Project Report for

CE4041 – Artificial Intelligence

Classifier implementation for the MNIST handwritten digits database

Project Team

Student Name	**Student Id**
Qinyuan Liu	20137095
Eamon Moloney	8457077
Ibrahim Saana Aminu	25381993
Des Powell	9513833
Terence Coffey	15223124

Contents

Introduction

This project creates a Classifier for the Modified National Institute of Standards and Technology (MNIST[1]) handwritten digits database, using the Keras Neural Network library[2] (Tensorflow-hosted version) and programmed in Python. The classifier achieves a testing accuracy of at least 99 percent. The MNIST[1] dataset contains 60,000 28 x 28 grey scale images of handwritten digits and their associated labels for training and 10,000 images for testing. The classifier is a 10-way classifier with the 10 possible digits (0-9) and 10 possible digit classes.

Approach

In-person meetings were used as the initial basis for agreeing on the project requirements and the approach to adopt in the code implementations. The group initially used Google Colab to enable effective collaboration with input at various levels by the team members. However, as the project developed the python code was run on one group member's computer as the main version, with each member also running the codes individually to scan for error and necessary adjustments where necessary. The final project report was created, shared and edited by all team members via OneDrive. The content of the submission was agreed upon by all members after a final review.

Investigate Solution Space

After taking guidance from our instructor Dr. Colin Flanagan and reviewing the Artificial Intelligence module's course material[3], the team started to investigate options to develop the required program and associated report. We realised that there were potentially many ways of achieving a test accuracy performance level of 99 percent. The Wikipedia[1] entry for MNIST digits database provides a table which showed the error rate for different types of Neural Networks. The range of possible solutions included Convoluted Neural Networks (CNNs), K-Nearest Neighbours, Linear and non-linear classifiers. However, we decided to limit our choices to CNNs as this was most closely related to our course and lecture material.

Initial testing using Google Colab for one test case for each model indicated that it would take approximately 15 minutes per model to run one test with a very limited set of test parameters. The same test, if performed on a laptop took about 6 minutes. Therefore, we decided to focus python development on one team member's laptop rather than on Google Colab (more on this later)

Training, Validation and Test data

The MNIST dataset, available within Keras provided us with the necessary training and test data. The mnist.load() function loads the training and test data along with their associated labels. The training data is further subdivided to provide a validation portion. When the fit() function is called to train the model, a validation split of 0.2 or 20% of the training data is used exclusively for validation. The training data is also shuffled during the process as the fit() function has been request to perform this functionality. The shuffle type used in this case is shuffle-shuffle rather than batch-shuffling as this approach helps reduce the risk of over-fitting. See Fig. 1 which shows the relationship between training, validation and test data and how the test data is only used after training and validation phases have been completed.

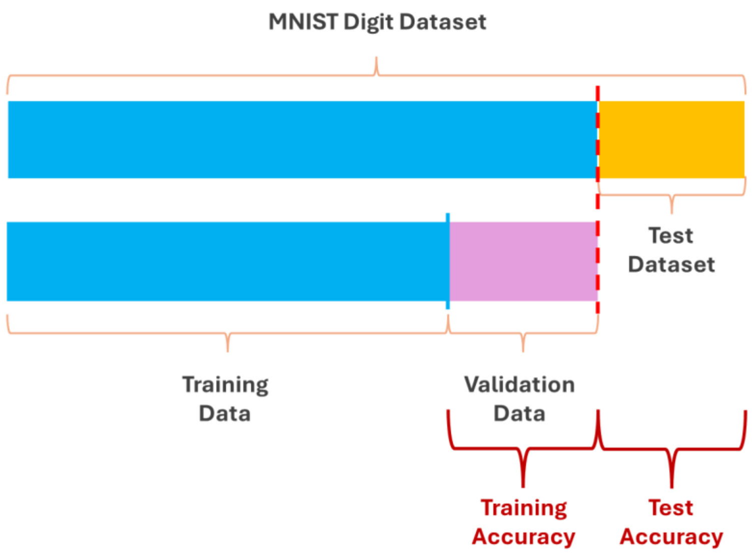


Fig. 1) Shows the relationship between training, validation and test data.

Choosing the models for Initial Evaluation

The models we ultimately selected for evaluation was the Sequential model. This model type uses a Linear sequential stack of layers, where the output of one layer becomes the input for the next layer. The initial stack order we implemented was motivated by our lecture notes before further refinement from experimentation.

Our first test results using one Convolve 2D layer typically produced an accuracy test result at or below 98 percent. We iterated through various models until we consistently produced accuracy results above 99 percent and these were then considered for the next stage.

Subsequently, models with 2 Convolve 2D layers continued to exceed accuracy test results above 99% (in the range 0.9905 to 0.992). Therefore, these were considered good candidates[4] for further testing and development.

These evaluations were run 3 times to ensure that they reliably produced results above 99 percent.

The 3 models are compared schematically in Fig. 2) and Table 1), see below.

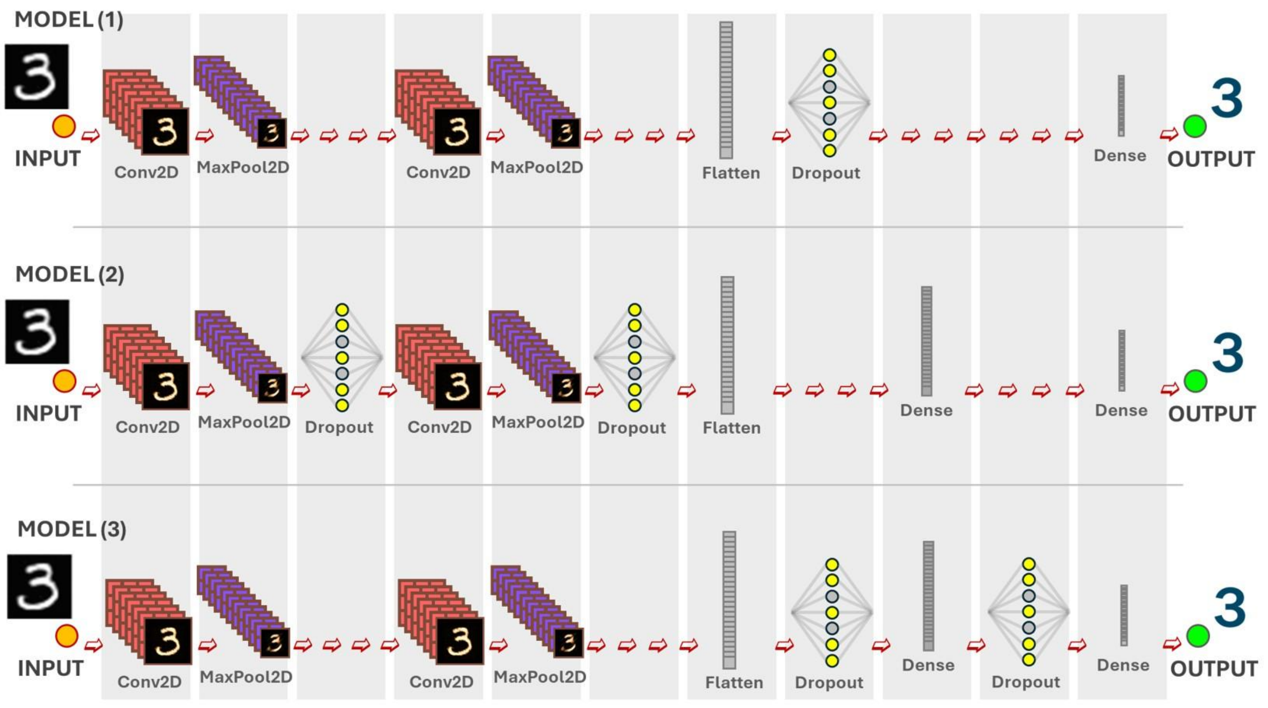


Fig. 2) Shows the corresponding layers used in the 3 candidate sequential models[5]

Model #	Test Accuracy
Model (1)	0.99140000 (99.14%)
Model (2)	0.99070000 (99.07%)
Model (3)	0.99210000 (99.21%)

Table 1) Shows the corresponding accuracy calculated for each model

Convolute Neural Network

Code Development Overview

A high-level description of the delivered Python code solution is described in this section:

1. The application imports the tensorflow.Keras library and subsequently loads in the training and test data from the Keras version of the MNIST dataset.
2. Set the hypertuning parameters to the values that were identified in the best performing CNN model.
3. Create the CNN sequential model. This adds multiple specialised layers to a sequential model or defines a model with the layers defined as part of the sequential model[6]. The later approach is what we used. See Layer review for more details.
4. Compile and train the model. The loss method selected is **categorical_crossentropy**[7] . A validation parameter tells the fit() method to reserve 20 percent of the test data for validation. The shuffle parameter is set to true so it will also be shuffled, so that the order of the training data is changed after each epoch. This measure can help with avoiding over-fitting.

5. Predict and Evaluate the test model using the test data.
6. Gather training and validation accuracy metrics, this was achieved via test logs, tracking training and validation accuracy data, as well as additional support data.
7. After each run the model was re-set and the program re-run (in this case three times).
8. A summary of these metrics such as test accuracy, training accuracy and loss precision where plotted to visually demonstrate the success of each classifier model.

CNN Layer review

In this section, we provide a description of what happens in each layer of the sequential model:

- **Input layer:** The Keras library function, which provides this functionality is introduced. The relevant Keras function (shown in bold face) is shown as the first part of each description.

```
layers.Input (shape=(28, 28, 1)
```

  This is the input layer. It defines a grey scale (1) input object ,28 by 28 pixels in size representing our input image. The image is a two dimensional representation of the image. The input layer takes the whole image as input.

- **Convolution 2d layer:** The Conv2D layer(s) in the model can subsequently work on different parts/features of the image.

```
layers.Conv2D(64, kernel_size=(3, 3), activation='relu'),
```

  defines a Convolution layer, which is important for processing images in a **C**onvolutional **N**eural **N**etwork (**CNN**). Each filter learns to detect different patterns in the input image. Additional filters can identify more complex features but computational costs also increase. The kernel size means that the filter will examine a 3x3 pixel region of the image at a time. The activation function will apply **R**ectified **L**inear **U**nit (**ReLU**) function after the convolution operation. This introduces non-linearity into the model. It replaces negative values with zero and helps the model to learn more complex patterns. Convolution layers are important as they are excellent at automatically learning and extracting relevant features from images. They maintain the spatial relationship thereby making them suitable for image recognition tasks. If you stack multiple convolution layers, CNNs can learn increasingly complex features. This can vary from edges and texture in early layers to more abstract object parts in later layers.

- **MaxPooling layer:** similarly, the MaxPooling[8] layer(s) can be repeated later in the model, as required:

```
layers.MaxPooling2D(pool_size=(2, 2))
```

  This MaxPooling[8] layer is used for down-sampling feature maps, reducing their size, making the model more efficient and robust. It's a key component in many successful CNN architectures. The feature map is a grid of numbers. Max pooling uses a small window called a sliding window. This sliding window slides over the feature map without overlapping. For each window position, it selects the maximum value within that window and discards the rest. This creates a new, smaller grid where each value represents the maximum value from the corresponding region in the feature map. This reduces the size of the feature map making the model more efficient and robust while preserving important features.

- **Dropout layer:** as per other layers, the Dropout layer(s) can be repeated subsequently in the model, as required:

```
layers.Dropout(0.25)
```

  A **Dropout layer** is a regularisation technique used to prevent overfitting and improves generalisation. The dropout rate controls the probability of a neuron being dropped and is passed as an argument to the Dropout layer. This layer works by randomly deactivating neurons during training. This forces the network to learn more robust features. It does this by selecting a subset of neurons in the training step and setting their output to 0. The model then performs forward and backward passes with the active neutrons. The weights of the networks are updated based on the error calculated with the dropped-out neurons. This process is then repeated for each training step, with a different set of neurons being dropped out each time.

- **Flatten layer:** as per other layers, the Flatten layer(s) can be used subsequently in the model, as required, however in the case of these three test models, it was used as a single layer:

```
layers.Flatten()
```

  Flattening is necessary to bridge the gap between convolutional/pooling layers and dense layers (Fully connected layers) in CNNs. This layer converts multidimensional output from convolution and pooling layers into a 1D vector that can be used by dense layers for classification. e.g If the output from the previous layer is a 3D tensor with shape (10, 10 ,32) (e.g. 32 features maps of size 10 x 10), the flattening layer will transform it into a 1D array of size 3200(10 x 10 x 32) Dense layers expect their input to be a 1D vector. By flattening the output of previous layers, it combines all the learned features into a single vector.

- **Dense layer:** as per other layers, the Dense layer(s) can be repeated in the model, as required:

```
layers.Dense(10, activation='softmax')
```

  This defines the output layer in the model. Since it is predicting the digits 0 to 9, there will be 10 outputs. Dense means that each unit in this layer is connected to every unit in the previous layer. **Softmax** calculates the probability that the input digit image belongs to each of the ten output digits. This layer can be thought of as a decision-making part of the model as it classify the input image into one of 10 digit categories based on the probabilities. The highest probability corresponds to the model's predicted digit for the input image.

Early Stopping

This is useful to stop the training regime when some parameter (typically validation loss) starts to increase, while training loss is continuing to decrease. This is usually a sign of overfitting on the model .

```
EarlyStopping(monitor='val_loss' , #Validation lss monitoring

     min-delta=0, ' Min allowed changes to be ignored

    patience=2, # Wait for epochs after to stop

    mode ='auto') #Halt if val_loss changes direction.
```

These details were passed to the fit function. More details about this later in the report.

Selection and Tuning Keras function parameters

The next phase in the exploratory phase was to tune the models to identify additional performance improvement by adjusting parameters in the main Keras function calls. One of our goals was to try to execute all the test models and test case containing all combinations of the relevant Keras function parameters within 8 hours. This would allow an hour or two in the case of test runs exceeding 8 hours. We also limited our model selection to 3 models due to this constraint. We ran over 48 tests per model.

Due to time constraints due to long duration test runs, it was decided to run all the tests on one team member's laptop as the tests would be faster to complete. Moreover the risk of exceeding the amount of free computational resources using Google Colab for a particular individual would be minimised. The laptop used was a HP zbook and had the following specification:

<u>HP zbook Specification</u>:

- 12th Gen Intel Core i7-1265U × 12 Processors,
- Graphics NVIDIA Corporation / Mesa Intel® Graphics (ADL GT2)*

Software used:

- Linux Ubuntu 20.04 Operating System,
- VS Code 1.95.0, Python 3.12.4,
- TensorFlow 2.18.0,
- Keras 3.6.0,
- Numpy 1.26.4,
- Cuda Toolkit (NVIDIA),
- cuDNN (NVIDIA GPU-acceleration library)

Parameter Selection

The parameters for Convolve 2D and the dropout layers were identified and a range of values for these were selected. Our intension was to determine if we could further improve the test accuracy by a particular combination of parameter settings. We had to limit our choice of possible function parameters and the range of values to reduce the feasible execution time of the program. e.g. We only tested with values of 0.2 and 0.5 for the Dropout layer.

The relevant Keras functions and their parameters deemed to have contributed most to an improvement in performance were identified as shown in Table 2).

Training–Validation/Test Data	**Parameter**	**Range or values**	**Description**
Training and Test data are available from MNIST using the **mnist.load_data()** function.	Training /Validation Split	0.8/0.2	The training and test data are provided in MNIST. The validaion_split parameter passed into the fit() function, determines the amount of the training set which will be reserved for validation. In the fit() function, the shuffle parameter tells the training function whether or not to perform shuffling on the training data.

Table 2). Showing Keras function calls and 'tuned' parameters

Keras Function Name	**Variables for model**		
Conv2D	filter_size	15, 32, 64	[2] It is used in feature extraction. It slides across the input image. It is the dimension of the output space.
	kernel_size	3,5	Specifies the size of the convolution window
	padding	**'same'**	Padding is applied evenly to the left/right or up/down of the input.
	kernel_initializer	**HeNormal**	Initialiser for the convolution kernel
	Activation	'relu'	Activation function
	kernel_regularizer	**Regularizers.l2()**	Also in Dense layer. Regularization is a technique to help prevent overfitting.
Dense	filter	15,32,64	
	activation	'softmax'	
	kernel_regularizer	**Regularizers.l2()**	Also in Convolve2D Layer. It helps to prevent overfitting.
MaxPooling2D	pool_size		Factors by which to downscale dimension 1 and 2
Model.fit()	Batch size	32, 64	The batch_size specifies the number of training samples to be processed before the model's internal parameters are updated
Dropout	dropout_rate	0.2, 0.5	Fraction of the input units to drop.
Model.compile	loss	“categorical_crossentropy”	This is a loss function. See description below for reason to use it.
	optimizer	“adam” learning_rate	This is a stochastic gradient decent method based on adaptive estimation of first and second-order moments. The learning rate defaults to 0.001 0.0001, 0.01

Table 3) Showing Keras Function Names and Variables for the models

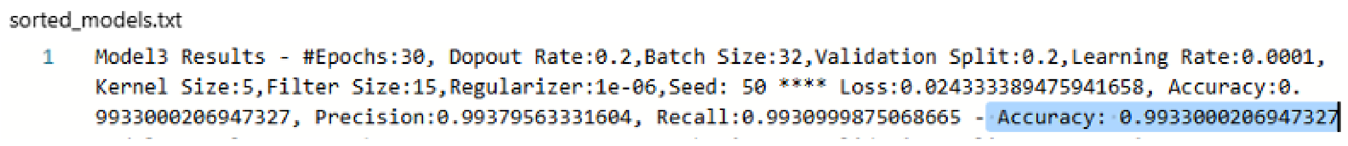
Items in Table 3), above, are highlighted in **Blue**, are additional parameters/values which have been added to the 3 sequential models.

Test Application Overview

The Python test application was designed around the 3 identified sequential test models. The initial hard coded parameters settings were replaced with a list variable and code was added so one of the 3 models could be selected from the **model_id_list** for each run. This code was then within a larger **for** loop which worked its way through the various combination of the model parameters, one model parameter at a time.
A final loop, controlled by **NUM_RUNS** determined the number of times you wanted these overall tests to run to ensure the reliability of the test accuracy result. Validate and test accuracy were written to standard output. A history object return by **model.fit()** stores the results, which can be used to plot graphs and extract information by model and epoch. When we were sure that all combinations could be tested via this test program, we were then ready to run the tests and get the settings for the parameters for the models which generated the best test accuracy.

Finding the best model and parameter settings

After the testing application had finished running, the test results were sorted into descending order by test accuracy using a shell script, see Fig. 3 below. The parameters for the best performing model were noted for the next step (Please note: the top 16 results are shown in Appendix A).



```
sorted_models.txt
  1    Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
       Kernel Size:5,Filter Size:15,Regularizer:1e-06,Seed: 50 **** Loss:0.024333389475941658, Accuracy:0.
       9933000206947327, Precision:0.99379563331604, Recall:0.9930999875068665 - Accuracy: 0.9933000206947327
```

Fig. 3) The top performing model with parameter setting and accuracy

Results

The results (shown in Appendix A) are summarised here. **Model 3** is the best performing model with a test accuracy of **0.99330002**. This occurred on **Epoch 30**. The details are as follows:

Dropout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,Kernel Size:5,Filter Size:15,Regularizer:1e-06,Seed: 50 * Loss:0.024333389475941658, **Accuracy:0.9933000206947327**, Precision:0.99379563331604, Recall:0.9930999875068665 - Accuracy: 0.9933000206947327

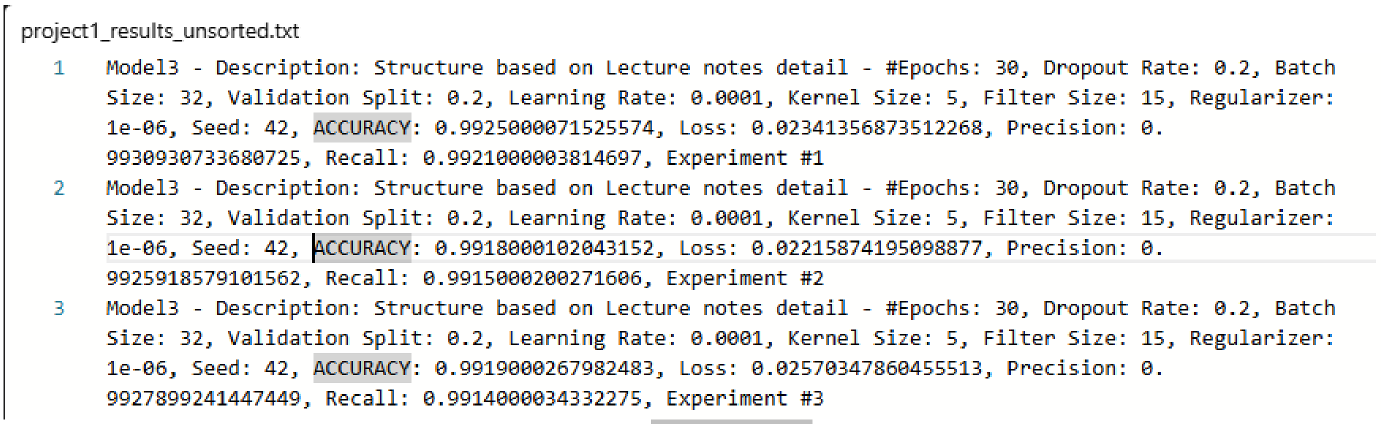
The following 14 positions was also held by model 3 with model 2 showing up at position 15 and model 1 showing up at position 16.

Model 2 & Model 1 had a test accuracy of 0.99169999 and 0.991599977, respectively – see Fig. 4 & 5.

However, we still had to test model 3 to see if its results were repeatable/reliable, see Fig. 4 & 5. **project1.py** was updated with the parameters corresponding to the best performing experiment The number of runs in this script was set to 3 and the test was executed.

The test accuracy results this time were not as good as the earlier reported one, however it was still above the 99% test accuracy threshold, so it satisfied this project requirement.

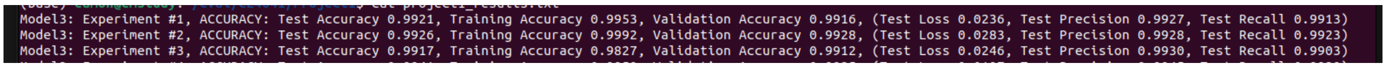
Final Results:



project1_results_unsorted.txt

```
Model3 - Description: Structure based on Lecture notes detail - #Epochs: 30, Dropout Rate: 0.2, Batch
Size: 32, Validation Split: 0.2, Learning Rate: 0.0001, Kernel Size: 5, Filter Size: 15, Regularizer:
1e-06, Seed: 42, ACCURACY: 0.9925000071525574, Loss: 0.02341356873512268, Precision: 0.
9930930733680725, Recall: 0.9921000003814697, Experiment #1
Model3 - Description: Structure based on Lecture notes detail - #Epochs: 30, Dropout Rate: 0.2, Batch
Size: 32, Validation Split: 0.2, Learning Rate: 0.0001, Kernel Size: 5, Filter Size: 15, Regularizer:
1e-06, Seed: 42, ACCURACY: 0.9918000102043152, Loss: 0.02215874195098877, Precision: 0.
9925918579101562, Recall: 0.9915000200271606, Experiment #2
Model3 - Description: Structure based on Lecture notes detail - #Epochs: 30, Dropout Rate: 0.2, Batch
Size: 32, Validation Split: 0.2, Learning Rate: 0.0001, Kernel Size: 5, Filter Size: 15, Regularizer:
1e-06, Seed: 42, ACCURACY: 0.9919000267982483, Loss: 0.02570347860455513, Precision: 0.
9927899241447449, Recall: 0.9914000034332275, Experiment #3
```

Fig. 4) Shows test accuracy (ACCURACY) and training accuracy



```
Model3: Experiment #1, ACCURACY: Test Accuracy 0.9921, Training Accuracy 0.9953, Validation Accuracy 0.9916, (Test Loss 0.0236, Test Precision 0.9927, Test Recall 0.9913)
Model3: Experiment #2, ACCURACY: Test Accuracy 0.9926, Training Accuracy 0.9992, Validation Accuracy 0.9928, (Test Loss 0.0283, Test Precision 0.9928, Test Recall 0.9923)
Model3: Experiment #3, ACCURACY: Test Accuracy 0.9917, Training Accuracy 0.9827, Validation Accuracy 0.9912, (Test Loss 0.0246, Test Precision 0.9930, Test Recall 0.9903)
```

Fig. 5) Shows test accuracy (ACCURACY) and training accuracy

The overall performance of Model 3 was very good and reliable as it has produced very similar results with all 3 runs. However, when you look at the diagram in Fig. 6, it can be seen that slight overfitting has occurred.

Accuracy – Overfitting Discussion

Overfitting happens when a model learns the training data and its peculiarities too well. It achieves near perfect accuracy on the training data but performs less favourably on the test data. A model that overfits fails to generalise to new data.

When we generated the graphs (see Fig. 6) for the training and validation accuracy for each of the sequential models are viewed over 3 runs, model 3 shows some over fitting. From the graph the over-fitting appears to happen at around epoch 16. This is the epoch where the training accuracy continues to increase but the validation accuracy appears stationary. An alternative way to see overfitting is to plot the loss against each epoch.

On model 1 and 2 the training and validation graph align on epoch 30 and demonstrate this overfitting effect.

See Fig. 6) below.

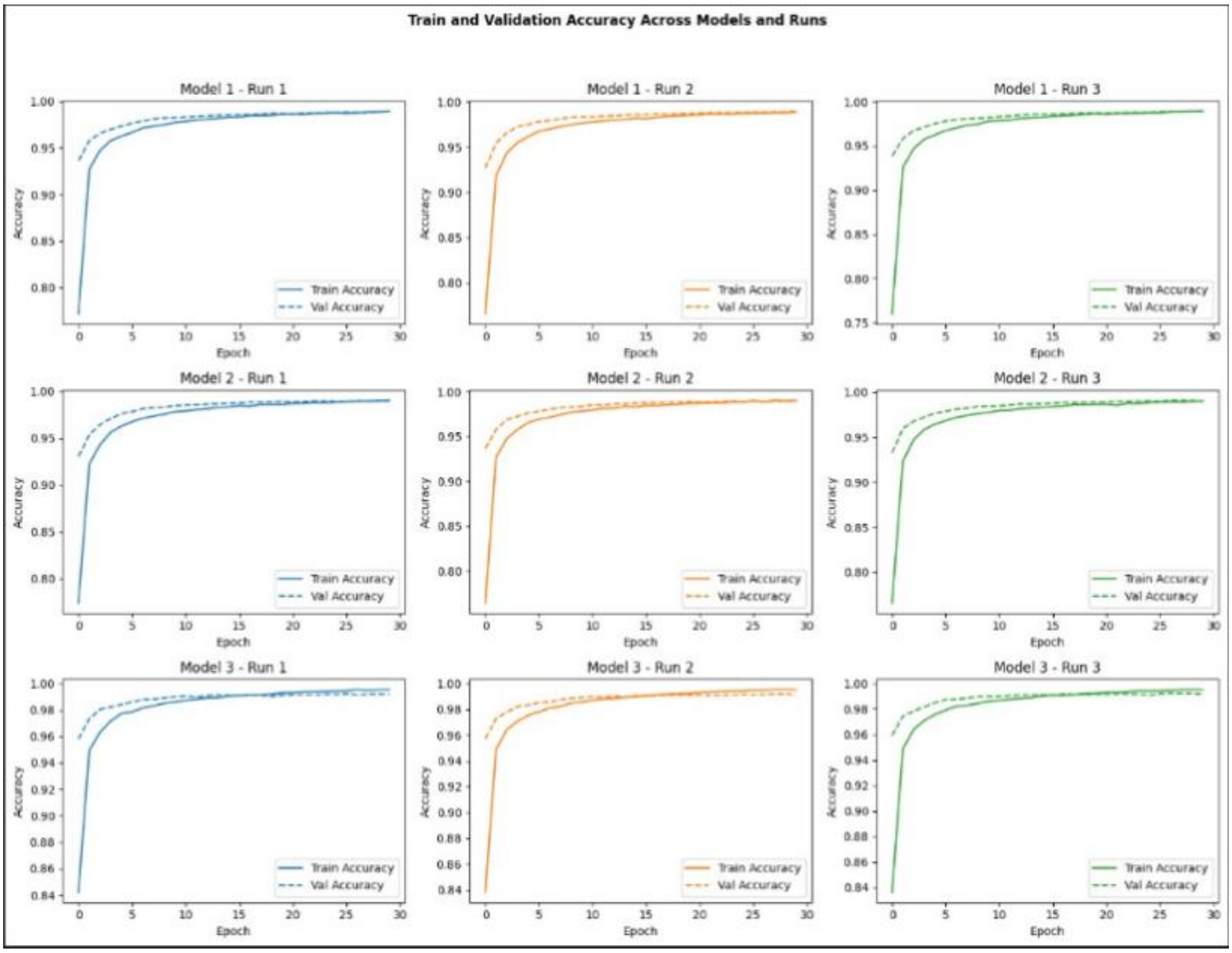
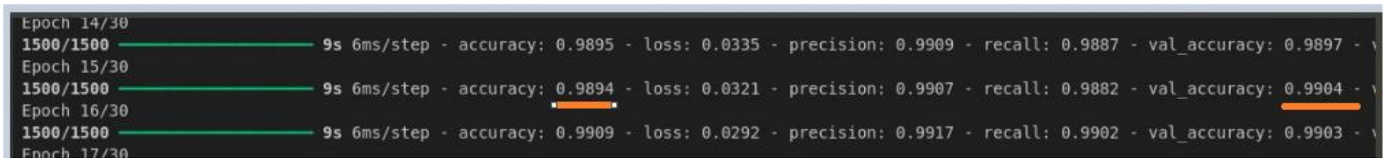


Fig. 6) Shows for each of the 3 models, the Training and Validation accuracy scores for 3 runs



```
Epoch 14/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9895 - loss: 0.0335 - precision: 0.9909 - recall: 0.9887 - val_accuracy: 0.9897 - v
Epoch 15/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9894 - loss: 0.0321 - precision: 0.9907 - recall: 0.9882 - val_accuracy: 0.9904 - v
Epoch 16/30
1500/1500 ━━━━━━━━━━━━━━━━━━━━ 9s 6ms/step - accuracy: 0.9909 - loss: 0.0292 - precision: 0.9917 - recall: 0.9902 - val_accuracy: 0.9903 - v
Epoch 17/30
```

Fig. 7) Shows at Epoch 16 the validation and testing accuracy scores

Deliverables

The items delivered for this assignment includes this pdf report and a python script called **project1.py**. This python script can be run directly from the Python command line.

Lessons Learned (And Future Projects)

Given the time constraints we knew it was important to meet early to discuss and agree the scope of the project work. This worked out well as it gave us a good plan to refer to for the weeks that followed. We were fortunate to have a good cross-section of skills across the team. This included having good organisational, coding, documentation, research and soft skills. We also set up a one drive sharepoint folder, which allowed us to work productive together and share documents.

When changes were required for improving the project, team members contributed their input , updating the files using track changes and documenting all updates. Teams meetings were mainly held face-to-face either with the whole group during the initial stages of the project and later on in smaller group where individuals needed to work together to complete a work item (documenting the report or updating part of the python code). We also used Whatsapp conference calls and texts to coordinate and occasionally meet remotely, when it was not possible to all meet face to face. Overall, the project team work well together and used modern communication technology to facilitate the delivery of this project.

One enhancement we would recommend making to this project is to attempt to get the EarlyStopping functionality to behave as expected. In this implementation, we found that it stopped after the third epoch. However, when EarlyStopping was removed the model achieved over 99% test accuracy.

Conclusion

The Sequential CNN model implemented with 2*Conv2D and 2*Dropout layers achieved a test accuracy in excess of 99%. However, care needs to be taken to reduce the risk of overfitting when optimizing for test accuracy, as excessive fine-tuning may lead to poor generalization on new, unseen data.

References

[1] https://en.m.wikipedia.org/wiki/MNIST_database

[2] https://keras.io/api/datasets/mnist/

[3] https://learn.ul.ie/d2l/le/lessons/45554/topics/680113, PDF: 051-keras-intro-24.4u.pdf (Slides 30 and 32)

[4] https://learn.ul.ie/d2l/le/lessons/45554/topics/680113, PDF: 051-keras-intro-24.4u.pdf (Slides 32) # Slide 32, recommending the use of 2 Conv2D/MaxPooling2D layer series for improved results

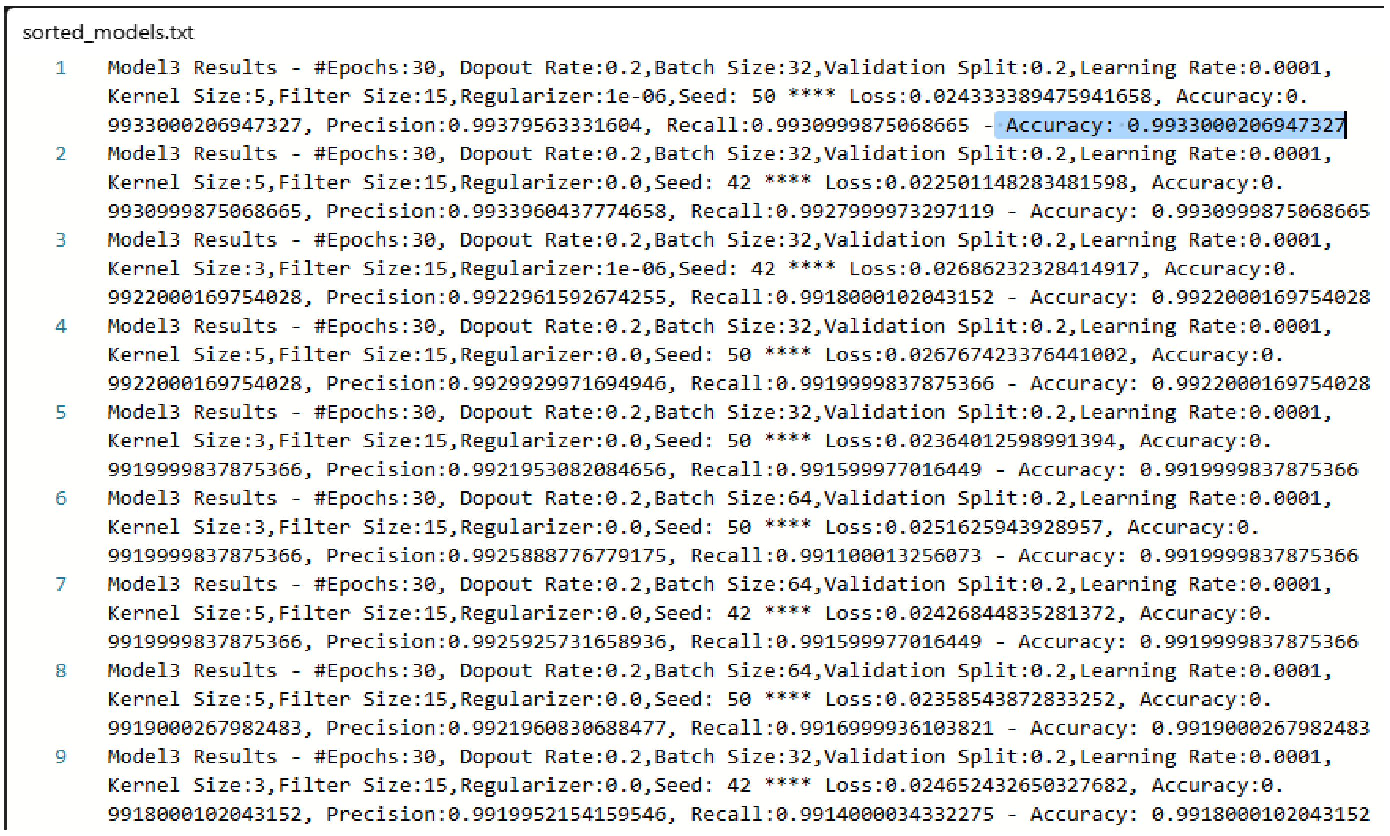
[5] https://learn.ul.ie/d2l/le/lessons/45554/topics/664611, 050-convnet.4up.pdf (Slides 2) # Slide 2, Inspiration from Slide 2

[6] https://learn.ul.ie/d2l/le/lessons/45554/topics/680113, PDF: 051-keras-intro-24.4u.pdf (Slides 11) Similar to how it is shown on slide 11, 051-keras-intro-24.4up.pdf

[7] https://learn.ul.ie/d2l/le/lessons/45554/topics/680113, PDF: 051-keras-intro-24.4u.pdf (Slides 24) As described on slide 24, 05-keras-intro-24.4up.pdf

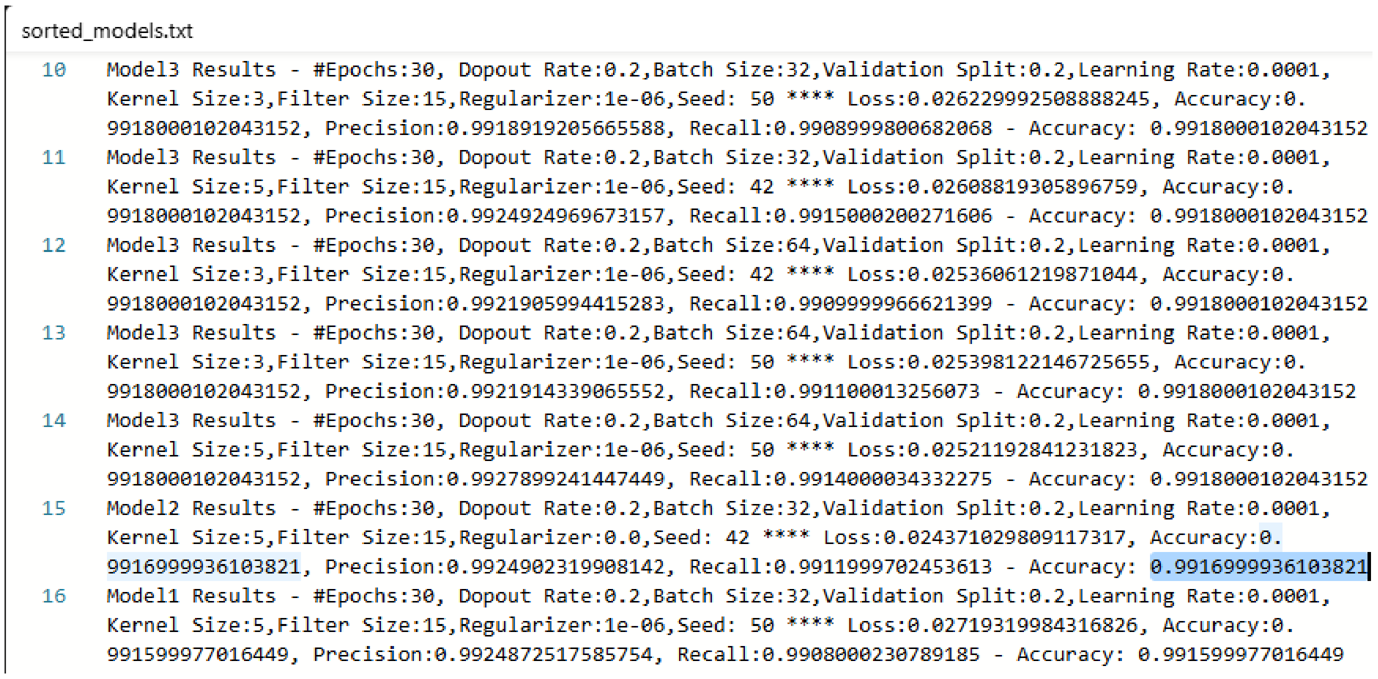
[8] https://learn.ul.ie/d2l/le/lessons/45554/topics/680113, PDF: 051-keras-intro-24.4u.pdf (Slides 29)

Appendix A



sorted_models.txt

```
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:1e-06,Seed: 50 **** Loss:0.024333389475941658, Accuracy:0.
9933000206947327, Precision:0.99379563331604, Recall:0.9930999875068665 - Accuracy: 0.9933000206947327
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:0.0,Seed: 42 **** Loss:0.02250114828348l598, Accuracy:0.
9930999875068665, Precision:0.9933960437774658, Recall:0.9927999973297119 - Accuracy: 0.9930999875068665
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:3,Filter Size:15,Regularizer:1e-06,Seed: 42 **** Loss:0.02686232328414917, Accuracy:0.
9922000169754028, Precision:0.9922961592674255, Recall:0.9918000102043152 - Accuracy: 0.9922000169754028
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:0.0,Seed: 50 **** Loss:0.026767423376441002, Accuracy:0.
9922000169754028, Precision:0.9929929971694946, Recall:0.9919999837875366 - Accuracy: 0.9922000169754028
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:3,Filter Size:15,Regularizer:0.0,Seed: 50 **** Loss:0.02364012598991394, Accuracy:0.
9919999837875366, Precision:0.9921953082084656, Recall:0.991599977016449 - Accuracy: 0.9919999837875366
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:64,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:3,Filter Size:15,Regularizer:0.0,Seed: 50 **** Loss:0.0251625943928957, Accuracy:0.
9919999837875366, Precision:0.9925888776779175, Recall:0.991100013256073 - Accuracy: 0.9919999837875366
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:64,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:0.0,Seed: 42 **** Loss:0.02426844835281372, Accuracy:0.
9919999837875366, Precision:0.9925925731658936, Recall:0.991599977016449 - Accuracy: 0.9919999837875366
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:64,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:0.0,Seed: 50 **** Loss:0.02358543872833252, Accuracy:0.
9919000267982483, Precision:0.9921960830688477, Recall:0.9916999936103821 - Accuracy: 0.9919000267982483
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:3,Filter Size:15,Regularizer:0.0,Seed: 42 **** Loss:0.024652432650327682, Accuracy:0.
9918000102043152, Precision:0.9919952154159546, Recall:0.9914000034332275 - Accuracy: 0.9918000102043152
```



sorted_models.txt

```
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:3,Filter Size:15,Regularizer:1e-06,Seed: 50 **** Loss:0.026229992508888245, Accuracy:0.
9918000102043152, Precision:0.9918919205665588, Recall:0.9908999800682068 - Accuracy: 0.9918000102043152
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:1e-06,Seed: 42 **** Loss:0.02608819305896759, Accuracy:0.
9918000102043152, Precision:0.9924924969673157, Recall:0.9915000200271606 - Accuracy: 0.9918000102043152
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:64,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:3,Filter Size:15,Regularizer:1e-06,Seed: 42 **** Loss:0.02536061219871044, Accuracy:0.
9918000102043152, Precision:0.9921905994415283, Recall:0.9909999966621399 - Accuracy: 0.9918000102043152
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:64,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:3,Filter Size:15,Regularizer:1e-06,Seed: 50 **** Loss:0.025398122146725655, Accuracy:0.
9918000102043152, Precision:0.9921914339065552, Recall:0.991100013256073 - Accuracy: 0.9918000102043152
Model3 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:64,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:1e-06,Seed: 50 **** Loss:0.02521192841231823, Accuracy:0.
9918000102043152, Precision:0.9927899241447449, Recall:0.9914000034332275 - Accuracy: 0.9918000102043152
Model2 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:0.0,Seed: 42 **** Loss:0.024371029809117317, Accuracy:0.
9916999936103821, Precision:0.9924902319908142, Recall:0.9911999702453613 - Accuracy: 0.9916999936103821
Model1 Results - #Epochs:30, Dopout Rate:0.2,Batch Size:32,Validation Split:0.2,Learning Rate:0.0001,
Kernel Size:5,Filter Size:15,Regularizer:1e-06,Seed: 50 **** Loss:0.02719319984316826, Accuracy:0.
991599977016449, Precision:0.9924872517585754, Recall:0.9908000230789185 - Accuracy: 0.991599977016449
```