461          1.2115746
 0.3186561 ]
 [-0.71045107-1.7778647 0.43008155...-0.42037937   1.2255307
 0.47380894]]], shape=(64, 5, 512), dtype=float32)
```

Output 18.1: Output from the decoder

18.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

18.5 Summary

In this chapter, you discovered how to implement the transformer decoder from scratch in TensorFlow and Keras. Specifically, you learned:

- ◁ The layers that form part of the transformer decoder
- ◁ How to implement the transformer decoder from scratch

In the next chapter, you will combine the encoder and decoder to finish a transformer.

Joining the Transformer Encoder and Decoder with Masking

19

We have arrived at a point where we have implemented and tested the transformer encoder and decoder separately, and we may now join the two together into a complete model. We will also see how to create padding and look-ahead masks by which we will suppress the input values that will not be considered in the encoder or decoder computations. Our end goal remains to apply the complete model to natural language processing (NLP).

In this chapter, you will discover how to implement the complete transformer model and create padding and look-ahead masks. After completing this chapter, you will know:

- ◀ How to create a padding mask for the encoder and decoder
- ◀ How to create a look-ahead mask for the decoder
- ◀ How to join the transformer encoder and decoder into a single model
- ◀ How to print out a summary of the encoder and decoder layers

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ◀ Recap of the Transformer Architecture
- ◀ Masking
 - ◀ Joining the Transformer Encoder and Decoder
 - ◀ Creating an Instance of the Transformer Model

19.1 Recap of the Transformer Architecture

Recall from Section 15.1 that the transformer architecture follows an encoder-decoder structure. The encoder, on the left-hand side, is tasked with mapping an input sequence to a sequence of continuous representations; the decoder, on the right-hand side, receives the output of the encoder together with the decoder output at the previous time step to generate an output sequence.

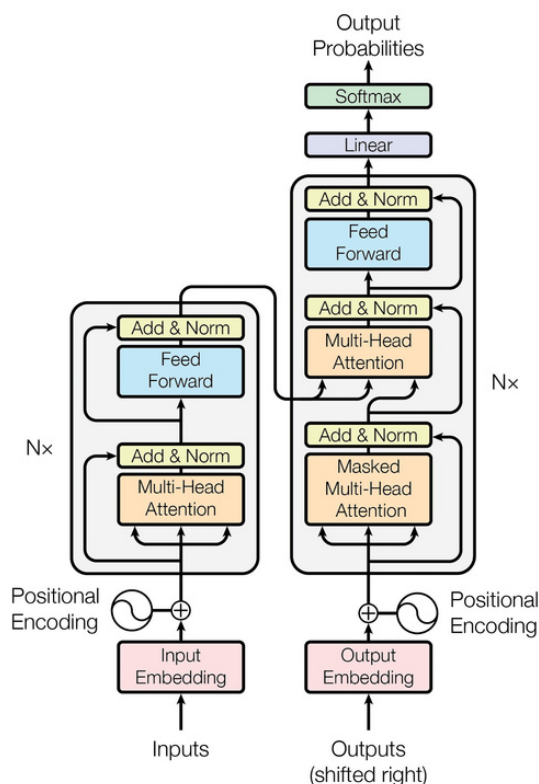


Figure 19.1: The encoder-decoder structure of the transformer architecture. From "Attention Is All You Need"

In generating an output sequence, the transformer does not rely on recurrence and convolutions. You have seen how to implement the transformer encoder and decoder separately. In this chapter, you will join the two into a complete transformer model and apply padding and look-ahead masking to the input values.

19.2 Masking

Let's start first by discovering how to apply masking.

Creating a Padding Mask

In Chapter 15, you should already be familiar with the importance of masking the input values before feeding them into the encoder and decoder. As you will see when you proceed to train the transformer model, the input sequences fed into the encoder and decoder will first be zero-padded up to a specific sequence length. The importance of having a padding mask is to make sure that these zero values are not processed along with the actual input values by both the encoder and decoder.

Let's create the following function to generate a padding mask for both the encoder and decoder:

```
from tensorflow import math, cast, float32

def padding_mask(input):
    # Create mask which marks the zero padding values in the input by a 1
    mask = math.equal(input, 0)
    mask = cast(mask, float32)

    return mask
```

Listing 19.1: A function to create padding masks

Upon receiving an input, this function will generate a tensor that marks by a value of *one* wherever the input contains a value of *zero*. Hence, if you input the following array:

```
from numpy import array

input = array([1, 2, 3, 4, 0, 0, 0])
print(padding_mask(input))
```

Listing 19.2: Example of generating a padding mask

Then the output of the `padding_mask()` function would be the following:

```
tf.Tensor([0. 0. 0. 0. 1. 1. 1.], shape=(7,), dtype=float32)
```

Output 19.1: Padding mask as generated

Creating a Look-Ahead Mask

A look-ahead mask is required to prevent the decoder from attending to succeeding words, such that the prediction for a particular word can only depend on known outputs for the words that come before it. For this purpose, let's create the following function to generate a look-ahead mask for the decoder:

```
from tensorflow import linalg, ones

def lookahead_mask(shape):
    # Mask out future entries by marking them with a 1.0
    mask = 1 - linalg.band_part(ones((shape, shape)), -1, 0)

    return mask
```

Listing 19.3: A function to create look-ahead masks

You will pass to it the length of the decoder input. Let's make this length equal to 5, as an example:

```
print(lookahead_mask(5))
```

Listing 19.4: Generating a look-ahead mask

Then the output that the `lookahead_mask()` function returns is the following:


```
tf.Tensor(
[[[0. 1. 1. 1. 1.]
 [0. 0. 1. 1. 1.]
 [0. 0. 0. 1. 1.]
 [0. 0. 0. 0. 1.]
 [0. 0. 0. 0. 0.]], shape=(5, 5), dtype=float32)
```

Output 19.2: The look-ahead mask as generated

Again, the *one* values mask out the entries that should not be used. In this manner, the prediction of every word only depends on those that come before it.

19.3 Joining the Transformer Encoder and Decoder

Let's start by creating the `TransformerModel` class, which inherits from the `Model` base class in Keras:

```
class TransformerModel(Model):
    def __init__(self, enc_vocab_size, dec_vocab_size, enc_seq_length, dec_seq_length,
h, d_k, d_v, d_model, d_ff_inner, n, rate, **kwargs):
    super().__init__(**kwargs)

    # Set up the encoder
    self.encoder = Encoder(enc_vocab_size, enc_seq_length, h, d_k, d_v, d_model,
d_ff_inner, n, rate)

    # Set up the decoder
    self.decoder = Decoder(dec_vocab_size, dec_seq_length, h, d_k, d_v, d_model,
d_ff_inner, n, rate)

    # Define the final dense layer
    self.model_last_layer = Dense(dec_vocab_size)
    ...
```

Listing 19.5: The transformer model with encoder and decoder

Your first step in creating the `TransformerModel` class is to initialize instances of the `Encoder` and `Decoder` classes implemented earlier and assign their outputs to the variables `encoder` and `decoder` respectively. If you saved these classes in separate Python scripts, do not forget to import them. Let's save the code in the Python scripts `encoder.py` and `decoder.py`, so you need to import them accordingly. You will also include one final dense layer that produces the final output, as in the transformer architecture of “Attention Is All You Need”.

Next, you shall create the class method `call()` to feed the relevant inputs into the encoder and decoder. A padding mask is first generated to mask the encoder input, as well as the encoder output, when this is fed into the second self-attention block of the decoder:

```
...
def call(self, encoder_input, decoder_input, training):

    # Create padding mask to mask the encoder inputs and the encoder outputs in
```

```
# the decoder
enc_padding_mask = self.padding_mask(encoder_input)
...
```

Listing 19.6: Implementing the `call()` method for the transformer model class

A padding mask and a look-ahead mask are then generated to mask the decoder input. These are combined together through an element-wise maximum operation:

```
...
# Create and combine padding and look-ahead masks to be fed into the decoder
dec_in_padding_mask = self.padding_mask(decoder_input)
dec_in_lookahead_mask = self.lookahead_mask(decoder_input.shape[1])
dec_in_lookahead_mask = maximum(dec_in_padding_mask,
dec_in_lookahead_mask) ...
```

Listing 19.7: Creating the masks

Next, the relevant inputs are fed into the encoder and decoder, and the transformer model output is generated by feeding the decoder output into one final dense layer:

```
...
# Feed the input into the encoder
encoder_output = self.encoder(encoder_input, enc_padding_mask, training)

# Feed the encoder output into the decoder
decoder_output = self.decoder(decoder_input, encoder_output,
                             dec_in_lookahead_mask, enc_padding_mask, training)

# Pass the decoder output through a final dense layer
model_output = self.model_last_layer(decoder_output)

return model_output
```

Listing 19.8: Using the encoder and decoder

Combining all the steps gives us the following complete code listing:

```
from encoder import Encoder
from decoder import Decoder
from tensorflow import math, cast, float32, linalg, ones, maximum, newaxis
from tensorflow.keras import Model
from tensorflow.keras.layers import Dense

class TransformerModel(Model):
    def __init__(self, enc_vocab_size, dec_vocab_size, enc_seq_length, dec_seq_length,
h, d_k, d_v, d_model, d_ff_inner, n, rate, **kwargs):
        super().__init__(**kwargs)

    # Set up the encoder
    self.encoder = Encoder(enc_vocab_size, enc_seq_length, h, d_k, d_v,
d_model, d_ff_inner, n, rate)

    # Set up the decoder
```

```

self.decoder = Decoder(dec_vocab_size, dec_seq_length, h, d_k, d_v,
                        d_model, d_ff_inner, n, rate)

# Define the final dense layer
self.model_last_layer = Dense(dec_vocab_size)

def padding_mask(self, input):
    # Create mask which marks the zero padding values in the input by a 1.0
    mask = math.equal(input, 0)
    mask = cast(mask, float32)

    # The shape of the mask should be broadcastable to the shape
    # of the attention weights that it will be masking later on
    return mask[:, newaxis, newaxis, :]

def lookahead_mask(self, shape):
    # Mask out future entries by marking them with a 1.0
    mask = 1 - linalg.band_part(ones((shape, shape)), -1, 0)

    return mask

def call(self, encoder_input, decoder_input, training):

    # Create padding mask to mask the encoder inputs and the encoder
    # outputs in the decoder
    enc_padding_mask = self.padding_mask(encoder_input)

    # Create and combine padding and look-ahead masks to be fed into the decoder
    dec_in_padding_mask = self.padding_mask(decoder_input)
    dec_in_lookahead_mask = self.lookahead_mask(decoder_input.shape[1])
    dec_in_lookahead_mask = maximum(dec_in_padding_mask, dec_in_lookahead_mask)

    # Feed the input into the encoder
    encoder_output = self.encoder(encoder_input, enc_padding_mask, training)

    # Feed the encoder output into the decoder
    decoder_output = self.decoder(decoder_input, encoder_output,
                                  dec_in_lookahead_mask, enc_padding_mask, training)

    # Pass the decoder output through a final dense layer
    model_output = self.model_last_layer(decoder_output)

    return model_output

```

Listing 19.9: Complete code to use the encoder and decoder

Note that you have performed a small change to the output that is returned by the `padding_mask` function. Its shape is made broadcastable to the shape of the attention weight tensor that it will mask when you train the transformer model.

19.4 Creating an Instance of the Transformer Model

You will work with the parameter values specified in the paper, “Attention Is All You Need”:

```

h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_ff = 2048 # Dimensionality of the inner fully connected layer
d_model = 512 # Dimensionality of the model sub-layers' outputs
n = 6 # Number of layers in the encoder stack

dropout_rate = 0.1 # Frequency of dropping the input units in the dropout layers ...

```

Listing 19.10: Parameters from “Attention Is All You Need”

As for the input-related parameters, you will work with dummy values for now until you arrive at the stage of training the complete transformer model. At that point, you will be using actual sentences:

```

...
enc_vocab_size = 20 # Vocabulary size for the encoder
dec_vocab_size = 20 # Vocabulary size for the decoder

enc_seq_length = 5 # Maximum length of the input sequence
dec_seq_length = 5 # Maximum length of the target sequence
...

```

Listing 19.11: Dummy values for testing the input

You can now create an instance of the `TransformerModel` class as follows:

```

from model import TransformerModel

# Create model
training_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
                                  dec_seq_length, h, d_k, d_v, d_model, d_ff, n,
                                  dropout_rate)

```

Listing 19.12: Create the transformer model

The complete code listing is as follows:

```

enc_vocab_size = 20 # Vocabulary size for the encoder
dec_vocab_size = 20 # Vocabulary size for the decoder

enc_seq_length = 5 # Maximum length of the input sequence
dec_seq_length = 5 # Maximum length of the target sequence

h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_ff = 2048 # Dimensionality of the inner fully connected layer
d_model = 512 # Dimensionality of the model sub-layers' outputs
n = 6 # Number of layers in the encoder stack

dropout_rate = 0.1 # Frequency of dropping the input units in the dropout layers

```

```
# Create model
training_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
                                  dec_seq_length, h, d_k, d_v, d_model, d_ff, n,
                                  dropout_rate)
```

Listing 19.13: Complete code to invoke a transformer model

Printing Out a Summary of the Encoder and Decoder Layers

You may also print out a summary of the encoder and decoder blocks of the transformer model. The choice to print them out separately will allow you to be able to see the details of their individual sub-layers. In order to do so, add the following line of code to the `__init__()` method of both the `EncoderLayer` and `DecoderLayer` classes:

```
self.build(input_shape=[None, sequence_length, d_model])
```

Listing 19.14: A line of code to add to the encoder and decoder layer

Then you need to add the following method to the `EncoderLayer` class:

```
def build_graph(self):
    input_layer = Input(shape=(self.sequence_length, self.d_model))
    return Model(inputs=[input_layer], outputs=self.call(input_layer, None, True))
```

Listing 19.15: The `build_graph()` method for the encoder layer

And the following method to the `DecoderLayer` class:

```
def build_graph(self):
    input_layer = Input(shape=(self.sequence_length, self.d_model))
    return Model(inputs=[input_layer],
                  outputs=self.call(input_layer, input_layer, None, None, True))
```

Listing 19.16: The `build_graph()` method for the decoder layer

This results in the `EncoderLayer` class being modified as follows (the `call()` method remains the same as the one that was implemented in Chapter 17):

```
from tensorflow.keras.layers import Input
from tensorflow.keras import Model

class EncoderLayer(Layer):
    def __init__(self, sequence_length, h, d_k, d_v, d_model, d_ff, rate, **kwargs):
        super().__init__(**kwargs)
        self.build(input_shape=[None, sequence_length, d_model])
        self.d_model = d_model
        self.sequence_length = sequence_length
        self.multihead_attention = MultiHeadAttention(h, d_k, d_v, d_model)
        self.dropout1 = Dropout(rate)
        self.add_norm1 = AddNormalization()
        self.feed_forward = FeedForward(d_ff, d_model)
        self.dropout2 = Dropout(rate)
```

```

self.add_norm2 = AddNormalization()

def build_graph(self):
    input_layer = Input(shape=(self.sequence_length, self.d_model))
    return Model(inputs=[input_layer], outputs=self.call(input_layer, None, True))

def call(self, x, padding_mask, training):
    ...

```

Listing 19.17: Modified encoder layer

Similar changes can be made to the DecoderLayer class too.

Once you have the necessary changes in place, you can proceed to create instances of the EncoderLayer and DecoderLayer classes and print out their summaries as follows:

```

from encoder import EncoderLayer
from decoder import DecoderLayer

encoder = EncoderLayer(enc_seq_length, h, d_k, d_v, d_model, d_ff, dropout_rate)
encoder.build_graph().summary()

decoder = DecoderLayer(dec_seq_length, h, d_k, d_v, d_model, d_ff, dropout_rate)
decoder.build_graph().summary()

```

Listing 19.18: Creating the encoder and decoder layer

The resulting summary for the encoder is the following:

```

Model: "model"
-----
Layer(type) OutputShapeParam#Connectedto
=====
=====
input_1(InputLayer) [(None,5,512)]0 []

multi_head_attention_18(Multi(None,5,512) 131776 ['input_1[0][0]',
HeadAttention) 'input_1[0][0]',
'input_1[0][0]']

                                dropout_32(Dropout) (None,5,512) 0 ['multi_head_attention_18[0][0]']

add_normalization_30(AddNorma(None,5,512) 1024 ['input_1[0][0]',
lization) 'dropout_32[0][0]']

feed_forward_12(FeedForward)(None,5,512) 2099712 ['add_normalization_30[0][0]']

dropout_33(Dropout) (None,5,512) 0 ['feed_forward_12[0][0]']

add_normalization_31(AddNorma(None,5,512) 1024 ['add_normalization_30[0][0]',
lization) 'dropout_33[0][0]']

=====
===== Total params: 2,233,536
Trainable params: 2,233,536

```

Non-trainable params: 0

Output 19.3: Summary from the encoder layer

While the resulting summary for the decoder is the following:

```
Model: "model_1"
-----
Layer(type) OutputShapeParam#Connectedto
=====
input_2(InputLayer) [(None,5,512)]0 []

multi_head_attention_19(Multi(None,5,512) 131776 ['input_2[0][0]',
HeadAttention) 'input_2[0][0]',
'input_2[0][0]']

                        dropout_34(Dropout) (None,5,512) 0 ['multi_head_attention_19[0][0]']

add_normalization_32(AddNorma(None,5,512) 1024 ['input_2[0][0]',
lization) 'dropout_34[0][0]',
'add_normalization_32[0][0]',
'dropout_35[0][0]']

multi_head_attention_20(Multi(None,5,512) 131776 ['add_normalization_32[0][0]',
HeadAttention) 'input_2[0][0]',
'input_2[0][0]']

                        dropout_35(Dropout) (None,5,512) 0 ['multi_head_attention_20[0][0]']

feed_forward_13(FeedForward)(None,5,512) 2099712 ['add_normalization_32[1][0]']

dropout_36(Dropout) (None,5,512) 0 ['feed_forward_13[0][0]']

add_normalization_34(AddNorma(None,5,512) 1024 ['add_normalization_32[1][0]',
lization) 'dropout_36[0][0]']

=====
===== Total params: 2,365,312
Trainable params: 2,365,312
Non-trainable params: 0
-----
```

Output 19.4: Summary from the decoder layer

19.5 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

19.6 Summary

In this chapter, you discovered how to implement the complete transformer model, and create padding and look-ahead masks. Specifically, you learned:

- ◁ How to create a padding mask for the encoder and decoder
- ◁ How to create a look-ahead mask for the decoder
- ◁ How to join the transformer encoder and decoder into a single model
- ◁ How to print out a summary of the encoder and decoder layers

In the next chapter, you will feed in data to the transformer to train it.

Training the Transformer Model

20

We have put together the complete transformer model, and now we are ready to train it for neural machine translation. We shall use a training dataset for this purpose, which contains short English and German sentence pairs. We will also revisit the role of masking in computing the accuracy and loss metrics during the training process.

In this chapter, you will discover how to train the transformer model for neural machine translation. After completing this chapter, you will know:

- ◁ How to prepare the training dataset
- ◁ How to apply a padding mask to the loss and accuracy computations
- ◁ How to train the transformer model

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◁ Preparing the Training Dataset
 - ◁ Applying a Padding Mask to the Loss and Accuracy Computations
- ◁ Training the Transformer Model

Let's start first by preparing the dataset for training.

20.1 Preparing the Training Dataset

You will also use a dataset that contains short English and German sentence pairs, which you may download here:

◁
<https://github.com/Rishav09/Neural-Machine-Translation-System/blob/master/english-german-both.pkl>

This particular dataset has already been cleaned by removing non-printable and non-alphabetic characters and punctuation characters, further normalizing all Unicode characters to ASCII,

and changing all uppercase letters to lowercase ones. Hence, you can skip the cleaning step, which is typically part of the data preparation process.

Let's proceed by creating the `PrepareDataset` class that implements the following steps:

1. Loads the dataset from a specified filename.

```
clean_dataset = load(open(filename, 'rb'))
```

2. Selects the number of sentences to use from the dataset. Since the dataset is large, you will reduce its size to limit the training time. However, you may explore using the full dataset as an extension to this chapter.

```
dataset = clean_dataset[:self.n_sentences, :]
```

3. Appends start (<START>) and end-of-string (<EOS>) tokens to each sentence.

For example, the English sentence "iliketorun" now becomes "<START>iliketorun<EOS>". This also applies to its corresponding translation in German "ich gehe gerne joggen" which now becomes "<START> ich gehe gerne joggen <EOS>".

```
for i in range(dataset[:, 0].size):
    dataset[i, 0] = "<START> " + dataset[i, 0] + " <EOS>"
    dataset[i, 1] = "<START> " + dataset[i, 1] + " <EOS>"
```

4. Shuffles the dataset randomly.

```
shuffle(dataset)
```

5. Splits the shuffled dataset based on a pre-defined ratio.

```
train = dataset[:int(self.n_sentences * self.train_split)]
```

6. Creates and trains a tokenizer on the text sequences that will be fed into the encoder and finds the length of the longest sequence as well as the vocabulary size.

```
enc_tokenizer = self.create_tokenizer(train[:, 0])
enc_seq_length = self.find_seq_length(train[:, 0])
enc_vocab_size = self.find_vocab_size(enc_tokenizer, train[:, 0])
```

7. Tokenizes the sequences of text that will be fed into the encoder by creating a vocabulary of words and replacing each word with its corresponding vocabulary index.

The <START> and <EOS> tokens will also form part of this vocabulary. Each sequence is also padded to the maximum phrase length.

```
trainX = enc_tokenizer.texts_to_sequences(train[:, 0])
trainX = pad_sequences(trainX, maxlen=enc_seq_length,
padding='post') trainX = convert_to_tensor(trainX, dtype=int64)
```

8. Creates and trains a tokenizer on the text sequences that will be fed into the decoder, and finds the length of the longest sequence as well as the vocabulary size.

```
dec_tokenizer = self.create_tokenizer(train[:, 1])
dec_seq_length = self.find_seq_length(train[:, 1])
dec_vocab_size = self.find_vocab_size(dec_tokenizer, train[:, 1])
```

9. Repeats a similar tokenization and padding procedure for the sequences of text that will be fed into the decoder.

```
trainY = dec_tokenizer.texts_to_sequences(train[:, 1])
trainY = pad_sequences(trainY, maxlen=dec_seq_length,
padding='post') trainY = convert_to_tensor(trainY, dtype=int64)
```

The complete code listing is as follows:

```
from pickle import load
from numpy.random import shuffle
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow import convert_to_tensor, int64

class PrepareDataset:
    def __init__(self, **kwargs):
        super().__init__(**kwargs)
        self.n_sentences = 10000 # Number of sentences to include in the dataset
        self.train_split = 0.9 # Ratio of the training data split

    # Fit a tokenizer
    def create_tokenizer(self, dataset):
        tokenizer = Tokenizer()
        tokenizer.fit_on_texts(dataset)

    return tokenizer

    def find_seq_length(self, dataset):
        return max(len(seq.split()) for seq in dataset)

    def find_vocab_size(self, tokenizer, dataset):
        tokenizer.fit_on_texts(dataset)

    return len(tokenizer.word_index) + 1

    def __call__(self, filename, **kwargs):
```

```

# Load a clean dataset
clean_dataset = load(open(filename, 'rb'))

# Reduce dataset size
dataset = clean_dataset[:self.n_sentences, :]

# Include start and end of string tokens
for i in range(dataset[:, 0].size):
    dataset[i, 0] = "<START> " + dataset[i, 0] + " <EOS>"
    dataset[i, 1] = "<START> " + dataset[i, 1] + " <EOS>"

# Random shuffle the dataset
shuffle(dataset)

# Split the dataset
train = dataset[:int(self.n_sentences * self.train_split)]

# Prepare tokenizer for the encoder input
enc_tokenizer = self.create_tokenizer(train[:, 0])
enc_seq_length = self.find_seq_length(train[:, 0])
enc_vocab_size = self.find_vocab_size(enc_tokenizer, train[:, 0])

# Encode and pad the input sequences
trainX = enc_tokenizer.texts_to_sequences(train[:, 0])
trainX = pad_sequences(trainX, maxlen=enc_seq_length,
padding='post') trainX = convert_to_tensor(trainX, dtype=int64)

# Prepare tokenizer for the decoder input
dec_tokenizer = self.create_tokenizer(train[:, 1])
dec_seq_length = self.find_seq_length(train[:, 1])
dec_vocab_size = self.find_vocab_size(dec_tokenizer, train[:, 1])

# Encode and pad the input sequences
trainY = dec_tokenizer.texts_to_sequences(train[:, 1])
trainY = pad_sequences(trainY, maxlen=dec_seq_length,
padding='post') trainY = convert_to_tensor(trainY, dtype=int64)

return (trainX, trainY, train, enc_seq_length, dec_seq_length,
enc_vocab_size, dec_vocab_size)

```

Listing 20.1: Complete code to prepare data for neural machine translation

Before moving on to train the transformer model, let's first have a look at the output of the PrepareDataset class corresponding to the first sentence in the training dataset:

```

# Prepare the training data
dataset = PrepareDataset()
trainX, trainY, train_orig, enc_seq_length, dec_seq_length, \
    enc_vocab_size, dec_vocab_size = dataset('english-german-both.pkl')

print(train_orig[0, 0], '\n', trainX[0, :])

```

Listing 20.2: Testing the PrepareDataset class

```
<START> did tom tell you <EOS>
tf.Tensor([ 1 25 4 97 5 2 0], shape=(7,), dtype=int64)
```

Output 20.1: Output from `PrepareDataset` class

INFO-CIRCLE Note: Since the dataset has been randomly shuffled, you will likely see a different output.

You can see that, originally, you had a four-word sentence (did tom tell you) to which you have appended the start and end-of-string tokens. Then you proceeded to vectorize (you may notice that the <START> and <EOS> tokens are assigned the vocabulary indices 1 and 2, respectively). The vectorized text was also padded with zeros, such that the length of the end result matches the maximum sequence length of the encoder:

```
print('Encoder sequence length:', enc_seq_length)
```

Listing 20.3: Printing the encoder sequence length

```
Encoder sequence length: 7
```

Output 20.2: Encoder sequence length: Length of sentences with two extra tokens

You may similarly check out the corresponding target data that is fed into the decoder:

```
print(train_orig[0, 1], '\n', trainY[0, :])
```

Listing 20.4: Printing the decoder sequence

```
<START> hat tom es dir gesagt <EOS>
tf.Tensor([ 1 14 8 7042162 0], shape=(12,), dtype=int64)
```

Output 20.3: Sequence from the decoder

Here, the length of the end result matches the maximum sequence length of the decoder:

```
print('Decoder sequence length:', dec_seq_length)
```

Listing 20.5: Printing the decoder sequence length

```
Decoder sequence length: 12
```

Output 20.4: Decoder sequence length: A sentence with two extra tokens

20.2 Applying a Padding Mask

Recall seeing that the importance of having a padding mask at the encoder and decoder is to make sure that the zero values that we have just appended to the vectorized inputs are not processed along with the actual input values. This also holds true for the training process, where a padding mask is required so that the zero padding values in the target data are not

considered in the computation of the loss and accuracy. Let's have a look at the computation of loss first.

This will be computed using a sparse categorical cross-entropy loss function between the target and predicted values and subsequently multiplied by a padding mask so that only the valid non-zero values are considered. The returned loss is the mean of the unmasked values:

```
def loss_fn(target, prediction):
    # Create mask so that the zero padding values are not included in the
    # computation of loss
    mask = math.logical_not(equal(target, 0))
    mask = cast(mask, float32)

    # Compute a sparse categorical cross-entropy loss on the unmasked values
    loss = sparse_categorical_crossentropy(target, prediction, from_logits=True) * mask

    # Compute the mean loss over the unmasked values
    return reduce_sum(loss) / reduce_sum(mask)
```

Listing 20.6: Defining the loss metric

For the computation of accuracy, the predicted and target values are first compared. The predicted output is a tensor of size (batch_size,dec_seq_length,dec_vocab_size) and contains probability values (generated by the softmax function on the decoder side) for the tokens in the output. In order to be able to perform the comparison with the target values, only each token with the highest probability value is considered, with its dictionary index being retrieved through the operation: `argmax(prediction, axis=2)`. Following the application of a padding mask, the returned accuracy is the mean of the unmasked values:

```
def accuracy_fn(target, prediction):
    # Create mask so that the zero padding values are not included in the
    # computation of accuracy
    mask = math.logical_not(math.equal(target, 0))

    # Find equal prediction and target values, and apply the padding mask
    accuracy = equal(target, argmax(prediction, axis=2))
    accuracy = math.logical_and(mask, accuracy)

    # Cast the True/False values to 32-bit-precision floating-point numbers
    mask = cast(mask, float32)
    accuracy = cast(accuracy, float32)

    # Compute the mean accuracy over the unmasked values
    return reduce_sum(accuracy) / reduce_sum(mask)
```

Listing 20.7: Defining the accuracy metric

20.3 Training the Transformer Model

Let's first define the model and training parameters as specified by "Attention Is All You Need":

```

# Define the model parameters
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of model layers' outputs
d_ff = 2048 # Dimensionality of the inner fully connected layer
n = 6 # Number of layers in the encoder stack

# Define the training parameters
epochs = 2
batch_size = 64
beta_1 = 0.9
beta_2 = 0.98
epsilon = 1e-9
dropout_rate = 0.1

```

Listing 20.8: Parameters from “Attention Is All You Need”

INFO-CIRCLE Note: The training above run for two epochs only to limit the training time. However, you may explore training them chapter.

You also need to implement a learning rate scheduler that initially increases the learning rate linearly for the first `warmup_steps` and then decreases it proportionally to the inverse square root of the step number. Vaswani et al. express this by the following formula:

$$\text{learning_rate} = d - 0.5 \min(\text{step} - 0.5, \text{step_warmup_steps} - 1.5)$$

```

class LRScheduler(LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000, **kwargs):
        super().__init__(**kwargs)

        self.d_model = cast(d_model, float32)
        self.warmup_steps = warmup_steps

    def __call__(self, step_num):
        # Linearly increasing the learning rate for the first warmup_steps, and
        # decreasing it thereafter
        arg1 = step_num ** -0.5
        arg2 = step_num * (self.warmup_steps ** -1.5)

        return (self.d_model ** -0.5) * math.minimum(arg1, arg2)

```

Listing 20.9: Defining a custom learning rate scheduler

An instance of the `LRScheduler` class is subsequently passed on as the `learning_rate` argument of the Adam optimizer:

```
optimizer = Adam(LRScheduler(d_model), beta_1, beta_2, epsilon)
```

Listing 20.10: Setting up an optimizer with a custom learning rate schedule

Next, split the dataset into batches in preparation for training:

```
train_dataset = data.Dataset.from_tensor_slices((trainX, trainY))
train_dataset = train_dataset.batch(batch_size)
```

Listing 20.11: Prepare training dataset

This is followed by the creation of a model instance:

```
training_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
                                  dec_seq_length, h, d_k, d_v, d_model, d_ff, n,
                                  dropout_rate)
```

Listing 20.12: Setting up the transformer model

In training the transformer model, you will write your own training loop, which incorporates the loss and accuracy functions that were implemented earlier.

The default runtime in TensorFlow 2.0 is *eager execution*, which means that operations execute immediately one after the other. Eager execution is simple and intuitive, making debugging easier. Its downside, however, is that it cannot take advantage of the global performance optimizations that run the code using the *graph execution*. In graph execution, a graph is first built before the tensor computations can be executed, which gives rise to a computational overhead. For this reason, the use of graph execution is mostly recommended for large model training rather than for small model training, where eager execution can be more suited to perform simpler operations. Since the transformer model is sufficiently large, apply the graph execution to train it.

In order to do so, you will use the `@function` decorator as follows:

```
@function
def train_step(encoder_input, decoder_input, decoder_output):
    with GradientTape() as tape:
        # Run the forward pass of the model to generate a prediction
        prediction = training_model(encoder_input, decoder_input, training=True)

        # Compute the training loss
        loss = loss_fcn(decoder_output, prediction)

        # Compute the training accuracy
        accuracy = accuracy_fcn(decoder_output, prediction)

        # Retrieve gradients of the trainable variables with respect to the training loss
        gradients = tape.gradient(loss, training_model.trainable_weights)

        # Update the values of the trainable variables by gradient descent
        optimizer.apply_gradients(zip(gradients, training_model.trainable_weights))

    train_loss(loss)
    train_accuracy(accuracy)
```

Listing 20.13: Applying graph execution to a TensorFlow model

With the addition of the `@function` decorator, a function that takes tensors as input will be compiled into a graph. If the `@function` decorator is commented out, the function is, alternatively, run with eager execution.

The next step is implementing the training loop that will call the `train_step` function above. The training loop will iterate over the specified number of epochs and the dataset batches. For each batch, the `train_step` function computes the training loss and accuracy measures and applies the optimizer to update the trainable model parameters. A checkpoint manager is also included to save a checkpoint after every five epochs:

```
train_loss = Mean(name='train_loss')
train_accuracy = Mean(name='train_accuracy')

# Create a checkpoint object and manager to manage multiple checkpoints
ckpt = train.Checkpoint(model=training_model, optimizer=optimizer)
ckpt_manager = train.CheckpointManager(ckpt, "./checkpoints", max_to_keep=3)

for epoch in range(epochs):
    train_loss.reset_states()
    train_accuracy.reset_states()

    print("\nStart of epoch %d" % (epoch + 1))

    # Iterate over the dataset batches
    for step, (train_batchX, train_batchY) in enumerate(train_dataset):
        # Define the encoder and decoder inputs, and the decoder output
        encoder_input = train_batchX[:, 1:]
        decoder_input = train_batchY[:, :-1]
        decoder_output = train_batchY[:, 1:]

        train_step(encoder_input, decoder_input, decoder_output)

    if step % 50 == 0:
        print(f'Epoch {epoch + 1} Step {step} Loss {train_loss.result():.4f} '
              + f'Accuracy {train_accuracy.result():.4f}')

    # Print epoch number and loss value at the end of every epoch
    print(f'Epoch {epoch + 1}: Training Loss {train_loss.result():.4f}, '
          + f'Training Accuracy {train_accuracy.result():.4f}')

    # Save a checkpoint after every five epochs
    if (epoch + 1) % 5 == 0:
        save_path = ckpt_manager.save()
        print("Saved checkpoint at epoch %d" % (epoch + 1))
```

Listing 20.14: Implement a training loop in TensorFlow

An important point to keep in mind is that the input to the decoder is offset by one position to the right with respect to the encoder input. The idea behind this offset, combined with a look-ahead mask in the first multi-head attention block of the decoder, is to ensure that the prediction for the current token can only depend on the previous tokens.

“This masking, combined with the fact that the output embeddings are offset by one position

It is for this reason that the encoder and decoder inputs are fed into the transformer model in the following manner:

```
encoder_input = train_batchX[:, 1:]
decoder_input = train_batchY[:, :-1]
```

Listing 20.15: Splitting the encoder and decoder input

Putting together the complete code listing produces the following:

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers.schedules import LearningRateSchedule
from tensorflow.keras.metrics import Mean
from tensorflow import data, train, math, reduce_sum, cast, equal, argmax, \
float32, GradientTape, TensorSpec, function, int64
from tensorflow.keras.losses import sparse_categorical_crossentropy
from model import TransformerModel
from prepare_dataset import PrepareDataset
from time import time

# Define the model parameters
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of model layers' outputs
d_ff = 2048 # Dimensionality of the inner fully connected layer
n = 6 # Number of layers in the encoder stack

# Define the training parameters
epochs = 2
batch_size = 64
beta_1 = 0.9
beta_2 = 0.98
epsilon = 1e-9
dropout_rate = 0.1

# Implementing a learning rate scheduler
class LRScheduler(LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000, **kwargs):
        super().__init__(**kwargs)

    self.d_model = cast(d_model, float32)
    self.warmup_steps = warmup_steps

    def __call__(self, step_num):
        # Linearly increasing the learning rate for the first warmup_steps, and
        # decreasing it thereafter
        arg1 = step_num ** -0.5
        arg2 = step_num * (self.warmup_steps ** -1.5)

    return (self.d_model ** -0.5) * math.minimum(arg1, arg2)

# Instantiate an Adam optimizer
```

```

optimizer = Adam(LRScheduler(d_model), beta_1, beta_2, epsilon)

# Prepare the training and test splits of the dataset
dataset = PrepareDataset()
trainX, trainY, train_orig, enc_seq_length, dec_seq_length, \
enc_vocab_size, dec_vocab_size = dataset('english-german-both.pkl')

# Prepare the dataset batches
train_dataset = data.Dataset.from_tensor_slices((trainX, trainY))
train_dataset = train_dataset.batch(batch_size)

# Create model
training_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
dec_seq_length, h, d_k, d_v, d_model, d_ff, n,
dropout_rate)

# Defining the loss function
def loss_fn(target, prediction):
    # Create mask so that the zero padding values are not included in the
    # computation of loss
    mask = math.logical_not(equal(target, 0))
    mask = cast(mask, float32)

    # Compute a sparse categorical cross-entropy loss on the unmasked values
    loss = sparse_categorical_crossentropy(target, prediction, from_logits=True) * mask

    # Compute the mean loss over the unmasked values
    return reduce_sum(loss) / reduce_sum(mask)

# Defining the accuracy function
def accuracy_fn(target, prediction):
    # Create mask so that the zero padding values are not included in the
    # computation of accuracy
    mask = math.logical_not(equal(target, 0))

    # Find equal prediction and target values, and apply the padding mask
    accuracy = equal(target, argmax(prediction, axis=2))
    accuracy = math.logical_and(mask, accuracy)

    # Cast the True/False values to 32-bit-precision floating-point numbers
    mask = cast(mask, float32)
    accuracy = cast(accuracy, float32)

    # Compute the mean accuracy over the unmasked values
    return reduce_sum(accuracy) / reduce_sum(mask)

# Include metrics monitoring
train_loss = Mean(name='train_loss')
train_accuracy = Mean(name='train_accuracy')

# Create a checkpoint object and manager to manage multiple checkpoints
ckpt = train.Checkpoint(model=training_model, optimizer=optimizer)
ckpt_manager = train.CheckpointManager(ckpt, "./checkpoints", max_to_keep=3)

```

```

# Speeding up the training process
@function
def train_step(encoder_input, decoder_input, decoder_output):
    with GradientTape() as tape:
        # Run the forward pass of the model to generate a prediction
        prediction = training_model(encoder_input, decoder_input, training=True)

    # Compute the training loss
    loss = loss_fcn(decoder_output, prediction)

    # Compute the training accuracy
    accuracy = accuracy_fcn(decoder_output, prediction)

    # Retrieve gradients of the trainable variables with respect to the training loss
    gradients = tape.gradient(loss, training_model.trainable_weights)

    # Update the values of the trainable variables by gradient descent
    optimizer.apply_gradients(zip(gradients, training_model.trainable_weights))

    train_loss(loss)
    train_accuracy(accuracy)

    for epoch in range(epochs):
        train_loss.reset_states()
        train_accuracy.reset_states()

        print("\nStart of epoch %d" % (epoch + 1))

        start_time = time()

        # Iterate over the dataset batches
        for step, (train_batchX, train_batchY) in enumerate(train_dataset):
            # Define the encoder and decoder inputs, and the decoder output
            encoder_input = train_batchX[:, 1:]
            decoder_input = train_batchY[:, :-1]
            decoder_output = train_batchY[:, 1:]

            train_step(encoder_input, decoder_input, decoder_output)

        if step % 50 == 0:
            print(f"Epoch {epoch+1} Step {step} Loss {train_loss.result():.4f} "
                  + f"Accuracy {train_accuracy.result():.4f}")

        # Print epoch number and loss value at the end of every epoch
        print(f"Epoch {epoch+1}: Training Loss {train_loss.result():.4f}, "
              + f"Training Accuracy {train_accuracy.result():.4f}")

        # Save a checkpoint after every five epochs
        if (epoch + 1) % 5 == 0:
            save_path = ckpt_manager.save()
            print(f"Saved checkpoint at epoch {epoch+1}")

        print("Total time taken: %.2fs" % (time() - start_time))

```

Listing 20.16: Complete code to train a transformer model

Running the code produces a similar output to the following (you will likely see different loss and accuracy values because the training is from scratch, whereas the training time depends on the computational resources that you have available for training):

```
Start of epoch 1
Epoch 1 Step 0 Loss 8.4525 Accuracy 0.0000
Epoch 1 Step 50 Loss 7.6768 Accuracy 0.1234
Epoch 1 Step 100 Loss 7.0360 Accuracy 0.1713
Epoch 1: Training Loss 6.7109, Training Accuracy 0.1924

Start of epoch 2
Epoch 2 Step 0 Loss 5.7323 Accuracy 0.2628
Epoch 2 Step 50 Loss 5.4360 Accuracy 0.2756
Epoch 2 Step 100 Loss 5.2638 Accuracy 0.2839
Epoch 2: Training Loss 5.1468, Training Accuracy 0.2908 Total
time taken: 87.98s
```

Output 20.5: Screen output during training

It takes 155.13s for the code to run using eager execution alone on the same platform that is making use of only a CPU, which shows the benefit of using graph execution.

20.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

Articles

François Chollet. *Writing a training loop from scratch*. Keras developer guides, 2020.

https://keras.io/guides/writing_a_training_loop_from_scratch/

20.5 Summary

In this chapter, you discovered how to train the transformer model for neural machine translation. Specifically, you learned:

- ◀ How to prepare the training dataset
- ◀ How to apply a padding mask to the loss and accuracy computations
- ◀ How to train the transformer model

In the next chapter, you will modify this training procedure a bit to visualize the training performance.

Plotting the Training and Validation Loss Curves for the TransformerModel

21

We have previously seen how to train the transformer model for neural machine translation. Before moving on to inferencing the trained model, let us first explore how to modify the training code slightly to be able to plot the training and validation loss curves that can be generated during the learning process. The training and validation loss values provide important information because they give us a better insight into how the learning performance changes over the number of epochs and help us diagnose any problems with learning that can lead to an underfit or an overfit model. They will also inform us about the epoch with which to use the trained model weights at the inferencing stage.

In this chapter, you will discover how to plot the training and validation loss curves for the transformer model. After completing this chapter, you will know:

- ◁ How to modify the training code to include validation and test splits, in addition to a training split of the dataset
- ◁ How to modify the training code to store the computed training and validation loss values, as well as the trained model weights
- ◁ How to plot the saved training and validation loss curves

Let's get started.

Overview

This chapter is divided into three parts; they are:

- ◁ Preparing the Training, Validation, and Testing Splits of the Dataset
- ◁ Training the Transformer Model
- ◁ Plotting the Training and Validation Loss Curves

21.1 Preparing the Training, Validation, and Testing Splits of the Dataset

In order to be able to include validation and test splits of the data, you will modify the code that prepares the dataset in Chapter 20 by introducing the following lines of code, which:

◀ Specify the size of the validation data split. This, in turn, determines the size of the training and test splits of the data, which we will be dividing into a ratio of 80:10:10 for the training, validation and test sets, respectively:

```
self.val_split = 0.1          # Ratio of the validation data split
```

◀ Split the dataset into validation and test sets in addition to the training set:

```
val = dataset[int(self.n_sentences * self.train_split):
int(self.n_sentences * (1-self.val_split))]
test = dataset[int(self.n_sentences * (1 - self.val_split)):]
```

◀ Prepare the validation data by tokenizing, padding, and converting to a tensor. For this purpose, you will collect these operations into a function called `encode_pad`, as shown in the complete code listing below. This will avoid excessive repetition of code when performing these operations on the training data as well:

```
valX = self.encode_pad(val[:, 0], enc_tokenizer, enc_seq_length) valY =
self.encode_pad(val[:, 1], dec_tokenizer, dec_seq_length)
```

◀ Save the encoder and decoder tokenizers into pickle files and the test dataset into a text file to be used later during the inferencing stage:

```
self.save_tokenizer(enc_tokenizer, 'enc')
self.save_tokenizer(dec_tokenizer, 'dec')
savetxt('test_dataset.txt', test, fmt='%s')
```

The complete code listing is now updated as follows:

```
from pickle import load, dump, HIGHEST_PROTOCOL
from numpy.random import shuffle
from numpy import savetxt
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow import convert_to_tensor, int64

class PrepareDataset:
def __init__(self, **kwargs):
super().__init__(**kwargs)
self.n_sentences = 10000 # Number of sentences to include in the dataset
```



```

self.train_split = 0.8 # Ratio of the training data split
self.val_split = 0.1 # Ratio of the validation data split

# Fit a tokenizer
def create_tokenizer(self, dataset):
    tokenizer = Tokenizer()
    tokenizer.fit_on_texts(dataset)

    return tokenizer

def find_seq_length(self, dataset):
    return max(len(seq.split()) for seq in dataset)

def find_vocab_size(self, tokenizer, dataset):
    tokenizer.fit_on_texts(dataset)

    return len(tokenizer.word_index) + 1

# Encode and pad the input sequences
def encode_pad(self, dataset, tokenizer, seq_length):
    x = tokenizer.texts_to_sequences(dataset)
    x = pad_sequences(x, maxlen=seq_length, padding='post')
    x = convert_to_tensor(x, dtype=int64)

    return x

def save_tokenizer(self, tokenizer, name):
    with open(name + '_tokenizer.pkl', 'wb') as handle:
        dump(tokenizer, handle, protocol=HIGHEST_PROTOCOL)

def __call__(self, filename, **kwargs):
    # Load a clean dataset
    clean_dataset = load(open(filename, 'rb'))

    # Reduce dataset size
    dataset = clean_dataset[:self.n_sentences, :]

    # Include start and end of string tokens
    for i in range(dataset[:, 0].size):
        dataset[i, 0] = "<START> " + dataset[i, 0] + " <EOS>"
        dataset[i, 1] = "<START> " + dataset[i, 1] + " <EOS>"

    # Random shuffle the dataset
    shuffle(dataset)

    # Split the dataset in training, validation and test sets
    train = dataset[:int(self.n_sentences * self.train_split)]
    val = dataset[int(self.n_sentences * self.train_split):
int(self.n_sentences * (1 - self.val_split))]
        test = dataset[int(self.n_sentences * (1 - self.val_split)):]

    # Prepare tokenizer for the encoder input
    enc_tokenizer = self.create_tokenizer(dataset[:, 0])
    enc_seq_length = self.find_seq_length(dataset[:, 0])

```

```

enc_vocab_size = self.find_vocab_size(enc_tokenizer, train[:, 0])

# Prepare tokenizer for the decoder input
dec_tokenizer = self.create_tokenizer(dataset[:, 1])
dec_seq_length = self.find_seq_length(dataset[:, 1])
dec_vocab_size = self.find_vocab_size(dec_tokenizer, train[:, 1])

# Encode and pad the training input
trainX = self.encode_pad(train[:, 0], enc_tokenizer, enc_seq_length) trainY =
self.encode_pad(train[:, 1], dec_tokenizer, dec_seq_length)

# Encode and pad the validation input
valX = self.encode_pad(val[:, 0], enc_tokenizer, enc_seq_length)
valY = self.encode_pad(val[:, 1], dec_tokenizer, dec_seq_length)

# Save the encoder tokenizer
self.save_tokenizer(enc_tokenizer, 'enc')

# Save the decoder tokenizer
self.save_tokenizer(dec_tokenizer, 'dec')

# Save the testing dataset into a text file savetxt('test_dataset.txt', test,
fmt='%s')

return (trainX, trainY, valX, valY, train, val, enc_seq_length,
dec_seq_length, enc_vocab_size, dec_vocab_size)

```

Listing 21.1: Updated code of preparing the dataset

21.2 Training the Transformer Model

You shall introduce similar modifications to the code that trains the transformer model in Chapter 20 to:

◀ Prepare the validation dataset batches:

```

val_dataset = data.Dataset.from_tensor_slices((valX, valY))
val_dataset = val_dataset.batch(batch_size)

```

◀ Monitor the validation loss metric:

```
val_loss = Mean(name='val_loss')
```

◀ Initialize dictionaries to store the training and validation losses and eventually store the loss values in the respective dictionaries:

```

train_loss_dict = {}
val_loss_dict = {}

```

```
train_loss_dict[epoch] = train_loss.result()
val_loss_dict[epoch] = val_loss.result()
```

◀ Compute the validation loss:

```
loss = loss_fn(decoder_output, prediction)
val_loss(loss)
```

◀ Save the trained model weights at every epoch. You will use these at the inferencing stage to investigate the differences in results that the model produces at different epochs. In practice, it would be more efficient to include a callback method that halts the training process based on the metrics that are being monitored during training and only then save the model weights:

```
# Save the trained model weights
training_model.save_weights("weights/wgths" + str(epoch + 1) + ".ckpt")
```

◀ Finally, save the training and validation loss values into pickle files:

```
with open('./train_loss.pkl', 'wb') as file:
    dump(train_loss_dict, file)

with open('./val_loss.pkl', 'wb') as file:
    dump(val_loss_dict, file)
```

The modified code listing now becomes:

```
from tensorflow.keras.optimizers import Adam
from tensorflow.keras.optimizers.schedules import LearningRateSchedule
from tensorflow.keras.metrics import Mean
from tensorflow import data, train, math, reduce_sum, cast, equal, argmax, \
float32, GradientTape, function
from tensorflow.keras.losses import sparse_categorical_crossentropy
from model import TransformerModel
from prepare_dataset import PrepareDataset
from time import time
from pickle import dump

# Define the model parameters
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of model layers' outputs
d_ff = 2048 # Dimensionality of the inner fully connected layer
n = 6 # Number of layers in the encoder stack

# Define the training parameters
```

```

epochs = 20
batch_size = 64
beta_1 = 0.9
beta_2 = 0.98
epsilon = 1e-9
dropout_rate = 0.1

# Implementing a learning rate scheduler
class LRScheduler(LearningRateSchedule):
    def __init__(self, d_model, warmup_steps=4000, **kwargs):
        super().__init__(**kwargs)
        self.d_model = cast(d_model, float32)
        self.warmup_steps = warmup_steps

    def __call__(self, step_num):
        # Linearly increasing the learning rate for the first warmup_steps, and
        # decreasing it thereafter
        arg1 = step_num ** -0.5
        arg2 = step_num * (self.warmup_steps ** -1.5)

        return (self.d_model ** -0.5) * math.minimum(arg1, arg2)

# Instantiate an Adam optimizer
optimizer = Adam(LRScheduler(d_model), beta_1, beta_2, epsilon)

# Prepare the training dataset
dataset = PrepareDataset()
trainX, trainY, valX, valY, train_orig, val_orig, enc_seq_length, \
    dec_seq_length, enc_vocab_size, dec_vocab_size = dataset('english-german-both.pkl')

print(enc_seq_length, dec_seq_length, enc_vocab_size, dec_vocab_size)

# Prepare the training dataset batches
train_dataset = data.Dataset.from_tensor_slices((trainX, trainY))
train_dataset = train_dataset.batch(batch_size)

# Prepare the validation dataset batches
val_dataset = data.Dataset.from_tensor_slices((valX, valY))
val_dataset = val_dataset.batch(batch_size)

# Create model
training_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
    dec_seq_length, h, d_k, d_v, d_model, d_ff, n,
    dropout_rate)

# Defining the loss function
def loss_fn(target, prediction):
    # Create mask so that the zero padding values are not included in the
    # computation of loss
    mask = math.logical_not(equal(target, 0))
    mask = cast(mask, float32)

    # Compute a sparse categorical cross-entropy loss on the unmasked values
    loss = sparse_categorical_crossentropy(target, prediction, from_logits=True) * mask

```

```

# Compute the mean loss over the unmasked values
return reduce_sum(loss) / reduce_sum(mask)

# Defining the accuracy function
def accuracy_fcn(target, prediction):
# Create mask so that the zero padding values are not included in the
# computation of accuracy
mask = math.logical_not(equal(target, 0))

# Find equal prediction and target values, and apply the padding mask
accuracy = equal(target, argmax(prediction, axis=2))
accuracy = math.logical_and(mask, accuracy)

# Cast the True/False values to 32-bit-precision floating-point numbers
mask = cast(mask, float32)
accuracy = cast(accuracy, float32)

# Compute the mean accuracy over the unmasked values
return reduce_sum(accuracy) / reduce_sum(mask)

# Include metrics monitoring
train_loss = Mean(name='train_loss')
train_accuracy = Mean(name='train_accuracy')
val_loss = Mean(name='val_loss')

# Create a checkpoint object and manager to manage multiple checkpoints
ckpt = train.Checkpoint(model=training_model, optimizer=optimizer)
ckpt_manager = train.CheckpointManager(ckpt, "./checkpoints", max_to_keep=None)

# Initialise dictionaries to store the training and validation losses
train_loss_dict = {}
val_loss_dict = {}

# Speeding up the training process
@function
def train_step(encoder_input, decoder_input, decoder_output):
with GradientTape() as tape:

# Run the forward pass of the model to generate a prediction
prediction = training_model(encoder_input, decoder_input, training=True)

# Compute the training loss
loss = loss_fcn(decoder_output, prediction)

# Compute the training accuracy
accuracy = accuracy_fcn(decoder_output, prediction)

# Retrieve gradients of the trainable variables with respect to the training loss
gradients = tape.gradient(loss, training_model.trainable_weights)

# Update the values of the trainable variables by gradient descent
optimizer.apply_gradients(zip(gradients, training_model.trainable_weights))

```

```

train_loss(loss)
train_accuracy(accuracy)

for epoch in range(epochs):
    train_loss.reset_states()
    train_accuracy.reset_states()
    val_loss.reset_states()

    print("\nStart of epoch %d" % (epoch + 1))

    start_time = time()

    # Iterate over the dataset batches
    for step, (train_batchX, train_batchY) in enumerate(train_dataset):
        # Define the encoder and decoder inputs, and the decoder output
        encoder_input = train_batchX[:, 1:]
        decoder_input = train_batchY[:, :-1]
        decoder_output = train_batchY[:, 1:]

        train_step(encoder_input, decoder_input, decoder_output)

    if step % 50 == 0:
        print(f"Epoch {epoch + 1} Step {step} Loss {train_loss.result():.4f} "
              + f"Accuracy {train_accuracy.result():.4f}")

    # Run a validation step after every epoch of training
    for val_batchX, val_batchY in val_dataset:
        # Define the encoder and decoder inputs, and the decoder output
        encoder_input = val_batchX[:, 1:]
        decoder_input = val_batchY[:, :-1]
        decoder_output = val_batchY[:, 1:]

    # Generate a prediction
    prediction = training_model(encoder_input, decoder_input, training=False)

    # Compute the validation loss
    loss = loss_fn(decoder_output, prediction)
    val_loss(loss)

    # Print epoch number and accuracy and loss values at the end of every epoch
    print(f"Epoch {epoch+1}: Training Loss {train_loss.result():.4f}, "
          + f"Training Accuracy {train_accuracy.result():.4f}, "
          + f"Validation Loss {val_loss.result():.4f}")

    # Save a checkpoint after every epoch
    if (epoch + 1) % 1 == 0:
        save_path = ckpt_manager.save()
        print(f"Saved checkpoint at epoch {epoch+1}")

    # Save the trained model weights
    training_model.save_weights("weights/wgths" + str(epoch + 1) + ".ckpt")

    train_loss_dict[epoch] = train_loss.result()

```

```

        val_loss_dict[epoch] = val_loss.result()

# Save the training loss values
with open('./train_loss.pkl', 'wb') as file:
    dump(train_loss_dict, file)

# Save the validation loss values
with open('./val_loss.pkl', 'wb') as file:
    dump(val_loss_dict, file)

print("Total time taken: %.2fs" % (time() - start_time))

```

Listing 21.2: Modified code for training

21.3 Plotting the Training and Validation Loss Curves

In order to be able to plot the training and validation loss curves, you will first load the pickle files containing the training and validation loss dictionaries that you saved when training the transformer model earlier. Then you will retrieve the training and validation loss values from the respective dictionaries and graph them on the same plot.

The code listing is as follows, which you should save into a separate Python script:

```

from pickle import load
from matplotlib.pyplot import plt
from numpy import arange

# Load the training and validation loss dictionaries
train_loss = load(open('train_loss.pkl', 'rb'))
val_loss = load(open('val_loss.pkl', 'rb'))

# Retrieve each dictionary's values
train_values = train_loss.values()
val_values = val_loss.values()

# Generate a sequence of integers to represent the epoch numbers
epochs = range(1, 21)

# Plot and label the training and validation loss values plt.plot(epochs,
train_values, label='Training Loss') plt.plot(epochs, val_values,
label='Validation Loss')

# Add in a title and axes labels
plt.title('Training and Validation Loss')
plt.xlabel('Epochs')
plt.ylabel('Loss')

# Set the tick locations
plt.xticks(arange(0, 21, 2))

# Display the plot

```

```
plt.legend(loc='best')
plt.show()
```

Listing 21.3: The script for plotting the pickled data

Running the code above generates a similar plot of the training and validation loss curves to the one below:

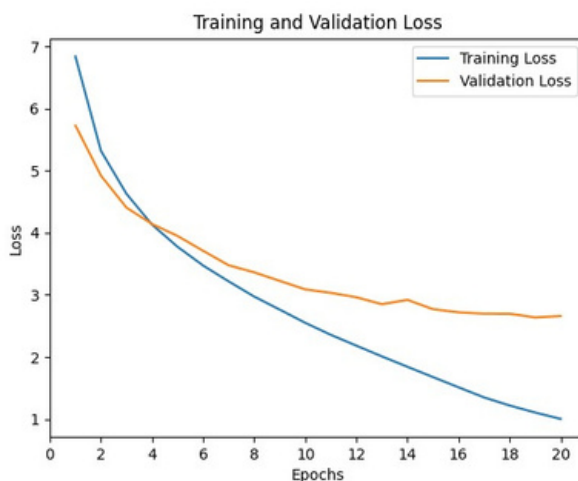


Figure 21.1: Line plots of the training and validation loss values over several training epochs

Note that although you might see similar loss curves, they might not necessarily be identical to the ones above. This is because you are training the transformer model from scratch, and the resulting training and validation loss values depend on the random initialization of the model weights. Nonetheless, these loss curves give us a better insight into how the learning performance changes over the number of epochs and help us diagnose any problems with learning that can lead to an underfit or an overfit model.

21.4 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

21.5 Summary

In this chapter, you discovered how to plot the training and validation loss curves for the transformer model. Specifically, you learned:

- ◀ How to modify the training code to include validation and test splits, in addition to a training split of the dataset
- ◀ How to modify the training code to store the computed training and validation loss values, as well as the trained model weights
- ◀ How to plot the saved training and validation loss curves

In the next chapter, you will learn how to make use of the trained transformer model.

Inference with the Transformer Model

22

We have seen how to train the transformer model on a dataset of English and German sentence pairs and how to plot the training and validation loss curves to diagnose the model's learning performance and decide at which epoch to run inference on the trained model. We are now ready to run inference on the trained transformer model to translate an input sentence.

In this chapter, you will discover how to run inference on the trained transformer model for neural machine translation. After completing this chapter, you will know:

◀ How to run inference on the trained transformer model

◀ How to generate text translations

Let's get started.

Overview

This chapter is divided into two parts; they are:

◀ Inferencing the Transformer Model

◀ Testing Out the Code

22.1 Inferencing the Transformer Model

Let's start by creating a new instance of the TransformerModel class that was previously implemented in Chapter 19.

You will feed into it the relevant input arguments as specified in the paper of “Attention Is All You Need” and the relevant information about the dataset in use:

```
# Define the model parameters
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of model layers' outputs
d_ff = 2048 # Dimensionality of the inner fully connected layer
n = 6 # Number of layers in the encoder stack
```

```
# Define the dataset parameters
enc_seq_length = 7 # Encoder sequence length
dec_seq_length = 12 # Decoder sequence length
enc_vocab_size = 2405 # Encoder vocabulary size
dec_vocab_size = 3858 # Decoder vocabulary size

# Create model
inferencing_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
                                     dec_seq_length, h, d_k, d_v, d_model, d_ff, n, 0)
```

Listing 22.1: Parameters from “Attention Is All You Need”

Here, note that the last input being fed into the `TransformerModel` corresponded to the dropout rate for each of the Dropout layers in the transformer model. These Dropout layers will not be used during model inferencing (you will eventually set the training argument to `False`), so you may safely set the dropout rate to 0. Furthermore, the `TransformerModel` class was already saved into a separate script named `model.py`. Hence, to be able to use the `TransformerModel`

class, you need to include “`from model import TransformerModel`”.

Next, let’s create a class `Translate` that inherits from the `Module` base class in Keras and assign the initialized inferencing model to the variable `transformer`:

```
class Translate(Module):
    def __init__(self, inferencing_model, **kwargs):
        super().__init__(**kwargs)
        self.transformer = inferencing_model
    ...
```

Listing 22.2: The `Translate` class

When you trained the transformer model, you saw that you first needed to tokenize the sequences of text that were to be fed into both the encoder and decoder. You achieved this by creating a vocabulary of words and replacing each word with its corresponding vocabulary index. You will need to implement a similar process during the inferencing stage before feeding the sequence of text to be translated into the transformer model.

For this purpose, you will include within the class the following `load_tokenizer` method, which will serve to load the encoder and decoder tokenizers that you would have generated and saved during the training stage in Chapter 21:

```
def load_tokenizer(self, name):
    with open(name, 'rb') as handle:
        return load(handle)
```

Listing 22.3: Function to load the trained tokenizers

It is important that you tokenize the input text at the inferencing stage using the same tokenizers generated at the training stage of the transformer model since these tokenizers would have already been trained on text sequences similar to your testing data. The next step is to create the class method, `call()`, that will take care to:

◀ Append the start (`<START>`) and end-of-string (`<EOS>`) tokens to the input sentence:

```
def __call__(self, sentence):
    sentence[0] = "<START> " + sentence[0] + " <EOS>"
```

- ◀ Load the encoder and decoder tokenizers (in this case, saved in the `enc_tokenizer.pkl` and `dec_tokenizer.pkl` pickle files from Chapter 21, respectively):

```
enc_tokenizer = self.load_tokenizer('enc_tokenizer.pkl')
dec_tokenizer = self.load_tokenizer('dec_tokenizer.pkl')
```

- ◀ Prepare the input sentence by tokenizing it first, then padding it to the maximum phrase length, and subsequently converting it to a tensor:

```
encoder_input = enc_tokenizer.texts_to_sequences(sentence)
encoder_input = pad_sequences(encoder_input,
                             maxlen=enc_seq_length, padding='post')
encoder_input = convert_to_tensor(encoder_input, dtype=int64)
```

- ◀ Repeat a similar tokenization and tensor conversion procedure for the `<START>` and `<EOS>` tokens at the output:

```
output_start = dec_tokenizer.texts_to_sequences(["<START>"])
output_start = convert_to_tensor(output_start[0], dtype=int64)

output_end = dec_tokenizer.texts_to_sequences(["<EOS>"])
output_end = convert_to_tensor(output_end[0], dtype=int64)
```

- ◀ Prepare the output array that will contain the translated text. Since you do not know the length of the translated sentence in advance, you will initialize the size of the output array to 0, but set its `dynamic_size` parameter to `True` so that it may grow past its initial size. You will then set the first value in this output array to the `<START>` token:

```
decoder_output = TensorArray(dtype=int64, size=0, dynamic_size=True)
decoder_output = decoder_output.write(0, output_start)
```

- ◀ Iterate, up to the decoder sequence length, each time calling the transformer model to predict an output token. Here, the training input, which is then passed on to each of the transformer's Dropout layers, is set to `False` so that no values are dropped during inference. The prediction with the highest score is then selected and written at the next available index of the output array. The for loop is terminated with a `break` statement as soon as an `<EOS>` token is predicted:

```
for i in range(dec_seq_length):
    prediction = self.transformer(encoder_input, transpose(decoder_output.stack()),
                                training=False)
```

```

prediction = prediction[:, -1, :]

predicted_id = argmax(prediction, axis=-1)
predicted_id = predicted_id[0][newaxis]

decoder_output = decoder_output.write(i + 1, predicted_id)

if predicted_id == output_end:
    break

```

◀ Decode the predicted tokens into an output list and return it:

```

output = transpose(decoder_output.stack())[0] output =
output.numpy()

output_str = []

# Decode the predicted tokens into an output list for
i in range(output.shape[0]):
    key = output[i]
    translation = dec_tokenizer.index_word[key]
    output_str.append(translation)

return output_str

```

The complete code listing, so far, is as follows:

```

from pickle import load
from tensorflow import Module
from tensorflow.keras.preprocessing.sequence import pad_sequences
from tensorflow import convert_to_tensor, int64, TensorArray, argmax, newaxis, transpose
from model import TransformerModel

# Define the model parameters
h = 8 # Number of self-attention heads
d_k = 64 # Dimensionality of the linearly projected queries and keys
d_v = 64 # Dimensionality of the linearly projected values
d_model = 512 # Dimensionality of model layers' outputs
d_ff = 2048 # Dimensionality of the inner fully connected layer
n = 6 # Number of layers in the encoder stack

# Define the dataset parameters
enc_seq_length = 7 # Encoder sequence length
dec_seq_length = 12 # Decoder sequence length
enc_vocab_size = 2404 # Encoder vocabulary size
dec_vocab_size = 3864 # Decoder vocabulary size

# Create model
inferencing_model = TransformerModel(enc_vocab_size, dec_vocab_size, enc_seq_length,
dec_seq_length, h, d_k, d_v, d_model, d_ff, n, 0)

```

```

class Translate(Module):
    def __init__(self, inferencing_model, **kwargs):
        super().__init__(**kwargs)
        self.transformer = inferencing_model

    def load_tokenizer(self, name):
        with open(name, 'rb') as handle:
            return load(handle)

    def __call__(self, sentence):
        # Append start and end of string tokens to the input sentence
        sentence[0] = "<START> " + sentence[0] + " <EOS>"

        # Load encoder and decoder tokenizers
        enc_tokenizer = self.load_tokenizer('enc_tokenizer.pkl')
        dec_tokenizer = self.load_tokenizer('dec_tokenizer.pkl')

        # Prepare the input sentence by tokenizing, padding and converting to tensor
        encoder_input = enc_tokenizer.texts_to_sequences(sentence)
        encoder_input = pad_sequences(encoder_input,
                                     maxlen=enc_seq_length, padding='post')
        encoder_input = convert_to_tensor(encoder_input, dtype=int64)

        # Prepare the output <START> token by tokenizing, and converting to tensor
        output_start = dec_tokenizer.texts_to_sequences(["<START>"])
        output_start = convert_to_tensor(output_start[0], dtype=int64)

        # Prepare the output <EOS> token by tokenizing, and converting to tensor
        output_end = dec_tokenizer.texts_to_sequences(["<EOS>"])
        output_end = convert_to_tensor(output_end[0], dtype=int64)

        # Prepare the output array of dynamic size
        decoder_output = TensorArray(dtype=int64, size=0, dynamic_size=True)
        decoder_output = decoder_output.write(0, output_start)

        for i in range(dec_seq_length):
            # Predict an output token
            prediction = self.transformer(encoder_input, transpose(decoder_output.stack()),
                                         training=False)
            prediction = prediction[:, -1, :]

            # Select the prediction with the highest score
            predicted_id = argmax(prediction, axis=-1)
            predicted_id = predicted_id[0][newaxis]

            # Write the selected prediction to the output array at the next
            # available index
            decoder_output = decoder_output.write(i + 1, predicted_id)

            # Break if an <EOS> token is predicted
            if predicted_id == output_end:
                break

        output = transpose(decoder_output.stack())[0]

```

```

output = output.numpy()

output_str = []

# Decode the predicted tokens into an output string
for i in range(output.shape[0]):
    key = output[i]
    output_str.append(dec_tokenizer.index_word[key])

return output_str

```

Listing 22.4: Complete code for inference

22.2 Testing Out the Code

In order to test out the code, let's have a look at the `test_dataset.txt` file that you would have saved when preparing the dataset for training in Chapter 21. This text file contains a set of English-German sentence pairs that you have been reserved for testing, from which you can select a couple of sentences to test.

Let's start with the first sentence:

```

# Sentence to translate
sentence = ['im thirsty']

```

Listing 22.5: A sentence for testing

The corresponding ground truth translation in German for this sentence, including the `<START>` and `<EOS>` decoder tokens, should be: “`<START> ich bin durstig <EOS>`”. If you have a look at the plotted training and validation loss curves for this model (here, you are training for 20 epochs), you may notice that the validation loss curve slows down considerably and starts plateauing at around epoch 16.

So let's proceed to load the saved model's weights at the 16th epoch and check out the prediction that is generated by the model:

```

# Load the trained model's weights at the specified epoch
inferencing_model.load_weights('weights/wgths16.ckpt')

# Create a new instance of the 'Translate' class translator =
Translate(inferencing_model)

# Translate the input sentence
print(translator(sentence))

```

Listing 22.6: Loading the model for inference

Running the lines of code above produces the following translated list of words:

```
['start', 'ich', 'bin', 'durstig', 'eos']
```

Output 22.1: Output of the model using weights from the 16th epoch

Which is equivalent to the ground truth German sentence that was expected.

INFOCIRCLE Always keep in mind that since you are training the transformer model from scratch, you may arrive at different results of the model weights.

Let's check out what would have happened if you had, instead, loaded a set of weights corresponding to a much earlier epoch, such as the 4th epoch. In this case, the generated translation is the following:

```
['start', 'ich', 'bin', 'nicht', 'nicht', 'eos']
```

Output 22.2: Output of the model using weights from the 4th epoch

In English, this translates to “*I in not not*”, which is clearly far off from the input English sentence, but which is expected since, at this epoch, the learning process of the transformer model is still at the very early stages.

Let's try again with a second sentence from the test dataset:

```
# Sentence to translate
sentence = ['are we done']
```

Listing 22.7: Another sentence for testing

The corresponding ground truth translation in German for this sentence, including the <START> and <EOS> decoder tokens, should be: “<START> sind wir dann durch <EOS>”. The model's translation for this sentence, using the weights saved at epoch 16, is:

```
['start', 'ich', 'war', 'fertig', 'eos']
```

Output 22.3: Output of the model using weights from the 16th epoch

Which, instead, translates to: “*I was ready*”. While this is also not equal to the ground truth, it is *close* to its meaning.

What the last test suggests, however, is that the transformer model might have required many more data samples to train effectively. This is also corroborated by the validation loss at which the validation loss curve plateaus remain relatively high. Indeed, transformer models are notorious for being very data hungry. Vaswani et al. (2017), for example, trained their English-to-German translation model using a dataset containing around 4.5 million sentence pairs.

“We trained on the standard WMT2014 English-German dataset consisting of

you. If you have the computational resources available, try to train the transformer model on a much larger set of sentence pairs and see if you can obtain better results than the translations obtained here with a limited amount of data.

22.3 Further Reading

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

Books

Ivan Vasilev. *Advanced Deep Learning with Python*. Packt Publishing, 2019.

<https://www.amazon.com/dp/178995617X>

Denis Rothman. *Transformers for Natural Language Processing*. Packt Publishing, 2021.

<https://www.amazon.com/dp/1800565798>

Papers

Ashish Vaswani et al. “Attention Is All You Need”. In: *Proc. 31st Conference on Neural Information Processing Systems (NIPS 2017)*. 2017.

<https://arxiv.org/pdf/1706.03762.pdf>

22.4 Summary

In this chapter, you discovered how to inference the trained transformer model for neural machine translation. Specifically, you learned:

- ◀ How to inference the trained transformer model
- ◀ How to generate text translations

In the next chapter, you will see how you can use a pre-trained model.

App
IllicV

A Brief Introduction to BERT

23

As we learned what a Transformer is and how we might train the Transformer model, we notice that it is a great tool to make a computer understand human language. However, the Transformer was originally designed as a model to translate one language to another. If we repurpose it for a different task, we would likely need to retrain the whole model from scratch. Given the time it takes to train a Transformer model is enormous, we would like to have a solution that enables us to readily reuse the trained Transformer for many different tasks. BERT is such a model. It is an extension of the encoder part of a Transformer.

In this chapter, you will learn what BERT is and discover what it can do. After completing this chapter, you will know:

- ◁ What is a Bidirectional Encoder Representations from Transformer (BERT)
- ◁ How a BERT model can be reused for different purposes
- ◁ How you can use a pre-trained BERT model

Let's get started.

Overview

This chapter is divided into four parts; they are:

- ◁ From Transformer Model to BERT
- ◁ What Can BERT Do?
- ◁ Using Pre-Trained BERT Model for Summarization
 - ◁ Using Pre-Trained BERT Model for Question-Answering

23.1 From Transformer Model to BERT

In the transformer model, the encoder and decoder are connected to make a seq2seq model in order for you to perform a translation, such as from English to German, as you saw before.

Recall that the attention equation says:

$$(\text{attention}_{Q,T})(Q,K,V)=\text{softmax}(Vd_k)$$

But each of the Q , K , and V above is an embedding vector transformed by a weight matrix in the transformer model. Training a transformer model means finding these weight matrices. Once the weight matrices are learned, the transformer becomes a *language model*, which means it represents a way to understand the language that you used to train it.

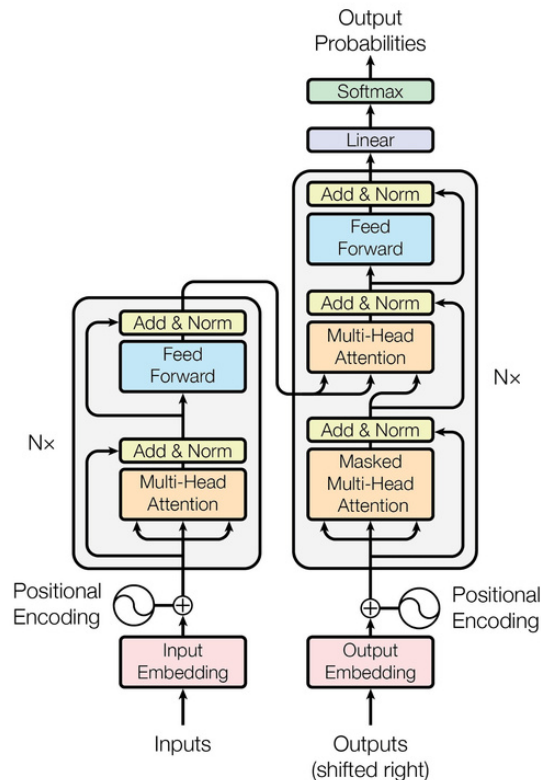


Figure 23.1: The encoder-decoder structure of the Transformer architecture. From “Attention Is All You Need”

A transformer has encoder and decoder parts. As the name implies, the encoder transforms sentences and paragraphs into an internal format (a numerical matrix) that understands the context, whereas the decoder does the reverse. Combining the encoder and decoder allows a transformer to perform seq2seq tasks, such as translation. If you take out the encoder part of the transformer, it can tell you something about the context, which can do something interesting.

The Bidirectional Encoder Representation from Transformer (BERT) leverages the attention model to get a deeper understanding of the language context. BERT is a stack of many encoder blocks. The input text is separated into tokens as in the transformer model, and each token will be transformed into a vector at the output of BERT.

23.2 What Can BERT Do?

A BERT model is trained using the *masked language model* (MLM) and *next sentence prediction* (NSP) simultaneously.

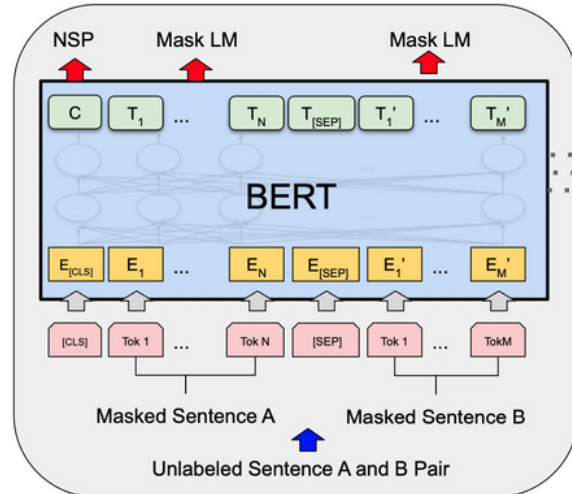


Figure 23.2: The BERT model

Each training sample for BERT is a pair of sentences from a document. The two sentences can be consecutive in the document or not. There will be a [CLS] token prepended to the first sentence (to represent the *class*) and a [SEP] token appended to each sentence (as a *separator*). Then, the two sentences will be concatenated as a sequence of tokens to become a training sample. A small percentage of the tokens in the training sample is *masked* with a special token [MASK] or replaced with a random token.

Before it is fed into the BERT model, the tokens in the training sample will be transformed into embedding vectors, with the positional encodings added, and particular to BERT, with *segment embeddings* added as well to mark whether the token is from the first or the second sentence.

Each input token to the BERT model will produce one output vector. In a well-trained BERT model, we expect:

◀ output corresponding to the masked token can reveal what the original token was

◀ output corresponding to the [CLS] token at the beginning can reveal whether the two sentences are consecutive in the document

Then, the weights trained in the BERT model can understand the language context well.

Once you have such a BERT model, you can use it for many *downstream tasks*. For example, by adding an appropriate classification layer on top of an encoder and feeding in only one sentence to the model instead of a pair, you can take the class token [CLS] as input

for

sentiment classification. It works because the output of the class token is trained to aggregate the attention for the entire input.

Another example is to take a question as the first sentence and the text (e.g., a paragraph) as the second sentence, then the output token from the second sentence can mark the position

where the answer to the question rested. It works because the output of each token reveals some information about that token in the context of the entire input.

23.3 Using Pre-Trained BERT Model for Summarization

A transformer model takes a long time to train from scratch. The BERT model would take even longer. But the purpose of BERT is to create one model that can be reused for many different tasks.

There are pre-trained BERT models that you can use readily. In the following, you will see a few use cases. The text used in the following example is from:

<

<https://www.project-syndicate.org/commentary/bank-of-england-gilt-purchases-necessary-but-mistakes-made-by-willem-h-buiter-and-anne-c-sibert-2022-10>

Theoretically, a BERT model is an encoder that maps each input token to an output vector, which can be extended to an infinite length sequence of tokens. In practice, there are limitations

imposed in the implementation of other components that limit the input size. Mostly, a few hundred tokens should work, as not every implementation can take thousands of tokens in one shot. You can save the entire article in `article.txt`. In case your model needs a smaller text, you can use only a few paragraphs from it.

First, let's explore the task for summarization. Using BERT, the idea is to *extract* a few sentences from the original text that represent the entire text. You can see this task is similar to next sentence prediction, in which if given a sentence and the text, you want to classify if they are related.

To do that, you need to use the Python module `bert-extractive-summarizer`

```
pip install bert-extractive-summarizer
```

Listing 23.1: Installing Python module

It is a wrapper to some Hugging Face models to provide the summarization task pipeline. Hugging Face is a platform that allows you to publish machine learning models, mainly on NLP tasks.

Once you have installed `bert-extractive-summarizer`, producing a summary is just a few lines of code:

```
from summarizer import Summarizer
text = open("article.txt").read()
model = Summarizer('distilbert-base-uncased')
result = model(text, num_sentences=3)
print(result)
```

Listing 23.2: Generate extractive summary from text using BERT

This gives the output:

Amid the political turmoil of outgoing British Prime Minister Liz Truss's short-lived government, the Bank of England has found itself in the fiscal-financial crossfire. Whatever government comes next, it is vital that the BOE learns the right lessons. According to a statement by the BOE's Deputy Governor for Financial Stability, Jon Cunliffe, the MPC was merely "informed of the issues in the gilt market and briefed in advance of the operation, including its financial-stability rationale and the temporary and targeted nature of the purchases."

Output 23.1: Extractive summary produced

That's the complete code! Behind the scene, spaCy was used on some preprocessing, and Hugging Face was used to launch the model. The model used was named `distilbert-base-uncased`. DistilBERT is a simplified BERT model that can run faster and use less memory. The model is an "uncased" one, which means the uppercase or lowercase in the input text is considered the same once it is transformed into embedding vectors.

The output from the summarizer model is a string. As you specified `num_sentences=3` in invoking the model, the summary is three selected sentences from the text. This approach is called the *extractive summary*. The alternative is an *abstractive summary*, in which the summary is generated rather than extracted from the text. This would need a different model than BERT.

23.4 Using Pre-Trained BERT Model for Question-Answering

The other example of using BERT is to match questions to answers. You will give both the question and the text to the model and look for the output of the beginning *and* the end of the answer from the text.

A quick example would be just a few lines of code as follows, reusing the same example text as in the previous example:

```
from transformers import pipeline
text = open("article.txt").read()
question = "What is BOE doing?"

answering = pipeline("question-answering",
                      model='distilbert-base-uncased-distilled-squad')
result = answering(question=question, context=text)
print(result)
```

Listing 23.3: Question-answering using BERT

Here, Hugging Face is used directly. If you have installed the module used in the previous example, the Hugging Face Python module is a dependence that you already installed. Otherwise, you may need to install it with pip:

```
pip install transformers
```

Listing 23.4: Installing Hugging Face module

And to actually use a Hugging Face model, you should have *both* PyTorch and TensorFlow installed as well:

```
pip install torch tensorflow
```

Listing 23.5: Installing PyTorch and TensorFlow modules

The output of the code above is a Python dictionary, as follows:

```
{'score': 0.42369240522384644,  
'start': 1261,  
'end': 1344,  
'answer': 'to maintain or restore market liquidity in systemically  
important\nfinancial markets'}
```

✱

23.6 Summary

In this chapter, you discovered what BERT is and how to use a pre-trained BERT model. Specifically, you learned:

- ◀ How is BERT created as an extension to Transformer models
 - ◀ How to use pre-trained BERT models for extractive summarization and question answering

This marks the end of this book.

Appe
Vnd

How to Setup Python on Your Workstation



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 software.

A.1 Overview

In this chapter, we will cover the following steps:

1. Download Anaconda
2. Install Anaconda
3. Start and Update Anaconda
4. Install Deep Learning Libraries
5. Install Visual Studio Code environment

Note: The specific versions may differ as the software and libraries are updated frequently.

INFOCIRCLE

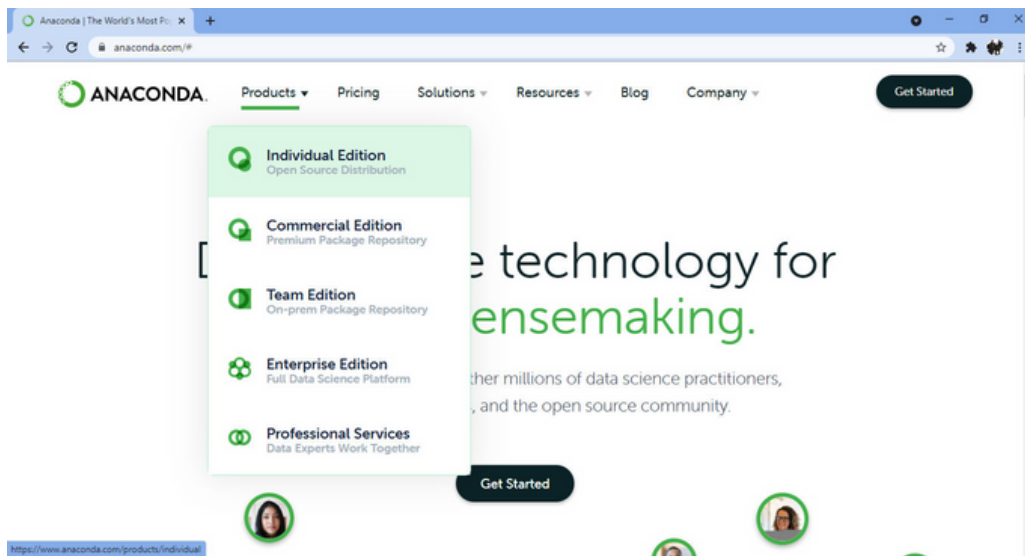


Figure A.1: Click “Products” and “Individual Edition”

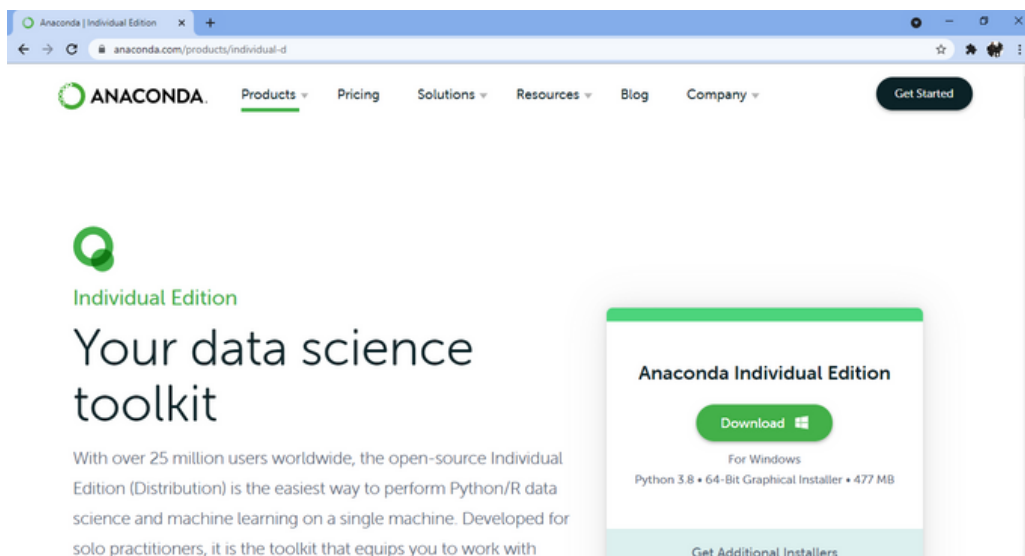


Figure A.2: Click Download

This will download the Anaconda Python package to your workstation. It will automatically give you the installer according to your OS (Windows, Linux, or MacOS). The file is about 480 MB. You should have a file with a name like:

```
Anaconda3-2021.05-Windows-x86_64.exe
```

A.3 Install Anaconda

In this step, you 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.

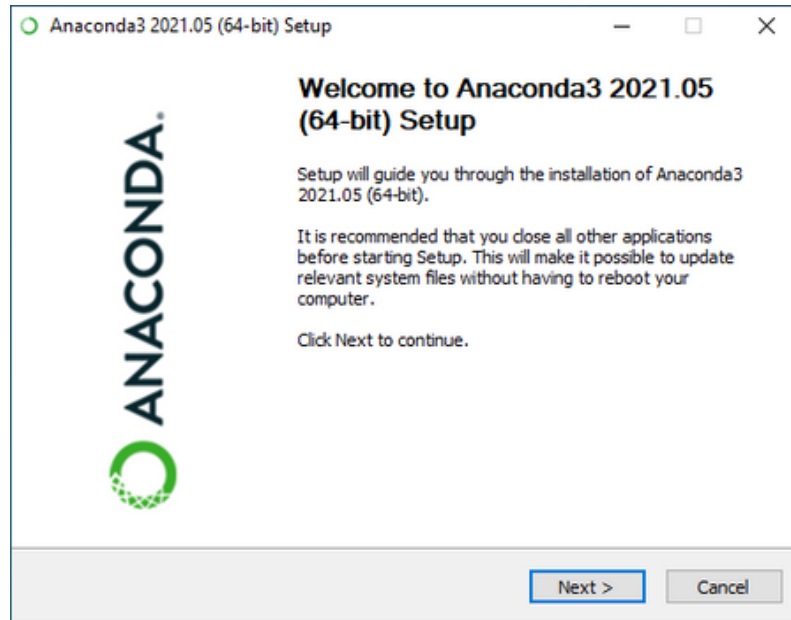


Figure A.3: Anaconda Python Installation Wizard

Installation is quick and painless. There should be no tricky questions or sticking points.

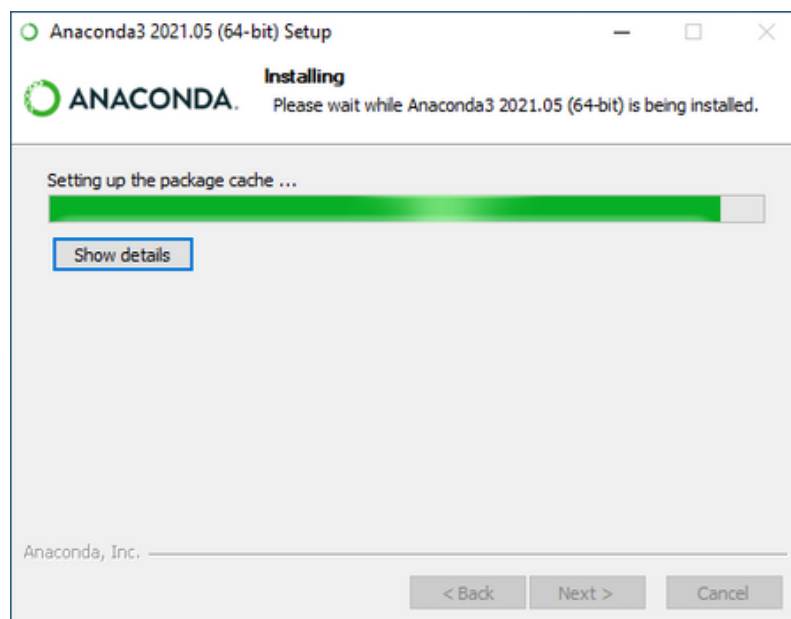


Figure A.4: Anaconda Python Installation Wizard Writing Files

The installation should take less than 10 minutes and take a bit more than 5 GB of space on your hard drive.

A.4 Start and Update Anaconda

In this step, you 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.

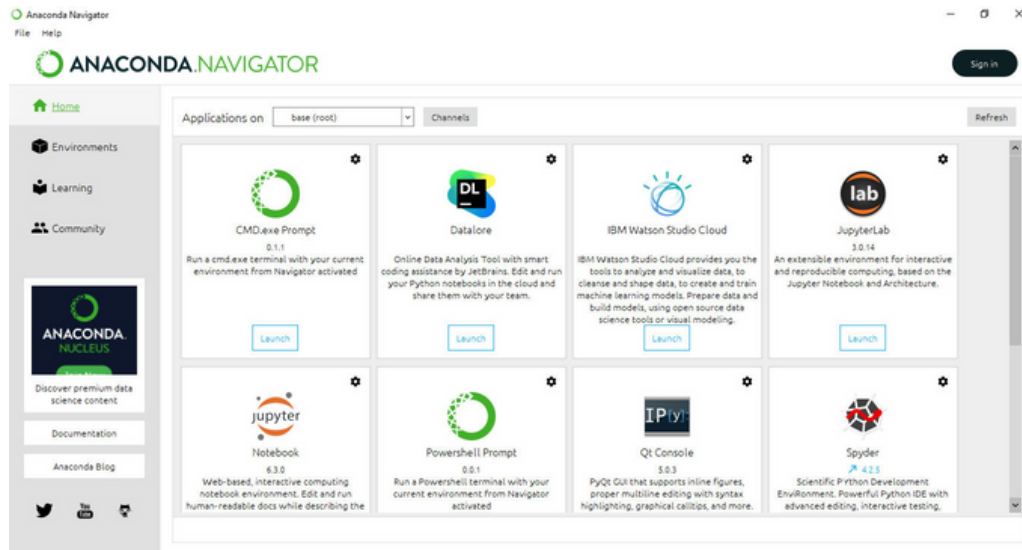


Figure A.5: Anaconda Navigator GUI

You can use the Anaconda Navigator and graphical development environments later; for now, it is recommended to start with the Anaconda command line environment called `conda1`. 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 or CMD.exe prompt (command line window).
2. Confirm conda is installed correctly, by typing:

```
conda -V
```

You should see the following (or something similar):

```
conda 4.10.1
```

3. Confirm Python is installed correctly by typing:

```
python -V
```

You should see the following (or something similar):

```
Python 3.8.8
```

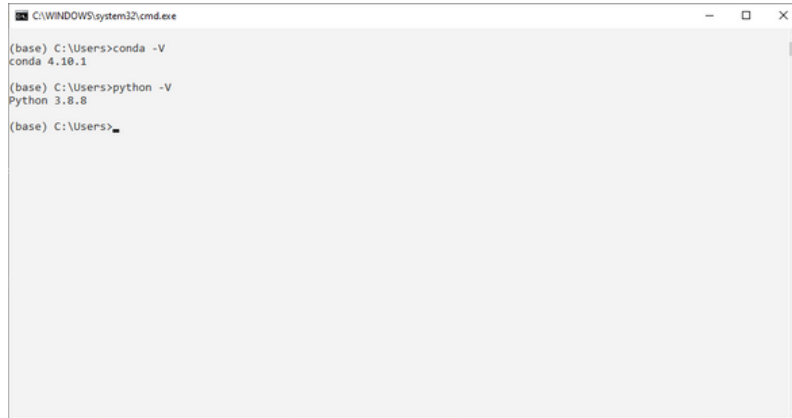
A screenshot of a Windows command prompt window titled 'C:\WINDOWS\system32\cmd.exe'. The window shows the following text: (base) C:\Users>conda -V, conda 4.10.1, (base) C:\Users>python -V, Python 3.8.8, and (base) C:\Users> with a cursor at the end. The window has standard Windows window controls (minimize, maximize, close) in the top right corner.

Figure A.6: Confirm Conda and Python are Installed

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. This step is optional. You can make community-supported packages available in conda by adding a “conda-forge” channel:

```
conda config --add channels conda-forge
```

5. Confirm your conda environment is up-to-date, type:

```
conda update conda
conda update anaconda
```

You may need to install some packages and confirm the updates.

6. 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, it is recommended to open a text editor and copy-pasting the script into your editor.

```
# 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 A.1: Code to check that key Python libraries are installed.

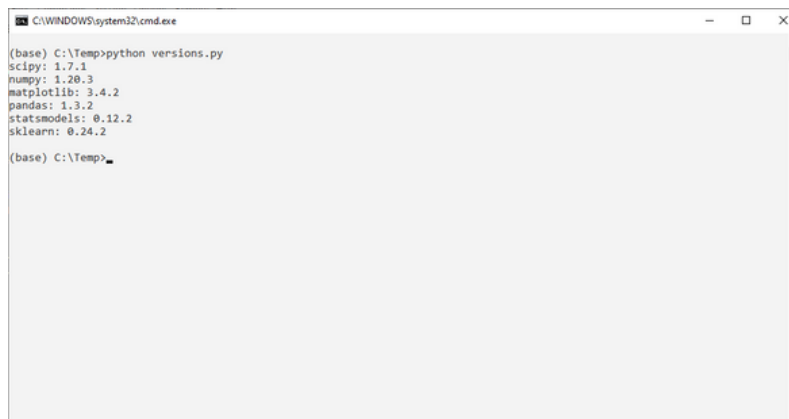
Save the script as a file with the name: `versions.py`. On the command line, change your directory to where you saved the script and type:

```
python versions.py
```

You should see output like the following:

```
scipy: 1.8.1
numpy: 1.23.1
matplotlib: 3.5.1
pandas: 1.4.3
statsmodels: 0.13.2
sklearn: 1.1.1
```

Output A.1: Sample output of thhe versions script

A screenshot of a Windows command prompt window titled "C:\WINDOWS\system32\cmd.exe". The prompt shows the command `(base) C:\Temp>python versions.py` being executed. The output lists the versions of several Python libraries: `scipy: 1.7.1`, `numpy: 1.20.3`, `matplotlib: 3.4.2`, `pandas: 1.3.2`, `statsmodels: 0.12.2`, and `sklearn: 0.24.2`. The prompt then returns to `(base) C:\Temp>`.

```
C:\WINDOWS\system32\cmd.exe
(base) C:\Temp>python versions.py
scipy: 1.7.1
numpy: 1.20.3
matplotlib: 3.4.2
pandas: 1.3.2
statsmodels: 0.12.2
sklearn: 0.24.2
(base) C:\Temp>
```

Figure A.7: Confirm Anaconda SciPy environment

A.5 Install Deep Learning Libraries

In this step, you will install Python libraries used for deep learning, specifically: TensorFlow and Keras.

1. Install the TensorFlow and Keras deep learning library by typing:

```
conda install tensorflow keras
```

2. Alternatively, you may choose to install using `pip` and a specific version of TensorFlow for your platform

```
pip install tensorflow==2.9.0
```

3. Confirm your deep learning environment is installed and working correctly.

Create a script that prints the version numbers of each library, as you did before for the SciPy environment.

```
# check deep learning version numbers
# tensorflow
import tensorflow
print('tensorflow: %s' % tensorflow.__version__) # keras
import keras
print('keras: %s' % keras.__version__)
```

Listing A.2: 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
```

Output A.2: Run script from the command line.

You should see output like:

```
tensorflow: 2.9.2
keras: 2.9.0
```

Output A.3: Sample output of the deep learning versions script

A.6 Install Visual Studio Code for Python

If you have installed Anaconda, you will have an IDE called Spyder installed. You can develop your Python project with Spyder. The other popular way of writing Python code nowadays is to use Visual Studio Code. It is a free programming environment from Microsoft. Visual Studio Code can do a lot of things via “extensions”. Python extension is what you should get.

These instructions are suitable for Windows, macOS, and Linux platforms. Below, macOS is used to demonstrate, so you may see some Windows dialogs and file extensions.

1. Visit the Visual Studio Code homepage <https://code.visualstudio.com>

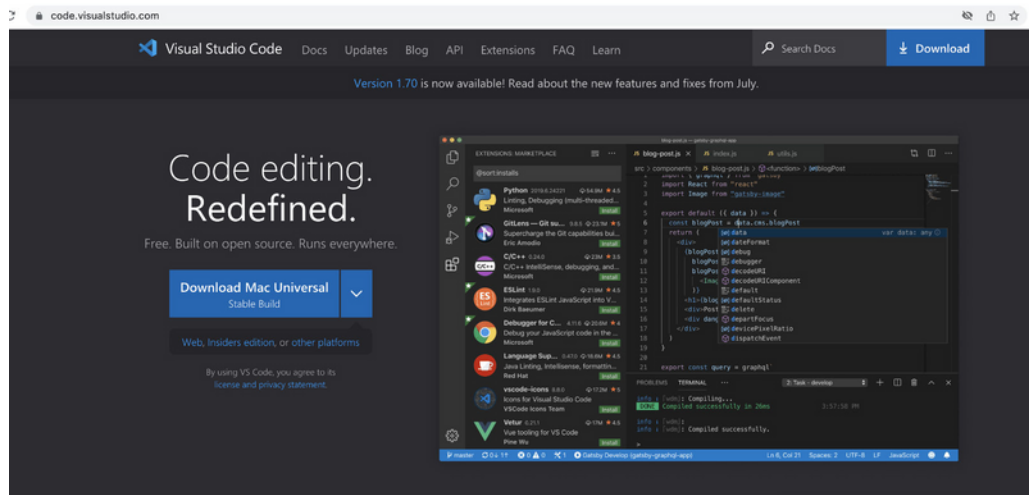


Figure A.8: Click “Products” and “Individual Edition”

2. Click the download button at the top toolbar or at the center of the screen to download a ZIP file (such as VSCode-darwin-universal.zip). It is around 200MB in size

3. Expand the ZIP you will find the application program. In macOS, you should move it into your /Applications folder

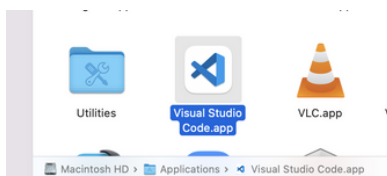


Figure A.9: Visual Studio Code in /Applications in macOS

4. Running Visual Studio Code will show you the main screen like the following:

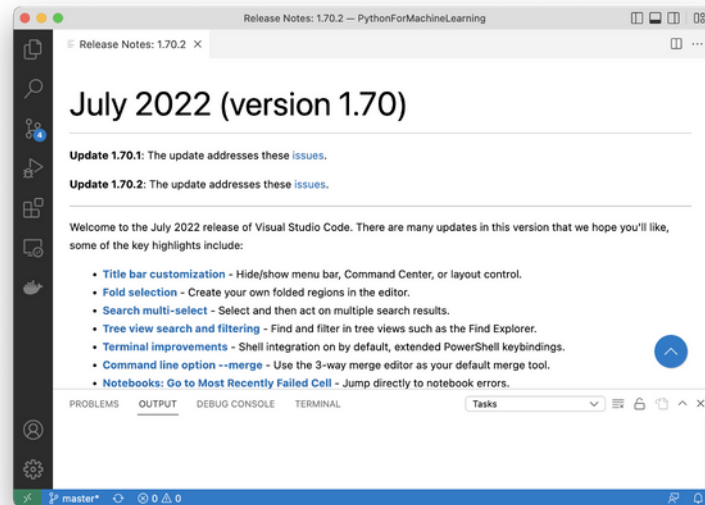


Figure A.10: Visual Studio Code main screen

You can click on the building block icon at left toolbar to open the extensions marketplace. Typing “python” on the search box will usually show the Microsoft-developed Python extension at top.

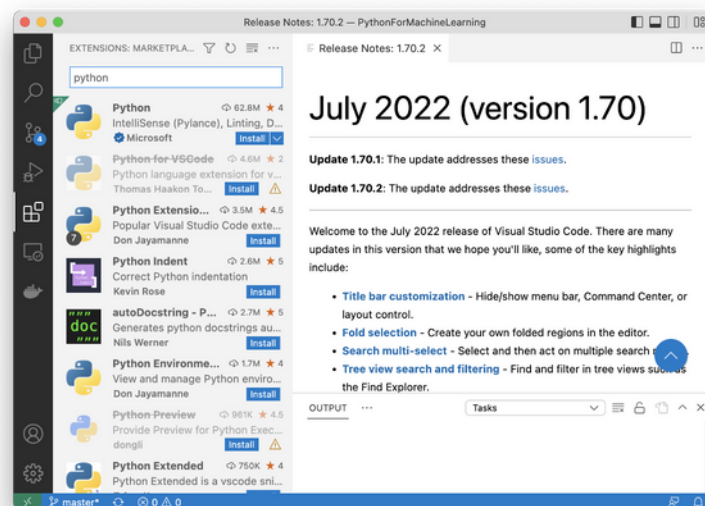


Figure A.11: Searching for Python extension in the extensions marketplace

5. Click on the “Install” button on the extension marketplace will get the extension installed. It should be very quick.