# 662086 – Machine Learning

## Table of Contents

# Data Exploration & Visualisation

The triple MNIST dataset consists of 100,000 84x84 grayscale images, containing three 28x28 handwritten digits. For each unique combination of digits, there is a labelled folder containing 100 images, where each image shows a three-digit number.

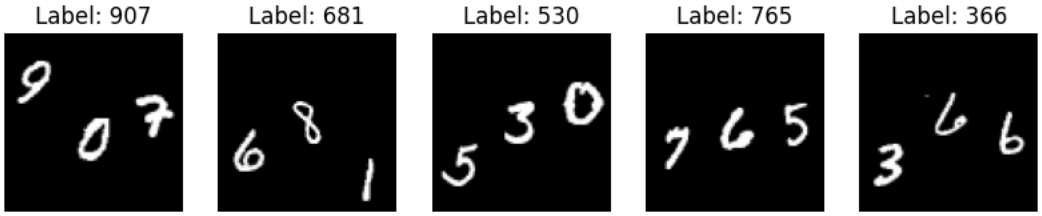



*Figure 1 - Random samples from triple MNIST dataset*

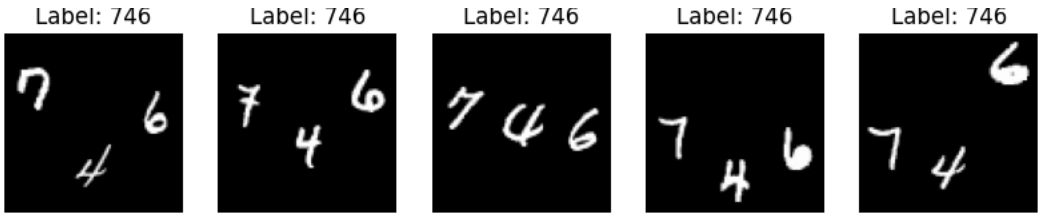



*Figure 2 - Variations of samples with same label*

The dataset will be used for machine learning, where a model will be tasked with classifying a label when given an image, making it a supervised learning task (Cireşan et al., 2012).

There are 1000 unique labels in triple MNIST, which makes classification much more complex, compared to the 10 unique labels in the single MNIST dataset. The test-train-validation splits provided in this task contain an even distribution of samples, with 100 samples allocated to each label. The dataset does not suffer from class imbalance, however 100 samples per label may not be enough to effectively train a machine learning model across 1000 potential outputs.

Training, test and validation data is provided pre-split, with each split having 64,000, 20,000 and 16,000 samples respectively. Each split contains a unique set of labels, which are not present in the other splits and with this being a classification problem, simpler models may struggle to identify labels missing from the training data.

Each 84x84 image must be converted to a format that can be more easily processed by a machine learning model. This can be done by converting the images into arrays, making the data suitable for computational analysis. The samples being in grayscale mean that the amount of computational power required to process each image is much lower, compared to that of a colour image.

Colour images are represented in three-dimensional arrays, where each pixel contains three values, corresponding to the red, green, and blue (RGB) colour channels. These channels are stored as separate 2D arrays, where each value represents the brightness of the respective colour ranging from 0 (no intensity) to 255 (maximum intensity). These three layers are then combined to create full-colour image (Singh, 2023).

A grayscale image only contains a single layer of colour data, which means that instead of storing 21168 (84*84*3) values per sample, it has 7056 (84*84*1). These values can then be normalised to 0 or 1 (black or white), which further reduces the complexity of the sample. Additionally, the image can be downscaled; lowering image quality by a reducing the number of pixels in an image.

# Baseline model implementation

## Pre-processing

The image data was pre-processed by iterating through the samples in each label and using the “Image.open()” function from the Python Imaging Library (PIL) to call the sample, which could then be converted to a numpy array. The arrays would then be divided by 255.0 in order to normalise them, which can then be flattened into a single vector using the “.reshape” function when necessary

The images were also down sampled to 45x45 to improve computational efficiency.

The processed images and labels could then be used to train a logistic regression model from the SciKit-Learn library (SciKit-Learn).

## Logistic Regression

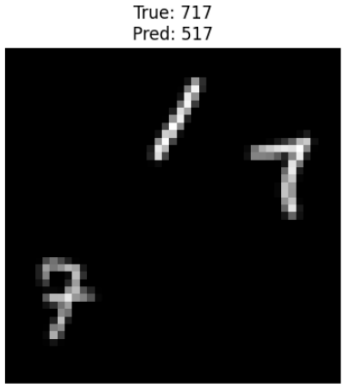
Due to the complexity of the dataset, the logistic regression model did not fare well, achieving 0% accuracy on testing and validation data, and taking over 45 minutes to train. A limited-memory Broyden–Fletcher–Goldfarb–Shanno (LBFGS) solver was used, as the model was tackling a multi-class classification problem, and had a regularisation strength of 1, which served as a reasonable starting point for the problem.

The model had a training accuracy of 82%, and 0% accuracy on the test and validation data, predicting 20,000 false positives. This suggests the model may be overfitting, so the regularisation strength was lowered to 0.1 in the hopes of improving generalisation of unseen labels, however, the model was met with the even worse results, achieving only 40% accuracy on the training data.

The suspected issue with the model was its inability to identify unseen labels, as none of the labels overlap between the training, test and validation data. The model seems to be unable to recognise the patterns between the numbers in each label.

To prove this hypothesis, the train, test and validation sets were combined together and re-split using “test_train_split” from SciKit-Learn (SkiKit-Learn). By splitting the data this way, some classes will be shared between each of the sets, meaning the model will be less likely to encounter an unseen label. These new splits will contain all 1000 labels, with the training data having 60 samples per label, and the test and validation with 20.

This hypothesis was proved correct, as when the model was trained on different splits, it was achieved an accuracy of 0.001%, which still indicated poor performance, but showed that it had much more of chance successfully predicting a label if it has seen it before.




The logistic regression models can be seen to be learning and making predictions that are close, but not quite accurate, which can be seen in the following example:

The model predicted the label to be 517, which closely resembles the actual value of 717, but as 717 was not present in the training data, it was unable to identify it as a label.

*Figure 3 - Logistic regression false prediction*

## Cognitive Neural Network (Full Image)

Cognitive neural networks (CNN) were trained using the same data sets as the logistic regression model.

The “basic” CNN contains a single convolutional layer using 32 3x3 filters and ReLu activation to introduce non-linearity and mitigate vanishing gradients, followed by a 2x2 max pooling layer for feature reduction, where the output is then flattened and passed through a drop out layer at 50% to reduce overfitting. A dense layer with 100 neurons Is added for further feature extraction, with a final dense layer with 1000 neurons and softmax activation for multi-class classification.

The “complex” CNN contains three convolutional blocks, using a similar structure to the single block used in the “basic” model, with the addition of batch normalisation to stabilise and speed up training, and drop out layers to reduce the risk of overfitting. The number of filters doubles in each block to capture more complex features. After the third convolutional block, the output is flattened and passed through a dense layer with 512 neurons and ReLu activation, followed by a 50% drop out layer, where it is finally passed through another dense layer with 1000 neurons and softmax activation.

The models are then compiled using the “Adam” optimiser and use sparse categorical cross-entropy for the loss function, to accommodate for the multi-class integer label predictions. Each model will be trained using linear rate scheduling to adapt the learning rate dynamically during training. The scheduler will halve the learning rate if the validation loss has not improved after 5 epochs, which can allow the model to converge to a better solution.

The “EarlyStopping” callback will also be applied to each model, which will stop training if the validation accuracy of a model has not improved after 10 epochs. This is implemented to avoid overfitting and overconsumption of computational resources.

The “basic” and “complex” models will be trained using the “unseen” and “seen” datasets, where each model will be run four times, with the following variation in parameters:

- Linear Rate Scheduling + Early Stopping
- Early Stopping
- 25 epochs
- 50 epochs

The difference in training strategies allows for models' performances to be systematically analysed.

Both the "complex" and "basic" models failed to correctly identify a single unseen label from the test and validation sets, with most models achieving a training accuracy of over 70%.

The "basic" models trained with early stopping achieved a training accuracy of ≈72% after 11 epochs, with the linear rate scheduling increasing training accuracy to ≈73%. The model training to 25 epochs had a training accuracy of ≈85%, and after 50, the models training accuracy of ≈88%. None of the "basic" models were able to raise the validation accuracy at all, with all "basic" models having a validation loss between 35 and 40.

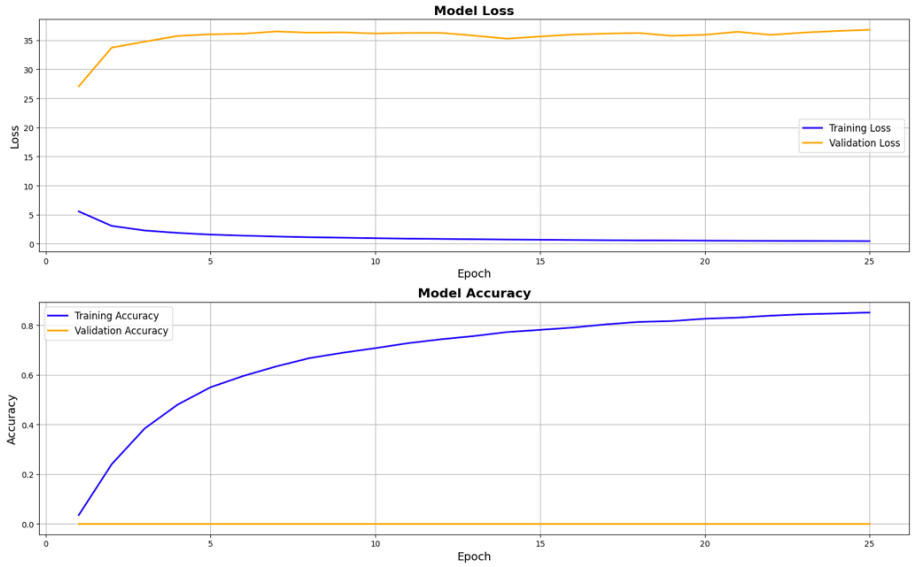



*Figure 4 - Unseen Labels, 25 Epochs, Accuracy 0% ("Basic")*

The "complex" models performed significantly worse than the "basic" model, with the early stopping models achieving extremely low training accuracy and validation loss over 40. As the number of epochs increased, so did the training accuracy, however the validation accuracy remained at zero.

This suggests that the more "complex" models are prone to overfitting and that both models struggle with generalisation, as they are unable to identify and learn meaningful features to apply to new data. The poor performance is likely due to the high number of classes (1000) and low number of samples per label (100), where the models may not be presented with enough data to accurately distinguish between the large number of potential labels.

The models trained on the "seen" dataset showed marginally better performance, with the basic CNN using LRS and early stopping being able to successfully identify 14 labels correctly. It achieved a training accuracy of ≈95% and a validation accuracy of ≈58%, indicating it was able to make some accurate predictions on unseen samples. As the validation accuracy was increasing with each epoch, the early stopping callback was not activating, meaning that the early stopper models ran to 50 epochs.

The model running for 25 epochs was able to successfully identify 26 labels from the test set, making it the highest performing model out of the full image CNNs. The training and validation loss and accuracy closely followed each other throughout the training process.

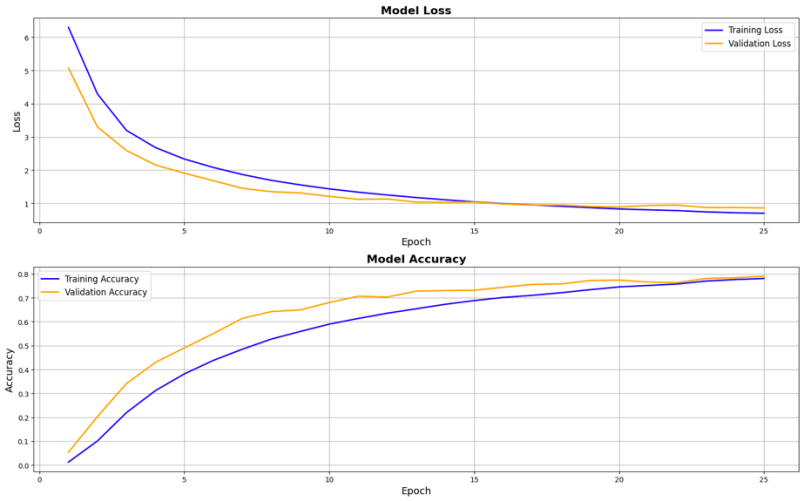



*Figure 5 - Seen Labels, 25 Epochs, Accuracy 0.0013% ("Basic")*

The "complex" models were much more prone to overfitting, reflected in the fluctuations in validation and training accuracy in Figure 6. The early stopping model successfully identified 4 labels after 16 epochs. The other seen complex models failed to make any correct predictions.

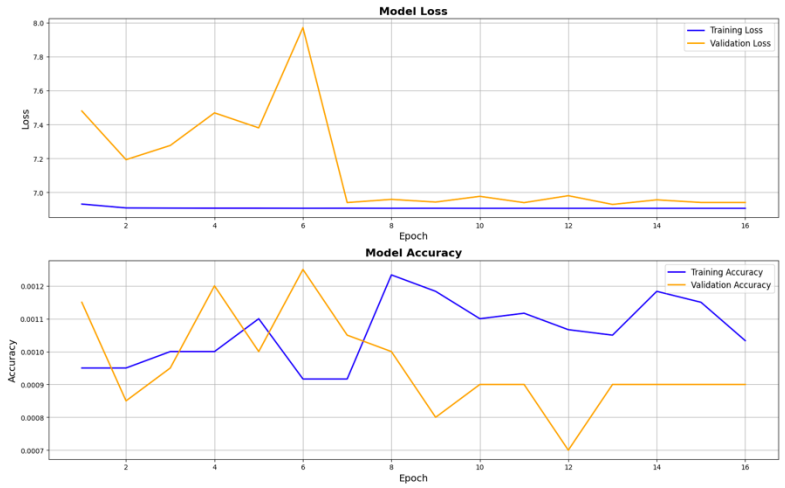



*Figure 6 - Seen Labels, Early Stopping, 16 Epochs, Accuracy 0.0002% ("Complex")*

The CNNs and logistic regression models all performed poorly on full image predictions, even with the use of validation data to tune hyperparameters. Both models saw a marginal improvement when making predictions on labels they had already seen, however the high number of classes meant the models could not accurately predict the labels of unseen samples. This suggests that if given more samples or less labels, the models are likely to improve in performance.

# Cognitive Neural Network (Split Image)

## Pre-processing

To improve the performance of the CNN, the image data was split into three distinct pieces and trained using three separate models, making the number of potential labels 10 instead of 1000. By splitting the data this way and reducing the number of labels, it also means there will be significantly more samples per label, allowing the models to learn with more depth and produce more accurate predictions.

The samples were processed by vertically slicing each image into 3 equal parts and allocating each slice to the respective number in the label, which can be seen in figure 7.1 and 7.2:

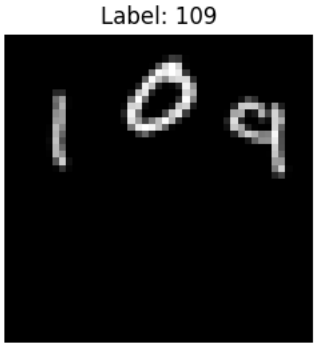



*Figure 7.1 - 109 Label*

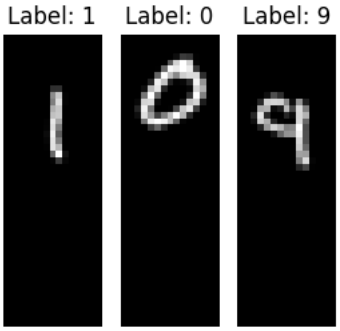



*Figure 7.2 - Split 109 Label*

These splits are then saved into an array, as 45x15 images, where they can then be used to train, test and validate machine learning models.

In order to train a CNN to make predictions on a three-digit image, separate models are trained on each digit and their outputs are stored in arrays, where their outputs can be combined upon completion.

Two CNN architectures were used, similarly to task 2, with a “basic” and “complex” model, with differences in the input shape and output layer, with each model predicting for 10 labels for a single digit, instead of the 1000 labels used in the previous 3-digit full image prediction model.

## “Basic” model

The “basic” model had an accuracy of ≈91% and an F1 score of 0.24, making 1697 incorrect predictions. When analysing the model prediction on each digit, all three digits had an accuracy of ≈97%, indicating that the model was moderately successful at identifying individual numbers, with each model making 550-600 incorrect predictions, summing to 1751 total across individual digits. This suggests that only ≈3% of incorrect predictions had 2 or more incorrectly predicted digits, with the rest only failing to identify a single digit.

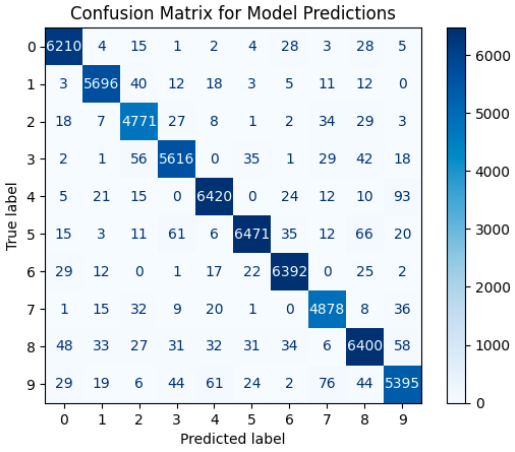



*Figure 8 - Confusion Matrix of All Digits ("Basic")*

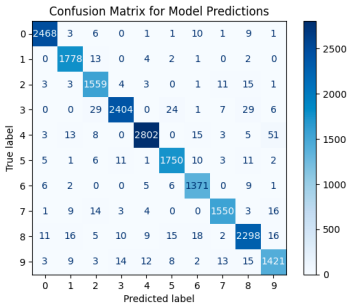



*Figure 9.1 - Digit 1 Confusion Matrix*

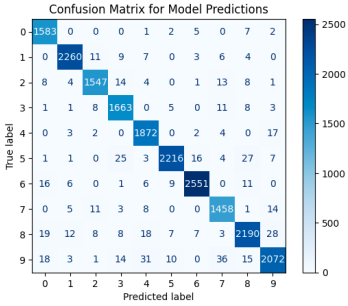



*Figure 9.2 - Digit 2 Confusion Matrix*

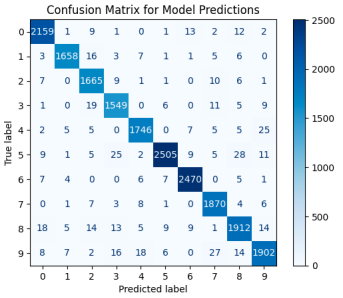



*Figure 9.3 - Digit 3 Confusion Matrix*

The loss and accuracy for training and validation converged smoothly on the basic model, with no signs of overfitting or underfitting.

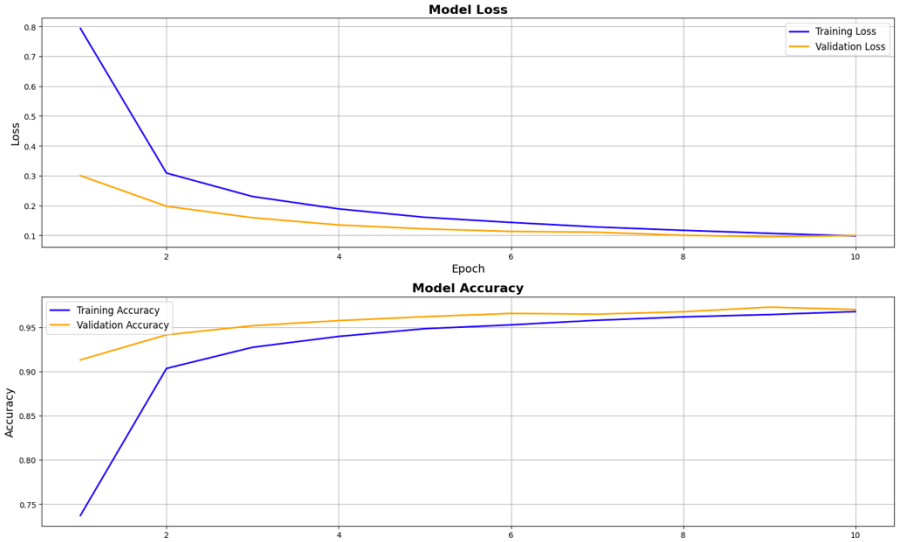



*Figure 10 - Digit 2 ("Basic")*

Compared to the CNN making predictions on full images, the "basic" model performed significantly better. The model making predictions on the split data had an output layer of 10, instead of 1000, which drastically reduces the classification required by the model. Additionally, the new model does not have to make predictions on unseen data, and receives considerably more samples per label, with each digit 0-10 having 1000-2000 samples, compared to the 100 samples per label in the previous model. These changes allow the model to learn with more depth and make predictions more accurately.

## “Complex” Model

To further improve performance, more convolutional layers can be added to the model. The results of which can be seen in the “complex” model.

The “complex” model had an overall accuracy of ≈97% on the test data, with an F1 score of 0.4. It had an accuracy of ≈99% across all digit predictions, making 611 incorrect predictions on individual digits, resulting in 602 total incorrect 3-digit predictions, indicating that only ≈1% of incorrect 3-digit predictions contained 2 or more incorrect predictions. Similarly to the “basic” model, the “complex” model also struggled to distinguish between 4’s and 9’s, which is reflected in the confusion matrices of each model.

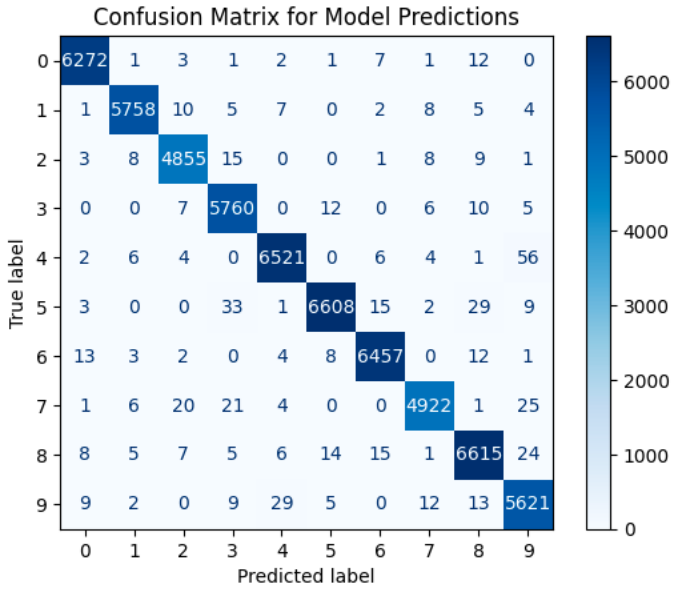



*Figure 11 - Confusion Matrix of all digits ("Complex")*

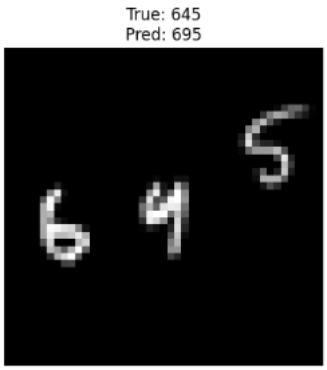



*Figure 12.1 - False Prediction 1 ("Complex")*

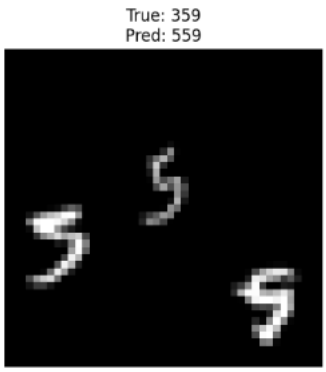



*Figure 12.2 - False Prediction 2 (“Complex”)*

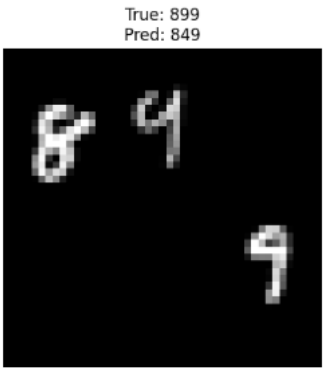



*Figure 12.3 - False Prediction 3 (“Complex”)*

The increase in complexity leads to the model being susceptible to overfitting, which can be seen in figure 13.1 and figure 13.2, which in epochs 7 and 8, the models' accuracy spikes, where they have struggled to identify the image presented in the validation set.

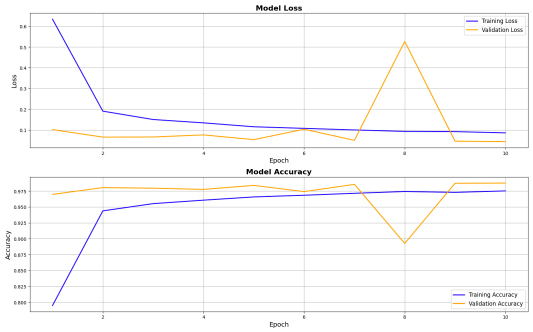



*Figure 13.1 - Digit 1 ("Complex")*

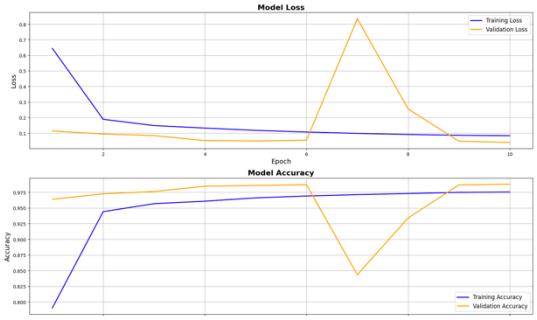



*Figure 13.2 - Digit 2 ("Complex")*

## Data augmentation

In an attempt to further improve accuracy, the complex model was trained on an augmented dataset, which was achieved using ImageDataGenerator from TensorFlow (TensorFlow). This allowed the images in the dataset to be randomly rotated, zoomed, shifted and blurred, which would diversify the data and allow the model to improve its pattern recognition.

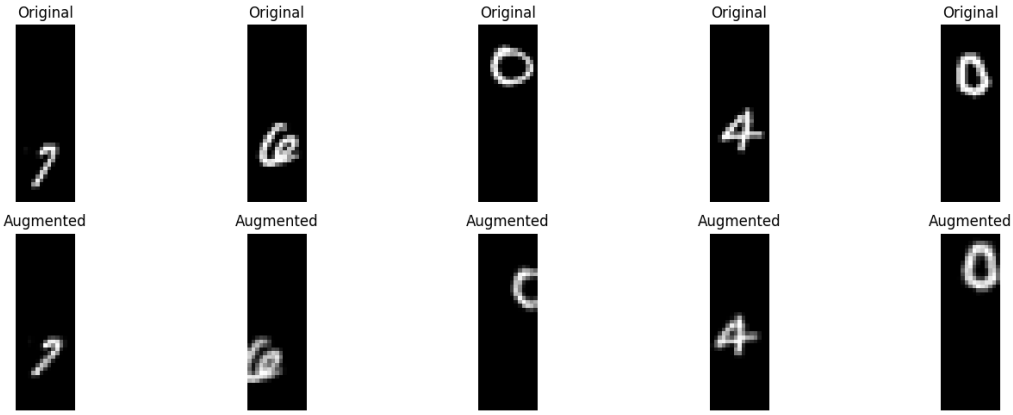



*Figure 14 - Augmented image samples*

When training the "complex" model using the augmented dataset, the accuracy did not improve, with an accuracy of ≈96% when making predictions on the test data. The models also seemed just as susceptible to overfitting, with similar spikes in validation accuracy.

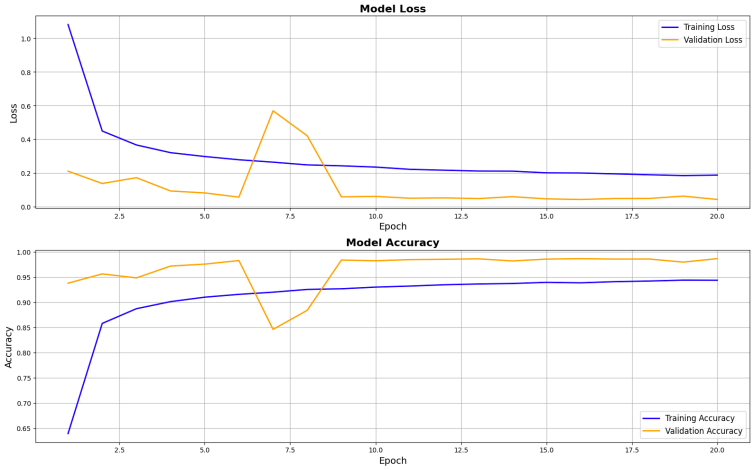



*Figure 15 - Digit 1 (Augmented "Complex")*

# Generative Adversarial Network

A generative adversarial network (GAN) could be used to create images for data augmentation. A GAN consists of a generator and discriminator, where the generator model starts by producing noise, which is then fed into the discriminator model, which predicts whether the image passed from the generator is real or fake. This prediction is passed back to the generator, containing information the generator can use to reshape the noise closer to a real image. This process then repeats until the discriminator cannot distinguish real and fake images, meaning the generator is creating images similar to what is in the dataset.

When trained on a full image, the GAN was unable to produce realistic samples. It was able to identify the three columns where the numbers should go, but was only able to create blobs that vaguely resembled a combination of number shapes.

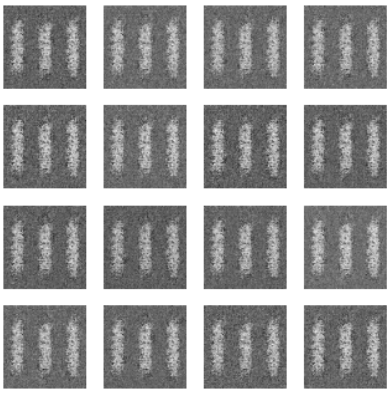


*Figure 15.1 - GAN, 50 epochs*

*Figure 16.2 - GAN, 225 epochs*

With an improved CNN performance on split images, the GAN was also fed single digits. However, it achieved similar results.

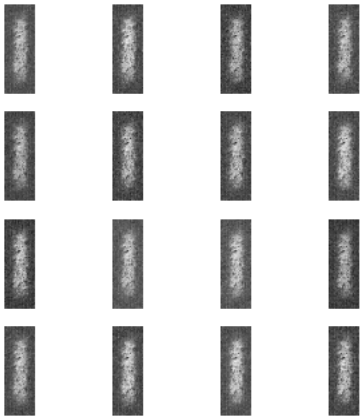


*Figure 18.1 - Slice GAN, 75 epochs*

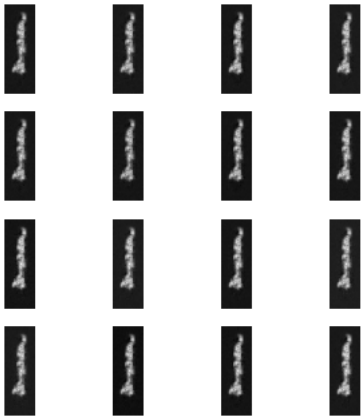


*Figure 17.2 - Slice GAN, 225 epochs*

It is apparent that the GAN architecture is not suitable for the task, and a more label-based approach would be more effective, such as a conditional deep convolutional GAN (CDCGAN). This architecture combines the conditional GAN with deep convolutional networks, meaning the generator would take in both noise and a label, which would specify the desired class. However, after multiple failed attempts, a functional CDCGAN was not successfully implemented.

# References

Cireşan, D., Meier, U. & Schmidhuber, J. (2012) Multi-column deep neural networks for image classification.

PIL *Image module.* https://pillow.readthedocs.io/en/stable/reference/reference/Image.html [Accessed Jan 8 2025].

SciKit-Learn *LogisticRegression.* https://scikit-learn/stable/modules/generated/sklearn.linear_model.LogisticRegression.html [Accessed Jan 8 2025].

Singh, H. (2023) *How do computers store images?* https://www.analyticsvidhya.com/blog/2021/03/grayscale-and-rgb-format-for-storing-images/ [Accessed 08/12 2024].

SkiKit-Learn *Train_test_split.* https://scikit-learn/stable/modules/generated/sklearn.model_selection.train_test_split.html [Accessed Jan 8 2025].

TensorFlow *Tf.keras.preprocessing.image.ImageDataGenerator.* https://www.tensorflow.org/api_docs/python/tf/keras/preprocessing/image/ImageDataGenerator