Multi-class classification with neural networks

Matthew Finster
Master of Data Science student (ID: 21702717)
LaTrobe University

September 2024

Contents

1 Introduction

This paper details the approach to solving a multi-class classification problem using neural networks. The **MNIST** dataset, which is used for this problem, can be downloaded as part of the keras package. The dataset consists of images of handwritten digits ranging from 0 to 9. Each **MNIST** data point contains two components: an image of a handwritten digit, denoted as x, and a corresponding label, denoted as y. The y label specifies the digit that the given image represents.

In this paper, several neural networks (NNs) will be developed and trained to classify handwritten digits. After training, the performance of each NN will be evaluated.

The **MNIST** dataset is well-known and has already been preprocessed. As a result, there was no need to handle missing values, duplicated rows, or manually divide the data into training and testing sets. However, a few pre-processing steps were performed to prepare the data for the neural networks. Specifically, the x values (pixel intensities) were normalized to range from 0 to 1 by dividing each x value by 255. The y values (labels) were one-hot encoded using the to_categorical function from keras.

2 Part 1: Neural networks (NNs)

Multilayer neural networks consist of three types of layers: an *input* layer, which receives the input data, one or more *hidden* layers, which consist of computational neurons, and an *output* layer, which produces the output[1]. In the hidden layers, the input signals are multiplied by weights, and biases are added. The resulting values are passed through activation functions, which determine whether a neuron should be activated. The signal is propagated forward through the network, layer by layer until the final layer. The final layer passes through a final activation function. Afterwards, the output values represent probabilities.

The weights in the network are then adjusted through a process called backpropagation. Backpropagation works by calculating errors at the output layer (using a loss function) and propagating those errors backward through the network to update the weights. This iterative process is repeated until the network learns to make accurate predictions.

2.1 NN model 1: Description

In the first NN model, a baseline multilayer neural network was built without convolutional layers to complete the classification task. The images were first flattened from 28×28 matrices into vectors of 784 elements. These vectors were passed as input to the NN's input layer and subsequently to a hidden layer consisting of 128 neurons. The `ReLU` activation function was applied at this layer, which was chosen because it a) introduces non-linearity to the model (as all activation functions aim to do) and, b) avoids the vanishing gradient problem that arises with the `sigmoid` and `tanh` activation functions. Following the hidden layer, the output from the `ReLU` function was passed to the output layer, which consisted of 10 neurons — one for each possible class (digits 0 to 9). The `Softmax` activation function was applied in the output layer, as this is a multi-class classification problem and `Softmax` converts the outputs into a probability distribution across the 10 classes.

For backpropagation, the categorical `CrossEntropy` loss function was used as is appropriate for multi-class classification problems where the labels are one-hot encoded. `CrossEntropy` measures the difference between the predicted probability distribution (from the `Softmax` output) and the true distribution (from the one-hot encoded values). This loss function is minimised during backpropagation by adjusting the weights of the network using `Stochastic Gradient Descent (SGD)` as the optimizer. `SGD` was chosen as it iteratively updates the model's weights based on subsets of the training data, rather than using the entire dataset at once, usually leading to faster convergence.

The architecture for NN model 1 can be found in Table 1 and Figure 1.

Layer	**Operation**	**Output Dimensions**
Input Layer	Flatten input from 28×28 to 784×1	$\mathbf{x}^{(0)} \in \mathbf{R}^{784}$
Hidden Layer 1 (Dense, ReLU)	$\mathbf{z}^{(1)} = \mathbf{W}^{(0)}\mathbf{x}^{(0)}$ $x^{(1)} = \max(0, \mathbf{z}^{(1)})$ (ReLU activation)	$\mathbf{x}^{(1)} \in R^{128}$
Output Layer (Softmax)	$\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{x}^{(1)}$ $\mathbf{x}^{(2)} = \text{Softmax}(\mathbf{z}^{(2)})$ Probability distribution over 10 classes	$\mathbf{x}^{(2)} \in R^{10}$

Table 1: NN model 1 architecture

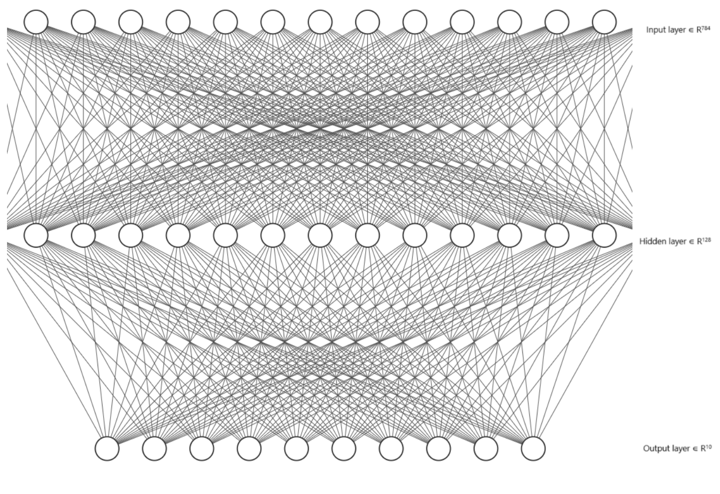

Figure 1: NN model 1 archictecture, created using NN-SVG[2]

2.2 NN model 2: Description

Another three different NN architectures were trialled to build upon and improve the accuracy of NN model 1 (see Table 3). The following changes were made from NN model 1 to NN model 2:

- Increased the number of dense layers from 1 to 2
- Increased the number of neurons in dense layers from 128 in the first dense layer to 512. Included 256 neurons in the second dense layer.

The architecture for NN model 2 can be found in Table 2 and Figure 2.

Layer	**Operation**	**Output Dimensions**
Input Layer	Flatten input from 28×28 to 784×1	$\mathbf{x}^{(0)} \in R^{784}$
Hidden Layer 1 (Dense, ReLU)	$\mathbf{z}^{(1)} = \mathbf{W}^{(0)}\mathbf{x}^{(0)}$ $\mathbf{x}^{(1)} = \max(0, \mathbf{z}^{(1)})$ (ReLU activation)	$\mathbf{x}^{(1)} \in R^{512}$
Hidden Layer 2 (Dense, ReLU)	$\mathbf{z}^{(2)} = \mathbf{W}^{(1)}\mathbf{x}^{(1)}$ $\mathbf{x}^{(2)} = \max(0, \mathbf{z}^{(2)})$ (ReLU activation)	$\mathbf{x}^{(2)} \in R^{256}$
Output Layer (Softmax)	$\mathbf{z}^{(3)} = \mathbf{W}^{(2)}\mathbf{x}^{(2)}$ $\mathbf{x}^{(3)} = \text{Softmax}(\mathbf{z}^{(3)})$ Probability distribution over 10 classes	$\mathbf{x}^{(3)} \in R^{10}$

Table 2: NN model 2 architecture

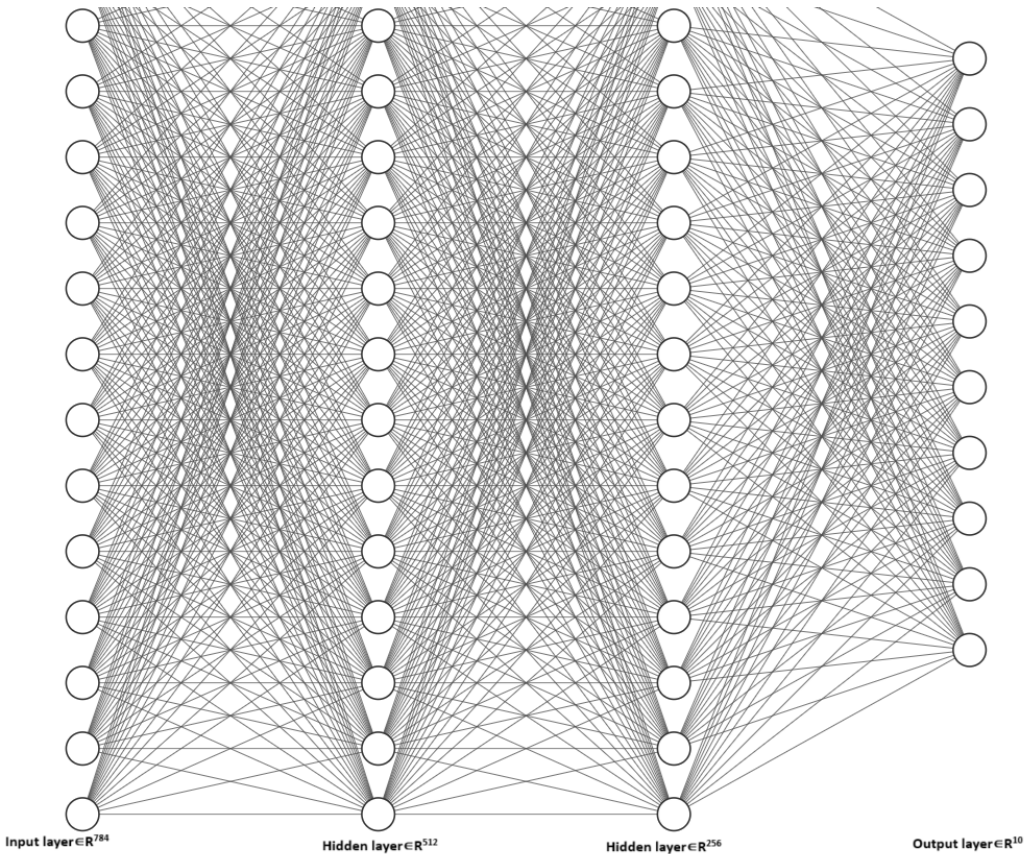

Figure 2: NN model 2 archictecture, created using NN-SVG[2]

2.3 Comparison of results

The performance of these NNs was measured by their prediction accuracy in classifying images from the test set (i.e. number of the correctly predicted images / number of the images in the test set). The results can be found in Table 3.

Based on the results, it is evident that model performance improves as the model depth increases. NN model 1.2, with a more shallow architecture (fewer neurons), resulted in worse performance. In contrast, NN models 1.1 and 2, which had

Model	Model Architecture	Accuracy
NN model 1	Input layer $\mathbf{x}^{(0)} \in R^{784}$ Hidden Layer 1 (Dense, ReLU) $\mathbf{x}^{(1)} \in R^{128}$ Output Layer (Softmax) $\mathbf{x}^{(2)} \in R^{10}$	95.14%
NN model 1.1	Input layer $\mathbf{x}^{(0)} \in R^{784}$ Hidden Layer 1 (Dense, ReLU) $\mathbf{x}^{(1)} \in R^{256}$ Hidden Layer 2 (Dense, ReLU) $\mathbf{x}^{(2)} \in R^{128}$ Output Layer (Softmax) $\mathbf{x}^{(3)} \in R^{10}$	96.81%
NN model 1.2	Input layer $\mathbf{x}^{(0)} \in R^{784}$ Hidden Layer 1 (Dense, ReLU) $\mathbf{x}^{(1)} \in R^{64}$ Output Layer (Softmax) $\mathbf{x}^{(2)} \in R^{10}$	95.03%
NN model 2	Input layer $\mathbf{x}^{(0)} \in R^{784}$ Hidden Layer 1 (Dense, ReLU) $\mathbf{x}^{(1)} \in R^{512}$ Hidden Layer 2 (Dense, ReLU) $\mathbf{x}^{(2)} \in R^{256}$ Output Layer (Softmax) $\mathbf{x}^{(3)} \in R^{10}$	97.04%

Table 3: Comparison of different neural network models

additional hidden layers and more neurons, showed improved accuracy. This suggests that deeper models, with more layers and neurons, can capture more complex relationships in the data. However, while increased depth generally leads to better performance, it is prudent to be wary that there is a risk of overfitting as models become too deep, which may reduce performance on unseen data. Using dropout can help mitigate this[3].

3 Part 2: Convolutional neural networks (CNNs)

Convolutional neural networks (CNNs) consist of four types of layers: an *input layer* that receives the input data, one or more *convolutional layers* that create feature maps by sliding a kernel over the image to generate convolutions, *pooling layers* that reduce the dimensionality of each feature map, and *fully connected layers* that are responsible for producing the final output[4].

The weights in the convolutional layers and fully connected layers are adjusted through backpropagation. Like with an NN, backpropagation works by calculating errors at the output layer (using a loss function) and propagating these errors backward through the network, updating the weights accordingly. This iterative process continues until the network learns to make accurate predictions.

3.1 CNN model 1: Description

A baseline convolutional neural network (CNN) was built to complete the classification task. In order to fit the **MNIST** dataset into a *Conv2D* layer, the input shape was changed to meet the required format of (batch_size, image_height, image_width, image_channels) by reshaping the dataset to (dataset_length, 28, 28, 1). This fourth argument is typically set to 3 for RGB images, but since the **MNIST** dataset consists of black-and-white images, the channel dimension was set to 1.

The reshaped input was passed to the first convolutional layer, which consisted of 32 filters with a kernel size of 3×3 and same padding to preserve the spatial dimensions. Different weights were assigned by the model to each of the 32 filters in order to find 32 different types of features. The output of the first convolutional layer had a shape of (28, 28, 32). The second convolutional layer, also with 32 filters of size 3×3, was applied to extract more complex features, keeping the output shape the same at (28, 28, 32). After the second convolutional layer, a 2×2 `MaxPooling` layer was applied, reducing the spatial dimensions by half to (14, 14, 32). `MaxPooling` downsamples the feature maps by selecting the maximum value within a 2×2 window, thereby reducing the spatial dimensions. This downsizing not only reduces the computational cost but also helps in focusing on the most important features. `MaxPooling`, which selects the maximum value in a 2×2 grid, was selected over `AveragePooling` due to its effectiveness in preserving sharp features, such as edges, which could be important for this task. The output from the pooling layer was then flattened into a 1D vector (size $14 \times 14 \times 32 = 6272$) and passed into a fully connected (dense) layer with 512 neurons. To prevent overfitting, a `Dropout` layer with a rate of 0.2 was applied, randomly turning off 20% of the neurons during training. From the 512 neurons, the data was passed to the output layer, which consists of 10 neurons, corresponding to the 10 possible classes (digits 0-9), with a `Softmax` activation function applied to output the class probabilities.

Like the previous NNs; for backpropagation, the categorical `CrossEntropy` loss function was used, which is appropriate for multi-class classification problems where the labels are one-hot encoded. The `SGD` optimizer was employed to minimize this loss by iteratively updating the weights of the network in both the convolutional layers and the fully connected layers.

The architecture for CNN model 1 can be found in Table 4 and Figure 3.

Layer	**Operation**	**Output Dimensions**
Input Layer	$(28 \times 28 \times 1)$	$\mathbf{x}^{(0)} \in R^{28\times28\times1}$
Conv2D Layer 1	$\mathbf{z}^{(1)} = \text{Conv2D}(\mathbf{W}^{(0)}\mathbf{x}^{(0)})$ 32 filters of size 3×3, ReLU activation	$\mathbf{x}^{(1)} \in R^{28\times28\times32}$
Conv2D Layer 2	$\mathbf{z}^{(2)} = \text{Conv2D}(\mathbf{W}^{(1)}\mathbf{x}^{(1)})$ 32 filters of size 3×3, ReLU activation	$\mathbf{x}^{(2)} \in R^{28\times28\times32}$
MaxPooling Layer	$\mathbf{x}^{(3)} = \text{MaxPooling}(\mathbf{x}^{(2)})$ MaxPooling size 2×2	$\mathbf{x}^{(3)} \in R^{14\times14\times32}$
Flatten Layer	Flatten output from $14 \times 14 \times 32$ to a vector of size 6,272	$\mathbf{x}^{(4)} \in R^{6272}$
Dense Layer 1 (ReLU)	$\mathbf{z}^{(5)} = \mathbf{W}^{(4)}\mathbf{x}^{(4)}$ $\mathbf{x}^{(5)} = \max(0, \mathbf{z}^{(5)})$ ReLU activation	$\mathbf{x}^{(5)} \in R^{512}$
Dropout Layer	Dropout with rate 0.2	$\mathbf{x}^{(5)} \in R^{512}$
Output Layer (Softmax)	$\mathbf{z}^{(6)} = \mathbf{W}^{(5)}\mathbf{x}^{(5)}$ $\mathbf{x}^{(6)} = \text{Softmax}(\mathbf{z}^{(6)})$ Probability distribution over 10 classes	$\mathbf{x}^{(6)} \in R^{10}$

Table 4: CNN model 1 architecture

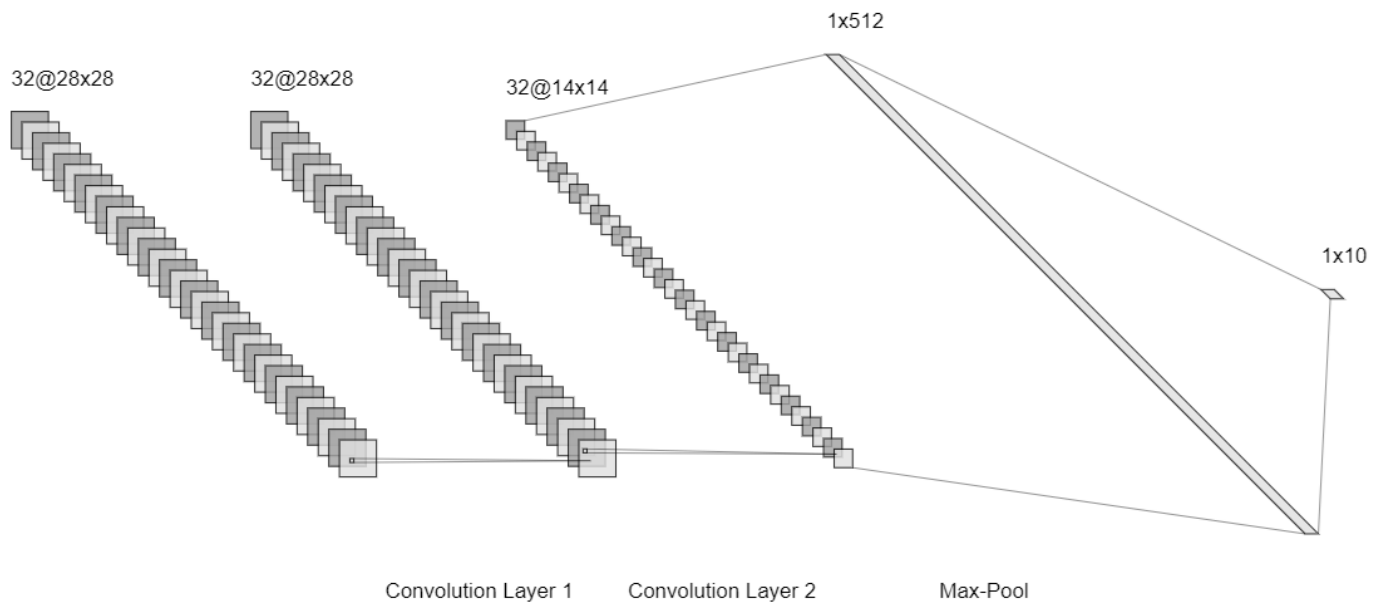

Figure 3: CNN model 1 architecture, created using NN-SVG[2]

3.2 CNN model 2: Description

Another three different CNN architectures were trialled to build upon and improve the accuracy of CNN model 1 (see Table 6). The following changes were made from CNN model 1 to CNN model 2:

- Increased the number of convolutional layers from 2 to 3 and applied MaxPooling after two of these layers
- Increased the number of filters increasingly throughout the model from 32 to 64 to 128

The architecture for CNN model 2 can be found in Table 5 and Figure 4.

Layer	Operation	Output Dimensions
Input Layer	$(28 \times 28 \times 1)$	$\mathbf{x}^{(0)} \in R^{28\times28\times1}$
Conv2D Layer 1	$\mathbf{z}^{(1)} = \text{Conv2D}(\mathbf{W}^{(0)}\mathbf{x}^{(0)})$ 32 filters of size 3×3, ReLU activation	$\mathbf{x}^{(1)} \in R^{28\times28\times32}$
Conv2D Layer 2	$\mathbf{z}^{(2)} = \text{Conv2D}(\mathbf{W}^{(1)}\mathbf{x}^{(1)})$ 64 filters of size 3×3, ReLU activation	$\mathbf{x}^{(2)} \in R^{28\times28\times64}$
MaxPooling Layer 1	$\mathbf{x}^{(3)} = \text{MaxPooling}(\mathbf{x}^{(2)})$ MaxPooling size 2×2	$\mathbf{x}^{(3)} \in R^{14\times14\times64}$
Conv2D Layer 3	$\mathbf{z}^{(4)} = \text{Conv2D}(\mathbf{W}^{(3)}\mathbf{x}^{(3)})$ 128 filters of size 3×3, ReLU activation	$\mathbf{x}^{(4)} \in R^{14\times14\times128}$
MaxPooling Layer 2	$\mathbf{x}^{(5)} = \text{MaxPooling}(\mathbf{x}^{(4)})$ MaxPooling size 2×2	$\mathbf{x}^{(5)} \in R^{7\times7\times128}$
Flatten Layer	Flatten output from $7 \times 7 \times 128$ to a vector of size 6,272	$\mathbf{x}^{(6)} \in R^{6272}$
Dense Layer 1 (ReLU)	$\mathbf{z}^{(7)} = \mathbf{W}^{(6)}\mathbf{x}^{(6)}$ $\mathbf{x}^{(7)} = \max(0, \mathbf{z}^{(7)})$ ReLU activation	$\mathbf{x}^{(7)} \in R^{512}$
Dropout Layer	Dropout with rate 0.2	$\mathbf{x}^{(7)} \in R^{512}$
Output Layer (Softmax)	$\mathbf{z}^{(8)} = \mathbf{W}^{(7)}\mathbf{x}^{(7)}$ $\mathbf{x}^{(8)} = \text{Softmax}(\mathbf{z}^{(8)})$ Probability distribution over 10 classes	$\mathbf{x}^{(8)} \in R^{10}$

Table 5: CNN model 2 architecture

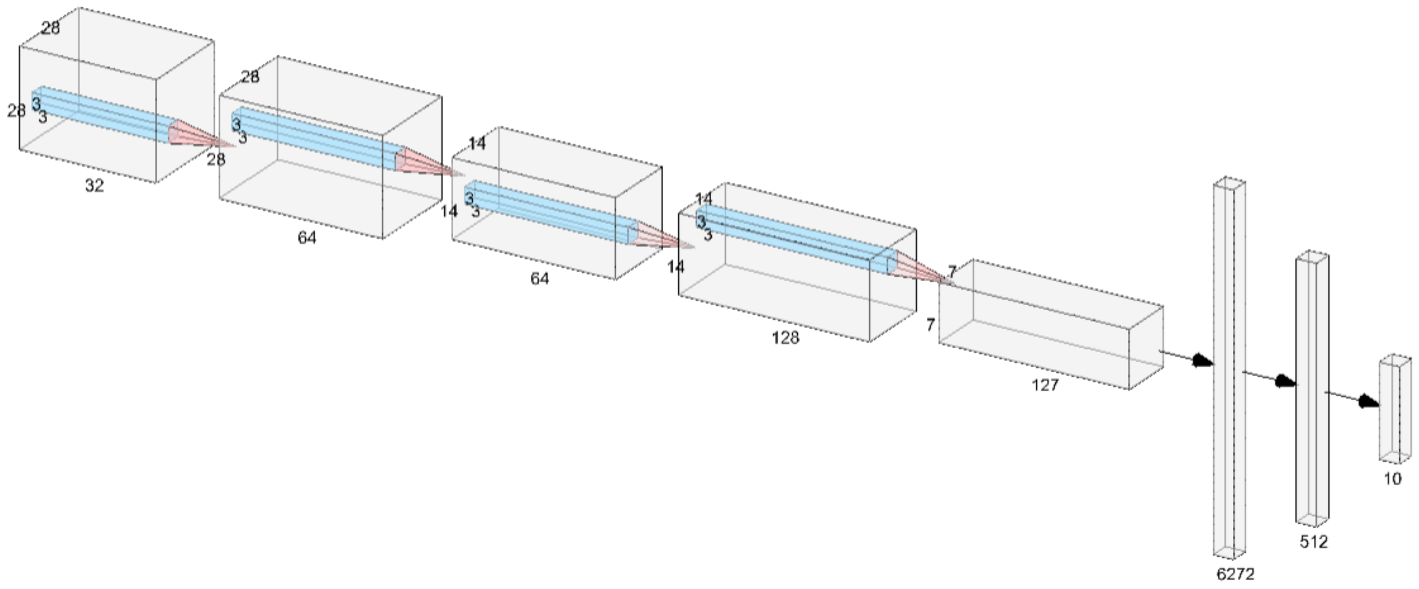

Figure 4: CNN model 2 architecture, created using NN-SVG[2]

3.3 Comparison of results

The performance of the CNNs was measured by their prediction accuracy. The results can be found in Table 6.

Based on the results, it appears that model performance generally improved as the model depth increased, similar to the NNs. However, there was some variability in performance across different runs. For example, CNN model 2 achieved 97.45% and 97.47% accuracy in two runs but dropped to 94.33% in another. CNN models 1.1 and 1.2 initially performed worse but showed improvements in subsequent runs. These fluctuations could be attributed to factors such as overfitting in certain runs or differences in the order of the training batches.

While all CNN models produced relatively similar results, CNN model 2 was ultimately chosen as the best model, as it demonstrated the ability to achieve the highest accuracy. In contrast, CNN model 1.2, while more complex, was the most computationally expensive and did not produce enough benefits to justify the additional complexity. For this reason, it was considered the worst CNN model.

To address the inconsistencies observed across runs and to produce more reliable performance, hyperparameter optimisation strategies were explored in Part 3.

Model	Model Architecture	Best accuracy
CNN model 1	Input layer $\mathbf{x}^{(0)} \in R^{28\times28\times1}$ CvLayer 1 (Conv2D, ReLU, 32 filters, 3×3) $\mathbf{x}^{(1)} \in R^{28\times28\times32}$ CvLayer 2 (Conv2D, ReLU, 32 filters, 3×3) $\mathbf{x}^{(2)} \in R^{28\times28\times32}$ MaxPooling (2×2) $\mathbf{x}^{(3)} \in R^{14\times14\times32}$ Flatten Layer $\mathbf{x}^{(4)} \in R^{6272}$ Dense Layer (ReLU, 512 units) $\mathbf{x}^{(5)} \in R^{512}$ Dropout (0.2) Output Layer (Softmax, 10 units) $\mathbf{x}^{(6)} \in R^{10}$	96.79%
CNN model 1.1	Input layer $\mathbf{x}^{(0)} \in R^{28\times28\times1}$ CvLayer 1 (Conv2D, ReLU, 64 filters, 3×3) $\mathbf{x}^{(1)} \in R^{28\times28\times64}$ CvLayer 2 (Conv2D, ReLU, 64 filters, 3×3) $\mathbf{x}^{(2)} \in R^{28\times28\times64}$ MaxPooling (2×2) $\mathbf{x}^{(3)} \in R^{14\times14\times64}$ Flatten Layer $\mathbf{x}^{(4)} \in R^{12544}$ Dense Layer 1 (ReLU, 1024 units) $\mathbf{x}^{(5)} \in R^{1024}$ Dropout (0.2) Dense Layer 2 (ReLU, 512 units) $\mathbf{x}^{(6)} \in R^{512}$ Output Layer (Softmax, 10 units) $\mathbf{x}^{(7)} \in R^{10}$	96.92%
CNN model 1.2	Input layer $\mathbf{x}^{(0)} \in R^{28\times28\times1}$ CvLayer 1 (Conv2D, ReLU, 64 filters, 3×3) $\mathbf{x}^{(1)} \in R^{28\times28\times64}$ CvLayer 2 (Conv2D, ReLU, 64 filters, 3×3) $\mathbf{x}^{(2)} \in R^{28\times28\times64}$ MaxPooling (2×2) $\mathbf{x}^{(3)} \in R^{14\times14\times64}$ CvLayer 3 (Conv2D, ReLU, 128 filters, 3×3) $\mathbf{x}^{(4)} \in R^{14\times14\times128}$ CvLayer 4 (Conv2D, ReLU, 128 filters, 3×3) $\mathbf{x}^{(5)} \in R^{14\times14\times128}$ MaxPooling (2×2) $\mathbf{x}^{(6)} \in R^{7\times7\times128}$ CvLayer 5 (Conv2D, ReLU, 256 filters, 3×3) $\mathbf{x}^{(7)} \in R^{7\times7\times256}$ CvLayer 6 (Conv2D, ReLU, 256 filters, 3×3) $\mathbf{x}^{(8)} \in R^{7\times7\times256}$ MaxPooling (2×2) $\mathbf{x}^{(9)} \in R^{3\times3\times256}$ Flatten Layer $\mathbf{x}^{(10)} \in R^{2304}$ Dense Layer 1 (ReLU, 512 units) $\mathbf{x}^{(11)} \in R^{512}$ Dropout (0.2) Output Layer (Softmax, 10 units) $\mathbf{x}^{(12)} \in R^{10}$	96.79%
CNN model 2	Input layer $\mathbf{x}^{(0)} \in R^{28\times28\times1}$ CvLayer 1 (Conv2D, ReLU, 32 filters, 3×3) $\mathbf{x}^{(1)} \in R^{28\times28\times32}$ CvLayer 2 (Conv2D, ReLU, 64 filters, 3×3) $\mathbf{x}^{(2)} \in R^{28\times28\times64}$ MaxPooling (2×2) $\mathbf{x}^{(3)} \in R^{14\times14\times64}$ CvLayer 3 (Conv2D, ReLU, 128 filters, 3×3) $\mathbf{x}^{(4)} \in R^{14\times14\times128}$ MaxPooling (2×2) $\mathbf{x}^{(5)} \in R^{7\times7\times128}$ Flatten Layer $\mathbf{x}^{(6)} \in R^{6272}$ Dense Layer (ReLU, 512 units) $\mathbf{x}^{(7)} \in R^{512}$ Dropout (0.2) Output Layer (Softmax, 10 units) $\mathbf{x}^{(8)} \in R^{10}$	97.47%

Table 6: Comparison of different CNN models

4 Part 3: Hyperparameter optimisation

4.1 Hyperparameters 1: Description

The network structure from CNN model 2 was used as the final model. Our default hyperparameters in CNN model 2 included a learning rate of 0.002, the use of the `SGD` optimiser, 5 epochs and a batch size of 60 (therefore 1000 batches per epoch).

The learning rate controls how large the updates to the model's weights are after each batch. Larger learning rates (such as 0.01) make bigger updates and lead to faster convergence, but if the updates are too large, the model can skip past the global minima in the loss function, leading to decreased performance. On the other hand, smaller learning rates (like 0.001) make smaller updates which helps to avoid overshooting the optimal point, but if the rate is too small, the model can get stuck in a local minima and fail to reach the best possible solution (see Figure 5)[5].

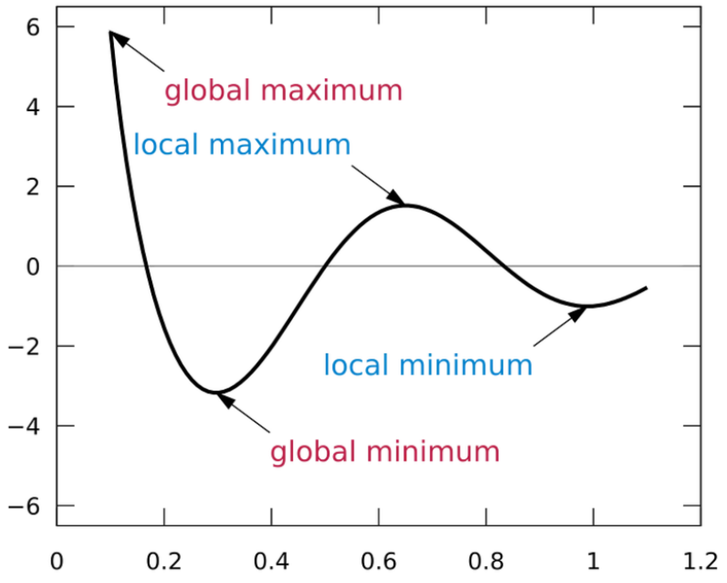

Figure 5: Local minima (trap) vs global minima (optimal point)[5]

The optimiser determines how the gradients, calculated from the loss function, are used to update the model's weights during training. For example, `SGD` uses a fixed learning rate and applies uniform updates, making it simpler and more computationally efficient than many other optimisers. When combined with momentum and learning rate decay, `SGD` can generalise well to new data, though it converges more slowly, particularly in complex models. In contrast, `Adam` adapts the learning rate and update direction for each parameter by tracking both the running average of past gradients (momentum) and past squared gradients. This enables faster convergence, especially in more complex models, but can sometimes lead to overfitting, as `Adam` adapts to the training set, sometimes at the cost of generalisation on unseen data.

Choosing the best optimiser and learning rate is essential to model performance, however choosing the best batch size and number of epochs are also important. After finding the best learning rate and optimiser combination (Table 7); various combinations of batch size and epochs were tested (Table 8). Smaller batch sizes result in a model updating its weights more frequently per epoch. For our example using the **MNIST** dataset of 60 000 images, a batch size of 30 would result in 2000 updates per epoch. This means that training will take longer per epoch (than using a batch size of 60), however it can help the model generalise better as each batch is smaller and influenced more by each data point. Larger batches lead to faster computation, however require more memory and can sometimes lead to overfitting as there is less noise (non-typical examples) per batch.

Similarly, fewer epochs result in faster training times but can lead to underfitting as the model does not learn enough patterns from the data. In contrast, more epochs result in slower training times but can provide ample time for the model to learn patterns in the data. If trained for too many epochs, however, the model can memorise the training data, leading to overfitting and poor generalisation on unseen data. Striking the right balance here is also critical.

4.2 Hyperparameters 2: Description

Several combinations of different learning rates and optimisers were trialled to optimise the performance of CNN model 2 (see Table 7). After finding the optimal learning rate and optimiser, several combinations of different batch sizes and number of epochs were also trialled (see Table 8). Finally, the following changes were made from CNN model 2 to the final model with finely tuned hyperparameters:

- Changed the optimiser from ’SGD’ to ’Adam’

The full list of hyperparameters that resulted in the optimal performance can be found in Table 9.

4.3 Comparison of results

Table 7 and Table 8 show the performance of CNN model 2, using different hyperparameter combinations, as measured by its prediction accuracy. Table 9 shows the difference between CNN model 2 and the final model with finely tuned hyperparameters.

Trial	**Optimizer**	**Learning Rate**	**Accuracy**
1	SGD (momentum=0.7)	0.001	96.47%
2	SGD (momentum=0.7)	0.002	97.47%
3	SGD (momentum=0.7)	0.005	98.06%
4	SGD (momentum=0.7)	0.01	98.38%
5	Adam	0.001	99.06%
6	Adam	0.002	99.19%
7	Adam	0.005	98.32%
8	Adam	0.01	11.35%

Table 7: Hyperparameter search for Adam and SGD with various learning rates

Trial	**Batch size**	**Number of epochs**	**Accuracy**	**Training time**
1	30	5	98.85%	14 mins 12 secs
2	30	10	99.29 %	29 mins 55 secs
3	60	5	99.19%	12 mins 10 secs
4	60	10	99.09%	22 mins 41 secs
5	120	5	99.16%	10 mins 36 secs
6	120	10	99.26%	20 mins 16 secs

Table 8: Hyperparameter search for batch size and number of epochs (using the Adam optimiser and a 0.002 learning rate)

Based on the results, it appears that model performance improved as the `Adam` optimiser was used and the initial learning rate was maintained between 0.001 and 0.005. This is likely due to `Adam`'s ability to dynamically adjust the learning rate for each parameter during training, allowing it to make more fine-tuned updates based on gradient information. In summary, using a LR of 0.002 ensured that the model did not make excessively large updates initially that that could cause it to overshoot the optimal solution (as seen in the performance drop at 0.01), while the adaptive nature of `Adam` helped prevent the model getting stuck in local minima.

Although a batch size of 30 and 10 epochs led to the highest accuracy, this came at a cost as the learning time elapsed 29 minutes and 55 seconds. In contrast, when using the original batch size of 60 and 5 epochs, it took only 12 minutes and 10 seconds. This latter combination of parameters take $\approx 40\%$ of the time that the former combination of parameters take, with only 10 fewer correct classifications (0.1% of 10,000). While this 0.1% difference may be valued more highly by others, for the purposes of this paper, the extra computational efficiency was considered more important than the marginal improvement in accuracy. Therefore, 5 epochs with a batch size of 60 was chosen for the final finely tuned model.

Hyperparameters	**CNN model 2**	**CNN model 2 (finely tuned)**
Learning Rate (lrate)	0.002	0.002
Epochs	5	5
Decay	lrate / epochs	Internally managed by Adam
Optimizer	SGD with momentum 0.7	Adam
Batch Size	60	60
Performance (Accuracy)	97.47%	99.19%

Table 9: Hyperparameters and performance: CNN model 2 vs CNN model 2 (finely tuned)

5 Conclusion

This paper has demonstrated that model performance is highly influenced by both the architecture of neural networks and the choice of hyperparameters used for optimizing the model during training. The experiments showed that the combination of layers can significantly enhance the model's ability to learn and, ultimately, improve its accuracy. In particular, it has been demonstrated that the use of convolutional layers is highly effective for image classification tasks. Additionally, increasing the depth of the neural network can improve the model's ability to recognize patterns, provided it does not become too deep, which can lead to overfitting or the model becoming too computationally expensive.

To understand the rankings of all models discussed, refer to Table 3, Table 6, and Table 8. These tables provide a comprehensive ranking of the various models and optimisers tested throughout this paper. It should be noted, however, that while the final model achieved an accuracy of over 99% on the **MNIST** dataset, this level of performance may not generalise well to other datasets. As demonstrated in this paper; one should experiment with different neural network architectures, as well as with hyperparameters, to optimise performance on other diverse datasets.

References

[1] S. Haykin, *"Neural networks and learning machines"* (Third edition.), Pearson, 2009.

[2] A. Lenail (2024). NN-SVG: Publication-quality neural network architecture schematics. Retrieved from `https://alexlenail.me/NN-SVG/LeNet.html`

[3] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, & R. Salakhutdinov, *"Dropout: A simple way to prevent neural networks from overfitting"*, Journal of Machine Learning Research, vol. 15, pp. 1929-1958, 2014. Retrieved from `https://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`

[4] A. Choulwar, "The art of the convolutional neural network", Medium, 2019. Retrieved from `https://medium.com/machine-learning-researcher/convlutional-neural-network-cnn-2fc4faa7bb63`

[5] M. Mishra, "The curse of local minima: How to escape and find the global minimum," Medium, 2023. Retrieved from `https://mohitmishra786687.medium.com/the-curse-of-local-minima-how-to-escape-and-find-the-global-minimum-fdabceb2cd6a`