



# **L7 - Artificial Intelligence (ACCA7026)**

## **MNIST Dataset Classification**

Student: Austin Hooper (1700481)

MSc Software Engineering and Artificial Intelligence

**Wales Institute for Science and Art (WISA)**  
**University of Wales Trinity Saint David**

Lecturer: Dr Gordon Dickers

25<sup>th</sup> March 2022

# Table of Contents

|                                                                      |    |
|----------------------------------------------------------------------|----|
| 1. Introduction .....                                                | 1  |
| 2. Literature Review .....                                           | 2  |
| 3. Dataset Context, Analysis and Pre-processing .....                | 4  |
| 3.1. Imbalanced Classes .....                                        | 4  |
| 3.1.1. Class Weights .....                                           | 4  |
| 3.1.2. Class Resampling .....                                        | 6  |
| 3.2. Encoding and Normalisation .....                                | 6  |
| 3.3. Dimensionality Reduction via Principal Component Analysis ..... | 7  |
| 4. Classification Algorithms and Evaluation Methodology .....        | 8  |
| 4.1. Decision Trees and Random Forests .....                         | 9  |
| 4.2. Support Vector Machines .....                                   | 12 |
| 4.3. Convolutional Neural Networks .....                             | 14 |
| 5. Results and Discussion .....                                      | 16 |
| 5.1. Model Classification Capability .....                           | 16 |
| 5.2. Model Train and Inference Time .....                            | 18 |
| 6. Conclusion and Future Work .....                                  | 20 |
| Appendix – Python Code .....                                         | 25 |

## List of Figures

|                                                                           |    |
|---------------------------------------------------------------------------|----|
| Figure 1: MNIST samples [1] .....                                         | 4  |
| Figure 2: Class balance graph (train and test combined).....              | 5  |
| Figure 3: Class weights generation.....                                   | 5  |
| Figure 4: One-hot encoding .....                                          | 6  |
| Figure 5: Image normalisation .....                                       | 6  |
| Figure 6: Principal Component Analysis .....                              | 7  |
| Figure 7: DT confusion matrix.....                                        | 9  |
| Figure 8: DT ROC curve .....                                              | 10 |
| Figure 9: RF confusion matrix.....                                        | 11 |
| Figure 10: RF ROC curve .....                                             | 11 |
| Figure 11: SVM confusion matrix.....                                      | 13 |
| Figure 12: SVM ROC curve .....                                            | 13 |
| Figure 13: CNN (AlexNet) accuracy per epoch.....                          | 15 |
| Figure 14: CNN (AlexNet) confusion matrix.....                            | 15 |
| Figure 15: Classifier performance comparison bar chart .....              | 17 |
| Figure 16: Model train and inference time vs. accuracy scatter plot ..... | 19 |

## List of Tables

|                                                           |    |
|-----------------------------------------------------------|----|
| Table 1: Class balance .....                              | 5  |
| Table 2: Classifier performance comparison.....           | 16 |
| Table 3: Training environment configuration.....          | 18 |
| Table 4: Model train and inference time vs. accuracy..... | 18 |

## Abstract

Computer Vision is an increasingly expanding and important field. This report explores the MNIST dataset of hand written digits, with a focus on classification. The dataset was loaded into a Python environment. Various libraries were utilised, facilitating data examination, augmentation, dimensionality reduction, and classification. Decision Trees, Random Forests, Support Vector Machines, and various Convolutional Neural Network architectures were applied to the dataset. These classifiers were evaluated using a wide range of methods, primarily confusion matrices, test accuracy, and Receiver Operating Characteristic analysis. Support Vector Machines performed the best of the traditional methods, achieving 98% accuracy in 39 seconds. Random Forests achieved the best time efficiency, achieving 95% accuracy in only 9 seconds. AlexNet was the best performing Convolutional Neural Network tested, achieving 99.3% accuracy in 751 seconds. Traditional Machine Learning methods are capable of achieving high accuracies on this dataset due to its simplicity. Neural Networks achieve only marginally higher accuracy, at a high cost to training and inference time.

# 1. Introduction

This report focuses on Machine Learning (ML) for Computer Vision (CV), specifically of hand-written Arabic numerals. The Modified National Institute of Standards and Technology (MNIST) dataset [1] was specified as the data source. This report conducts a review of literature, details analysis and pre-processing of data, and evaluates four classification algorithms. The four chosen algorithms are Random Forests (RFs) including Decision Trees (DTs), Support Vector Machines (SVMs), and Convolutional Neural Networks (CNN). Models' classification capability and run time are evaluated and compared. ML techniques were implemented in Python, primarily with the Scikit-Learn [2] library.

## 2. Literature Review

As described by IBM, CV enables computers to see, observe, and understand [3]. As identified by Norvig and Russell, CV was founded in the 1960s, primarily by Roberts' 1963 Doctoral thesis [4]. This paper introduced model-based matching and edge detection, which was further developed by Canny in 1986 [5].

CNNs are the foundation of most CV models. CNNs are a type of Artificial Neural Network (ANN). ANNs were introduced in 1943, based on the McCulloch-Pitts neuron, proposed by McCulloch and Pitts in their landmark paper [6]. This neuron is a basic model of a biological neuron in a brain. Rosenblatt's 1958 paper [7] introduced the Rosenblatt Perceptron, a more advanced model featuring thresholds, inhibitory and excitatory inputs, and weighted connections. CNNs were popularised by LeCun et al. [8], employing them to recognise handwritten digits. LeCun et al. later proposed the LeNet-5 CNN in 1998 [1], along with the MNIST dataset. LeNet-5 achieved a test error rate of 0.95% on this dataset.

Deep Neural Networks (DNNs) are another model type. They have many hidden layers; the phrase "deep learning" was popularised by Hinton et al. in 2006 [9], also investigating MNIST. DNNs are particularly effective when trained on large datasets. This is facilitated through training on Graphics Processing Units (GPUs) as opposed to Central Processing Units (CPUs), due to the highly parallelised nature of GPU architectures. Hinton, together with Rumelhart and Williams, introduced backpropagation [10], [11], an algorithm using gradient descent to automatically compute gradients. As Géron summarises, backpropagation finds optimal connection weights and biases to reduce error [12, pp. 289–291].

In 2007, Lauer, Suen, and Bloch achieved a test error rate of 0.54% on the MNIST dataset [13]. They achieved this through developing a Trainable Feature Extractor (TFE) based on LeNet-5, connected to binary SVM to aid generalisability. This was aided by elastic distortion, an image transformation method used for data augmentation, introduced by Simard et al. in 2003 [14]. The authors noted that replacing LeNet-5 outputs with SVMs considerably increased complexity and training time.

Cireşan, Meier, and Schmidhuber implemented a multi-column DNN in 2012 [15], based on Fukushima's "Neocognitron" neural network [16]. This model achieved 0.23% test error rate, approaching the ~0.20% human error rate [17]. This work also highlights managing training time by down sampling data using max pooling technique. This performance was accredited partly to six additional datasets generated by normalising digit widths.

In 2019, Alvear-Sandoval et al. achieved a test error rate of 0.14% using a combination of Stacked Denoising Auto-Encoding (SDAE) classifiers and an ensemble of CNNs [18]. They found CNNs to be superior to SDAE classifiers, though combining both improved performance. Although record performance, the authors noted that this performance increase over previous best models (~0.1% reduction in error rate) cost an order of magnitude of computational load. This work did not generate as much augmented data as Cireşan's approach, thus, combining both approaches could be beneficial. Nonetheless, there is little room to improve, as the dataset is "too easy," as described by LeCun [17].

The Digit Dataset (DIDA) and DIGITNET classifier were introduced by Kusetogullari et al. in 2021 [19]. It contains more samples than MNIST, 250,000 versus 70,000. Images are RGB format, at 5184×3456 resolution. Images are of documents containing many digits, such as dates. DIGITNET achieved 76% accuracy on the DIDA dataset, in contrast to the ease of classifying the MNIST dataset. This is likely due to the inclusion of many digits per image.

The Brown University Stylus Handwriting (BRUSH) dataset was created by Kotani et al. in 2020 [20], containing images of English words. It is beneficial over MNIST as it contains 27,649 samples provided by 170 individuals, varying the writing style. Also detailed was the Decoupled Style Descriptor (DSD) model, which achieved accuracy of 99% from 50-word samples. It was also applied to generate new handwriting. Authors noted the model does not factor character-independent writer style.

The IAM dataset, created by Marti and Bunke [21], consists of 13,353 images of handwritten English sentences. Sourced from 657 writers, data totalled 115,320 words. Images are A4, scanned at 300 DPI. Though there are few samples compared to MNIST, each sample contains many individual characters. This paper did not propose a classification model, instead focusing highly on dataset analysis.

Baruah and Hazarika created the Tezpur University Assamese language dataset in 2014 [22]. It contains 10 numeric and 52 alphabetic characters across 8,235 images, from 45 writers. These characters are more complex than English characters. Each image contains a single character, at 4,392×4,868 resolution – far higher than MNIST. No new model was proposed, instead, pre-processing and an SVM classifier were applied, achieving 78% accuracy using a polynomial kernel. The primary limitation is it did not apply more capable classifiers.

Clanuwat et al. introduced two datasets in 2018: Kuzushiji-MNIST and Kuzushiji-49. Kuzushiji-MNIST is a replacement for MNIST, while Kuzushiji-49 is more complex. Kuzushiji-49 consists of Japanese cursive writing with 49 classes across 266,407 images, at 28x28 resolution. This dataset is imbalanced, increasing classification difficulty while providing opportunities for experimentation. Five existing models were applied to the data, making comparisons to MNIST. A 4-Nearest Neighbour model achieved 97% accuracy on MNIST, but only 86% on Kuzushiji-49, while more advanced models like CNNs closed the gap significantly to only 3%.

### 3. Dataset Context, Analysis and Pre-processing

The MNIST dataset was compiled by LeCun et al. in 1998 [1], consisting of 60,000 train images and 10,000 test images of handwritten digits. Images are 28×28 pixels resolution, greyscale, and are anti-aliased, as shown below in Figure 1.



Figure 1: MNIST samples [1]

The data was sourced from LeCun’s website [23] in IDX format, then converted into NumPy arrays. Image samples were shuffled prior to training, to ensure they were in a random order. This did not impact model accuracy, suggesting they were already shuffled.

#### 3.1. Imbalanced Classes

The images’ class balance is detailed in Table 1, visualised in Figure 2. There is a minor variance in samples per class, with a minimum of 6,318 and maximum of 7,878.

##### 3.1.1. Class Weights

Manually generated class weights input to models had no impact on performance (see Figure 3). Thus, resampling was used (see Section 3.1.2). This is likely because there is not a significant variance in class size.

Table 1: Class balance

| Class (Digit) | No. Train     | No. Test      | Total Per Class |
|---------------|---------------|---------------|-----------------|
| 0             | 5,923         | 980           | 6,903           |
| 1             | 6,742         | 1135          | 7,878           |
| 2             | 5,958         | 1032          | 6,992           |
| 3             | 6,131         | 1010          | 7,144           |
| 4             | 5,842         | 982           | 6,828           |
| 5             | 5,421         | 892           | 6,318           |
| 6             | 5,918         | 958           | 6,882           |
| 7             | 6,265         | 1028          | 7,300           |
| 8             | 5,851         | 974           | 6,833           |
| 9             | 5,949         | 1009          | 6,967           |
| <b>Totals</b> | <b>60,000</b> | <b>10,000</b> | <b>70,000</b>   |

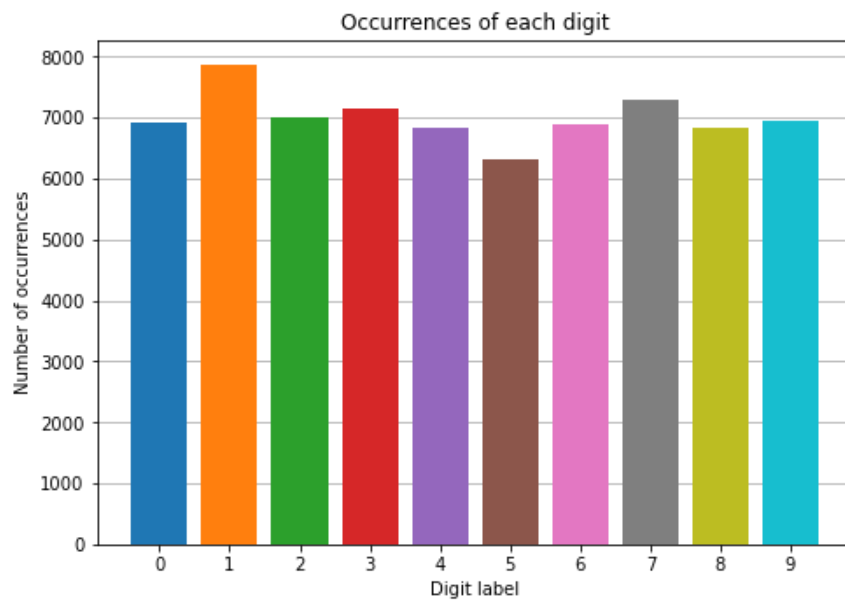


Figure 2: Class balance graph (train and test combined)

```

1 unique, class_counts_train = np.unique(y_train, return_counts=True) # Count of class occurrences
2 class_weights_dict = dict(zip(unique, class_counts_train)) # Store as dict in format {0: 1.234, 1: 1.234 ...}
3
4 for key in class_weights_dict:
5     # Number of this class divided by total number of items
6     class_weights_dict[key] = class_weights_dict[key] / len(y_train)
7     print("Class", key, "has weight of", round(class_weights_dict[key], 4))

```

```

Class 0 has weight of 0.0987
Class 1 has weight of 0.1124
Class 2 has weight of 0.0993
Class 3 has weight of 0.1022
Class 4 has weight of 0.0974
Class 5 has weight of 0.0904
Class 6 has weight of 0.0986
Class 7 has weight of 0.1044
Class 8 has weight of 0.0975
Class 9 has weight of 0.0992

```

Figure 3: Class weights generation

### 3.1.2. Class Resampling

Class resampling alters class distribution [24], often to balance class sample count. The Imbalanced-Learn [25] library facilitated implementing oversampling, undersampling, Synthetic Minority Oversampling Technique (SMOTE) [26], and Adaptive Synthetic (ADASYN) oversampling [27]. Oversampling randomly duplicates minority class samples, while undersampling deletes majority class samples [24], [28]. SMOTE generates new samples from existing ones [26], while ADASYN uses a weighed distribution for minority class images based on their level of difficulty to learn [27]. Of these methods, only SMOTE improved model accuracy, by 1-2% in some models.

### 3.2. Encoding and Normalisation

This dataset required little data cleaning or manipulation. One-hot encoding converts categorical labels to a list format [29], [30], shown in Figure 4. One-hot encoding increased training time by ~80% in some classifiers, and decreased classification performance in ensemble classifiers, while being incompatible with some classifiers, though necessary for CNNs. Normalisation was performed to convert greyscale pixel values from the 0-255 range to the 0-1 range, shown in Figure 5.

```
1 print("Before one-hot encoding, first y_train item looks like this:\t", y_train[0])
2
3 y_train = to_categorical(np.asarray(y_train))
4 y_test = to_categorical(np.asarray(y_test))
5
6 print("\nAfter one-hot encoding, first y_train item looks like this:\t", y_train[0])
```

Before one-hot encoding, first y\_train item looks like this: 5

After one-hot encoding, first y\_train item looks like this: [0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]

Figure 4: One-hot encoding

```
1 print("Pixel range before normalising:", np.amin(X_train), "-", np.amax(X_train))
2
3 # Divide every greyscale level by 255 - max pixel value
4 X_train = X_train / 255
5 X_test = X_test / 255
6
7 print("Pixel range after normalising: ", np.amin(X_train), "-", np.amax(X_train))
```

Pixel range before normalising: 0.0 - 255.0  
Pixel range after normalising: 0.0 - 1.0

Figure 5: Image normalisation

### 3.3. Dimensionality Reduction via Principal Component Analysis

As Géron explains, dimensionality reduction involves reducing feature count (dimensions) in a dataset, to facilitate data visualisation or speed up training [12, p. 213]. This report explores Principal Component Analysis (PCA) for dimensionality reduction. PCA reduces dimensions while retaining as much statistical variance (useful information) as possible, by identifying patterns [12, pp. 213–214], [31].

PCA with a target of 95% variance retention reduced dimensionality from 60,000×784 to 60,000×154, shown in Figure 6. This dimensionality decrease reduced accuracy by only ~0.3% across all models. Unexpectedly, PCA-computed data did not benefit all algorithms. This could be due to the models' expected data formats.

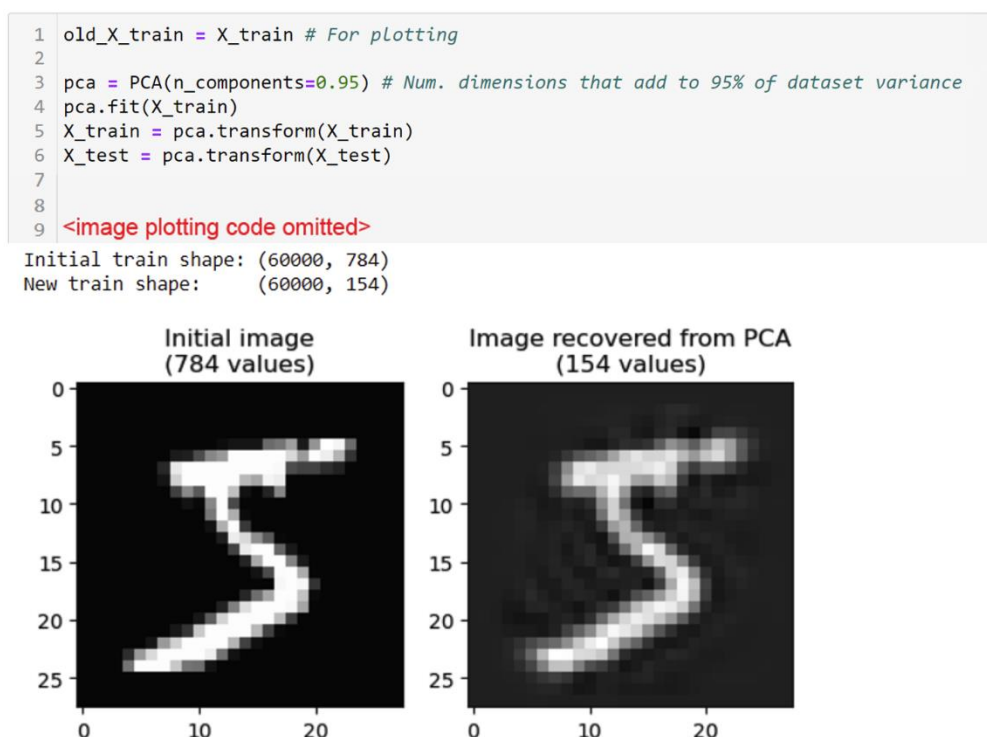


Figure 6: Principal Component Analysis

## 4. Classification Algorithms and Evaluation Methodology

This section explores a range of classifiers. Hyperparameter tuning is explored and evaluated in relation to classifier capability and train time performance. A range of classifiers were chosen to illustrate various approaches to classification. Python code is included in the Appendix.

A wide range of metrics were used to thoroughly evaluate classifiers. Confusion matrices are used to evaluate classifiers, by plotting the number of true and false positive predictions per class [32], [33]. Other metrics, derived from the confusion matrix, are:

- Recall, the true positive rate [33]
- Precision, the proportion of predicted positive cases which were correct [33], [34]
- F<sub>1</sub> score, the harmonic mean of precision and recall [12, p. 92]
- Accuracy, the ratio between the number of correctly classified samples and the total sample count [32]

Receiver Operating Characteristic (ROC) curves and Area Under the Curve (AUC) measure classifier performance. ROC curves visualise the ability of binary classifiers to differentiate classes [33], [35], from which the AUC is calculated. Predicting probabilities of an input being a member of a given class [12, pp. 98–100], [33] produces unrealistically high AUC values in this context – accuracies of ~85% translated to AUC scores of above 99.999%. Predicting discrete class labels by wrapping the classifier in Scikit-Learn's `OneVsRestClassifier` provides more representative results. As a result, each ROC curve has only one point.

Hyperparameters were adjusted manually based on literature recommendations, followed by applying Grid Search (GS). GS exhaustively searches a grid of hyperparameters to find the optimal combination [36]–[38]. GS did not find better hyperparameters than those manually selected.

## 4.1. Decision Trees and Random Forests

Decision Trees (DTs) derive a set of decisions from input data, generating a tree structure containing decisions [39], [40]. This tree is usually binary. DTs are supervised, in that they train using labelled data [40].

The default DT produced accuracy, precision, recall and F1 scores of 0.83. Pruning the tree by limiting the maximum number of features and leaf nodes parameters provided no improvement. Limiting the maximum depth to 14 improved all metrics by only 1%, while halving time taken to train. Limits to the minimum number of samples required to split a node, and minimum samples required to be at a leaf node did not improve the model. Low values had no impact, while high values inhibited learning. Lack of improvement from pruning suggests that the DT was not overfitting. Changing the function measuring tree split quality had no impact, neither did changing the splitter option from “best” to “random.”

Final DT performance was 84% on all classification report metrics. A confusion matrix of classification predictions is provided in Figure 7. A ROC curve is shown in Figure 8, visualising prediction performance per digit. The digit 1 is evidently more accurately classified, likely due to its simple nature, where less capable models such as this one experience difficulty identifying more complex digits.

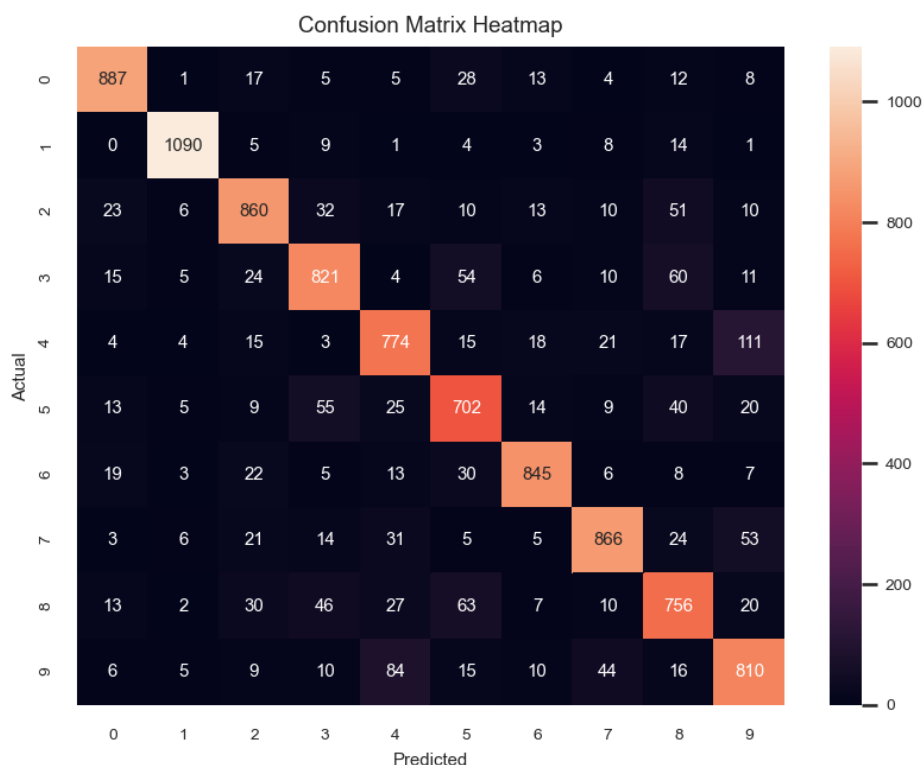


Figure 7: DT confusion matrix

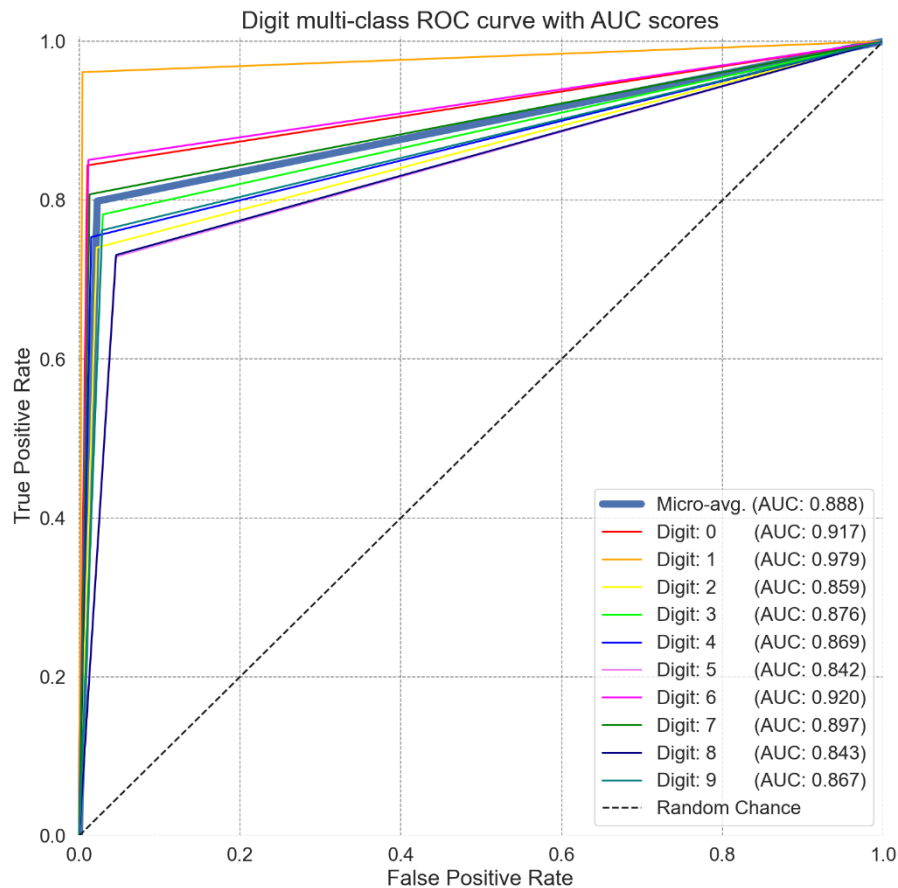


Figure 8: DT ROC curve

Random Forests (RFs) were proposed by Tin Kam Ho in 1995 [41]. They are an ensemble classifier, as they output the mode result of many DTs [12, pp. 175–179], [40], [42]. RFs tend to overfit [40], [43], though this issue did not arise during testing.

Unexpectedly, RFs benefited little from modifications to the default model, as with DTs. Accuracy, precision, recall and F1 scores were all 0.94 using the default model. Any limitation to tree depth, maximum number of leaf nodes or maximum number of features, below initial values reduced performance. As with DTs, imposing minimum samples per leaf and per split were detrimental. Choice of function measuring tree split quality, along with choice of node split method, had no impact on performance. Increasing the number of DTs in the forest from the default 100 to 170 increased accuracy by 1% across all metrics. This increased training time by approximately 70%, which corresponds to the 70% increase in DTs.

A confusion matrix for the RF is shown in Figure 9, followed by its ROC curve in Figure 10. Notably, the ROC plot shows the gap in classification accuracy of each digit significantly decreased from the DT.

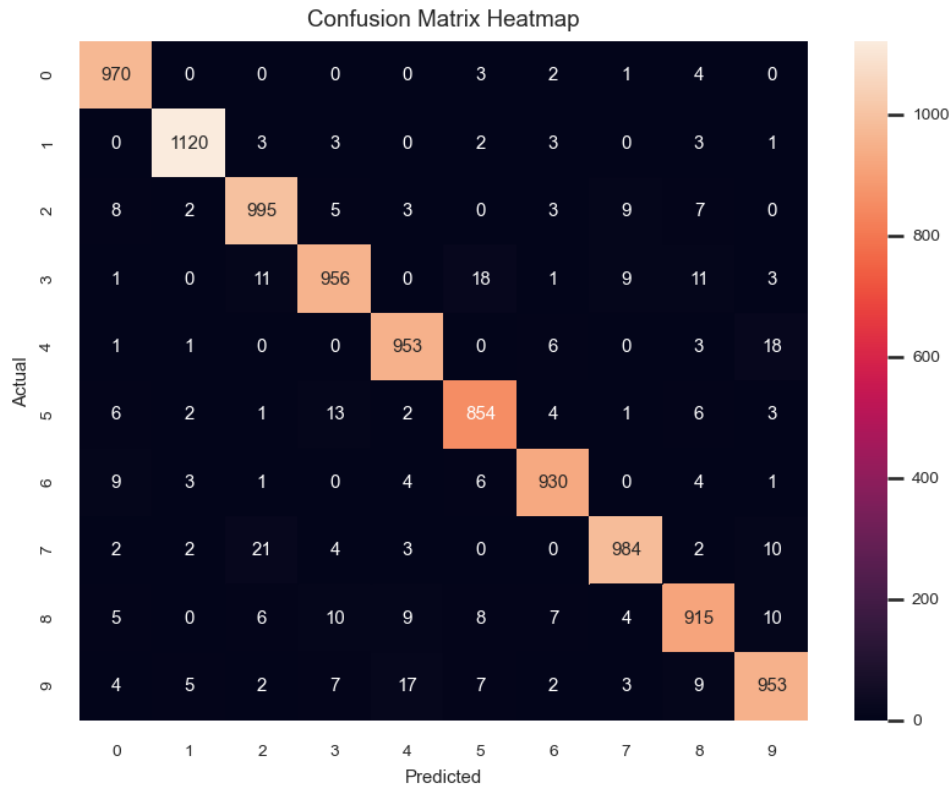


Figure 9: RF confusion matrix

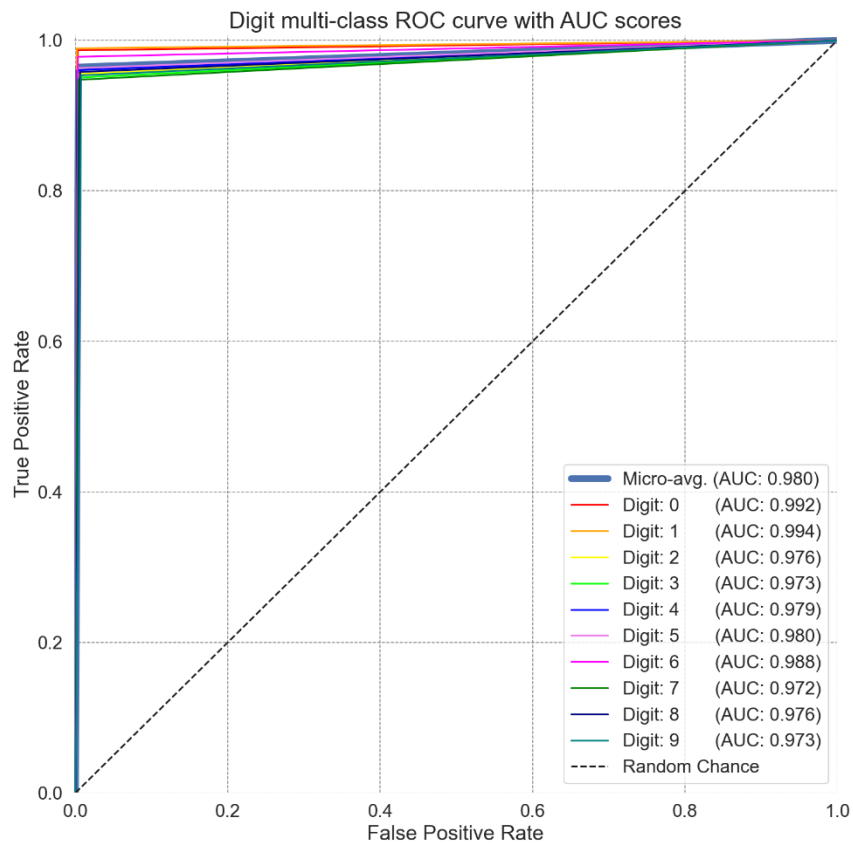


Figure 10: RF ROC curve

## 4.2. Support Vector Machines

Support Vector Machines (SVMs) try to fit the widest gap (“street”) between classes, known as large margin classification. Instances at the edge of each class cluster, known as support vectors, define the width of the street [12, p. 153].

The default SVM scored 0.98 on all classification report metrics. Similarly to previous models, no hyperparameter adjustments improved classification capability. Kernel option changes, being sigmoid, linear, and polynomial, decreased performance over the default Radial Basis Function (RBF) kernel, even when the polynomial kernel’s degree parameter was adjusted. Enabling additional probability estimates provided no benefits. For the RBF kernel, decreasing the tolerance threshold for stopping criterion, from the 0.001 default, had no impact, despite the findings of [22]. Increasing it provided no classification improvement, though a large value of 2 halved train time. Similarly, increasing the C parameter from the default 1.0 had no impact, while decreasing it reduced performance. This is a regularisation parameter, for reducing likelihood of overfitting – regularisation strength is inversely proportional to C [44]. Impact of the gamma parameter, being the coefficient used for RBF, polynomial, and sigmoid kernels [44], varied per kernel. No kernel and gamma combination improved accuracy over the default value of  $1/\text{feature count}$ . Changing the decision function from one versus rest to one versus one had no impact, likely as the model was already performing so well.

Limiting the maximum number of iterations decreased classification report metrics up to 250 iterations. Past this point, classification report metrics were indistinguishable from those of the model without a limit, though ROC and AUC were noticeably reduced. A limit of 350 provided near-identical AUC scores to those of the model without a limit. This reduced train time from the non-limited model by ~35%.

The SVM’s confusion matrix is shown in Figure 11, and its ROC curve in Figure 12. Both figures illustrate the SVM’s similar performance to the RF model. The SVM was overall slightly better than the RF, but far slower to train due to its single-thread limitation.

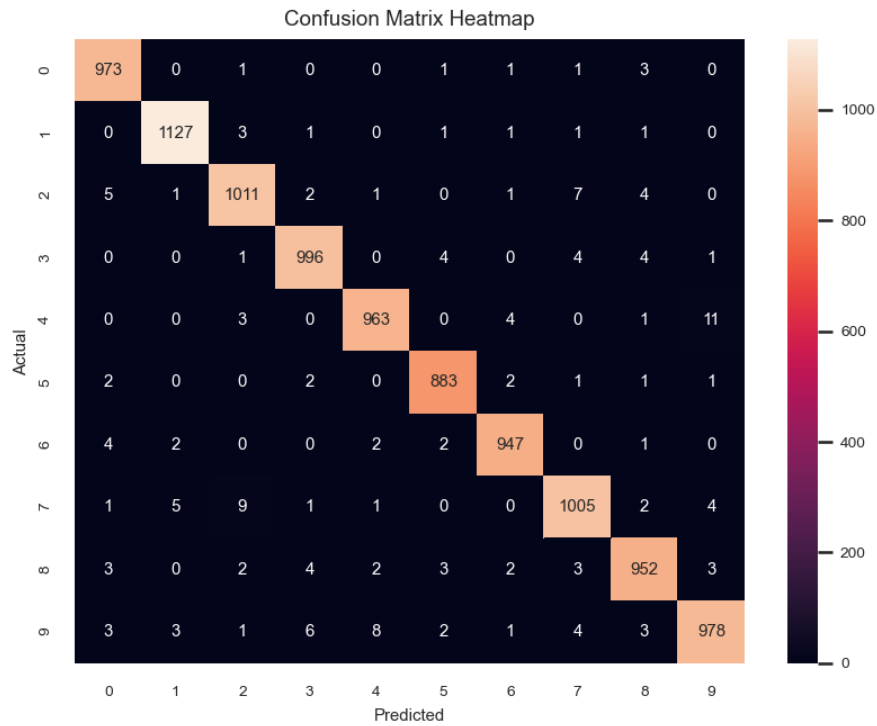


Figure 11: SVM confusion matrix

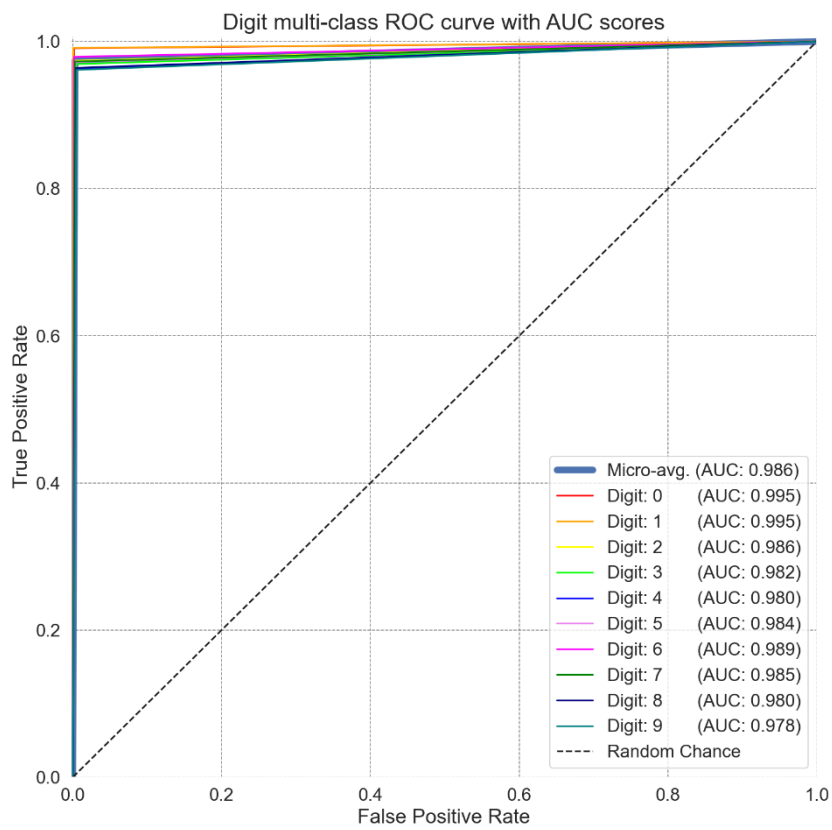


Figure 12: SVM ROC curve

### 4.3. Convolutional Neural Networks

For CNNs, three prominent models were applied. These were LeNet-5 [1], AlexNet [45], and ResNet-34 [46]. The stochastic gradient descent optimiser was used with a learning rate of 0.1 and loss of categorical cross entropy for all models. Experimentation showed this to be reasonably fast with good performance across each model. An early stopping call-back was used to end training early when validation accuracy plateaus. Resampling was found to decrease accuracy by ~0.5% for CNNs, thus was not used for these experiments. The reason for this is unknown.

The Keras LeNet-5 implementation achieved 99.0% accuracy, as reported in its paper, with slight variance likely due to setup variance. Inclusion of dropout regularisation layers, with values of 0.1, 0.2, 0.3, 0.4, and 0.5, had no impact on performance.

AlexNet achieved a base performance of 99.1% accuracy, though taking considerably longer due to its increased complexity over LeNet-5. Reducing model complexity by removing both 384-filter convolutional layers allowed the model to reach 99.3% accuracy. Inclusion of dropout layers did not benefit this model.

ResNet-34 achieved 98.8% accuracy prior to adjustments. Simplifying the model by removing the 256 and 512 filter count convolutions provided a small accuracy increase to 99.1%. As with previous models, no dropout values improved performance. This may be because validation accuracy was already within 1% of training accuracy, which reached 100%.

LeNet-5 performed best with the hyperbolic tangent activation functions, while both other models were best using Rectified Linear Unit (RELU). Figure 13 shows training and validation accuracy per epoch for AlexNet, the best performing CNN tested. Note the magnified scale of the Y axis due to the high performance. The graph highlights the very gradual increase in validation accuracy. A confusion matrix is shown in Figure 14, demonstrating that many actual to predicted class combinations had zero misclassifications. At this performance, a ROC curve's graphic is not useful as all curve areas approach 1.0.

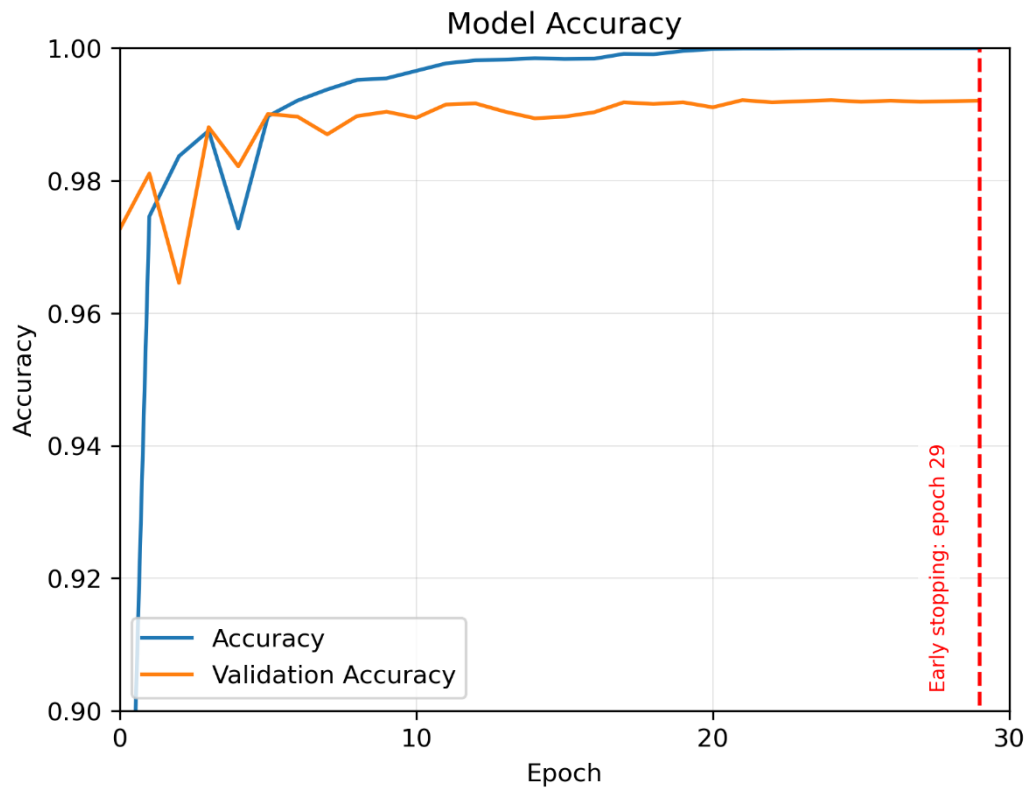


Figure 13: CNN (AlexNet) accuracy per epoch

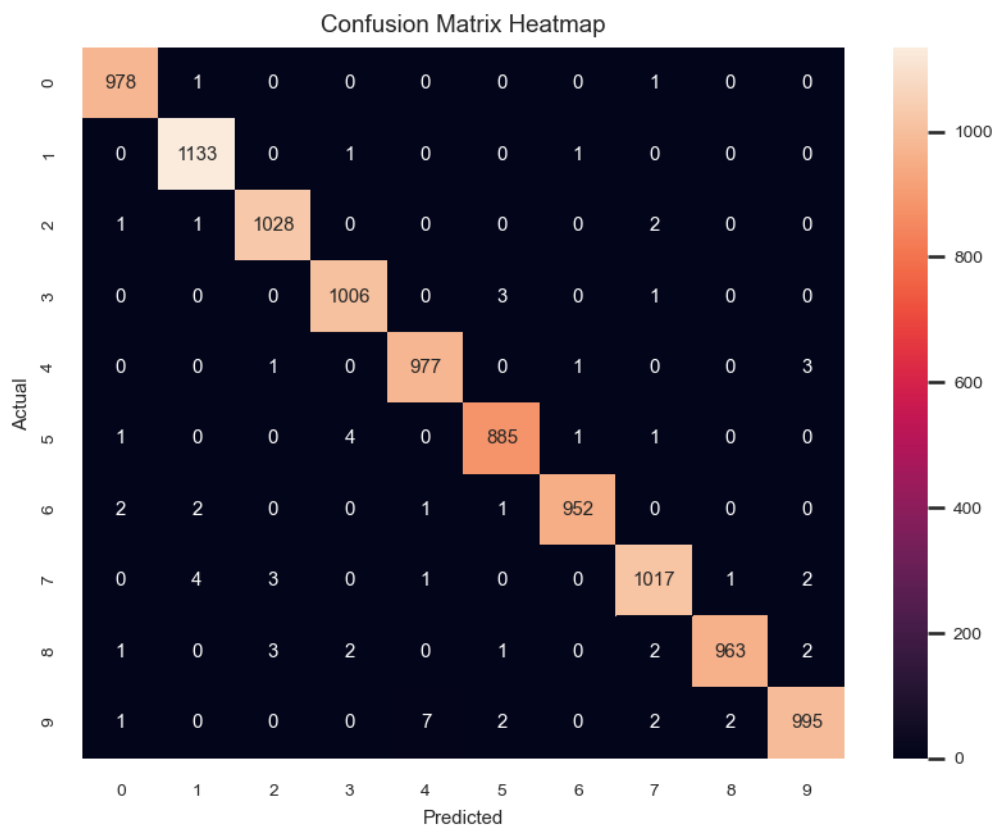


Figure 14: CNN (AlexNet) confusion matrix

## 5. Results and Discussion

This section further evaluates and compares the classifiers from the previous section. Of primary concern is classifiers' accuracy and the time in which they took to achieve this.

### 5.1. Model Classification Capability

Model accuracy ranged from 84% to >99%, detailed in Table 2 and visualised in Figure 15. The magnified Y axis highlights small performance differences. CNNs performed the best, by a very small margin. The dataset is likely not complex enough to differentiate more capable classifiers [17]. More complex images would probably increase the gap between CNNs and traditional ML models. This work demonstrates that RFs and SVMs are very capable on this simple dataset.

Table 2: Classifier performance comparison

| <b>Classifier</b>    | Decision Tree<br>( <i>max_depth=14, rest default</i> ) | Random Forest<br>( <i>n_jobs=-1, n_estimators=170, rest default</i> ) | Support Vector Machine<br>( <i>max_iter=350, tol=2, rest default</i> ) | Convolutional Neural Network<br>(adjusted AlexNet) |
|----------------------|--------------------------------------------------------|-----------------------------------------------------------------------|------------------------------------------------------------------------|----------------------------------------------------|
| <b>Accuracy</b>      | 0.841                                                  | 0.950                                                                 | 0.981                                                                  | 0.993                                              |
| <b>Precision</b>     | 0.840                                                  | 0.950                                                                 | 0.980                                                                  | 0.990                                              |
| <b>Recall</b>        | 0.840                                                  | 0.950                                                                 | 0.980                                                                  | 0.990                                              |
| <b>F1</b>            | 0.840                                                  | 0.950                                                                 | 0.980                                                                  | 0.990                                              |
| <b>AUC Min.</b>      | 0.842                                                  | 0.972                                                                 | 0.978                                                                  | 0.992                                              |
| <b>AUC Max.</b>      | 0.979                                                  | 0.994                                                                 | 0.995                                                                  | 0.998                                              |
| <b>AUC Micro-avg</b> | 0.888                                                  | 0.980                                                                 | 0.986                                                                  | 0.996                                              |

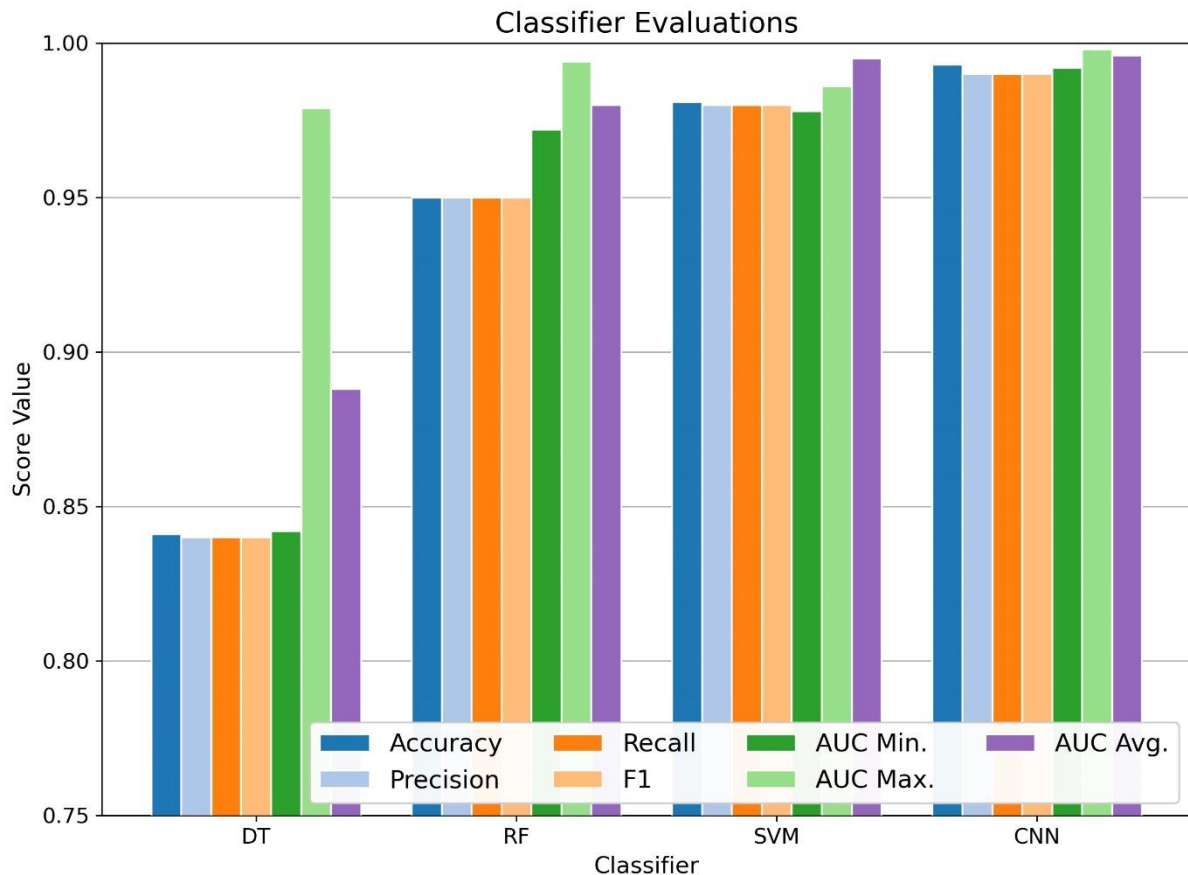


Figure 15: Classifier performance comparison bar chart

Few changes to the Scikit-Learn classifiers' hyperparameters impacted performance. Less advanced classifiers' default states probably matched their tuned versions because they were fundamentally limited for this dataset. RFs and SVMs experienced little benefit from modifications, likely as their performance was already so high, reaching their limit. Hyperparameter tuning primarily benefited training time with negligible impact on accuracy.

Though AlexNet achieved 99.3% accuracy, this appears to only be applicable to this dataset. On a test of 30 of the author's handwritten digits, processed similarly to the MNIST dataset, accuracy was ~60%. This result is not comparable to the MNIST results, though demonstrates a drawback of this classifier and dataset. Other datasets (see literature review) may partially alleviate this problem due to their range of writing styles.

As highlighted in Figure 15, per-digit classification capability varied considerably. Less capable models such as DTs exhibited high variance per digit. More competent models reduced this variance, especially CNNs. The 1 digit was classified best, likely due to its simple, unique nature.

## 5.2. Model Train and Inference Time

Each algorithm's combined training and inference time was recorded using the Python `timeit` library [47], and validated using a hardware timer. This enabled evaluating models by time efficiency along with raw performance. The mean of 10 times was used per model – limited by high train times. Models were trained using the CPU (except CNNs), enabling multi-threading where supported. Relevant hardware and software environment information is provided in Table 3. Timing was conducted with no background applications running.

Table 3: Training environment configuration

| Component       | Specification / Details                                 |
|-----------------|---------------------------------------------------------|
| <b>Hardware</b> |                                                         |
| Processor       | AMD Ryzen 9 3900X: 12 core, 24 thread, 3.8GHz-4.6GHz    |
| Motherboard     | ASUS Prime X570-P, AMD AM4 platform                     |
| System Memory   | 32GB Corsair 3200MHz CL16                               |
| Storage         | ADATA XPG SX8200 Pro 1TB 3,500/3,000MB/s R/W            |
| Graphics        | NVIDIA GeForce GTX 1060 6GB                             |
| <b>Software</b> |                                                         |
| Windows         | Windows 10 Pro build 19043.1526. High perf. power plan. |
| Python          | Python 3.7.9, Conda 4.11.0, IPython 7.31.1              |
| Scikit-Learn    | sklearn 0.23.2. Training using CPU.                     |

Table 4 highlights time versus accuracy for each best model configuration. A single DT took longer to run than a RF, probably due to the difference in parallelisation of the two models. Further details are shown in Figure 16. CNN models ran on the GPU, and so are not strictly comparable to traditional models, though CNNs would take longer on the CPU, which is not viable.

Table 4: Model train and inference time vs. accuracy

| Best Model Configuration                                            | Mean Train + Inference Time (s) | Accuracy (%) | Normalised Acc./Time Ratio (higher is better) |
|---------------------------------------------------------------------|---------------------------------|--------------|-----------------------------------------------|
| Decision Tree ( <i>max_depth=14, rest default</i> )                 | 15.825                          | 84.1         | 0.545                                         |
| Random Forest ( <i>n_jobs=-1, n_estimators=170, rest default</i> )  | 9.745                           | 95.0         | 1.000                                         |
| Support Vector Machine ( <i>max_iter=350, tol=2, rest default</i> ) | 39.044                          | 98.1         | 0.258                                         |
| Convolutional Neural Network (adjusted AlexNet)                     | 751.344                         | 99.3         | 0.014                                         |

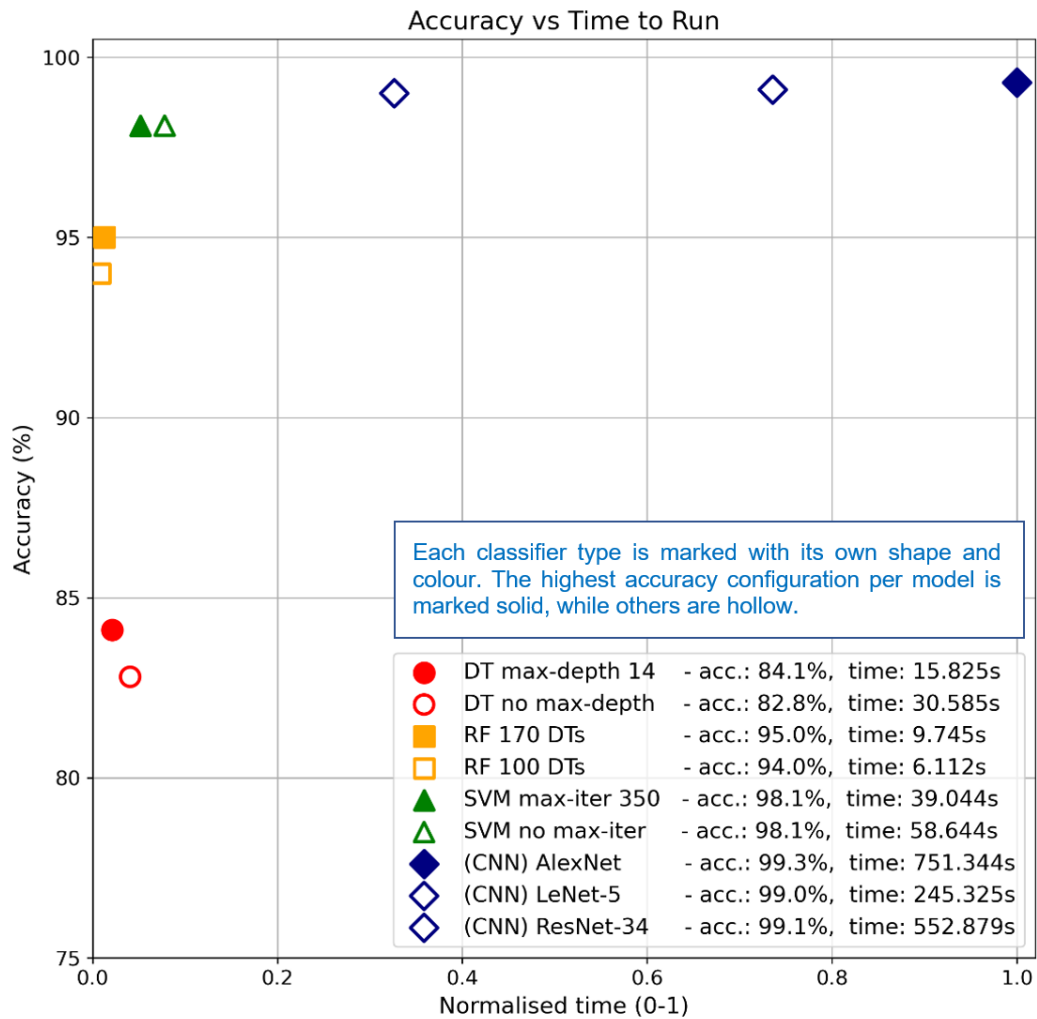


Figure 16: Model train and inference time vs. accuracy scatter plot

The scatter plot highlights the extreme range of classifier times, due to inclusion of CNNs. CNNs had little value – a SVM achieved 98.1% accuracy in 39 seconds, while AlexNet achieved 99.3% accuracy in 751 seconds. This is a 1,800% time increase for a 1.2% increase in accuracy. A linear time scale, as opposed to logarithmic, highlights this difference. For this dataset, simpler ML models are far more time efficient.

## 6. Conclusion and Future Work

This report has investigated data examination, augmentation, and classification of the MNIST dataset. PCA significantly decreased train time with minimal accuracy cost. DTs and RFs performed the worst, with SVM and CNN performance very close together. Final accuracy was 99.3% with a modified AlexNet CNN. Model run time was considered, demonstrating a minimal accuracy improvement at the cost of a large time increase.

Future work would primarily entail further model exploration, such as further depth of CNNs, stochastic gradient descent classifiers, and histogram of oriented gradients. The developed models could also be applied to different handwriting styles, to test their generalisability – this is beyond the scope of this report. CNN run time could be evaluated more thoroughly, as this was time-constrained by GPU performance.

## References

- [1] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.
- [2] scikit-learn, "scikit-learn: machine learning in Python — scikit-learn 0.23.2 documentation." <https://scikit-learn.org/stable/> (accessed Nov. 10, 2020).
- [3] IBM, "What is Computer Vision? | IBM." <https://www.ibm.com/uk-en/topics/computer-vision> (accessed Mar. 07, 2022).
- [4] L. Roberts, "Machine perception of three-dimensional solids," Massachusetts, 1963.
- [5] J. Canny, "A Computational Approach to Edge Detection," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-8, no. 6, pp. 679–698, 1986, doi: 10.1109/TPAMI.1986.4767851.
- [6] W. S. McCulloch and W. Pitts, "A logical calculus of the ideas immanent in nervous activity," *The bulletin of mathematical biophysics*, vol. 5, no. 4, pp. 115–133, 1943, doi: 10.1007/BF02478259.
- [7] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958, doi: 10.1037/h0042519.
- [8] Y. LeCun *et al.*, "Handwritten digit recognition: applications of neural network chips and automatic learning," *IEEE Communications Magazine*, vol. 27, no. 11, pp. 41–46, Nov. 1989, doi: 10.1109/35.41400.
- [9] G. Hinton, S. Osindero, and Y.-W. Teh, "A Fast Learning Algorithm for Deep Belief Nets," *Neural computation*, vol. 18, pp. 1527–1554, 2006, doi: 10.1162/neco.2006.18.7.1527.
- [10] D. E. Rumelhart and J. L. McClelland, "Learning Internal Representations by Error Propagation," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition: Foundations*, 1987, pp. 318–362.
- [11] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, 1986, doi: 10.1038/323533a0.
- [12] A. Géron, *Hands-on machine learning with Scikit-Learn, Keras and TensorFlow: concepts, tools, and techniques to build intelligent systems*. Sebastopol, CA: O'Reilly, 2019. [Online]. Available: <https://www.oreilly.com/library/view/hands-on-machine-learning/9781492032632/>
- [13] F. Lauer, C. Y. Suen, and G. Bloch, "A trainable feature extractor for handwritten digit recognition," *Pattern Recognition*, vol. 40, no. 6, pp. 1816–1824, 2007, doi: 10.1016/j.patcog.2006.10.011.
- [14] P. Y. Simard, D. Steinkraus, and J. C. Platt, "Best practices for convolutional neural networks applied to visual document analysis," in *Seventh International*

- Conference on Document Analysis and Recognition, 2003. Proceedings.*, 2003, pp. 958–963. doi: 10.1109/ICDAR.2003.1227801.
- [15] D. Cireşan, U. Meier, and J. Schmidhuber, “Multi-column deep neural networks for image classification,” in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, 2012, pp. 3642–3649.
  - [16] K. Fukushima, “Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position,” *Biological Cybernetics*, vol. 36, no. 4, pp. 193–202, 1980, doi: 10.1007/BF00344251.
  - [17] Y. LeCun *et al.*, “Learning algorithms for classification: A comparison on handwritten digit recognition,” 1995.
  - [18] R. F. Alvear-Sandoval, J. L. Sancho-Gómez, and A. R. Figueiras-Vidal, “On improving CNNs performance: The case of MNIST,” *Information Fusion*, vol. 52, pp. 106–109, 2019, doi: <https://doi.org/10.1016/j.inffus.2018.12.005>.
  - [19] H. Kusetogullari, A. Yavariabdi, J. Hall, and N. Lavesson, “DIGITNET: A Deep Handwritten Digit Detection and Recognition Method Using a New Historical Handwritten Digit Dataset,” *Big Data Research*, vol. 23, p. 100182, 2021, doi: <https://doi.org/10.1016/j.bdr.2020.100182>.
  - [20] A. Kotani, S. Tellex, and J. Tompkin, “Generating Handwriting via Decouple Style Descriptors,” *ArXiv*, vol. abs/2008.1, 2020.
  - [21] U.-V. Marti and H. Bunke, “The IAM-database: an English sentence database for offline handwriting recognition,” *International Journal on Document Analysis and Recognition*, vol. 5, no. 1, pp. 39–46, 2002, doi: 10.1007/s100320200071.
  - [22] U. Baruah and S. M. Hazarika, “A Dataset of Online Handwritten Assamese Characters,” *J. Inf. Process. Syst.*, vol. 11, pp. 325–341, 2015.
  - [23] Y. LeCun, “MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges.” <http://yann.lecun.com/exdb/mnist/> (accessed Feb. 14, 2022).
  - [24] S. Pouyanfar *et al.*, “Dynamic Sampling in Convolutional Neural Networks for Imbalanced Data Classification,” in *2018 IEEE Conference on Multimedia Information Processing and Retrieval (MIPR)*, Apr. 2018, pp. 112–117. doi: 10.1109/MIPR.2018.00027.
  - [25] G. Lemaitre, F. Nogueira, D. Oliveira, and C. Aridas, “imbalanced-learn API — imbalanced-learn 0.5.0 documentation,” 2017. <https://imbalanced-learn.readthedocs.io/en/stable/api.html> (accessed Jan. 01, 2021).
  - [26] N. v Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, “SMOTE: Synthetic Minority over-Sampling Technique,” *J. Artif. Int. Res.*, vol. 16, no. 1, pp. 321–357, Jun. 2002.
  - [27] H. He, Y. Bai, E. A. Garcia, and S. Li, “ADASYN: Adaptive synthetic sampling approach for imbalanced learning,” in *2008 IEEE International Joint Conference on Neural Networks (IEEE World Congress on Computational Intelligence)*, 2008, pp. 1322–1328. doi: 10.1109/IJCNN.2008.4633969.
  - [28] K. E. Bennin, A. Tahir, S. G. MacDonell, and J. Börstler, “An empirical study on the effectiveness of data resampling approaches for cross-project software

- defect prediction,” *IET Software*, vol. 1, no. 15, 2021, doi: <https://doi.org/10.1049/sfw2.12052>.
- [29] A. C. H. Choong and N. K. Lee, “Evaluation of convolutionary neural networks modeling of DNA sequences using ordinal versus one-hot encoding method,” in *2017 International Conference on Computer and Drone Applications (IConDA)*, Nov. 2017, pp. 60–65. doi: 10.1109/ICONDA.2017.8270400.
  - [30] M. Zabidi, M. Mokji, I. Kamisian, and N. Ramli, “Implementing One Hot State Machines Using Schematics,” *Application Note AN6, Digital Electronics Laboratory*, vol. 2019, pp. 1–4, 2019.
  - [31] J. R. Beattie and F. W. L. Esmonde-White, “Exploration of Principal Component Analysis: Deriving Principal Component Analysis Visually Using Spectra,” *Applied Spectroscopy*, vol. 75, no. 4, pp. 361–375, 2021, doi: 10.1177/0003702820987847.
  - [32] D. Chicco and G. Jurman, “The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation,” *BMC Genomics*, vol. 21, no. 1, p. 6, 2020, doi: 10.1186/s12864-019-6413-7.
  - [33] T. Fawcett, “An introduction to ROC analysis,” *Pattern Recognition Letters*, vol. 27, no. 8, pp. 861–874, 2006, doi: 10.1016/j.patrec.2005.10.010.
  - [34] D. M. W. Powers, “Evaluation: from precision, recall and F-measure to ROC, informedness, markedness and correlation.” 2020.
  - [35] A. N. Kamarudin, T. Cox, and R. Kolamunnage-Dona, “Time-dependent ROC curve analysis in medical research: current methods and applications,” *BMC Medical Research Methodology*, vol. 17, no. 1, p. 53, 2017, doi: 10.1186/s12874-017-0332-6.
  - [36] scikit-learn, “sklearn.model\_selection.GridSearchCV — scikit-learn 1.0.2 documentation,” 2021. [https://scikit-learn.org/stable/modules/generated/sklearn.model\\_selection.GridSearchCV.html](https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html) (accessed Feb. 20, 2022).
  - [37] S. M. I. Shamsah and T. O. Owolabi, “Modeling the Maximum Magnetic Entropy Change of Doped Manganite Using a Grid Search-Based Extreme Learning Machine and Hybrid Gravitational Search-Based Support Vector Regression,” *Crystals*, vol. 10, no. 4, 2020, doi: 10.3390/cryst10040310.
  - [38] L. Yang and A. Shami, “On hyperparameter optimization of machine learning algorithms: Theory and practice,” *Neurocomputing*, vol. 415, pp. 295–316, 2020, doi: <https://doi.org/10.1016/j.neucom.2020.07.061>.
  - [39] D. Moses, T. Deepa, B; Patri, and M. Sowmya, “A Review of Decision Tree Algorithms for Predictive Analysis in Data Mining,” *i-Manager’s Journal on Software Engineering*, vol. 12, no. 1, pp. 38–46, 2018.
  - [40] T. Hillel, M. Bierlaire, M. Elshafie, and Y. Jin, “A systematic review of machine learning classification methodologies for modelling passenger mode choice,” *Journal of Choice Modelling*, no. August, 2020, doi: 10.1016/j.jocm.2020.100221.

- [41] T. K. Ho, "Random decision forests," in *Proceedings of 3rd International Conference on Document Analysis and Recognition*, Aug. 1995, vol. 1, pp. 278–282. doi: 10.1109/ICDAR.1995.598994.
- [42] Z. Wang, R. Wang, J. Gao, Z. Gao, and Y. Liang, "Fault recognition using an ensemble classifier based on Dempster–Shafer Theory," *Pattern Recognition*, vol. 99, p. 107079, 2020, doi: <https://doi.org/10.1016/j.patcog.2019.107079>.
- [43] A. Singh and C. S. Tucker, "A machine learning approach to product review disambiguation based on function, form and behavior classification," *Decision Support Systems*, vol. 97, no. March, pp. 81–91, 2017, doi: 10.1016/j.dss.2017.03.007.
- [44] scikit-learn, "sklearn.svm.SVC — scikit-learn 1.0.2 documentation," 2021. <https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html> (accessed Feb. 17, 2022).
- [45] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet Classification with Deep Convolutional Neural Networks," in *Advances in Neural Information Processing Systems*, 2012, vol. 25, pp. 1097–1105. [Online]. Available: <https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf>
- [46] K. He, X. Zhang, S. Ren, and J. Sun, "Deep Residual Learning for Image Recognition," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Jun. 2016, pp. 770–778. doi: 10.1109/CVPR.2016.90.
- [47] Python Software Foundation, "timeit — Measure execution time of small code snippets — Python 3.10.2 documentation," 2022. <https://docs.python.org/3/library/timeit.html> (accessed Feb. 17, 2022).

## Appendix – Python Code

```
#!/usr/bin/env python
# coding: utf-8

# # Imports and Project Setup

# In[1]:

from IPython.core.display import display, HTML
import winsound
import timeit
from random import randrange
from tensorflow.keras.utils import to_categorical
from matplotlib import pyplot as plt
import numpy as np
import pandas as pd
import seaborn as sn
import matplotlib as mpl
from matplotlib import pyplot
import warnings
import os
import cv2
import random
import struct
from pathlib import Path
import gzip
from math import floor

from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.utils import shuffle
from scipy.ndimage import gaussian_filter
from sklearn.preprocessing import LabelBinarizer
from sklearn.metrics import roc_curve, auc, roc_auc_score, accuracy_score
from imblearn.over_sampling import RandomOverSampler, SMOTE, ADASYN
from imblearn.under_sampling import RandomUnderSampler
from sklearn.model_selection import GridSearchCV

import tensorflow as tf
from tensorflow import keras
from tensorflow.keras.layers import Conv2D, Conv1D, MaxPooling2D, Flatten, Dense, Dropout,
BatchNormalization, Activation, AveragePooling2D
from tensorflow.keras.layers import SeparableConv2D, LeakyReLU, PReLU, Input, ZeroPadding2D
from tensorflow.keras.layers import GlobalMaxPooling2D, GaussianNoise
from tensorflow.keras.optimizers import Adam, RMSprop, SGD, Adadelta
from tensorflow.keras.models import Sequential
from keras.regularizers import l2, l1
from keras.callbacks import ReduceLROnPlateau

from sklearn import tree
from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, confusion_matrix
from sklearn import svm
from sklearn.multiclass import OneVsRestClassifier
from sklearn.model_selection import cross_val_score

warnings.filterwarnings('ignore')
seed = 0

# # Data Import and Setup

# In[2]:

# ##### GET DATA FROM NPZ FILE
# numpy_data = np.load('mnist-data/mnist.npz')

# nsamples, nx, ny = numpy_data["x_train"].shape # Reshape X_train
# X_train = numpy_data["x_train"].reshape((nsamples,nx*ny))
# y_train = numpy_data["y_train"]
```

```

# nsamples, nx, ny = numpy_data["x_test"].shape          # Reshape X_test
# X_test = numpy_data["x_test"].reshape((nsamples,nx*ny))
# y_test = numpy_data["y_test"]

##### GET DATA FROM IDX FILES
def read_idx_file(path):
    with gzip.open(path) as file:
        zero, data_type, dims = struct.unpack('>HBB', file.read(4))
        shape = tuple(struct.unpack('>I', file.read(4))[0] for d in range(dims))
        return np.frombuffer(file.read(), dtype=np.uint8).reshape(shape)

# Get X TEST and X TRAIN. Reshape so that each image is stored in one dimension, thus compatible with
the models
X_test = read_idx_file("mnist-data/t10k-images-idx3-ubyte.gz").reshape(10000, 784)
X_train = read_idx_file("mnist-data/train-images-idx3-ubyte.gz").reshape(60000, 784)

# Get Y TEST and Y TRAIN
y_test = read_idx_file("mnist-data/t10k-labels-idx1-ubyte.gz")
y_train = read_idx_file("mnist-data/train-labels-idx1-ubyte.gz")

##### CHECK DATA
print("First 10 train labels before shuffling:", y_train[0:10])
y_train, X_train = shuffle(y_train, X_train, random_state=seed)
print("First 10 train labels after shuffling: ", y_train[0:10])

# Check sizes are correct
print("Number of X_test images: ", len(X_test))
print("Number of y_test labels: ", len(y_test))
print("Number of X_train images: ", len(X_train))
print("Number of y_train labels: ", len(y_train))

print("\nFirst y_test label: ", y_test[0])
print("First X_test image:")
image = np.reshape(X_test[0], (28, 28)) # Reshape to 2d array
fig = plt.figure
plt.imshow(image, cmap='gray')
plt.show()

# ### Class Imbalance and Resampling

# In[ ]:

unique, class_counts_train = np.unique(y_train, return_counts=True) # Count of class occurrences
class_weights_dict = dict(zip(unique, class_counts_train)) # Store as dict in format {0: 1.234, 1:
1.234 ...}

for key in class_weights_dict:
    # Number of this class divided by total number of items
    class_weights_dict[key] = class_weights_dict[key] / len(y_train)
    print("Before resampling, class", key, "has weight of", round(class_weights_dict[key], 4))

# In[ ]:
def plot_class_balance(class_counts, title):
    mpl.rcParams.update(mpl.rcParamsDefault) # Set to default so no other changes affect this
    fig = plt.figure()
    ax = fig.add_axes([0,0,1,1])
    plt.xticks(np.arange(min(labels), max(labels)+1, 1.0)) # Show all digit labels
    ax.set_title(title)
    ax.set_ylabel('Number of occurrences')
    ax.set_xlabel('Digit label')
    ax.yaxis.grid(True)
    ax.yaxis.grid(zorder=0)

    for i in range(len(labels)): # Print each bar
        ax.bar(labels[i], class_counts[i], zorder = 3)

    plt.show()

```

```

unique, class_counts_train = np.unique(y_train, return_counts=True)
labels = range(0, 10) # 10 classes
plot_class_balance(class_counts_train, 'Occurrences of each training digit before resampling')

# In[ ]:

# resampler = SMOTE(sampling_strategy='not majority', n_jobs=-1) # Resample all except the one with
the most data
# #resampler = ADASYN(sampling_strategy='not majority', n_neighbors=5, n_jobs=-1) # Resample all
except the one with the most data
# #resampler = RandomOverSampler(sampling_strategy='not majority') # Resample all except the one with
the most data
# #resampler = RandomUnderSampler(sampling_strategy='not minority') # Resample all except the one with
the least data

# # Reshape to be compatible with resampler algorithms
# reshaped_x_train = X_train.reshape(len(X_train), 28*28)

# print("Number of images before resampling:", len(X_train))

# # Resample
# X_train, y_train = resampler.fit_resample(reshaped_x_train, y_train)

# print("Number of images after resampling: ", len(X_train))

# unique, class_counts_train = np.unique(y_train, return_counts=True) # Count of class occurrences
# class_weights_dict = dict(zip(unique, class_counts_train)) # Store as dict in format {0: 1234, 1:
1234 ...}

# for key in class_weights_dict:
#     # Number of this class divided by total number of items
#     class_weights_dict[key] = class_weights_dict[key] / len(y_train)
#     print("Class", key, "has weight of", round(class_weights_dict[key], 4))

# plot_class_balance(class_counts_train, 'Occurrences of each training digit after resampling')

# ### Data Pre-processing
# #### One-hot encoding - used only for CNN

# In[3]:

print("Before one-hot encoding, first y_train item looks like this:\t", y_train[0])

y_train = to_categorical(np.asarray(y_train))
y_test = to_categorical(np.asarray(y_test))

print("\nAfter one-hot encoding, first y_train item looks like this:\t", y_train[0])

# #### Normalisation
#

# In[4]:

print("Pixel range before normalising:", np.amin(X_train), "-", np.amax(X_train))

# Divide every greyscale level by 255 - max pixel value
X_train = X_train / 255
X_test = X_test / 255

print("Pixel range after normalising: ", np.amin(X_train), "-", np.amax(X_train))

```

```

### Dimensionality Reduction using Principal Component Analysis (PCA) - not used with CNN due to
data format

# In[ ]:

old_X_train = X_train # For plotting

pca = PCA(n_components=0.95) # Num. dimensions that add to 95% of dataset variance
pca.fit(X_train)
X_train = pca.transform(X_train)
X_test = pca.transform(X_test)

# VISUALISE THE ABOVE DATA
recovered_train = pca.inverse_transform(X_train)
recovered_image = np.reshape(recovered_train[0], (28, 28)) # Reshape to 2d array

initial_image = np.reshape(old_X_train[0], (28, 28))

print("Initial train shape:", old_X_train.shape)
print("New train shape:      ", X_train.shape)

fig, ((before_pca_ax, after_pca_ax)) = plt.subplots(1, 2)
before_pca_ax.imshow(initial_image, cmap='gray')
after_pca_ax.imshow(recovered_image, cmap='gray')
before_pca_ax.title.set_text('Initial image\n({0} values)'.format(len(old_X_train[0])))
after_pca_ax.title.set_text('Image recovered from PCA\n({0} values)'.format(len(X_train[0])))
plt.show()

# # Define and Train Model

# In[ ]:

# Define model
#classifier = tree.DecisionTreeClassifier(random_state=seed, max_depth=14)
#classifier = RandomForestClassifier(random_state = seed, n_jobs=-1, n_estimators=170)
classifier = svm.SVC(random_state=seed, max_iter=350, tol=2)

classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)

if isinstance(y_test[0], list) or isinstance(y_test[0], np.ndarray): # y_test encoded, so un-encode it
to evaluate
    y_test = y_test.argmax(axis=-1)
if isinstance(y_pred[0], list) or isinstance(y_pred[0], np.ndarray): # y_pred encoded, so un-encode it
to evaluate
    y_pred = y_pred.argmax(axis=-1)

print(classification_report(y_test, y_pred, zero_division="warn"))
print("Accuracy score: %0.3f" % accuracy_score(y_test, y_pred))

# cross_val_scores = cross_val_score(classifier,
#                                     np.concatenate((X_test, X_train), axis=0),
#                                     np.concatenate((y_test, y_train), axis=0),
#                                     cv=5)

#print("Cross Validation: %0.3f (+/- %0.3f)" % (cross_val_scores.mean(), cross_val_scores.std() * 2))

winsound.Beep(250, 1000) # Beep when done. Freq. in Hz, duration in ms

### Grid Search for finding best hyperparameter combination

# In[ ]:

# classifier = svm.SVC(random_state=seed, max_iter=350)

# #Grid search to check every combination of parameters - somewhat limited as it takes so long to run

```

```

# grid_parameters = {
#     "kernel": ["linear", "rbf", "sigmoid", "poly"],
#     "degree": [1, 2, 3, 4, 5],
#     "tol": [0.001, 0.01, 0.1, 1],
#     "C": [0.001, 0.01, 0.1, 1],
#     "gamma": [0.001, 0.01, 0.1, 1],
#     "decision_function_shape": ["ovo", "ovr"],
# }

# grid_search_clf = GridSearchCV(classifier, grid_parameters, n_jobs=24, scoring="accuracy",
# verbose=10, cv=2)
# grid_search_result = grid_search_clf.fit(X_train, y_train)
# print('Best Score: %s' % grid_search_result.best_score_)
# print('Best Hyperparameters: %s' % grid_search_result.best_params_)

### CNN - Define model and callbacks

# In[5]:

print("Num GPUs Available: ", len(tf.config.list_physical_devices('GPU')))

X_train = X_train.reshape(len(X_train), 28, 28, 1) # Reshape for CNN
X_test = X_test.reshape(10000, 28, 28, 1)

# Create early stopping callback
early_stopping_cb = keras.callbacks.EarlyStopping(monitor='val_accuracy',
min_delta=0, patience=8, verbose=1,
restore_best_weights=True)

# In[18]:

def make_compile_cnn():
    # LeNet-5
    clf = Sequential([
        # Convolutions
        Conv2D(filters=6, kernel_size=(5, 5), activation="tanh", input_shape=(28,28,1), strides=1),
        AveragePooling2D(pool_size=(2, 2), strides=1),

        Conv2D(filters=16, kernel_size=(5, 5), activation="tanh", strides=1),
        AveragePooling2D(pool_size=(2, 2), strides=1),

        Conv2D(filters=120, kernel_size=(5, 5), activation="tanh", strides=1),

        # Flatten to 1 dimensional
        Flatten(),

        # Fully connected layers
        Dense(units=84, activation="tanh"),
        Dense(units=10, activation="softmax"),
    ])

#ResNet-34
clf = Sequential([
    # Convolutions
    Conv2D(filters=64, kernel_size=(3, 3), activation="relu", input_shape=(28, 28, 1), strides=1),
    MaxPooling2D(pool_size=(2, 2), strides=1),

    Conv2D(filters=64, kernel_size=(3, 3), activation="relu", strides=1),
    Conv2D(filters=128, kernel_size=(3, 3), activation="relu", strides=1),
    #Conv2D(filters=256, kernel_size=(3, 3), activation="relu", strides=1),
    #Conv2D(filters=512, kernel_size=(3, 3), activation="relu", strides=1),
    AveragePooling2D(pool_size=(2, 2), strides=1),

    # Flatten to 1 dimensional
    Flatten(),

    # Fully connected layers

```

```

        Dense(units=1000, activation="relu"),
        Dense(units=10, activation="softmax")
    ])

    # AlexNet
    # clf = Sequential([
    #     # Convolutions
    #     Conv2D(filters=96, kernel_size=(11, 11), activation="relu", input_shape=(28, 28, 1),
    #     strides=1),
    #     MaxPooling2D(pool_size=(2, 2), strides=1),
    #
    #     Conv2D(filters=256, kernel_size=(5, 5), activation="relu", strides=1),
    #     MaxPooling2D(pool_size=(2, 2), strides=1),
    #
    #     #Conv2D(filters=384, kernel_size=(3, 3), activation="relu", strides=1),
    #     #Conv2D(filters=384, kernel_size=(3, 3), activation="relu", strides=1),
    #     Conv2D(filters=256, kernel_size=(3, 3), activation="relu", strides=1),
    #     MaxPooling2D(pool_size=(2, 2), strides=1),
    #
    #     # Flatten to 1 dimensional
    #     Flatten(),
    #
    #     # Fully connected layers
    #     Dense(units=4096, activation="relu"),
    #     Dense(units=4096, activation="relu"),
    #     Dense(units=1000, activation="relu"),
    #     Dense(units=10, activation="softmax")
    # ])

    clf.compile(optimizer=SGD(lr=0.1),
                loss='categorical_crossentropy',
                metrics=["accuracy"])

    return clf

classifier = make_compile_cnn()

### CNN - Define callbacks and fit

# In[19]:

get_ipython().run_cell_magic('html', '', '<!-- Makes the container wider so the training log is easier to read -->\n<style>\n.container {\n    width: 1400px !important;\n}')

# In[20]:

def config_fit_cnn(clf, do_show_output=True):
    batch_size = 128
    num_epochs = 100

    tf.config.experimental_run_functions_eagerly(True)
    # Create checkpoint callback that saves every epoch
    checkpoint_cb = keras.callbacks.ModelCheckpoint("cnn_model.h5",
        save_best_only=True, verbose=1, mode='min', monitor='val_loss')

    if do_show_output: # Trick to get around jupyter acting strange with verbose settings
        hist = clf.fit(X_train, y_train, batch_size=batch_size, epochs=num_epochs,
            validation_split=0.2, callbacks=[early_stopping_cb])
    else:
        hist = clf.fit(X_train, y_train, batch_size=batch_size, epochs=num_epochs,
            validation_split=0.2, verbose=0, callbacks=[early_stopping_cb])

    n_epochs = len(hist.history['loss'])

```

```

        return hist

train_history = config_fit_cnn(classifier)

y_pred = classifier.predict(X_test)
if isinstance(y_test[0], list) or isinstance(y_test[0], np.ndarray): # y_test encoded, so un-encode it
    to evaluate
    y_test = y_test.argmax(axis=-1)
if isinstance(y_pred[0], list) or isinstance(y_pred[0], np.ndarray): # y_pred encoded, so un-encode it
    to evaluate
    y_pred = y_pred.argmax(axis=-1)

print(classification_report(y_test, y_pred, zero_division="warn"))
print("Accuracy score: %0.3f" % accuracy_score(y_test, y_pred))

# ## Plot accuracy over number of epochs

# In[ ]:

mpl.rcParams['figure.dpi'] = 300

# Plot the accuracy and validation accuracy scores
plt.plot(train_history.history['accuracy'])
plt.plot(train_history.history['val_accuracy'])
n_epochs = len(train_history.history['loss'])

# Plot early stopping line and text
plt.plot([n_epochs - 1, n_epochs - 1], [0.0, 1.0], 'k--', lw=1.5, c="#ff0000")
early_stopping_text = plt.annotate(f'Early stopping: epoch {n_epochs - 1}', xy=(0.91, 0.035),
                                   xycoords='axes fraction', rotation=90, c="#ff0000", fontsize=8)

early_stopping_text.set_bbox(dict(facecolor='ffffff', alpha=0.75, edgecolor='ffffff'))

plt.title('Model Accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Accuracy', 'Validation Accuracy'], loc='lower left')

plt.xlim([0, n_epochs]) # Show up to the number of epochs
plt.ylim([0.9, 1]) # 0.0 to 1.0 accuracy

plt.grid()
plt.rc('grid', color='grey', lw=0.1)

# Force integer epoch numbers and multiples of 10
ax=plt.gca()
f = lambda x,pos: str(x).rstrip('0').rstrip('.')
ax.xaxis.set_major_locator(mpl.ticker.MultipleLocator(10))
ax.xaxis.set_major_formatter(mpl.ticker.FuncFormatter(f))

plt.show()

# ### Time model multiple times and average

# In[21]:

# Average time taken over X runs
NUM_RUNS = 10

def train_and_predict(clf):
    if (type(clf).__name__ == "Sequential"): # If the classifier is a keras classifier (this work uses
only sequential), run this code
        # Reset - make new classifier to forget previous training
        clf = make_compile_cnn()
        config_fit_cnn(clf, do_show_output=False)

```

```

        else: # Otherwise it will be a sklearn classifier, so run this code
            clf.fit(X_train, y_train)

        clf.predict(X_test)

times = [] # Hold time for each run
for i in range(0, NUM_RUNS):
    start_time = timeit.default_timer()
    train_and_predict(classifier)
    stop_time = timeit.default_timer()
    print('Run %s time: %0.4f seconds' % ((i + 1), (stop_time - start_time)))
    times.append(stop_time - start_time)

mean_time = sum(times) / len(times)
minutes = floor(mean_time / 60)
seconds = mean_time % 60

print("{} minutes, {} seconds".format(minutes, round(seconds, 1)))

display(HTML('<span style="color: navy; font-weight: bold; font-size: 18px;">' +
            'Average time of %s runs: %0.3f seconds <br>'
            '&nbsp; > %s minutes, %0.03f seconds'
            '</span>'
            % (NUM_RUNS, mean_time, minutes, seconds)))

winsound.Beep(250, 1000) # Beep when done. Freq. in Hz, duration in ms

# # Evaluate Model

# ## Confusion Matrix

# In[ ]:

confusion_matrix = pd.crosstab(y_test, y_pred,
                               rownames=['Actual'], colnames=['Predicted'])

mpl.rcParams.update(mpl.rcParamsDefault) # Set to default so no other changes affect this
mpl.rcParams['figure.dpi'] = 150 # Scale
sn.set(font_scale = 0.6) # Label size
sn.heatmap(confusion_matrix, annot=True, fmt="d") # Font size for numbers
plt.title('Confusion Matrix Heatmap', fontsize=9)
plt.show()

# ## Receiver Operating Characteristic Curve for sklearn classifiers

# In[ ]:

# Predict with a one vs rest classifier to get data in a ROC friendly format for sklearn
# Sequential keras models don't need this extra step
if type(classifier).__name__ != "Sequential":
    one_vs_rest_clf = OneVsRestClassifier(classifier)
    one_vs_rest_clf.fit(X_train, y_train)
    y_pred = one_vs_rest_clf.predict(X_test)

# Set plot figure size
fig, ax = plt.subplots(1,1, figsize = (12, 12))
mpl.rcParams['figure.dpi'] = 150 # Scale

roc_colours = ["red", "orange", "yellow", "lime", "blue", "violet", "magenta", "green", "navy",
               "teal"]

# Binarize to fit with ROC
lb = LabelBinarizer()
lb.fit(y_test)
y_test = lb.transform(y_test)
y_pred = lb.transform(y_pred)

```

```

fpr = dict()
tpr = dict()

# Calculate and plot micro average
fpr["micro"], tpr["micro"], _ = roc_curve(y_test.ravel(), y_pred.ravel())
micro = auc(fpr["micro"], tpr["micro"])
plt.plot(fpr["micro"], tpr["micro"], linewidth=6, label='Micro-avg. (AUC: {0:0.3f})'.format(micro))

# Plot result for each class
for (index, c_label) in enumerate(range(0, 10)):
    fpr[index], tpr[index], thresholds = roc_curve(y_test[:,index].astype(int), y_pred[:,index])
    ax.plot(fpr[index], tpr[index], linewidth = 1.5, color = roc_colours[index],
            label = 'Digit: %s (AUC: %0.3f)' % (c_label, auc(fpr[index], tpr[index])))

# Plot average line
ax.plot(fpr[index], fpr[index], 'k--', label = 'Random Chance')

# Set limits of axes = make the height go slightly past 1 to account for the line thickness
plt.xlim([-0.0025, 1.0])
plt.ylim([0.0, 1.0075])

ax.legend()
plt.title('Digit multi-class ROC curve with AUC scores', fontsize=20)
ax.set_xlabel('False Positive Rate', fontsize=18)
ax.set_ylabel('True Positive Rate', fontsize=18)
plt.xticks(fontsize=16)
plt.yticks(fontsize=16)
ax.set_facecolor('white')
plt.legend(facecolor='white', fontsize=16)
plt.grid(color='grey', linestyle='--', linewidth=0.75)
plt.show()

winsound.Beep(250, 1000) # Beep when done. Freq. in Hz, duration in ms

```