University of Hull

662086 - Machine Learning

Multi-Digit Classifier Report

Nathan Rignall

February 14, 2024

Contents

List of Figures

List of Tables

1 Visualisation

This report outlines the steps taken to develop a handwritten multi-digit classifier using multiple machine learning techniques. A random sample of images and labels from the provided dataset displays the challenges of implementing such classifier. The red markings in Figure 1.1 highlight a few poorly written digits and additional noise.

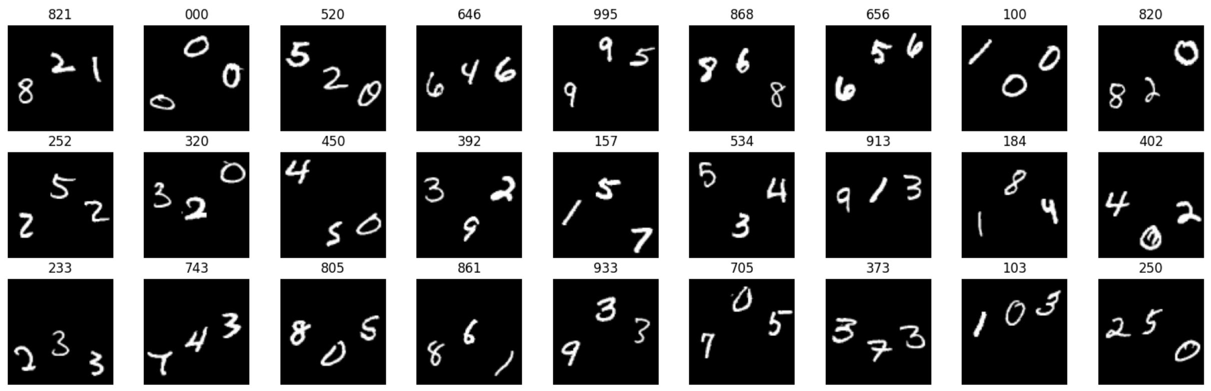


Figure 1.1: Random sample of images and labels

There are 640, 200 and 160 classes present in the train, test, and validation sets, all classes contain 100 images. The images are greyscale. The distribution graphs display when a class is present in a set. The red lines indicate which classes are present in the test or validation sets but not present in the train set. Unfortunately, this shows that all classes in the test and validation set are not in the train set. Using the dataset as it is, with single labels, learning from the training set cannot transfer into the validation and test sets.

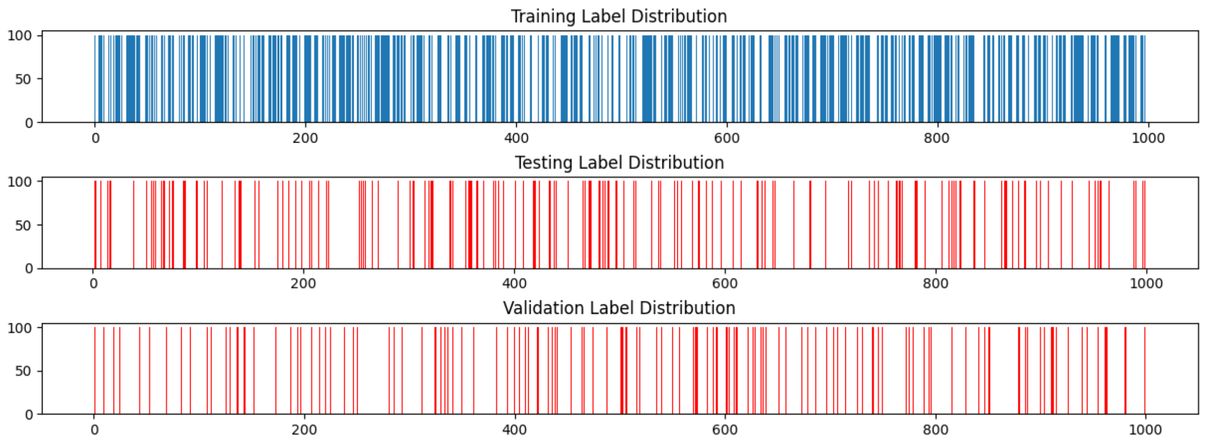


Figure 1.2: Distribution graph for test/train/validation datasets

Instead, this should be treated as a multi-label classification problem, with each label representing a digit. For example, the image containing 123 does not contain a single label of “123” but rather three labels of “1”, “2”, and “3”.

When treated as a multi-label classification problem, analysis of the placement of the digits can take place. The heatmap displays the proportion of the digits at a specific location within the images. Ideally the distribution is 0.1 per digit (10%) at each location. Train is the most evenly distributed set in Figure 1.3, likely due to its size. This is most optimal as during training the model shall receive an even exposure of all digits.

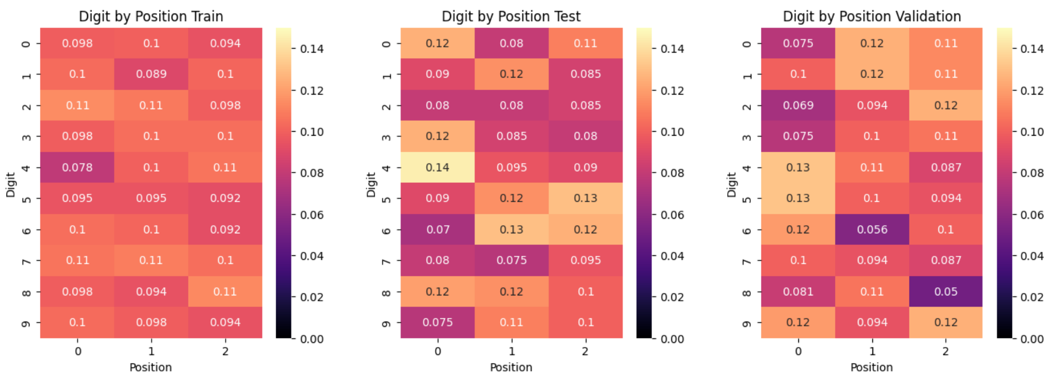


Figure 1.3: Heatmap for class distribution in dataset

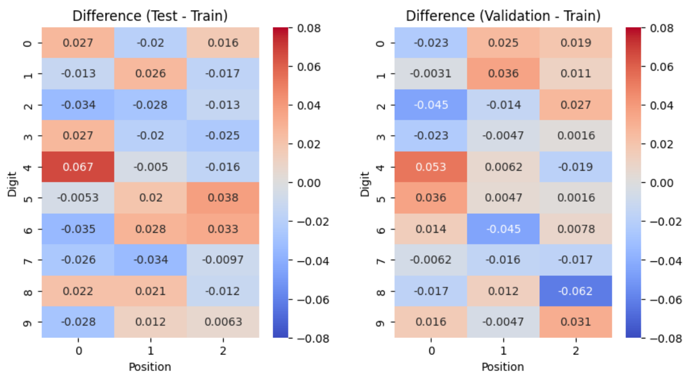


Figure 1.4: Heatmap for difference in class distribution across dataset

The additional heatmaps in Figure 1.4 display the difference between the test and validation sets against the train datasets, designed to highlight any gaps in training. Any digits marked with a positive difference (marked with a red shade) may perform worse in the validation and training evaluations. Digit four in position 1 appears to contain the most positive difference relative to the training set.

2 | Training

2.1 Single Model Training

To set a baseline we can treat this as a simple classification problem with one thousand possible classes. With a complete dataset this could perform moderately well. However, as all classes in the validation and test sets are not present in the train, it can be expected the models will achieve an accuracy of near zero. Assuming accuracy is measured using the total number of labels entirely classified as correct, rather than individual digits.

2.1.1 Convolutional Neural Network

To form the basis of our multi-digit classifier we shall use an existing convolutional neural network (CNN) architecture with slight modifications. (Biswas et al. 2021) conducted a survey into optimal network architectures, the most optimal performed at an accuracy of 99.53%. Figure 2.1 displays such layer design in combination with additional dropout layers in between the fully connected layers. Dropout layers, a regularization method, are a simple yet affective way to reduce the likelihood of overfitting (Srivastava et al. 2014). This model contains a six-layer CNN plus a two-layer dense classifier.

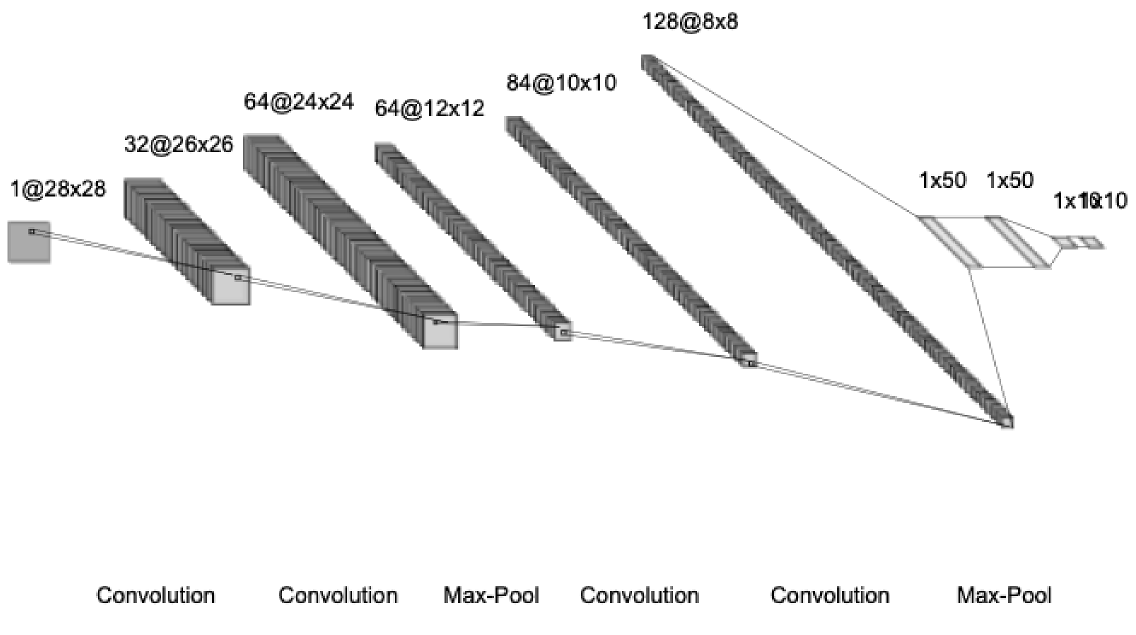


Figure 2.1: CNN architecture generated using Alex Lenail

Using the Adam optimiser with a learning rate of 0.001 and the categorical crossentropy loss function an accuracy of 0.0% is achieved on the test set.

The learning graphs in Figure 2.2 could suggest the model is overfitting, when a model performs well on the training data but does not generalize well (Géron 2019). However, this is because all test and validation classes are missing in the test set, likely not due to the model being too complex. No amount of regularization could improve this model's performance.

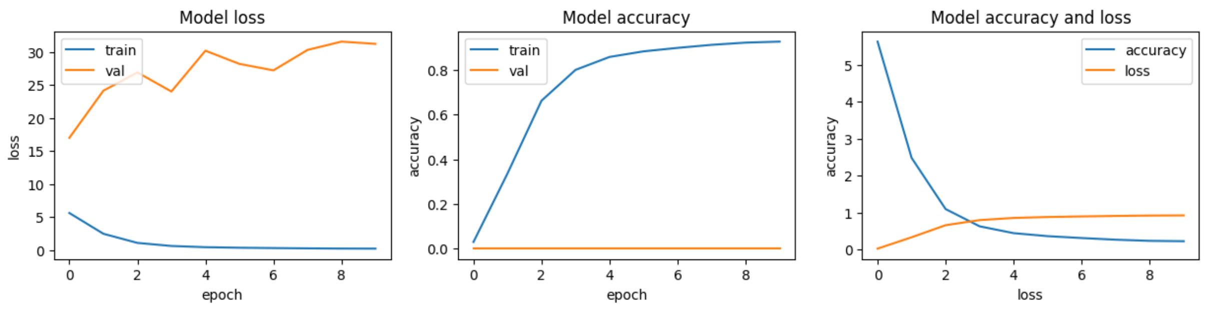


Figure 2.2: Learning graphs for CNN

This theory is further reenforced in Figure 2.3 whereby the predictions are mostly correct on unseen data. Approximately two digits within the predictions are true versus the labels, but never three. Missing data is the cause of these false predictions.

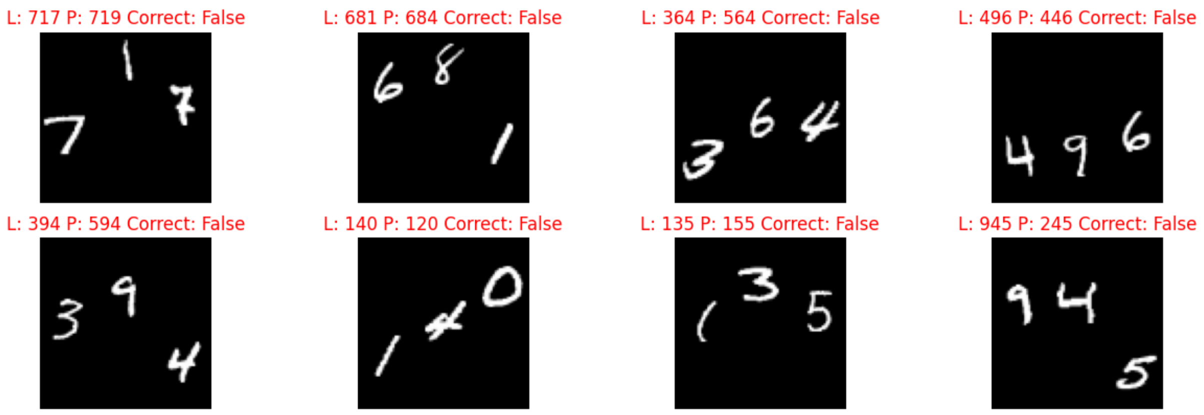

L: 717 P: 719 Correct: False
L: 681 P: 684 Correct: False
L: 364 P: 564 Correct: False
L: 496 P: 446 Correct: False
L: 394 P: 594 Correct: False
L: 140 P: 120 Correct: False
L: 135 P: 155 Correct: False
L: 945 P: 245 Correct: False

Figure 2.3: Predictions for CNN

In the model's current state, it is impossible to complete hyperparameter tuning as the accuracy metric on both the validation and test sets is zero. No hyperparameter tuner could deduce what parameters are better if the difference in accuracy is negligible. In order to conduct tuning, a new performance metric must be used if the overall model architecture is not changed. Currently accuracy is evaluated using an exact comparison between two labels. For instance, the label "717" is not equivalent to "719" however, this is technically 66.66% correct. If the accuracy is measured on a per-digit basis we can hope that the validation accuracy improves so that tuning can occur.

The original dataset shall not be modified to remove this poor distribution as it mirrors a real-world problem scenario where not all classes are present.

Using the hyperparameter values displayed in Table 2.1 an accuracy of 66.56% is achieved on the test set using the per-digit metric. This value is exactly as expected as the classifier is able to successfully classify approximately two thirds of the digits. This proves the model is not overfitting from an overly complex model, rather the classification layer is simply missing entire classes. Unfortunately using the traditional per-class metric, an accuracy of 0.0% is observed.

Parameter	Values	Selected
kernel_size_1	2, 3, 4, 5	4
kernel_size_2	2, 3, 4, 5	3
kernel_size_3	2, 3, 4, 5	4
kernel_size_4	2, 3, 4, 5	4
pool_size_1	2, 3, 4	4
pool_siz_2	2, 3, 4	3
dropout_1	0.2, 0.8, step=0.1	0.6
dropout_2	0.2, 0.8, step=0.1	0.5
dense_layer_units	1000, 2000, 3000, 4000, 5000	3000
learning_rate	min=0.0001, max=0.1, sampling="LOG"	0.00024

Table 2.1: Hyperparameter values for CNN

The learning graph in Figure 2.4, containing the unmodified loss metric, is almost identical to that of the loss graph in Figure 2.5. This is because the model's architecture hasn't fundamentally changed. The accuracy graph does reflect the new score, but the overall performance is similar to the unoptimized model. The underlying convolutional and pooling layers are likely correct, but the dense layers are lacking the exposure from the additional classes.

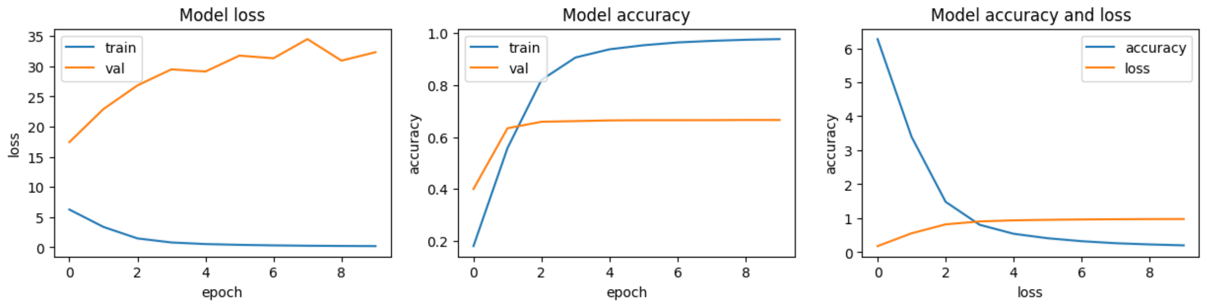
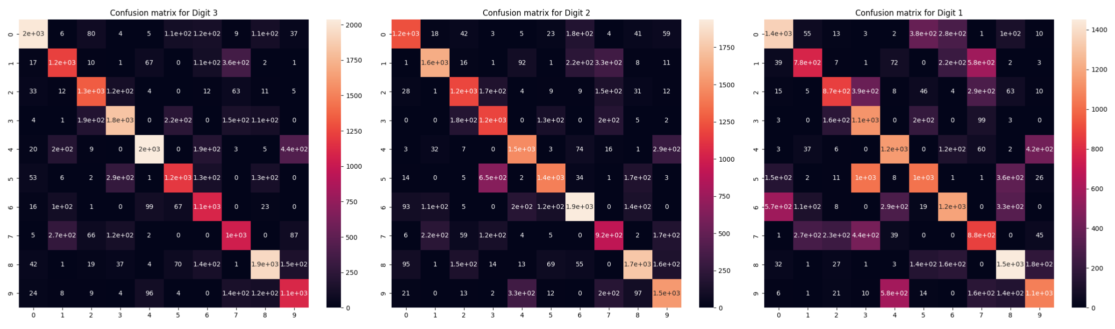


Figure 2.4: Learning graphs for hyperparameter tuned CNN



(a) Digit 1 **(b)** Digit 2 **(c)** Digit 3

Figure 2.5: Confusion matrices for hyperparameter tuned CNN

2.1.2 Convolutional Neural Network with Custom Encode

Without making any improvements to the model itself we can attempt to improve its accuracy on unseen classes by changing how the classes are encoded.

Currently the labels are one-hot encoded. As there are a thousand classes, this produces an array of size 1000. Learning from individual digits cannot transfer across the classifier. As far as the neural network is concerned, there is no similarity between 312 and 345, despite the digit three being repeated across the two classes. This is one of the reasons why the test set performs so poorly.

If instead the digits are encoded in such a way so that learning can be transferred across classes, the model performance could be improved. One possible technique is one-hot encoding each digit individually and concatenating the result. This produces three arrays of size ten that are concatenated together to produce an array of size thirty, Figure 2.6 versus Figure 2.7. The output space is no longer significantly expanded to one thousand and is instead thirty. Now training does not require all one thousand classes to be present, instead each digit location must just include a class from one to ten.

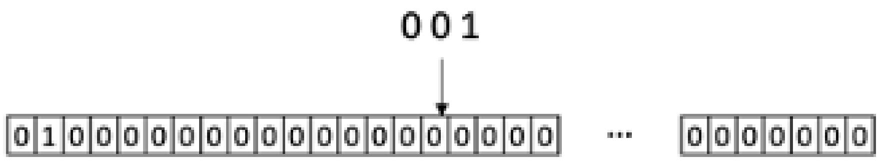


Figure 2.6: Traditional one-hot encoding method for classification

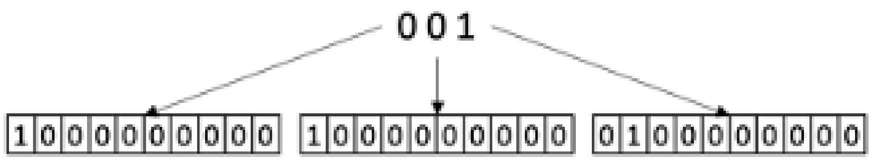


Figure 2.7: Custom one-hot encoding method for classification

In order for the model to understand this method of encoding, a custom loss function and accuracy metric must be created. For the loss function, we can use the categorical crossentropy loss function for each digit by slicing the input arrays and summing the loss. The custom accuracy metric can use the same technique of slicing but instead use the categorical accuracy metric and average. A per-digit accuracy of 98.12% is realised on the test set after ten epochs of training.

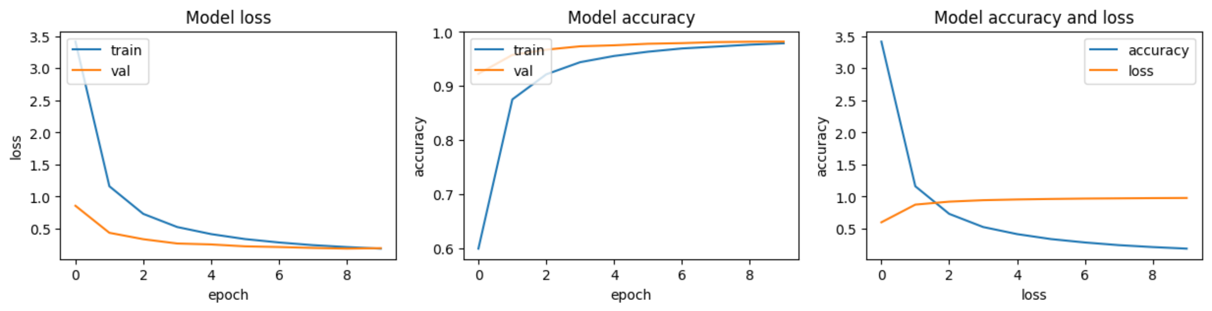


Figure 2.8: Learning graphs for CNN with custom encoding

The learning graphs display the model is not overfitting, the validation set outperforms the training set. This is most likely due to the validation set being of a higher quality and smaller sample size than the training.

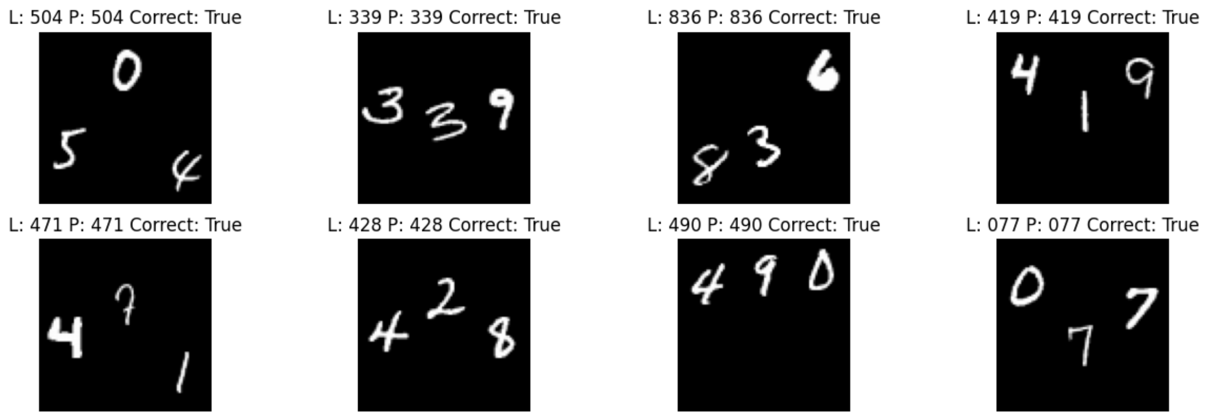
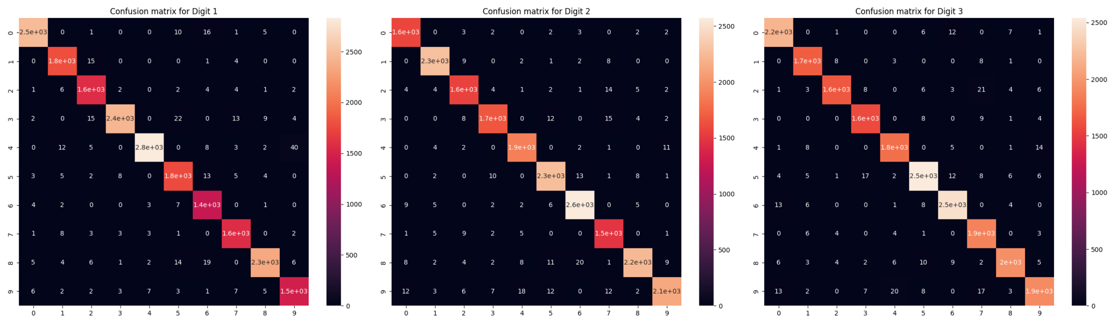

L: 504 P: 504 Correct: True
L: 339 P: 339 Correct: True
L: 836 P: 836 Correct: True
L: 419 P: 419 Correct: True
L: 471 P: 471 Correct: True
L: 428 P: 428 Correct: True
L: 490 P: 490 Correct: True
L: 077 P: 077 Correct: True

Figure 2.9: Predictions for CNN with custom encoding

The random sample of images in Figure 2.9 shows that the model is performing well but some challenging digits are still being miss-classified.

The model is also fairly balanced, no digit location or class is performing significantly worse than another. The even distribution of accuracy and F1 scores in Figure 2.10 and balanced confusion matrices in Table 2.2 reenforce this.

Using the validation set the model can undergo hyperparameter tuning. The dense layer is reduced to resemble the output layer more closely.



(a) Digit 1 (b) Digit 2 (c) Digit 3

Figure 2.10: Confusion matrices for CNN with custom encoding

	Accuracy	F1 Score
Digit 1	0.9810	0.9798
Digit 2	0.9821	0.9817
Digit 3	0.9811	0.9810

Table 2.2: Accuracy and F1 scores for CNN with custom encoding

Parameter	Values	Selected
kernel_size_1	2, 3, 4, 5	2
kernel_size_2	2, 3, 4, 5	5
kernel_size_3	2, 3, 4, 5	5
kernel_size_4	2, 3, 4, 5	2
pool_size_1	2, 3, 4	3
pool_siz_2	2, 3, 4	4
dropout_1	0.2, 0.8, step=0.1	0.8
dropout_2	0.2, 0.8, step=0.1	0.4
dense_layer_units	64, 128, 256, 512, 1024	128
learning_rate	min=0.0001, max=0.1, sampling="LOG"	0.00040

Table 2.3: Hyperparameter values for CNN with custom encoding

A per-digit accuracy of 99.51% is achieved after ten epochs of training, which is better than before hyperparameter tuning. The learning graphs displayed in Figure 2.11 demonstrate the model is not overfitting and the per-digit accuracy scores remain balanced in Figure 2.12.

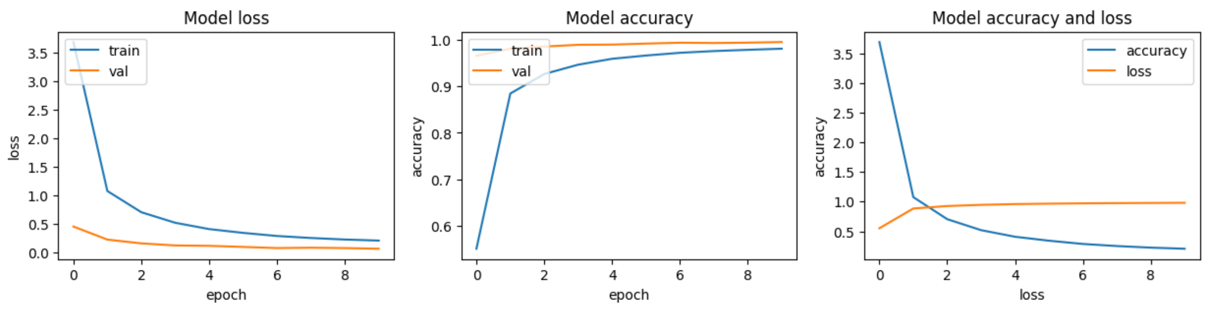
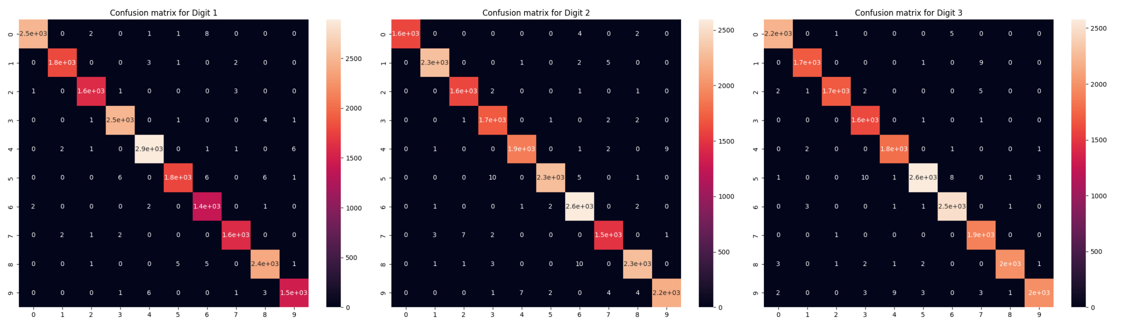


Figure 2.11: Learning graphs for hyperparameter tuned CNN with custom encoding



(a) Digit 1 **(b)** Digit 2 **(c)** Digit 3

Figure 2.12: Confusion matrices for hyperparameter tuned CNN with custom encoding

	Accuracy	F1 Score
Digit 1	0.9954	0.9952
Digit 2	0.9948	0.9947
Digit 3	0.9953	0.9953

Table 2.4: Accuracy and F1 scores for hyperparameter tuned CNN with custom encoding

Despite this model performing an order of magnitude better than the previous CNN plus classifier without custom encoding, there are still improvements that can be made with the architecture of the network. Currently no learning from each digit location can be transferred. For instance, if the digit zero never appears in position one in the training set but is present in position two and three. When tested with a zero in position one the model will likely be unable to classify the digit. Ideally the classification task should be shared across each digit location, to produce a more robust classification model.

2.1.3 Decision Tree

To collect another baseline performance metric, we shall use a simple decision tree model with no additional encoding techniques. Fitting a decision tree with varying depth levels should display if the model is overfitting when the train and test sets are compared.

Figure 2.13 displays the comparison between class and digit accuracies achieved, the model is initially underfitting then immediately overfitting. In contrast to the previous CNN the decision tree is unable to find the underlying patterns in the datasets. A per-digit accuracy of 18.72% is achieved on the unseen test set, which is marginally above statistical chance (10%), considerably worse than 66.56%.

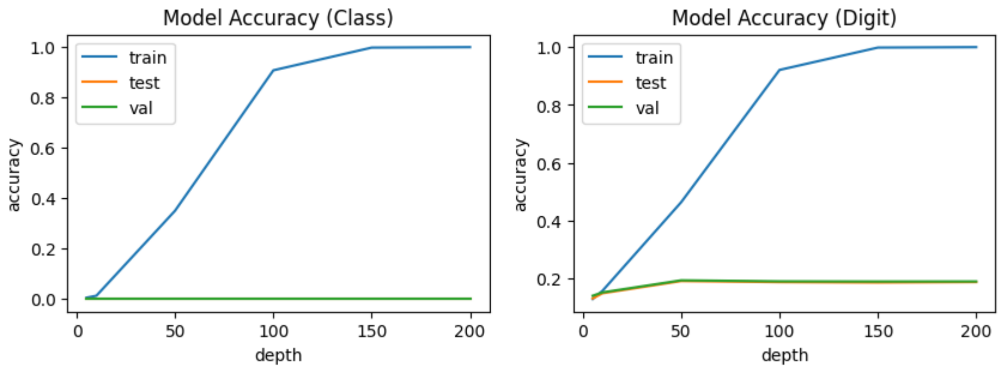
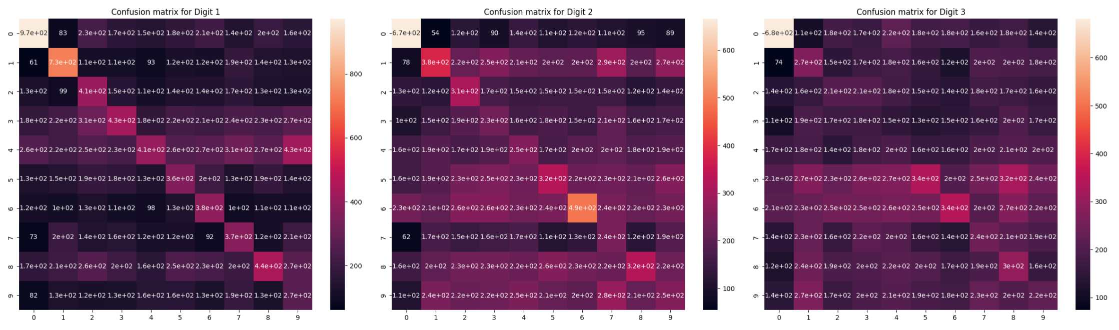


Figure 2.13: Learning graphs for decision tree

The decision tree appears to perform slightly better with the digit zero, this is likely a random pattern. Position one performs somewhat better in comparison to the other positions. Overall, the confusion matrices do show that the tree fails to classify most digits and the decision tree is a poor model.



(a) Digit 1 **(b)** Digit 2 **(c)** Digit 3

Figure 2.14: Confusion matrices for decision tree

In an attempt to reduce overfitting, the model shall undergo tuning using Sklearn's GridSearchCV. This function performs cross-validation, so that the model is discouraged from learning the training set too well. While cross validation is taking place, hyperparameters shall be tuned using the values in in Table 2.5.

Parameter	Values	Selected
max_depth	5, 10, 50, 100, 150, 200	200

Table 2.5: Hyperparameter values for decision tree

Due to the additional steps taken to reduce overfitting, the model is now underfitting. The decision tree is unable to capture the true pattern in the dataset. An overall accuracy of 18.96% is achieved using the test set and is displayed in Figure 2.15. The confusion matrices in Figure 2.16 display a tendency for the model to predict the class one, this is likely a random pattern as the distribution of digits in the training set is even.

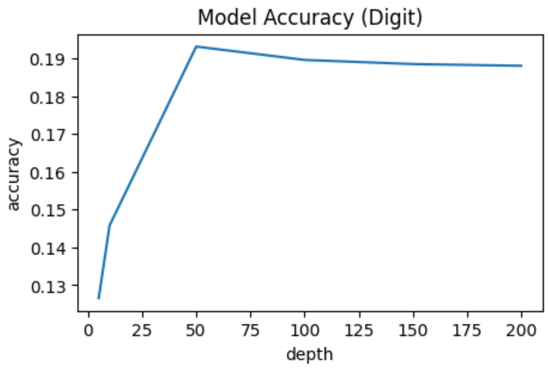
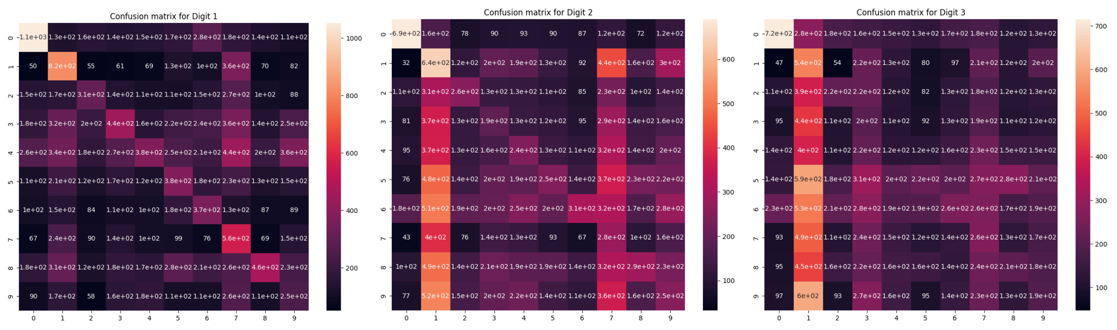


Figure 2.15: Learning graphs for hyperparameter tuned decision tree



(a) Digit 1 **(b)** Digit 2 **(c)** Digit 3

Figure 2.16: Confusion matrices for hyperparameter tuned decision tree

The decision trees have performed poorly in comparison to the CNN models because they do not contain methods to learn the hierarchical features within images. With increased depth and data, it could be possible for a decision tree to slowly learn these relationships.

2.2 Multi-Model Convolutional Neural Network

To individually classify digits, the digits must be first split into separate images. There are several methods for splitting multiple digits (words) into individual digits (characters) for use with a classification model.

2.2.1 Image Splitting

A slice operation separates the image along pre-defined boundaries. This assumes that there are exactly three digits and that they are similarly placed across the data set. To reduce the individual character image size, the digits are also cropped horizontally with the presence of a digit in the top pixel row.

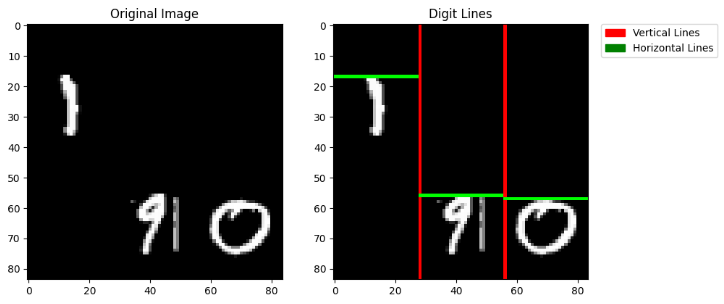


Figure 2.17: Simple image splitting process

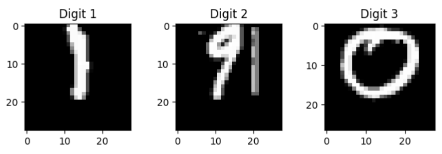


Figure 2.18: Simple image splitting result

This method could work well in a pre-defined scenario. Though, it is likely to struggle in a real-world implementation where there could be any number of digits at random locations within the image.

Another method dynamically divides the images using zero-filled columns and rows. When a column with every pixel value of zero is found, followed by a non-empty column, the image is cropped. The process is repeated in the horizontal dimension using rows.

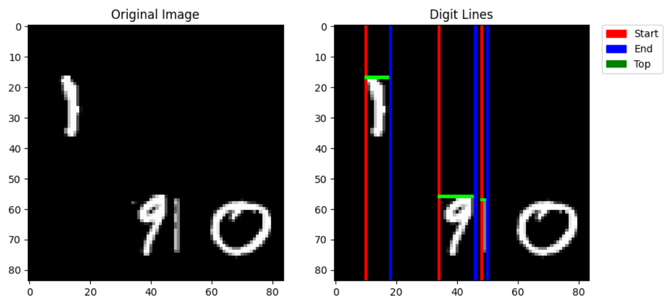


Figure 2.19: Dynamic image splitting process

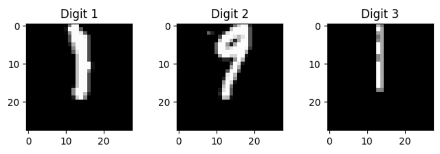


Figure 2.20: Dynamic image splitting result

This relies on the assumption the digits do not overlap in either the horizontal or vertical dimensions. To prevent noise or digits with broken lines being accidently split supplementary checks are required. Examples of the splitting process failing are shown in the report's appendix, these could benefit from such checks.

A more complex, but robust technique for separating the numbers into digits is using contour detection.

The difference of white to black is used to generate continuous contours around a digit, marked with the green outline. The contours are filtered so that objects with a very small area (14px) are removed. This should remove noise, such as pen smudges. Using the contours, a series of bounding boxes are created, marking where the images should be cropped. The cropped images are subsequently padded so that they fill a 28x28 image.

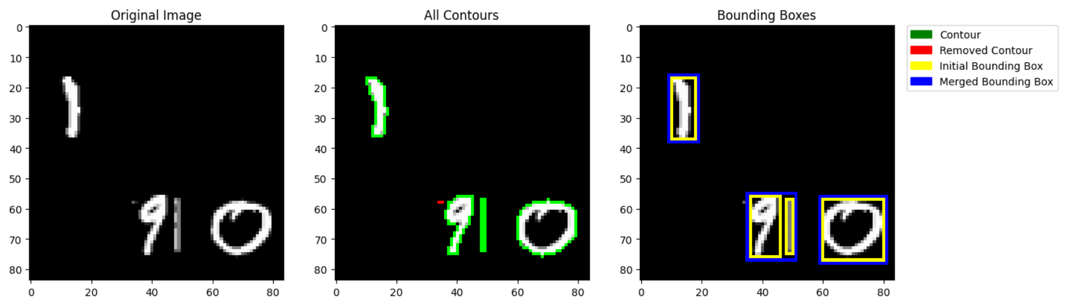


Figure 2.21: Contour image splitting process

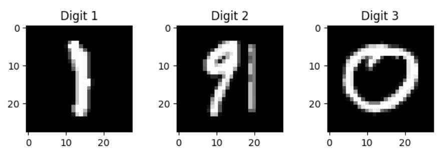


Figure 2.22: Contour image splitting result

This method works well, until a digit has a break in its line. To combat this, we can merge cropped images together if they meet a specified threshold. If all digits are assumed to be 28x28, every contour merge pair can be tested. If the merge is wider or taller than 28px, the contours are not merged. The specific thresholds can be fine-tuned using hyperparameter tuning.

For this technique of splitting to be effective, the handwriting must not be cursive. This is unlikely to be the case with digits but there is still the possibility of digits touching. In this scenario, the contours of two digits would unintentionally join.

2.2.2 Convolutional Neural Network

Although more complicated, integrating the image splitting directly into TensorFlow allows for hyperparameter tuning of the splitting layer itself. The optimum pair of splitter and classifier can be tuned together, rather than apart.

Multiple instances of the same base model are concatenated together into one larger model with the custom Keras splitting layer. Each base model takes one processed image from the splitting layer and performs classification. The output from each model is then concatenated back together. The same custom loss and accuracy functions are required from the previous CNN model. Figure 2.23 displays these repeated models using the dashed box border.

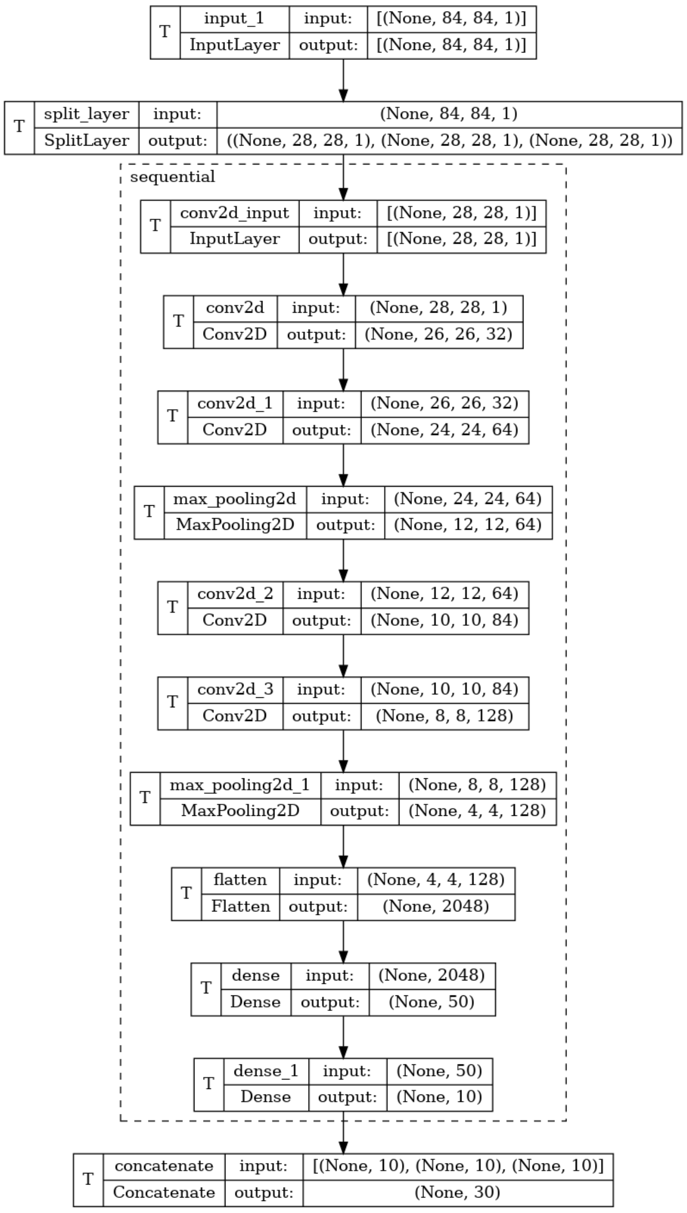


Figure 2.23: Multi-model architecture

Note that the dropout layers are removed in this splitting CNN model, this is because the model struggled to train with them present. With additional epochs and tuning it is likely they could work so during hyperparameter tuning they shall be re-introduced. Although very slow to train this model performs with an overall accuracy of 99.58%. The slow training is because the digit splitting layer is processed on the CPU, rather than GPU. The performance of this model could be improved if the splitting step was parallelised. No overfitting is apparent in Figure 2.24 and the model's performance is comparable to the single digit classifier from (Biswas et al. 2021). However, despite the extra complexity, the model is marginally better than the CNN with a custom encoding method.

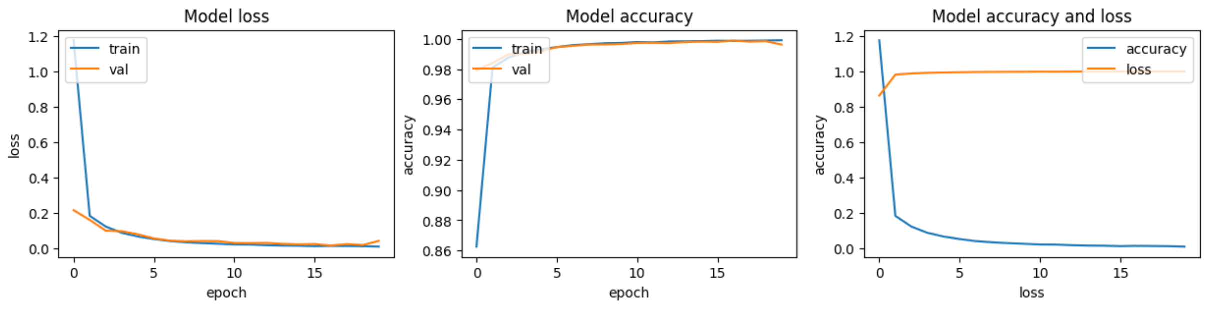


Figure 2.24: Learning graphs for multi-model CNN

The values used for hyperparameter tuning are shown in Table 2.6. Additional parameters for the splitting layer include contour size and threshold boundaries.

Parameter	Values	Selected
kernel_size_1	2, 3, 4, 5	4
kernel_size_2	2, 3, 4, 5	3
kernel_size_3	2, 3, 4, 5	3
kernel_size_4	2, 3, 4, 5	4
pool_size_1	2, 3, 4	2
pool_siz_2	2, 3, 4	2
dropout_1	0.2, 0.8, step=0.1	0.4
dropout_2	0.2, 0.8, step=0.1	0.2
dense_layer_units	10, 50, 100	10
contour_size	0.001, 0.01, step=0.001	0.001
threshold_lower	0, 120, step=20	80
threshold_upper	135, 255, step=20	175
learning_rate	min=0.0001, max=0.1, sampling="LOG"	0.00021

Table 2.6: Hyperparameter values for multi-model CNN

Using the optimal parameters, a per-digit accuracy of 99.65% is achieved on the test set with fifty epochs of training. The learning graphs in Figure 2.25 display that the parameters found produce a slower training curve for the train set. This could be because the regularisation dropout layers are strong. Training for additional epochs could improve this score.

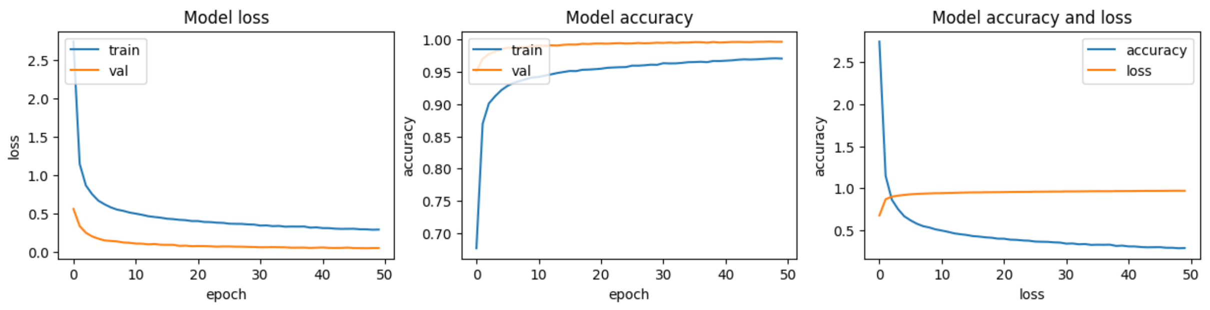
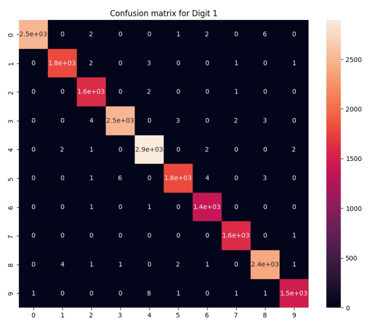
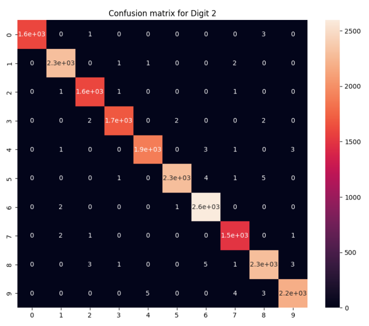


Figure 2.25: Learning graphs for hyperparameter tuned multi-model CNN

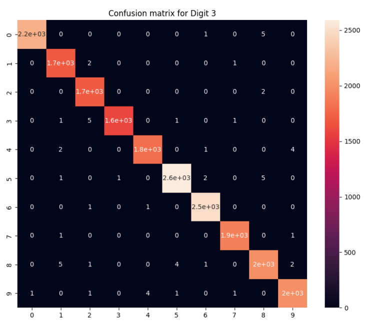
The final model performs excellently, with even digit distribution displayed in both Figure 2.26 and Table 2.4. The heatmaps from the confusion matrices closely resemble the digit distribution heatmaps from initial visualisation.



(a) Digit 1



(b) Digit 2



(c) Digit 3

Figure 2.26: Confusion matrices for hyperparameter tuned multi-model CNN

	Accuracy	F1 Score
Digit 1	0.99605	0.99597836
Digit 2	0.9966	0.9966265
Digit 3	0.997	0.9969314

Table 2.7: Accuracy and F1 scores for hyperparameter tuned multi-model CNN

The improved performance from this CNN model and splitting layer is realised because the single classifier is able to gain experience from every digit location. In a direct comparison to the simple CNN with encoding, the classification layers are trained on triple the amount of data. This reenforces the model's learning of handwritten digits.

2.3 Optimisation

The models developed all use advanced techniques to produce the most optimal multi-digit classifier. One of the best single MNIST classifiers achieves an accuracy of 99.87% using homogeneous vector capsules (Byerly et al. 2021). We could attempt to increase the current model's performance or build a new model for additional tasks.

One possible method to build a more generic handwriting recognition model is to use a recurrent neural network (RNN) (Advaith et al. 2021). Typically, when classifying text there is a semantic relationship between each letter, this can aid the classification process by suggesting what characters are likely. An RNN can store this sequential relationship between letters. This kind of model would be unnecessary and impractical for a simple multi-digit classification problem as there is little relationship between each digit in an image.

Another improvement technique could be to use an object recognition model to find text to classify within an image. This would involve taking a pre-trained model such as "ssd_mobilenet_v2" from TensorFlow and fine-tuning it on the handwritten digit dataset. However, object classification models are very large and are generally designed for much more complex image classification tasks.

Instead, to build a more diverse digit classification model, and hence more accurate, we will use data augmentation techniques. The CNN with custom encoding lacked exposure from every digit in comparison to the CNN with image splitting. The digits can be randomly rearranged so that every digit classifier can gain even experience from every digit, Figure 2.27. This can be implemented within the dataset loading layer.

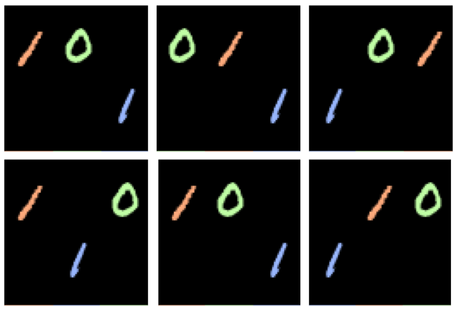


Figure 2.27: Digit augmentation example

In addition to the augmentation, the hyperparameter search can be expanded to find optimum layer sizes for the convolutional and pooling layers. This could significantly increase the tuning time but increase the model's performance.

Parameter	Values	Selected
kernel_size_1	2, 3, 4, 5	3
kernel_size_2	2, 3, 4, 5	5
kernel_size_3	2, 3, 4, 5	4
kernel_size_4	2, 3, 4, 5	5
pool_size_1	2, 3, 4	4
pool_size_2	2, 3, 4	4
conv2d_1	min=32, max=256, step=32	32
conv2d_2	min=32, max=256, step=32	128
conv2d_3	min=32, max=256, step=32	256
conv2d_4	min=32, max=256, step=32	32
dropout_1	0.2, 0.8, step=0.1	0.5
dense_layer_units	64, 128, 256, 512, 1024	64
learning_rate	min=0.0001, max=0.1, sampling="LOG"	0.00014

Table 2.8: Hyperparameter values for CNN with custom encoding and augmentation

A final accuracy of 99.91% is observed on the test set, using the tuned hyperparameters, after one hundred epochs of training. Despite the increased model complexity and training time, the model is not overfitting and performs well on the unseen data, Figure 2.28.

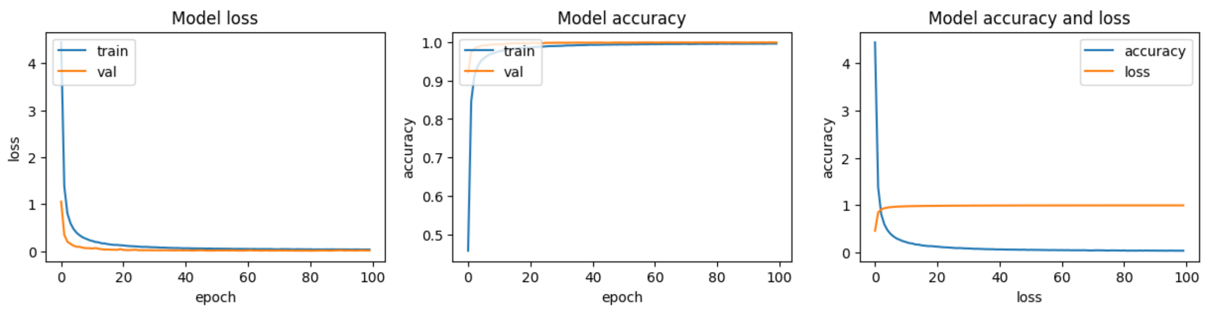


Figure 2.28: Learning graphs for hyperparameter tuned CNN with custom encoding and augmentation

The diagram in Figure 2.29 displays the slight modifications made to the combined CNN architecture. The digit classification head layers were separated, this provides no performance benefit but clarifies the model's behaviour. The image includes the tuned hyperparameter values for each layer size.

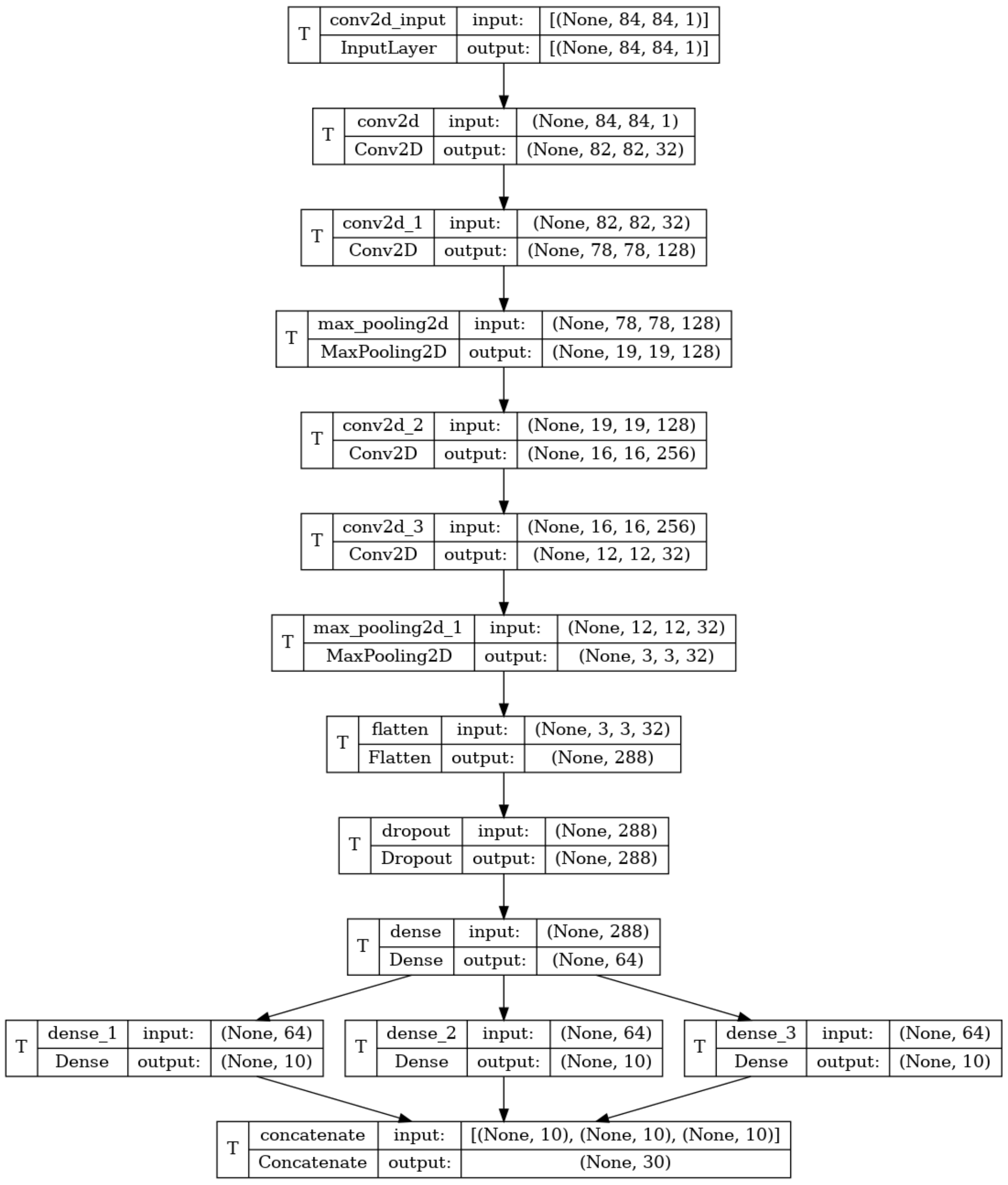
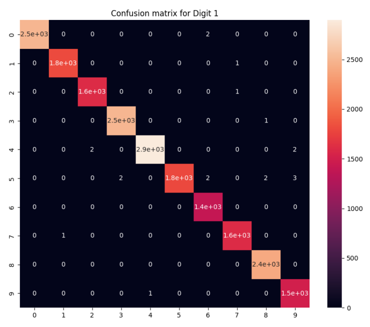
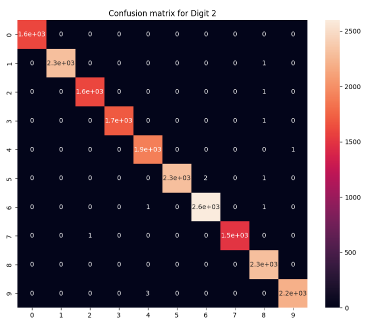

T conv2d_input InputLayer input: [(None, 84, 84, 1)] output: [(None, 84, 84, 1)]
T conv2d Conv2D input: (None, 84, 84, 1) output: (None, 82, 82, 32)
T conv2d_1 Conv2D input: (None, 82, 82, 32) output: (None, 78, 78, 128)
T max_pooling2d MaxPooling2D input: (None, 78, 78, 128) output: (None, 19, 19, 128)
T conv2d_2 Conv2D input: (None, 19, 19, 128) output: (None, 16, 16, 256)
T conv2d_3 Conv2D input: (None, 16, 16, 256) output: (None, 12, 12, 32)
T max_pooling2d_1 MaxPooling2D input: (None, 12, 12, 32) output: (None, 3, 3, 32)
T flatten Flatten input: (None, 3, 3, 32) output: (None, 288)
T dropout Dropout input: (None, 288) output: (None, 288)
T dense Dense input: (None, 288) output: (None, 64)
T dense_1 Dense input: (None, 64) output: (None, 10)
T dense_2 Dense input: (None, 64) output: (None, 10)
T dense_3 Dense input: (None, 64) output: (None, 10)
T concatenate Concatenate input: [(None, 10), (None, 10), (None, 10)] output: (None, 30)

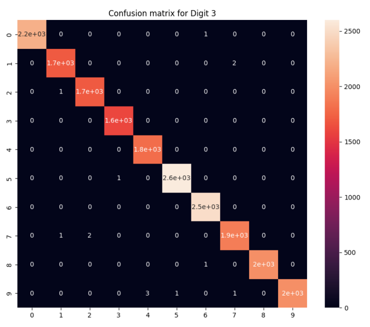
Figure 2.29: Modified multi-head CNN architecture



(a) Digit 1



(b) Digit 2



(c) Digit 3

Figure 2.30: Confusion matrices for hyperparameter tuned CNN with custom encoding and augmentation

3 Conclusion

Several machine learning techniques have been applied to a multi-digit classification problem. We reduced the classification space from one thousand to thirty by encoding the model using multi-label methods. The simple CNN models dramatically outperformed the decision tree models due to them being better suited to hierarchical feature learning.

Next, we tested splitting the images with a variety of techniques and performing classification on single digits at a time. This yields a small but significant increase in performance. The final model does not use a splitting layer, this allows the entire model to be trained on a GPU which is much faster. Because of the increased performance, more in-depth hyperparameter tuning could take place. An accuracy of 99.93% is achieved. The majority of the failed classifications would be exceptionally challenging for a human to interpret also. In some cases, the provided labels in the dataset could be considered incorrect, examples are shown in Table 3.1.

Simple CNN	Simple CNN (encoding)	Decision Tree	Splitting CNN	Final CNN
66.56%	99.51%	18.96%	99.65%	99.93%

Table 3.1: Accuracy of models

Bibliography

Advaith, Ala Sree et al. (2021). "Handwriting Recognition Using CNN and RNN". In: 26.1671.

Biswas, Angona and Md. Saiful Islam (04/27/2021). "An Efficient CNN Model for Automated Digital Handwritten Digit Classification". In: *Journal of Information Systems Engineering and Business Intelligence* 7.1, p. 42. DOI: 10.20473/jisebi.7.1.42-55. <https://e-journal.unair.ac.id/JISEBI/article/view/24237> (visited on 01/13/2024).

Byerly, Adam, Tatiana Kalganova, and Ian Dear (06/17/2021). *No Routing Needed Between Capsules*. Version 6. Comment: 13 pages, 7 figures, 9 tables. arXiv: 2001.09136 [cs]. <http://arxiv.org/abs/2001.09136> (visited on 01/16/2024). preprint.

Géron, Aurélien (2019). *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. Sebastopol, UNITED STATES: O'Reilly Media, Incorporated. <http://ebookcentral.proquest.com/lib/hull/detail.action?docID=5892320> (visited on 01/15/2024).

Srivastava, Nitish et al. (2014). "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In.