UNIVERSITY OF LIMERICK
OLLSCOIL LUIMNIGH

E&CE | Department of Electronic and Computer Engineering

Module	CE4041 Artificial Intelligence
Assignment	Assignment 1: Keras/TensorFlow MNIST Digit Classifier
Lecturer	Dr Colin Flanagan
Student Name	Ronan Keaveney
Student ID	19125356
Date of Submission	12/11/2021

Table of Contents

1. Introduction

The goal of this assignment is to develop a program in Python, using the Keras neural network library, which takes in 28x28 pixel images of handwritten numbers from the MNIST database, and classifies them as a digit from 1 to 10. This 10-way decision function must achieve at least a 99% classification accuracy on the testing set, after being trained using the training set. This will be done with the use of a convolutional neural network.

When beginning this project, first I did some research about convolutional neural networks [1], and had a look at some of the existing examples for solving the above problem. I found a few examples on Kaggle [3][4], but after a while I realised that they were a bit over the top for my project's specifications and were taking too long to execute on my laptop (45+ mins in some cases). I found another tutorial online [2], which was a bit more basic and less computationally expensive, which is sufficient for my specific use case and available hardware. I found these tutorials very helpful for gaining an understanding of the fundamentals of CNNs.

I ran my model 10 times using the code below, and achieved an average test accuracy of 99.13%, with a standard deviation of . The average test loss was 3.02%. On my laptop, the average model runtime was 18 minutes and 30 seconds.

2. Data Preparation

2.1 Import necessary libraries

I began by importing all the libraries I will need for this program:

```
# Import libraries
from tensorflow import keras
from numpy import mean
from numpy import std
from matplotlib import pyplot
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPool2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.optimizers import SGD
from keras.optimizers import RMSprop
from keras.callbacks import EarlyStopping
import time
```

2.2 Load the MNIST Data Set

I began by logging the start time and loading the MNIST data set. I printed the first 9 label values of both the training and testing set, just to check that they were integers between 0 and 9. I also plotted the first training 9 images, to check that they matched the labels.

```
# Load dataset
(trainX, trainY), (testX, testY) = mnist.load_data()

# Print label values to check that thet are integers from 0 to 9
print("First 9 training labels:\n", trainY[:9])
print("First 9 testing labels:\n", testY[:9])

for i in range(9):
    pyplot.subplot(330 + 1 + i)
    pyplot.imshow(trainX[i], cmap=pyplot.get_cmap('gray'))
pyplot.show()
```

```
First 100 training labels:
 [5 0 4 1 9 2 1 3 1 4 3 5 3 6 1 7 2 8 6 9 4 0 9 1 1 2 4 3 2 7 3 8 6 9 0 5 6
 0 7 6 1 8 7 9 3 9 8 5 9 3 3 0 7 4 9 8 0 9 4 1 4 4 6 0 4 5 6 1 0 0 1 7 1 6
 3 0 2 1 1 7 9 0 2 6 7 8 3 9 0 4 6 7 4 6 8 0 7 8 3 1]
First 100 testing labels:
 [7 2 1 0 4 1 4 9 5 9 0 6 9 0 1 5 9 7 3 4 9 6 6 5 4 0 7 4 0 1 3 1 3 4 7 2 7
 1 2 1 1 7 4 2 3 5 1 2 4 4 6 3 5 5 6 0 4 1 9 5 7 8 9 3 7 4 6 4 3 0 7 0 2 9
 1 7 3 2 9 7 7 6 2 7 8 4 7 3 6 1 3 6 9 3 1 4 1 7 6 9]
```

Figure 1: First 100 label values of the training and testing sets

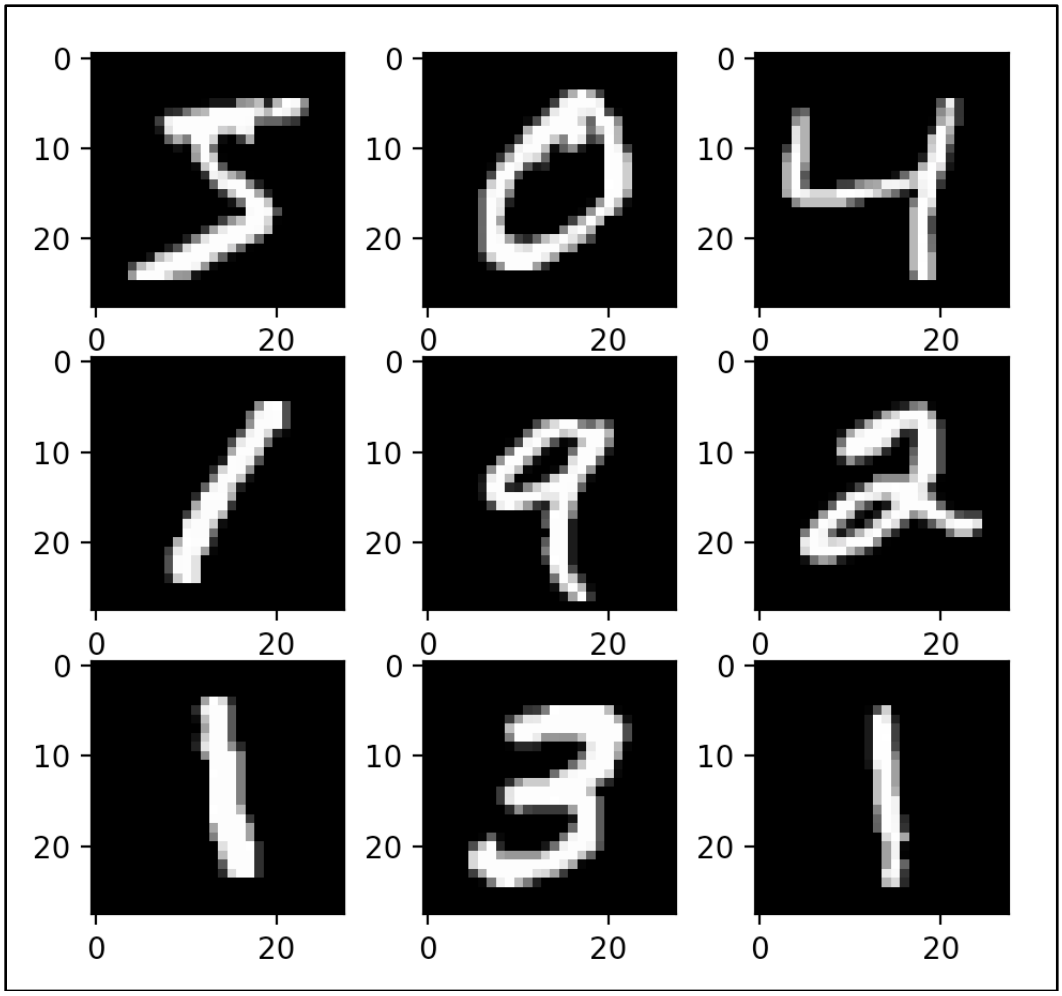

Figure 2: First 9 input images from the training set

As expected, there's a fairly even distribution of integers between 0 and 9 in the label data sets, and the first 9 input training images match up with the first 9 training labels.

2.3 Reshaping

Since we know all the images are 28x28 greyscale images, we can reshape them to this size and with only a single colour channel. If we were dealing with colour images, I would have to reshape the data to (28,28,3), to allocate channels for red, green and blue.

```
# Reshape dataset to have a single channel
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
```

2.4 Label Encoding

Since we know there are 10 different categories that the images can fall under, we can use one hot encoding to transform the integer label data sets into binary vectors with a 1 for the index of the correct value, and 0 for all other indices.

```
# One hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)
```

2.5 Normalization

The data for each pixel is encoded as an uint8, and ranges from 0 to 255. By converting this type to a float and dividing by 255, we can make every pixel's data lie between 0 and 1. This increases learning performance and leads to faster convergence [6].

```
# Normalize pixels to floats between 0 and 1
trainX = trainX.astype('float32')/255.0
testX = testX.astype('float32')/255.0
```

3. CNN Modelling

3.1 Model Definition

Next, I defined the CNN model. I decided to use 2 Conv2D/MaxPool2D layers in series to get the greatest test accuracy possible. The Conv2D layer is responsible for detecting patterns, and the MaxPool2D layer downsizes the image do reduce computational complexity and also helps when detecting corners and edges in the image. I also used dropout layers to reduce overfitting.

I finished with a Dense layer with 10 outputs and a softmax activation mode. This means that it will return a vector with a 1 at the index of the predicted label value and 0s everywhere else.

The I generated a summary of the model, which can be seen in Figure 3 o the next page.

```
# Define CNN model
model = Sequential()
model.add(Conv2D(filters=24,
                 kernel_size=4,
                 padding='same',
                 activation='relu',
                 input_shape=(28, 28, 1)
                 ))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))

model.add(Conv2D(filters=48,
                 kernel_size=4,
                 padding='same',
                 activation='relu',
                 ))
model.add(MaxPool2D(2,2))

model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Summarize model
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 28, 28, 24)        408
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 14, 14, 24)        0
_________________________________________________________________
dropout_1 (Dropout)          (None, 14, 14, 24)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 14, 14, 48)        18480
_________________________________________________________________
max_pooling2d_2 (MaxPooling2 (None, 7, 7, 48)          0
_________________________________________________________________
dropout_2 (Dropout)          (None, 7, 7, 48)          0
_________________________________________________________________
flatten_1 (Flatten)          (None, 2352)              0
_________________________________________________________________
dense_1 (Dense)              (None, 256)               602368
_________________________________________________________________
dense_2 (Dense)              (None, 10)                2570
=================================================================
Total params: 623,826
Trainable params: 623,826
Non-trainable params: 0
```

Figure 3: CNN model summary

3.2 Model Compilation

I compiled the model using the RMSprop optimizer.

```
# Compile model
model.compile(optimizer='RMSprop',
              loss='categorical_crossentropy',
              metrics=['accuracy']
              )
```

3.3 Model Training

```
# Define early stopping parameters
stop = EarlyStopping(monitor='val_loss',
                     min_delta=0,
                     patience=2,
                     mode='auto'
                     )

# Train model
history = model.fit(trainX, trainY,
                    epochs=20,
                    validation_split=0.2,
                    callbacks=[stop]
                    )
```

I used a validation split of 0.2. I.e., the model will be trained on 48000 images out of 60000, and the other 12000 will be used for validation. By using separate image sets for validation and testing, we are reducing overfitting, which is when a model is very good at detecting trends in a narrow, specific set of data, but is not as good at doing this on a wider, more general set of data.

An example of this would be if someone trained a model to classify digits, and used an input data sets made up entirely of their own handwriting. While the model would become very good at recognising the specific patterns found in a single persons handwriting, it would be much worse at doing the same on other people's handwriting.

I chose quite a large number of training epochs, with 20. The reason I did this is because I also used early stopping, which causes the model to stop training once its rate of improvement slows down. This meant that my model usually only ran for 7 or 8 epochs.

3.4 Model Validation Analysis

I used the History object returned by the model training method to get some statistics associated with the training and validation of the model. As you can see, I got the accuracy and loss for both the training and validation sets, and plotted them in Figures 4 & 5 below.

```
# Get Validation & Training stats
model_history = history.history

val_acc = model_history['val_accuracy']
val_loss = model_history['val_loss']
train_loss = model_history['loss']
train_acc = model_history['accuracy']
epochs = range(len(train_acc))

# Plot Validation and Training Accuracy
pyplot.figure(figsize = (14,10))
pyplot.subplot(211)
pyplot.plot(epochs, val_acc, 'ro', label='Validation Accuracy')
pyplot.plot(epochs, train_acc, 'b', label='Training Accuracy')

pyplot.xlabel('Epochs')
pyplot.ylabel('Accuracy')
pyplot.legend()

# Plot Validation and Training loss
pyplot.figure(figsize = (14,10))
pyplot.subplot(212)
pyplot.plot(epochs, val_loss, 'ro', label = 'Validation Loss')
pyplot.plot(epochs, train_loss, 'b', label = 'Training Loss')

pyplot.xlabel('Epochs')
pyplot.ylabel('Loss')
pyplot.legend()
```

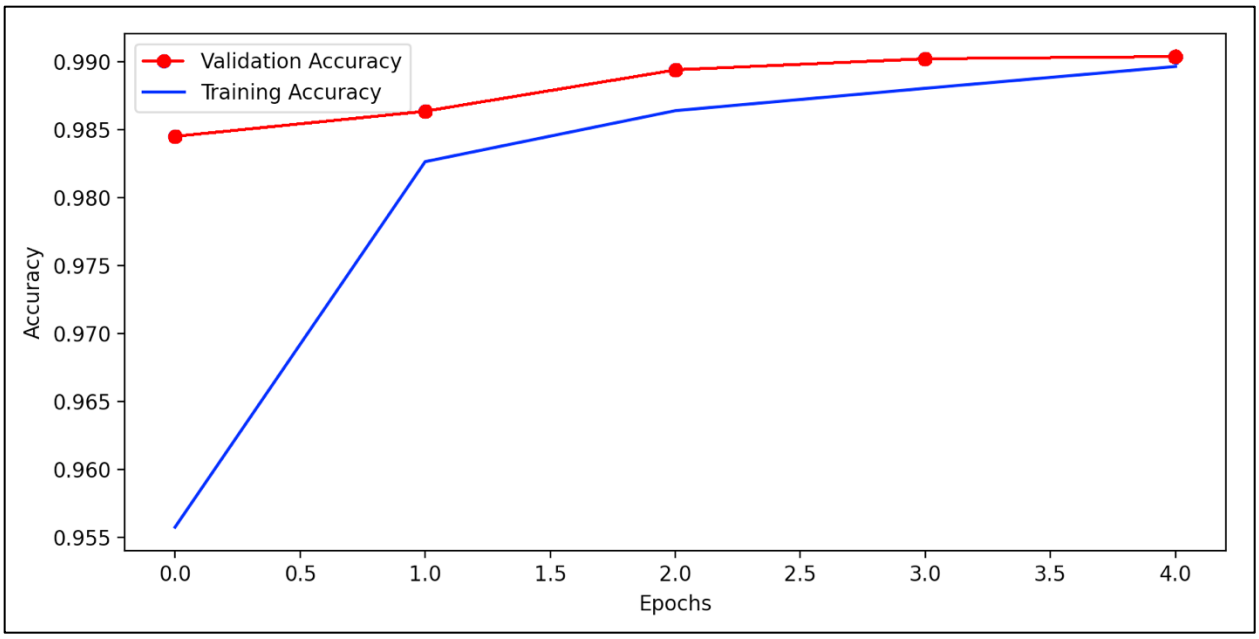

Figure 4: Validation Accuracy vs Training Accuracy

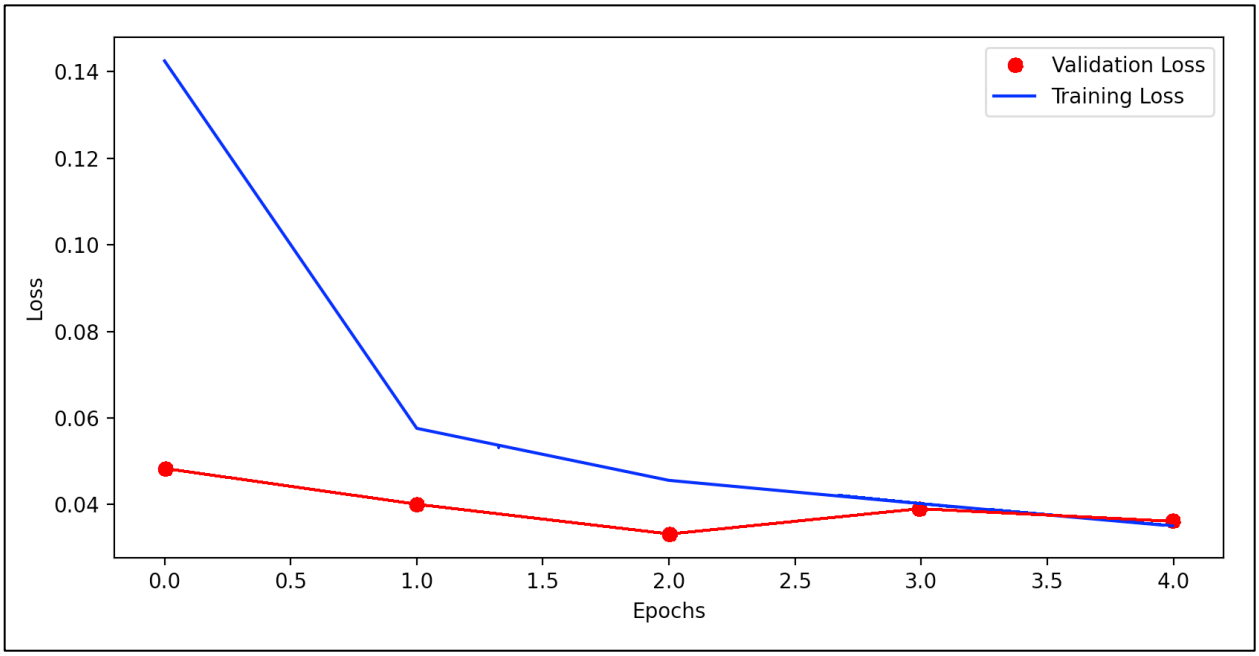

Figure 4: Validation Accuracy vs Training Loss

From the above plots, I can see that there is very little if any overfitting, as the training accuracy doesn't rise above the level of the validation accuracy. This is what I expected to see because I used separate sets for training, validation and testing, with no overlap. Also, there were no specific characteristics of the images from the training set that could be differentiated from images from another set, so overfitting is not as likely.

3.5 Model Evaluation

Lastly, I evaluated the trained model on the test data sets. I found the test loss and accuracy percentages, as well as the time taken to train and evaluate the model. I ran this model 10 times. The data from these 10 runs is seen in the table below.

```
# Evaluate the model with test data set
test_eval = model.evaluate(testX, testY, verbose=1)
print('Test loss = ', test_eval[0])
print('Test accuracy = ', test_eval[1])
print('Time taken = %s seconds' % int(time.time()-startTime))
```

Eval Iteration #	Test Accuracy %	Test Loss %	Time Taken (s)
1	99.23	3.02	1210
2	99.18	2.71	1193
3	99.19	2.72	871
4	98.99	3.45	1027
5	99.04	3.26	1081
6	99.26	2.92	1176
7	99.08	3.01	808
8	99.19	2.62	906
9	99.00	3.33	1100
10	99.18	3.14	1723
Average	99.13	3.02	1110
Std Deviation	0.0977	0.2807	257.07

4. Conclusion

This program achieves the targets set out at the start with a fair margin for error. I ran my model 10 times using the code below, and achieved an average test accuracy of 99.13%, with a standard deviation of . The average test loss was 3.02%. On my laptop, the average model runtime was 18 minutes and 30 seconds.

I realise now that using separate, unique sets for the training, validation and testing of the model reduces overfitting and thus gives the most accurate representation of the quality of the model.

I was curious as to what the images that my model was failing to classify looked like, so I added a bit to the bottom of my code to see what the failed predictions looked like. I would say that some of them would prove difficult or even impossible for a human to classify with 100% accuracy, so I don't believe that 100% testing accuracy is possible for this particular data set.

5. References

[1] Towards Data Science: Simple Introduction to Convolutional Neural Networks
https://towardsdatascience.com/simple-introduction-to-convolutional-neural-networks-cdf8d3077bac

[2] Machine Learning Mastery: How to Develop a CNN for MNIST Handwritten Digit Classification
https://machinelearningmastery.com/how-to-develop-a-convolutional-neural-network-from-scratch-for-mnist-handwritten-digit-classification/

[3] Kaggle: Introduction to CNN Keras – 0.997 (top 6%)
https://www.kaggle.com/yassineghouzam/introduction-to-cnn-keras-0-997-top-6/notebook

[4] Kaggle: MNIST Simple CNN Keras (Accuracy: 0.998)
https://www.kaggle.com/elcaiseri/mnist-simple-cnn-keras-accuracy-0-99-top-1

[6] Towards Data Science: Why data should be normalized before training a Neural Network
https://towardsdatascience.com/why-data-should-be-normalized-before-training-a-neural-network-c626b7f66c7d

6. Appendix

```
# Import libraries
from tensorflow import keras
from numpy import mean
from numpy import std
from numpy import round
import numpy as np
from matplotlib import pyplot
from keras.datasets import mnist
from keras.utils import to_categorical
from keras.models import Sequential
from keras.layers import Conv2D
from keras.layers import MaxPool2D
from keras.layers import Dense
from keras.layers import Flatten
from keras.layers import Dropout
from keras.layers import BatchNormalization
from keras.optimizers import SGD
from keras.optimizers import RMSprop
from keras.callbacks import EarlyStopping
import time

# Log start time
startTime=time.time()

# Load dataset
(trainX, trainY), (testX, testY) = mnist.load_data()

# Reshape dataset to have a single channel
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))

# One hot encode target values
trainY = to_categorical(trainY)
testY = to_categorical(testY)

# Normalize pixels to floats between 0 and 1
trainX = trainX.astype('float32')/255.0
testX = testX.astype('float32')/255.0

# Define CNN model
model = Sequential()
model.add(Conv2D(filters=24,
                 kernel_size=4,
                 padding='same',
                 activation='relu',
```

```
                 input_shape=(28, 28, 1)
                 ))
model.add(MaxPool2D(2,2))
model.add(Dropout(0.2))

model.add(Conv2D(filters=48,
                 kernel_size=4,
                 padding='same',
                 activation='relu',
                 ))
model.add(MaxPool2D(2,2))

model.add(Dropout(0.2))
model.add(Flatten())
model.add(Dense(256, activation='relu'))
model.add(Dense(10, activation='softmax'))

# Summarize model
model.summary()

# Compile model
model.compile(optimizer='RMSprop',
              loss='categorical_crossentropy',
              metrics=['accuracy']
              )

# Define early stopping parameters
stop = EarlyStopping(monitor='val_loss',
                     min_delta=0,
                     patience=2,
                     mode='auto'
                     )

# Train model
history = model.fit(trainX, trainY,
                    epochs=20,       ## Change to 20
                    validation_split=0.2,
                    callbacks=[stop]
                    )

# Evaluate the model on test set
test_eval = model.evaluate(testX, testY, verbose=1)
print('Test loss = ', test_eval[0])
print('Test accuracy = ', test_eval[1])
print('Time taken = %s seconds' % int(time.time()-startTime))

# Get Validation & Test stats
model_history = history.history
```

```
val_acc = model_history['val_accuracy']
val_loss = model_history['val_loss']
train_loss = model_history['loss']
train_acc = model_history['accuracy']
epochs = range(len(train_acc))

# Plot Validation and Training Accuracy
pyplot.figure(figsize = (14,10))
pyplot.subplot(211)
pyplot.plot(epochs, val_acc, 'bo', label='Validation Accuracy')
pyplot.plot(epochs, train_acc, 'b', label='Training Accuracy')

pyplot.xlabel('Epochs')
pyplot.ylabel('Accuracy')
pyplot.legend()

# Plot Validation and Training loss
pyplot.figure(figsize = (14,10))
pyplot.subplot(212)
pyplot.plot(epochs, val_loss, 'bo', label = 'Validation Loss')
pyplot.plot(epochs, train_loss, 'b', label = 'Training Loss')

pyplot.xlabel('Epochs')
pyplot.ylabel('Loss')
pyplot.legend()

pyplot.show()
```