365 DataScience A common CNN architecture

```
# We will use the MNIST dataset to show an easy yet effective CNN archi
tecture

# Importing the relevant packages
import tensorflow as tf
import tensorflow_datasets as tfds
```

Downloading and preprocessing the data

```
# Before continuing with our model and training, our first job is to pr
eprocess the dataset
# This is a very important step in all of machine learning

# The MNIST dataset is, in general, highly processed already - after al
l its 28x28 grayscale images of clearly visible digits
# Thus, our preprocessing will be limited to scaling the pixel values,
shuffling the data and creating a validation set

# NOTE: When finally deploying a model in practice, it might be a good
idea to include the prerpocessing as initial layers
# In that way, the users could just plug the data (images) directly, in
stead of being required to resize/rescale it before

# Defining some constants/hyperparameters
BUFFER_SIZE = 70_000 # for reshuffling
BATCH_SIZE = 128
NUM_EPOCHS = 20

# Downloading the MNIST dataset

# When 'with_info' is set to True, tfds.load() returns two variables:
# - the dataset (including the train and test sets)
# - meta info regarding the dataset itself

mnist_dataset, mnist_info = tfds.load(name='mnist', with_info=True, as_
supervised=True)

# Extracting the train and test datasets
mnist_train, mnist_test = mnist_dataset['train'], mnist_dataset['test']

# Creating a function to scale our image data (it is recommended to sca
le the pixel values in the range [0,1] )
def scale(image, label):
    image = tf.cast(image, tf.float32)
    image /= 255.

    return image, label
```

```
# Scaling the data
train_and_validation_data = mnist_train.map(scale)
test_data = mnist_test.map(scale)

# Defining the size of the validation set
num_validation_samples = 0.1 * mnist_info.splits['train'].num_examples
num_validation_samples = tf.cast(num_validation_samples, tf.int64)

# Defining the size of the test set
num_test_samples = mnist_info.splits['test'].num_examples
num_test_samples = tf.cast(num_test_samples, tf.int64)

# Reshuffling the dataset
train_and_validation_data = train_and_validation_data.shuffle(BUFFER_SI
ZE)

# Splitting the dataset into training + validation
train_data = train_and_validation_data.skip(num_validation_samples)
validation_data = train_and_validation_data.take(num_validation_samples)

# Batching the data
# NOTE: For proper functioning of the model, we need to create one big
batch for the validation and test sets
train_data = train_data.batch(BATCH_SIZE)
validation_data = validation_data.batch(num_validation_samples)
test_data = test_data.batch(num_test_samples)
```

Creating the model and training it

```
# Now that we have preprocessed the dataset, we can define our CNN and
train it

# Outlining the model/architecture of our CNN
# CONV -> MAXPOOL -> CONV -> MAXPOOL -> FLATTEN -> DENSE
model = tf.keras.Sequential([
    tf.keras.layers.Conv2D(64, 5, activation='relu', input_shape=(28, 2
8, 1)),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    # (2,2) is the default pool size so we could have just used MaxPool
ing2D() with no explicit arguments
    tf.keras.layers.Conv2D(64, 3, activation='relu'),
    tf.keras.layers.MaxPooling2D(pool_size=(2,2)),
    tf.keras.layers.Flatten(),
    tf.keras.layers.Dense(10) # You can apply softmax activation here,
see below for comentary
])

# The hyperparameters of this model (such as the number of filters) may
 need to be changed when dealing with a different dataset

# A brief summary of the model and parameters
model.summary(line_length = 75)
```

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 24, 24, 50)        1300
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 12, 12, 50)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 10, 10, 50)        22550
_________________________________________________________________
max_pooling2d_1 (MaxPooling2D) (None, 5, 5, 50)        0
_________________________________________________________________
flatten (Flatten)            (None, 1250)              0
_________________________________________________________________
dense (Dense)                (None, 10)                12510
=================================================================
Total params: 36,360
Trainable params: 36,360
Non-trainable params: 0
_________________________________________________________________
```

```
# Defining the loss function

# In general, our model needs to output probabilities of each class,
# which can be achieved with a softmax activation in the last dense lay
er

# However, when using the softmax activation, the loss can rarely be un
stable

# Thus, instead of incorporating the softmax into the model itself,
# we use a loss calculation that automatically corrects for the missing
 softmax
```

```
# That is the reason for 'from_logits=True'
loss_fn = tf.keras.losses.SparseCategoricalCrossentropy(from_logits=Tru
e)

# Compiling the model with Adam optimizer and the cathegorical crossent
ropy as a loss function
model.compile(optimizer='adam', loss=loss_fn, metrics=['accuracy'])

# Defining early stopping to prevent overfitting
early_stopping = tf.keras.callbacks.EarlyStopping(
    monitor = 'val_loss',
    mode = 'auto',
    min_delta = 0,
    patience = 2,
    verbose = 0,
    restore_best_weights = True
)

# Train the network
model.fit(
    train_data,
    epochs = NUM_EPOCHS,
    callbacks = [early_stopping],
    validation_data = validation_data,
    verbose = 2
)
```

```
Epoch 1/20
422/422 - 18s - loss: 0.2680 - accuracy: 0.9271 - val_loss: 0.0882 - va
l_accuracy: 0.9737
Epoch 2/20
422/422 - 18s - loss: 0.0696 - accuracy: 0.9792 - val_loss: 0.0692 - va
l_accuracy: 0.9817
Epoch 3/20
422/422 - 20s - loss: 0.0528 - accuracy: 0.9841 - val_loss: 0.0374 - va
l_accuracy: 0.9883
Epoch 4/20
422/422 - 20s - loss: 0.0412 - accuracy: 0.9874 - val_loss: 0.0291 - va
l_accuracy: 0.9917
Epoch 5/20
422/422 - 20s - loss: 0.0368 - accuracy: 0.9885 - val_loss: 0.0281 - va
l_accuracy: 0.9913
Epoch 6/20
422/422 - 21s - loss: 0.0304 - accuracy: 0.9905 - val_loss: 0.0251 - va
l_accuracy: 0.9918
Epoch 7/20
422/422 - 20s - loss: 0.0277 - accuracy: 0.9909 - val_loss: 0.0241 - va
l_accuracy: 0.9925
Epoch 8/20
422/422 - 19s - loss: 0.0230 - accuracy: 0.9928 - val_loss: 0.0216 - va
```

```
l_accuracy: 0.9928
Epoch 9/20
422/422 - 20s - loss: 0.0212 - accuracy: 0.9931 - val_loss: 0.0151 - va
l_accuracy: 0.9948
Epoch 10/20
422/422 - 19s - loss: 0.0185 - accuracy: 0.9938 - val_loss: 0.0111 - va
l_accuracy: 0.9970
Epoch 11/20
422/422 - 19s - loss: 0.0159 - accuracy: 0.9947 - val_loss: 0.0077 - va
l_accuracy: 0.9975
Epoch 12/20
422/422 - 19s - loss: 0.0153 - accuracy: 0.9951 - val_loss: 0.0141 - va
l_accuracy: 0.9947
Epoch 13/20
422/422 - 19s - loss: 0.0123 - accuracy: 0.9962 - val_loss: 0.0081 - va
l_accuracy: 0.9972

<tensorflow.python.keras.callbacks.History at 0x279e6bc6e80>
```

Testing our model

```
# Testing our model
test_loss, test_accuracy = model.evaluate(test_data)
```

```
1/1 [==============================] - ETA: 0s - loss: 0.0266 - accurac
y: 0.99 - 0s 2ms/step - loss: 0.0266 - accuracy: 0.9916
```

```
# Printing the test results
print('Test loss: {0:.4f}. Test accuracy: {1:.2f}%'.format(test_loss, t
est_accuracy*100.))
```

```
Test loss: 0.0266. Test accuracy: 99.16%
```

Start your 365 Journey!