Assignment 1

Group members

Edward Ji	ziji4098	510477226
Yao Ke	yake9601	510459679
Daniel Kovalenko	dkov2101	500499357

Abstract

We were given an unknown dataset with 128 features which we had to classify into 10 unique classes using a neural network. As such, we had to first construct a deep learning framework with automatic gradient calculation and descent using Numpy for the base vector operations. We then tested many variations of multi-layer perception designs and hyperparameters, achieving an optimal model with 54% accuracy on this classification task. Throughout our testing, we discovered that the optimal model is a small model with a 128 nodes input layer, 192 node hidden layer, a batch normalisation layer, a leaky ReLU activation layer, a 10 node output layer and a soft max activation layer. This shows that the pattern in our data is relatively simple requiring a small model and some regularisation for optimal performance.

1 Introduction

1.1 Aim

The aim of this assignment is to train a neural network which can classify instances with 128 features into 10 possible classes, achieving the highest possible accuracy. To achieve this, we built a CPU implementation of an automatic differentiation framework to classify the unknown dataset.

We also aimed to assess the performance and efficiency of the different components we implemented. In this report, we will briefly explain our implementation of automatic differentiation, and explore the different layers and optimisers of our deep learning framework. To assess the performance of different models, we tested them on the provided multi-classification dataset using hyperparameter, ablation and architectural testing.

1.2 Motivation

In the recent boom of artificial intelligence, deep learning frameworks such as PyTorch [1] have gained significant popularity through their easy-to-use Pythonic interfaces. Frameworks like these have become increasingly complex due to exploding demand, the need to support legacy design choices, and the necessity of supporting GPU acceleration. Here, we attempt to replicate the success of these frameworks using a pure CPU implementation and produce a bare-bone version of these frameworks. As a result, our implementation greatly resembles the interface of PyTorch and is easier to read and understand for didactic purposes. We hope that the framework can provide insight into the inner workings of automatic differentiation and further our understanding of the inner workings of neural networks.

2 Method

2.1 Datasets

We are provided a multi-class dataset with unknown context. It consists of 50,000 training samples and 10,000 test samples with 128 features. There are 10 classes evenly distributed in both the training and test sets. The target is provided as integer labels $y \in [0, 10)$.

2.2 Data Preprocessing

The limited context in the data set restricts the preprocessing options. The features are centred around 0, with variance decreasing from the first to the last, suggesting prior preprocessing with techniques like PCA. As a result, we only applied min-max scaling and standard scaling to test if there is an improvement in performance. Formally, the formula for min-max and standard scaling features x are

$$x_{\text{min-max}} = \frac{x - \min x}{\max x - \min x} \text{ and } x_{\text{standard}} = \frac{x - \mu}{\sigma}$$

respectively where μ is the mean and σ is the standard deviation.

2.3 Implemented Components

2.3.1 Tensor

The `Tensor` class wraps around numpy tensors and implements the data and method needed for automatic differentiation. There are 3 public attributes

1. `val` - holds the value of the tensor as a numpy array. Computations of forward and backward passes use and update this value,
2. `requires_grad` - indicates whether or not the tensor requires gradient computation. Critically, if a tensor requires gradient, then tensors that are computed using this tensor should also require gradient.
3. `grad` - holds the accumulated gradient of the tensor. As seen in the algorithm below, the backward function adds to this attribute.

There are two private attributes that represent the computation graph:

1. `_backward` - stores a function to update the gradient of the input tensors.
2. `_prev` - holds a Python set of previous nodes in the computation graph for backpropagation.

For example, Figure 1 shows the computation graph for a simple linear model with weights w and bias b. Tensors are represented as circles and operations are represented as squares. In this computation graph, tensor z stores the backward function for matrix multiplication with x and w as its previous nodes, and o stores the backward function for addition with z and b as its previous nodes. Notice that in this specific case, the graph happens to be a tree, however computation graphs are only guaranteed to be directed acyclic graphs (DAG) in general.

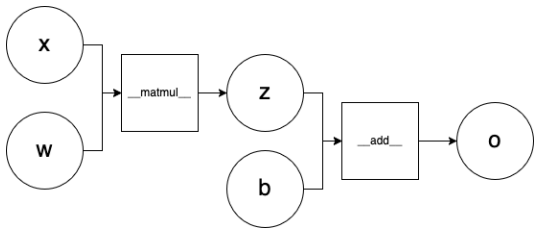


Figure 1: Example computational graph for a linear layer.

The `Tensor` class only implements some of the fundamental functions, such as Python operators `__matmul__`, `__add__`, and `__sub__`, as well as aggregating methods such as `sum` which sums over all entries in the tensor. These functions are essentially wrappers over the NumPy operations on the `val` attribute but also implement the backward function. All functions are implemented in pairs of forward and backward functions that follow the pattern in Algorithm 1. The parts coloured in red are specific to the function we are implementing. Using the Matrix Cookbook [2], we can fill in the forward functions and their respective gradients as per the following Table 1.

Python method name	$Y = f(X_1, \ldots, X_n)$	$\frac{\partial L}{\partial X_i} = g(\frac{\partial L}{\partial Y})$
`__matmul__`	$Y = AB$	$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Y} B^T$ $\frac{\partial L}{\partial B} = A^T \frac{\partial L}{\partial Y}$
`__add__`	$Y = A + B$	$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial Y}$
`__neg__`	$Y = -A$	$\frac{\partial L}{\partial A} = -\frac{\partial L}{\partial Y}$
`__mul__`	$Y = A \odot B$	$\frac{\partial L}{\partial A} = B \odot \frac{\partial L}{\partial Y}$
`abs`	$Y = \lvert A \rvert$	$\frac{\partial L}{\partial A} = \frac{\lvert A \rvert}{A} \odot \frac{\partial L}{\partial Y}$
`sum`	$y = \sum_i \sum_j a_{ij}$ given $A = [a_{ij}]$	$\frac{\partial L}{\partial A} = \frac{\partial L}{\partial y} I$

Table 1: Gradient for fundamental functions. Only one formula is shown for commutative operations, the other one can be inferred by symmetry. We use $A \odot B$ and $\frac{A}{B}$ to denote element-wise product and quotient respectively.

Algorithm 1 Forward and backward function pattern.

function FORWARD($X_1, X_2, \ldots, X_n$)
  $Y \leftarrow$ TENSOR($f(X_1, X_2, \ldots, X_n)$)
  $Y.requires\_grad \leftarrow \bigvee X_i.requires\_grad$
  function BACKWARD
    for all X_i **do**
      if $X_i.requires\_grad$ **then**
        $X_i.grad \leftarrow X_i.grad + g(Y.grad)$
      end if
    end for
  end function
  $Y.\_backward \leftarrow$ BACKWARD
  $Y.\_prev \leftarrow \{X_i\}$
  return y
end function

To automatically invoke the backward functions defined in all the nodes, we implement a `backward` method in the `Tensor` class as seen in Algorithm 2. The algorithm first computes a topological sorting of nodes in the graph and updates the nodes in reverse order. This guarantees that when a backward function is called for some node, the gradient of the nodes it depends on has been computed. This is required because the computation graph may not be a tree as seen in Figure 2. In this example, the gradient of x can only be computed after the gradients for both o and o' have been computed.

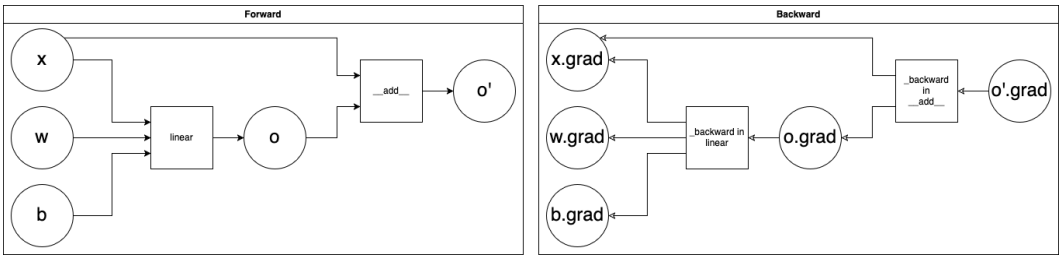


Figure 2: Example computational graph for a linear layer (as before in Figure 1) with an additional skip connection.

Algorithm 2 Invoke backward functions.

function BACKWARD(x)
  $x.grad \leftarrow I$ ▷ The gradient of a constant is the identity
  $T \leftarrow [\,]$ ▷ Topological sorting of computation graph
  $S \leftarrow \{\}$ ▷ Set of visited nodes for graph search
  function TRAVERSE(u)
    if $u \notin S$ **then**
      $S \leftarrow S \cup \{u\}$
      for all $v \in u.\_prev$ **do**
        TRAVERSE(u)
      end for
      $T \leftarrow T + [v]$
    end if
  end function
  TRAVERSE(x)
  for all $i = n, n-1, \ldots, 2, 1$ **do** ▷ Compute gradient in reverse order
    $T[i].\_backward()$
  end for
end function

The basis of backward function is the chain rule in multi-variable calculus. Suppose that L is the final loss scaler that we invoke the `backward` function on. We are interested in the gradient of every tensor that we are updating with respect to L. Suppose that an entry x of some tensor contributes to $z_1, z_2, \ldots, z_n$, formally, $L = f(z_1(x), z_2(x), \ldots, z_3(x))$ is a vector-valued function of x. Then, by the chain rule,

$$\frac{dL}{dx} = \sum_i \frac{\partial L}{\partial z_i} \cdot \frac{d z_i}{d x}.$$

This means that to compute the gradient for some value L w.r.t x, we only have to know the partial derivatives of L w.r.t z_i's to and the ordinary derivative of z_i w.r.t. x. For our algorithm, we update the gradient of x as a function of x, z, and the gradients of each z_i, irrespective of the tensors that comes afterwards in the computation graph. Note that we assume the gradient of each z_i is computed when we update x, hence the need for topological sorting.

2.3.2 Layer

Inspired by `torch.nn.Module` [1], we create `Layer` classes that implement more complicated operations than methods of the `Tensor` class. This has several benefits. First, it allows us to bundle common operations together. For example, the commonly used `Linear` class holds both the weight and the bias tensors - W and b and performs $Wx+b$. Moreover, it allows for more optimisation. For example, the cross-entropy loss integrates softmax operations and then computes the gradient using a shorthand. It also allows us to code the neural networks at a higher level of abstraction, enabling a more intuitive overview and reducing the likelihood of implementation errors.

Automatic Parameter Tracking. The base class provides a `get_all_tensors` method that recursively yields all trainable parameters (instances of the `Tensor` class) stored in the layer or its sub-layers. This mechanism mirrors PyTorch's `parameters()` method, and is essential for weight initialisation and optimisation.

Training and Evaluation Modes. Certain layers like `Dropout` and `BatchNormalisation` behave differently during training and inference. The `Layer` class exposes a `train` method that toggles a boolean flag `self.training` and propagates this mode to sub-layers, ensuring consistent behaviour during model evaluation or deployment. The flag is useful for the dropout and batch normalisation layers which have different behaviour depending on whether the program is training or testing.

Implemented Layer Types. We implement a variety of common neural network layers as subclasses of `Layer`:

- `Linear` implements an affine transformation $Y = XW + b$, holding both the weight matrix W and bias b as trainable `Tensor`s. Back-propagation is handled by the tensor magic methods.
- `ReLU` and `LeakyReLU` are non-linear activation layers that apply an element-wise transformation and implement custom back-propagation via the `_backward` method.
- `Dropout` randomly masks activations during training to prevent over-fitting. It creates a binary mask and rescales the input accordingly.
- `BatchNormalisation` standardises each feature dimension over the mini-batch, then applies learnable affine transformations. It computes mean and variance at runtime, and propagates gradients through the normalisation step.
- `Sigmoid` provides common nonlinearity used in the output layer for binary classification problems but can also be used as the activation function for hidden layers.

Loss Functions. We also define loss functions as layers:

- `MeanSquaredError` calculates the average squared difference between predicted and ground-truth values, suitable for regression tasks.
- `CrossEntropyLoss` combines softmax and log-loss for classification, optionally supporting label smoothing for improved generalization. Its gradient is computed efficiently using the softmax-logits shortcut without explicit softmax layers.

Composite Networks. The `Composite` class bundles multiple layers into a single forward pass. It provides a clean way to express feedforward networks by chaining layers in a list. It also overrides `train` and `get_all_tensors` to propagate iteratively for all layers in the list, maintaining consistent state and parameter access.

Design Benefits. This object-oriented design offers multiple advantages:

- **Readability:** Neural networks are constructed with intuitive syntax, mimicking PyTorch notation.
- **Extensibility:** New layer types can be implemented by simply subclassing `Layer` and overriding `forward` with implemented forward computation and internal `_backward` implementation.
- **Encapsulation:** Parameters, operations, and backpropagation logic are tightly scoped to each layer, reducing bugs.
- **Efficiency:** Some layers, such as `CrossEntropyLoss`, fuse multiple operations (softmax and log-loss) for numerical stability and speed.

Overall, this layered abstraction helps separate concerns between low-level tensor operations and higher-level model logic, enabling flexible experimentation and clean implementation of deep learning models.

2.3.3 Optimiser

To enable training of neural networks through gradient-based optimisation, we define an `Optimiser` base class, along with two concrete implementations: `GradientDescentOptimiser` (stochastic gradient descent, SGD) and `AdamOptimiser`. These optimisers are responsible for updating model parameters (tensors) based on their gradients, which are computed during back propagation.

The `Optimiser` class takes in a list of trainable tensors (typically weights and biases) and provides a `zero_grad` method to reset their gradients to zero before the next backward pass. This is analogous to PyTorch's `optim.zero_grad()` and helps avoid gradient accumulation across multiple forward-backward passes.

Gradient Descent Optimiser. The `GradientDescentOptimiser` implements standard stochastic gradient descent (SGD), one of the most fundamental optimisation algorithms. The update rule is:

$$\theta \leftarrow \theta - \eta \cdot \nabla_\theta L$$

where θ is the parameter (e.g., a weight), η is the learning rate, and $\nabla_\theta L$ is the gradient of the loss function with respect to θ. An optional **weight decay** term can be added, acting as L2 regularisation. This modifies the gradient as follows:

$$\nabla_\theta L \leftarrow \nabla_\theta L + \lambda \cdot \theta$$

where λ is the weight decay coefficient. This discourages overly large weights and can help reduce over fitting.

Adam Optimiser. The `AdamOptimiser` implements the Adam (Adaptive Moment Estimation) optimisation algorithm as described in [3]. Adam combines ideas from momentum and RMSProp by maintaining:

- A first moment estimate (moving average of the gradient) and
- A second moment estimate (moving average of the squared gradient)

Each parameter update is computed as:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t \quad v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

where g_t is the current gradient, and β_1, β_2 are decay rates for the moving averages. These are then bias-corrected:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$

Finally, the parameter update is performed with:

$$\theta \leftarrow \theta - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

The Adam optimiser in our implementation tracks the moments using dictionaries keyed by tensor objects and updates the parameters accordingly. It also supports weight decay in a similar manner to SGD.

Design Notes. The optimisers operate directly on `Tensor` objects and assume that gradients have already been computed via back-propagation (see 2.3.1). This modular framework makes it easy to plug in different training algorithms without modifying the core training loop logic. These classes can be easily extended to support learning rate schedules or other variants such as AdamW.

2.4 Architectures

There's no evidence or context suggesting that the data is of any particular type, like image, text, or audio. Therefore, we only experimented with multi-layer perceptrons instead of more complex convolution or recurrent neural networks. The model consists of one or more layers of linear transformations with activation functions such as rectified linear unit (ReLU), leaky ReLU, or sigmoid. We also tried models that include batch normalisation and dropout layers between the linear layers.

3 Experiments & Results

3.1 Experiment Structure

In this experiment, we conducted an architecture comparison to understand how the different modules we created affected the accuracy and runtime of a given model on our dataset. Using this information, we constructed the optimal model to maximise our accuracy within a reasonable runtime, and undertook hyperparameter testing to see which combination of epoch size, batch size, normalisation strategy, learning rate, weight decay, and optimiser would further increase our accuracy. After identifying the optimal model, we conducted an ablation study to identify what the different components of our model were doing, and how they impacted on the final accuracy of the model. We also evaluated the performance of the models based on their training time, as well as their inference time, to determine whether any increases in accuracy were justified by potential increases in training/inference time.

3.2 Architecture Comparison

To determine the best architecture for our model, we tested models of different depths (number of layers) and widths (number of neurons in each layer). These numbers include the input layer but exclude the output layer (so for example, a 2 layer model in this context means 1 input layer, 1 hidden layer, and 1 output layer). To test the optimal depth of our model, we fixed the width of the model's hidden layers to be 128 neurons and tested the performance of 1, 2, 3, 4 and 5 layers. Similarly, to test the optimal width of our model, we fixed the depth to 2 layers and set each hidden layer to contain 128, 192, 256, 384, and 512 neurons.

We also tested the effects of different activation functions applied to neurons in our networks. This was done by setting up a simple 2-layer network with 192 nodes, and then identifying how a lack of an activation function compared to using ReLU, Leaky ReLU, and Sigmoid on the neurons. The effects of dropout were also analysed by creating a 2-layer network, and placing a dropout layer between the hidden layer and the output layer, with dropout chances ranging from no dropout, 20%, 40%, 60%, and 80%. We also considered how dropout would affect accuracy in a 3-layer network across the same percentage chances, as a more complex neural network with dropout for regularisation could have proved to be a better overall model. Another piece of architecture we tested was the batch normalisation layer. Batch normalisation was applied to the output of all hidden layers, and was compared to a control model that did not contain batch normalisation.

To ensure fairness between the architecture tests, the same hyperparameters of number of epochs, batch size, normalisation, learning rate, weight decay and optimiser were all kept the same. The effects of changing these were investigated later in 3.3.

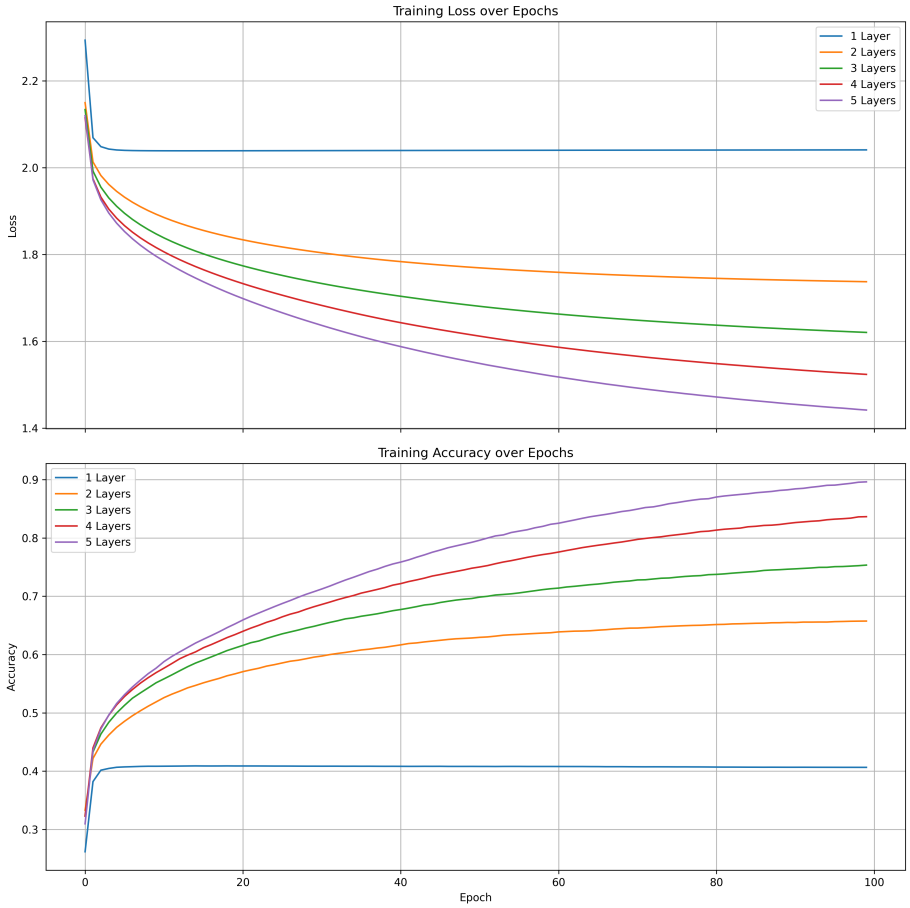
The following default values were used:

Hyperparameter	**Value**
Optimiser	`AdamOptimiser`
Weight Decay	1×10^{-5}
Normalisation	`standard_scale`
Learning Rate	0.001
Epochs	100
Batch Size	64

Table 2: Model training hyperparameters.

3.2.1 Depth Results

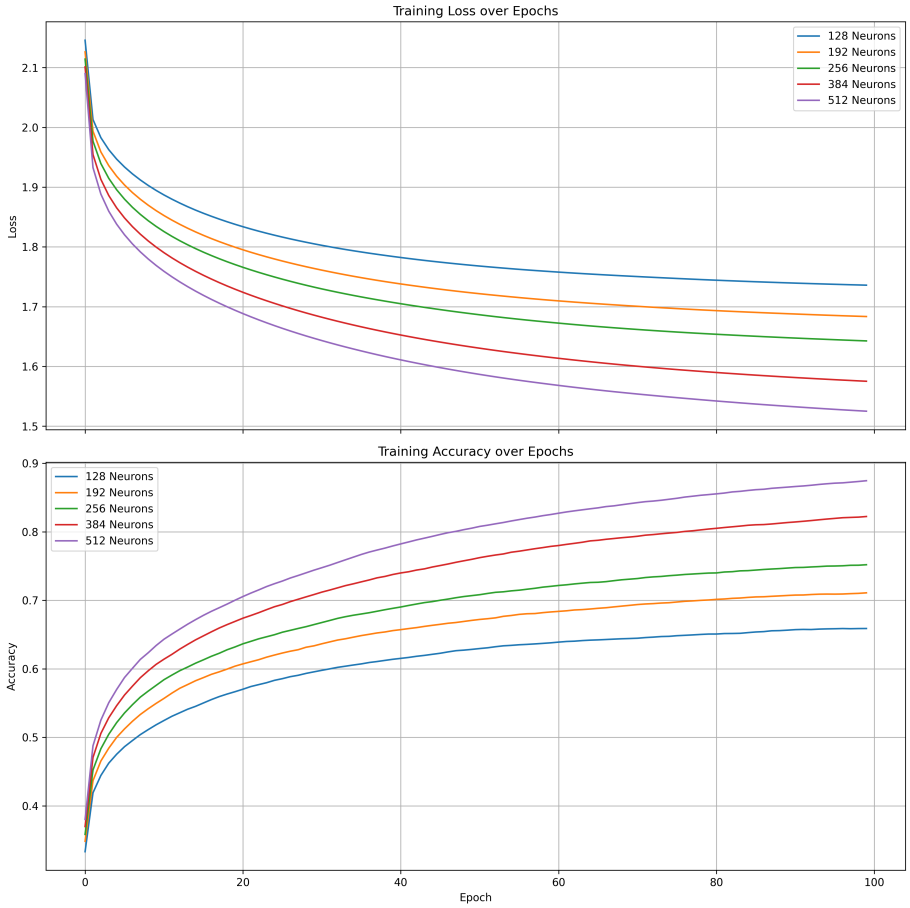
Model Depth	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
1 Layer	2.0408	40.64%	2.0423	40.28%	5.46	0.00145
2 Layers	1.7371	65.74%	1.9017	51.95%	15.76	0.00733
3 Layers	1.6202	75.35%	2.0044	47.88%	25.55	0.0133
4 Layers	1.5236	83.66%	2.1322	44.46%	35.73	0.0194
5 Layers	1.4414	89.63%	2.2409	42.99%	46.08	0.0271



These depth results indicate that 2 layer models tend to perform best and achieve the highest levels of test accuracy. Adding further layers seems to increase training accuracy due to the ability to learn more complex relationships, however the corresponding lower test accuracy for the deeper models implies the presence of overfitting. In addition, added layers increase both training time and inference time, as there are more calculations for the model to do both when backpropagating gradients and when predicting new unseen data, meaning that minimising the number of layers is important to ensure the practical usability of the model.

3.2.2 Width Results

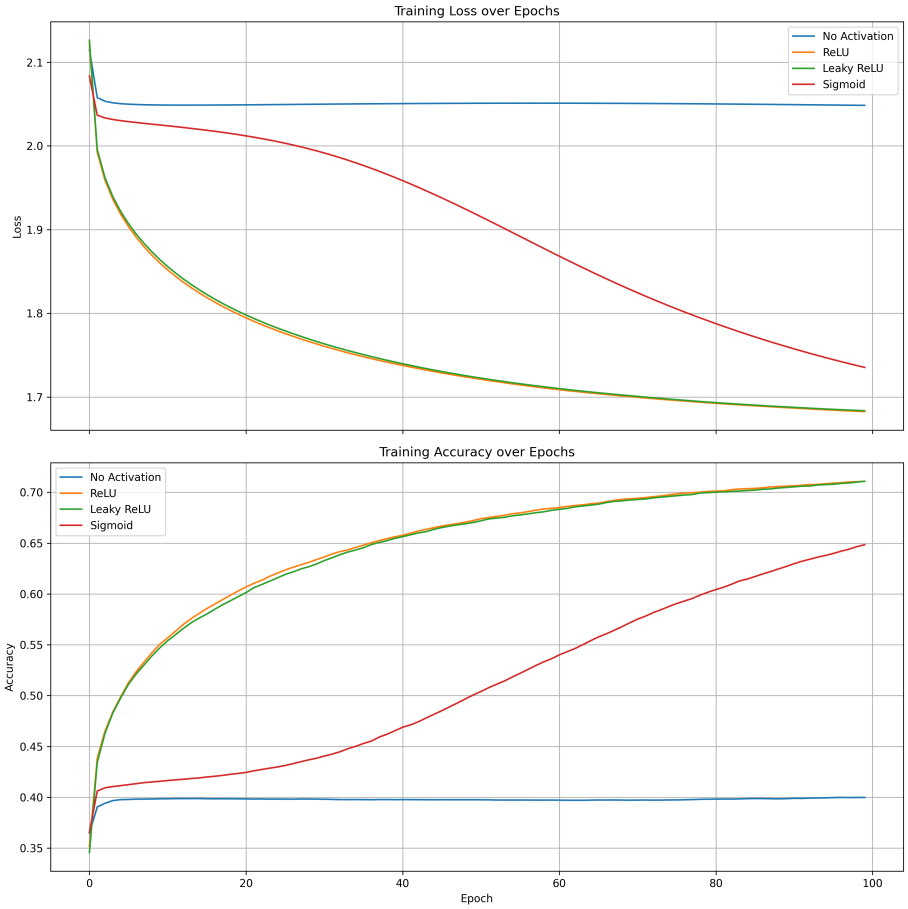
Model Width	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
128 Neurons	1.7360	65.89%	1.9102	51.44%	16.34	0.00923
192 Neurons	1.6835	71.09%	1.9232	51.86%	21.27	0.0159
256 Neurons	1.6427	75.20%	1.9480	51.77%	26.19	0.0161
384 Neurons	1.5752	82.23%	2.0033	50.58%	38.88	0.0184
512 Neurons	1.5251	87.46%	2.0598	48.14%	102.97	0.0297



These width results show the best width to be 192 neurons, with a test accuracy of 51.86%. Similar to the depth test, increasing the number of neurons beyond this point introduces overfitting, as seen by the higher training accuracy coupled with lower test accuracy in the experiments with more neurons. Additionally, increasing the number of neurons exponentially increases training time, as seen clearly by the huge jump in training time from 384 to 512 neurons, going 38.88 seconds to 102.97 seconds, indicating how potential increases in accuracy from increased neurons needs to be balanced with increased training and inference times.

3.2.3 Activation Function Results

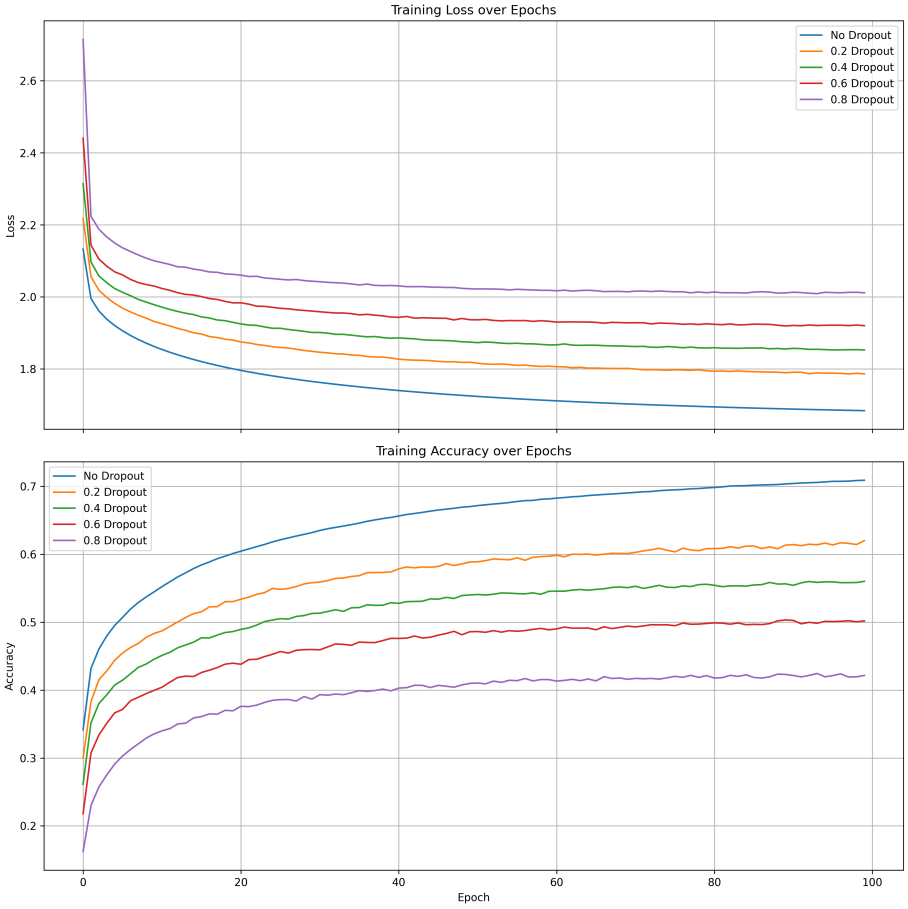
Activation Function	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
No Activation	2.048	39.98%	2.046	39.72%	19.3	0.0124
ReLU	1.682	71.09%	1.933	51.37%	21.3	0.00994
Leaky ReLU	1.684	71.09%	1.926	51.92%	25.8	0.0152
Sigmoid	1.735	64.85%	1.899	49.77%	21.7	0.0103



These results show out of the four activation functions we tested, LeakyReLU was the best performing one at a test accuracy of 51.92%. As expected, a lack of activation function performed the worst, as it removed the ability of the neurons to express non-linearity. Another expected result was the training time of the no activation model being much lower than that of the models with activation functions, as that contains less computations for the model to do. Interestingly however, the ReLU activation had the lowest inference time of all the models, even the no activation model - this is likely due to setting negative values to zero, making future computations using that output very fast as any multiplications just become zero.

3.2.4 Dropout Results

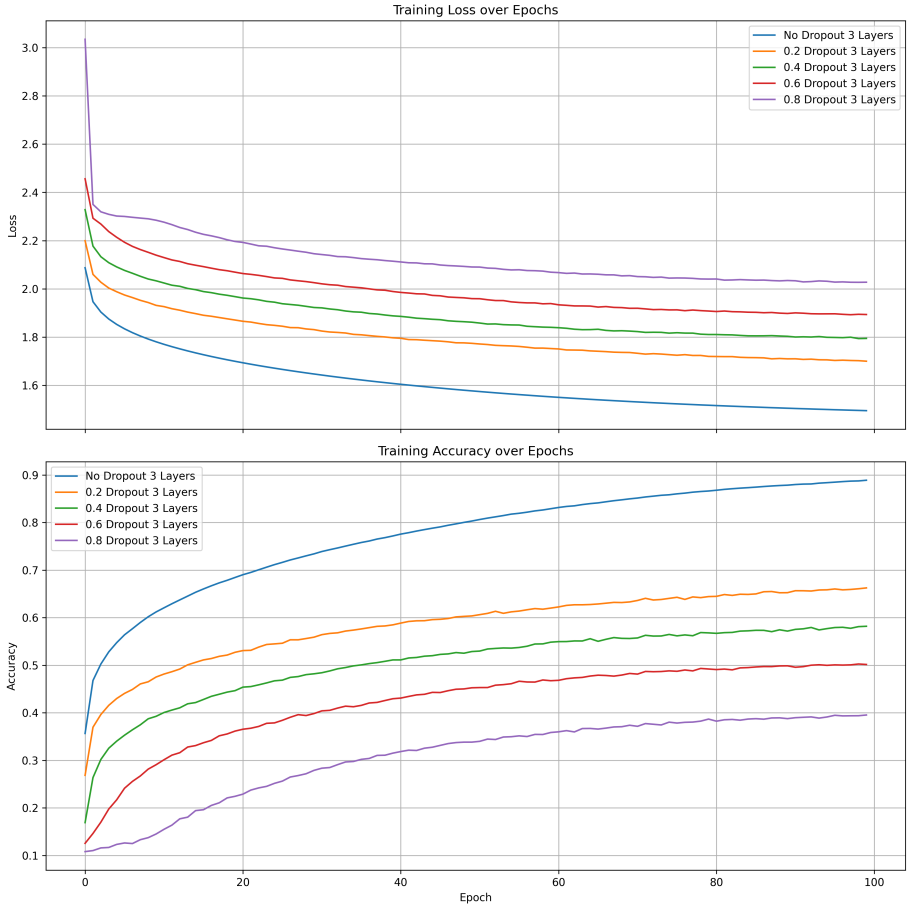
Dropout Proportion	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
No Dropout	1.684	70.91%	1.924	51.56%	22.0	0.00948
0.2 Dropout	1.786	62.01%	1.912	51.25%	32.1	0.0338
0.4 Dropout	1.853	56.02%	1.931	49.51%	35.1	0.0448
0.6 Dropout	1.920	50.19%	1.964	46.98%	34.8	0.0423
0.8 Dropout	2.011	42.16%	2.025	41.01%	31.6	0.0375



When using a 2-layer neural network, the inclusion of dropout actually harmed test accuracy, while not including it provided the best test accuracy of 51.56%. This may be due to the simplicity of a 2-layer model not being able to learn very complex patterns, meaning that the benefits of regularisation that dropout provides were significantly outweighed by the cost of certain nodes not being able to learn patterns in the first place. The inclusion of dropout also altered inference and training times slightly - dropout itself increased training and inference times, but including dropout that favoured dropping out more or dropping out less values (0.2 dropout and 0.8 dropout) seemed to increase this time by less than dropout amounts that favoured a more even chance between choosing to drop out values or not (0.4 and 0.6 dropout).

3.2.5 3-Layer Dropout Results

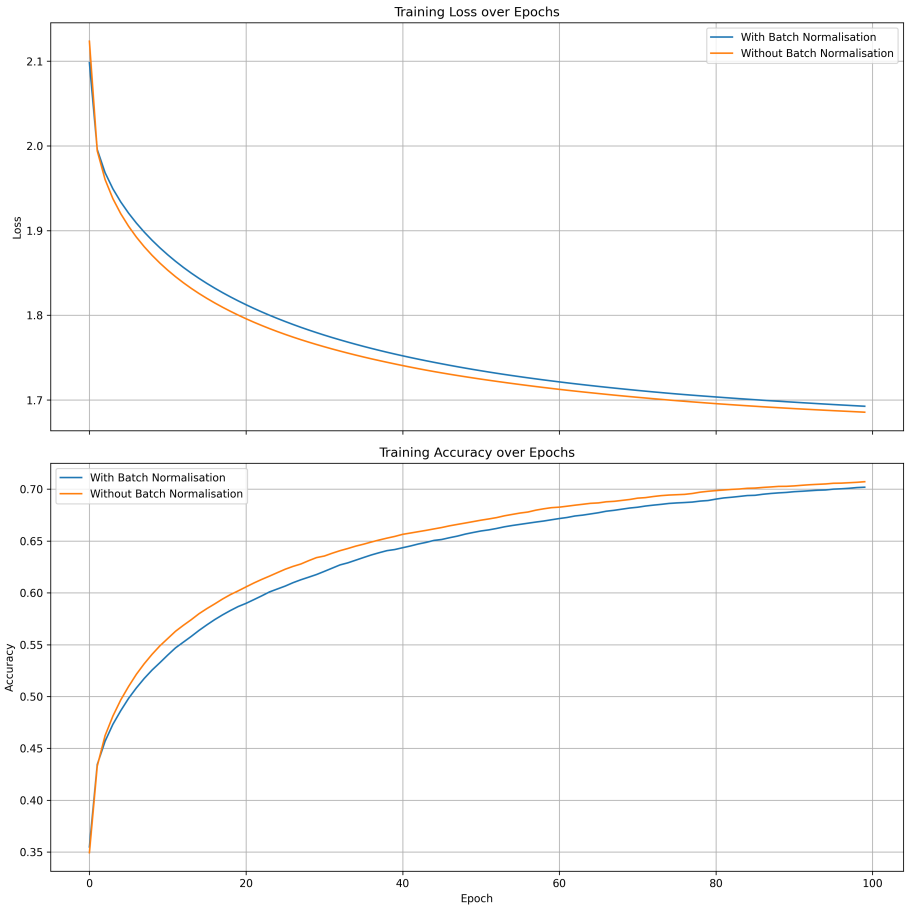
Dropout Proportion (3 Layers)	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
No Dropout	1.495	88.89%	2.134	44.91%	49.1	0.0229
0.2 Dropout	1.700	66.25%	1.917	49.97%	75.9	0.0777
0.4 Dropout	1.794	58.18%	1.908	49.67%	78.8	0.0807
0.6 Dropout	1.893	50.15%	1.949	46.22%	78.7	0.0806
0.8 Dropout	2.027	39.52%	2.040	38.13%	65.0	0.0756



The results of dropout applied to a 3-layer network supports the theory posited for the 2-layer dropout results. As the model was now more complex with 3 layers, it was able to learn more complex patterns, but could potentially suffer from some overfitting. As such, adding dropout with a relatively small dropout percentage helped add some regularisation to the model, with a percentage chance of 20% producing the best test accuracy of 49.97%. However, given that we found earlier that 2 layers tended to perform best for this task, we did not include dropout layers in our final model as the results above indicated that having a 2-layer model without dropout performed better than a 3-layer model with dropout. Of note are the increases in training time when including dropout - if researchers are concerned with training time, they must be careful with adding dropout as it seems to significantly increase training time.

3.2.6 Batch Normalisation Results

Model	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
With Batch Normalisation	1.693	70.18%	1.921	51.56%	37.5	0.0253
Without Batch Normalisation	1.686	70.71%	1.933	50.63%	25.6	0.0177



Batch normalisation seems to help the model improve its test accuracy by approximately 0.93%. This could primarily be due to the batch normalisation layers addressing internal covariate shift, although the limited improvement on test accuracy is likely due to the fact that the model this was tested on was very shallow (2 layers). A shallow model means the compounding effect of multiple layers on producing covariate shift is minimised, thus minimising the effectiveness of batch normalisation on improving accuracy. Indeed, this is further supported by the fact that during training, including batch normalisation actually *decreased* accuracy, lending credibility to the idea that the improvement in test accuracy was potentially just a matter of chance. However, this could also be a sign of less overfitting, given that despite this lower level of training accuracy, the model was able to generalise better to the test set with batch normalisation. Additionally, including batch normalisation had the expected result of increasing training time, as it introduces more computations and gradients for the model to learn and backpropagate over. Nevertheless, we went on to include batch normalisation in the hyperparameter testing given that it improved test accuracy in these experiments and the increase to inference times was not that large.

3.3 Hyperparameter Study

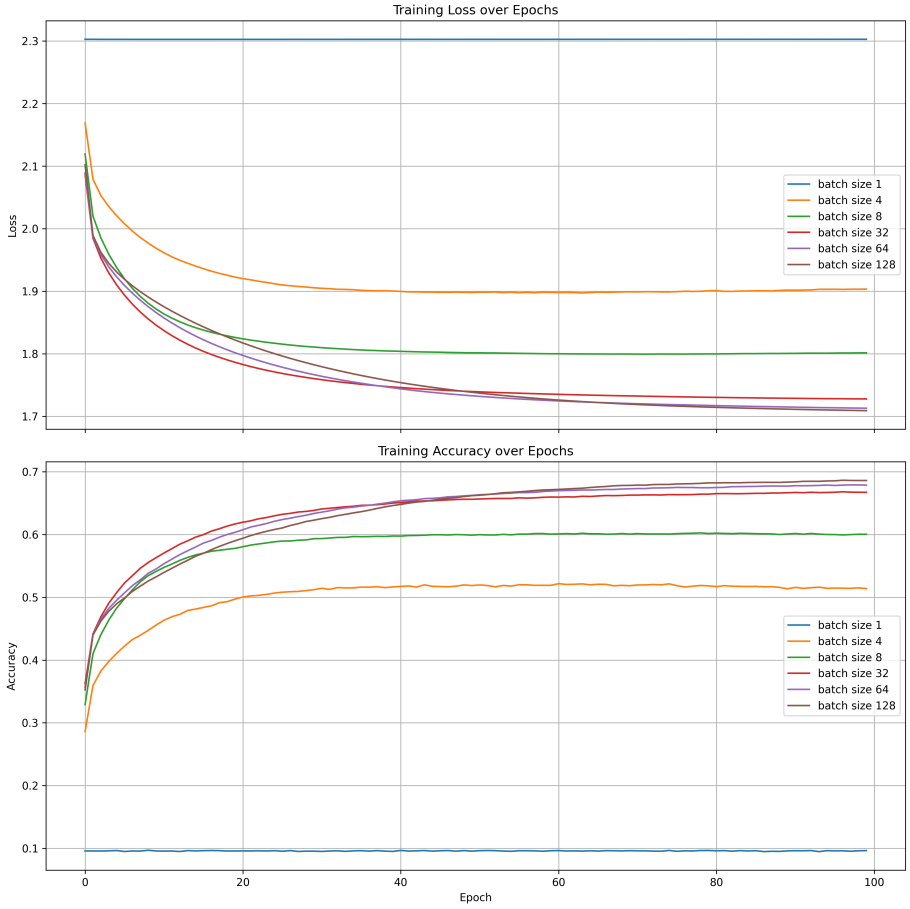
In line with the above results, we decided to conduct hyperparameter testing on a 2-layer model with a width of 192 neurons, the output of which was fed through a batch normalisation layer and then through an activation of LeakyReLU, without any dropout. To determine the optimal number of

epochs, batch size, learning rate, and weight decay for our model, we swept through several different reasonable values for each of these hyperparameters and then selected the values that maximised accuracy, while also keeping in mind training and inference times. For the normalisation method, we analysed both min-max scaling as well as standardisation, while for the optimiser we looked at traditional mini-batch SGD as well as the Adam optimiser.

Hyperparameter testing was performed by having several default values, which were then swapped out depending on the individual hyperparameter being tested. The base hyperparameters were 100 training epochs, batch size of 32, standard scale normalisation, a learning rate of 0.001, a weight decay rate of 0.001, and optimisation using the Adam optimiser.

3.3.1 Batch Size Results

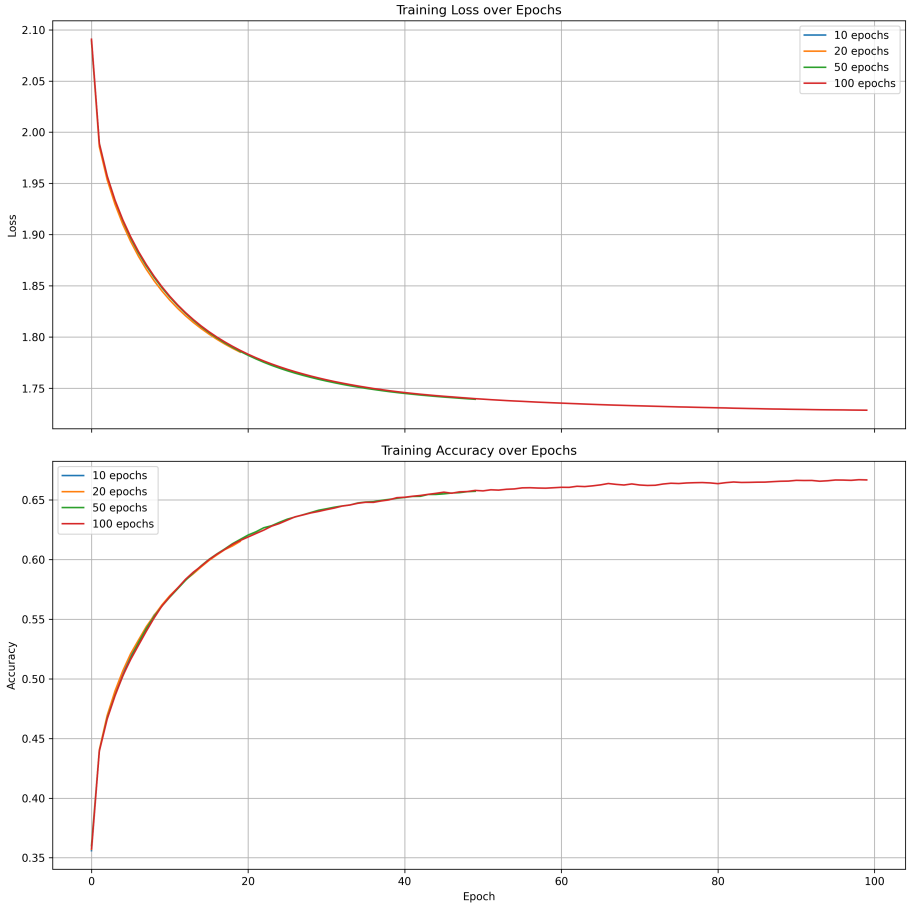
Batch Size	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
1	2.303	9.63%	2.303	10.00%	1350	0.0199
4	1.903	51.38%	1.934	48.00%	300	0.0187
8	1.802	60.04%	1.895	52.08%	159	0.0178
32	1.728	66.73%	1.860	54.55%	52.4	0.0181
64	1.713	67.88%	1.858	55.39%	34.2	0.0238
128	1.709	68.61%	1.859	54.22%	31.3	0.0204



The above results show that generally speaking, increasing the batch size increased test accuracy up to a batch size of approximately 64, after which it tended to decrease. This is consistent with prior research [4] that finds smaller batch sizes seem to work best, and that larger batch sizes tend to decrease the number of different acceptable learning rates, which can lead to lower test accuracies. Larger batch sizes also tended to decrease training time, as there were less backpropagation passes with more examples in each batch, however significantly increasing the batch size beyond 64 did not produce much better training times (compare the decrease in training time from a batch size of 32 to 64 going from 52.4 to 34.2, compared to going to 128 which only reduced it further to 31.3). This further solidifies 64 as the ideal batch size, as the improvements in training time diminish beyond this point.

3.3.2 Number of Epochs Results

Epochs	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
10	1.848	56.09%	1.893	51.36%	5.21	0.0266
20	1.785	61.52%	1.869	53.90%	10.4	0.0238
50	1.739	65.72%	1.866	53.95%	25.9	0.0249
100	1.728	66.67%	1.863	54.31%	52.2	0.0205

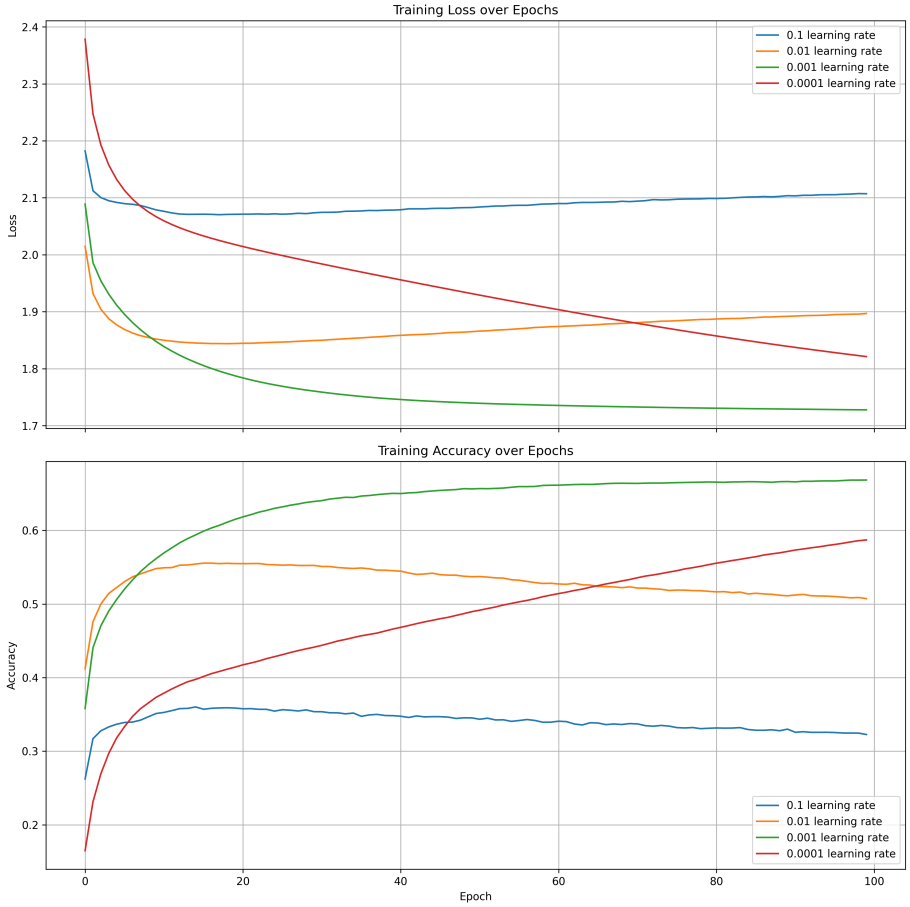


The results above indicate that 100 epochs was the optimal number of epochs to use during training. This is consistent with the traditional idea that the more epochs a model runs while training, the more it will be able to adjust its parameters to get closer to the local minimum of the loss function,

thus making it more accurate. However, the results also show diminishing returns as the number of epochs increases, which is a result of the model parameters being very close to producing the loss function local minimum, and thus not changing by much when optimising. If parameters do not change by a lot, this means that the results produced by the model will also not change a lot, thus not improving accuracy by much either. For this experiment, training time is not a valid measure as logically the more epochs a model trains on, the longer it will take to train, however of interest are the inference time results, which show that larger epochs produce smaller inference times. Given the very small difference in inference times however, and the inconsistency in this trend (50 epochs produces a longer inference time than 20 epochs), this may just be a result of chance, however it may be interesting to investigate in future research.

3.3.3 Learning Rate Results

Learning Rate	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
0.1	2.107	32.27%	2.124	32.47%	49.2	0.0123
0.01	1.897	50.74%	1.894	51.17%	50.5	0.0146
0.001	1.728	66.85%	1.863	53.74%	52.5	0.0168
0.0001	1.821	58.71%	1.891	51.34%	52.6	0.0171

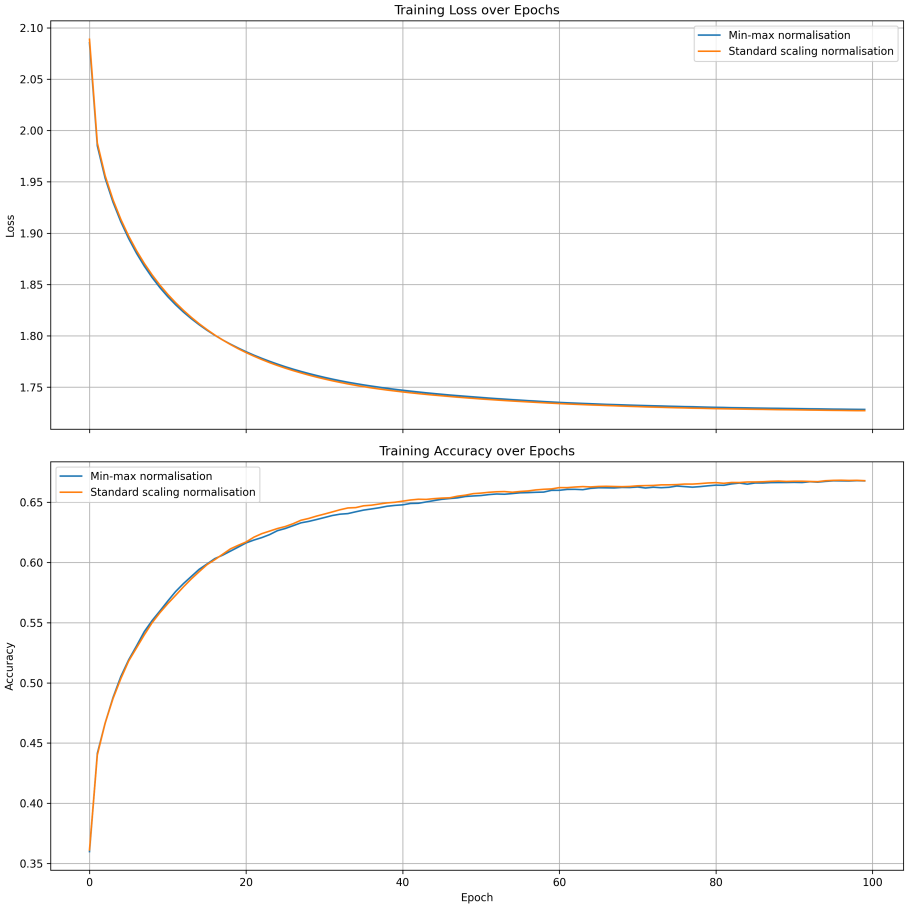


For the tested learning rates, a learning rate of 0.001 proved to be optimal, providing a test accuracy of 53.74%. While this learning rate provided the highest accuracy, the graph of training accuracy over epochs shows that while the learning rate of 0.001 was plateauing, the 0.0001 learning rate was

still steadily increasing, and was merely cut off early by the maximum number of training epochs being 100. This therefore suggests that given more training epochs, the 0.0001 learning rate may have performed better, and is an example of a classic trade-off in machine learning tasks - while smaller learning rates may be able to get closer to the local minimum of the loss function, they take much longer to do so due to their smaller step size, meaning that researchers must make a trade-off between training time and accuracy. Of note as well is the decreasing training accuracy of the 0.1 and 0.01 learning rates over the 100 epochs - this indicates overshooting, where the learning rate being too high causes the gradient update to be too large as well, in turn moving past the local minimum, and needing to move back in the opposite direction to achieve the local minimum. Interestingly, as learning rates increase, so do training times and inference times. This may be due to random chance, however the consistency in this pattern indicates an area for future research.

3.3.4 Normalisation Results

Normalisation	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
Min-Max scaling	1.728	66.76%	1.861	54.15%	52.3	0.0171
Standard scaling	1.727	66.78%	1.862	54.17%	52.6	0.0173

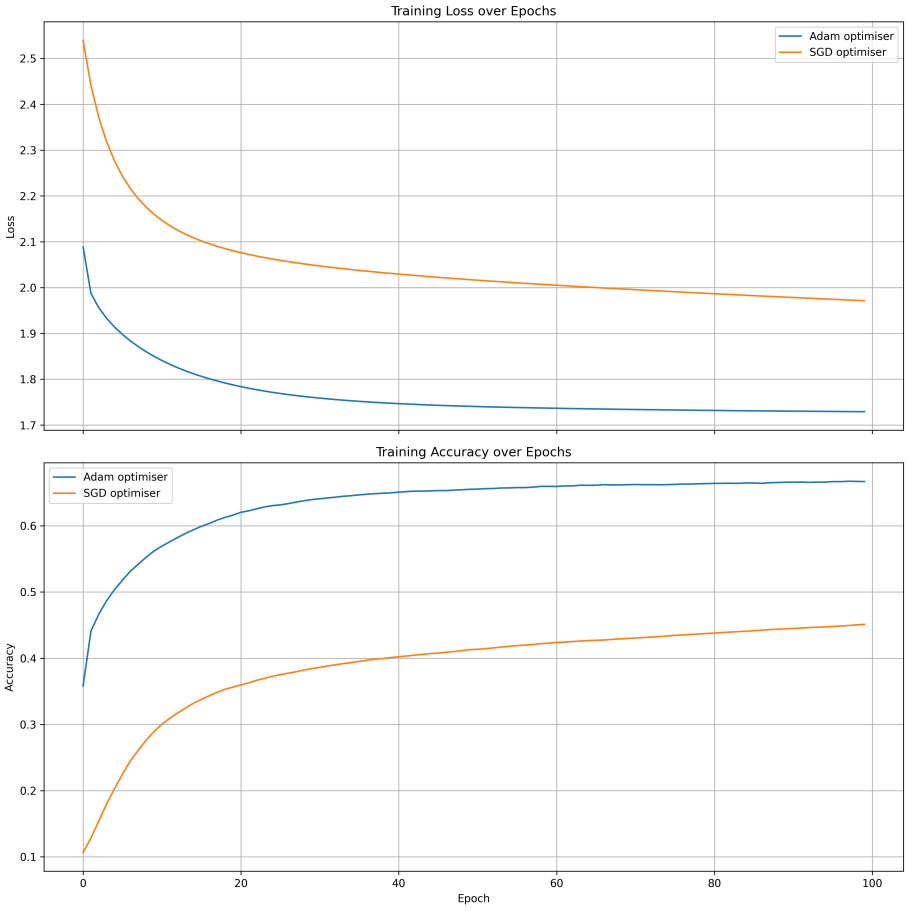


We tested two different normalisation techniques for the data on this model - traditional min-max scaling, and standard scaling. Min-max scaling was defined by $x'_i = \frac{x_i - min(x)}{max(x) - min(x)}$, while standard scaling was defined by $x'_i = \frac{x_i - \bar{x}}{\sigma}$ (where x'_i represents a normalised data point, x_i is a non-normalised data point, x is the set of all inputs, $\bar{x}$ is the average of all inputs, and σ is the standard deviation of the inputs). The two normalisation methods produced extremely similar results, only differing in their test accuracy by 0.02%, and differing in their training and inference times by only 0.3 and 0.0002 seconds respectively. This indicates that either of the normalisation methods are

valid to use in these types of classification problems, however purely due to the slight advantage in accuracy that standard scaling produced, we went ahead with using standard scaling in the ablation study.

3.3.5 Optimiser Results

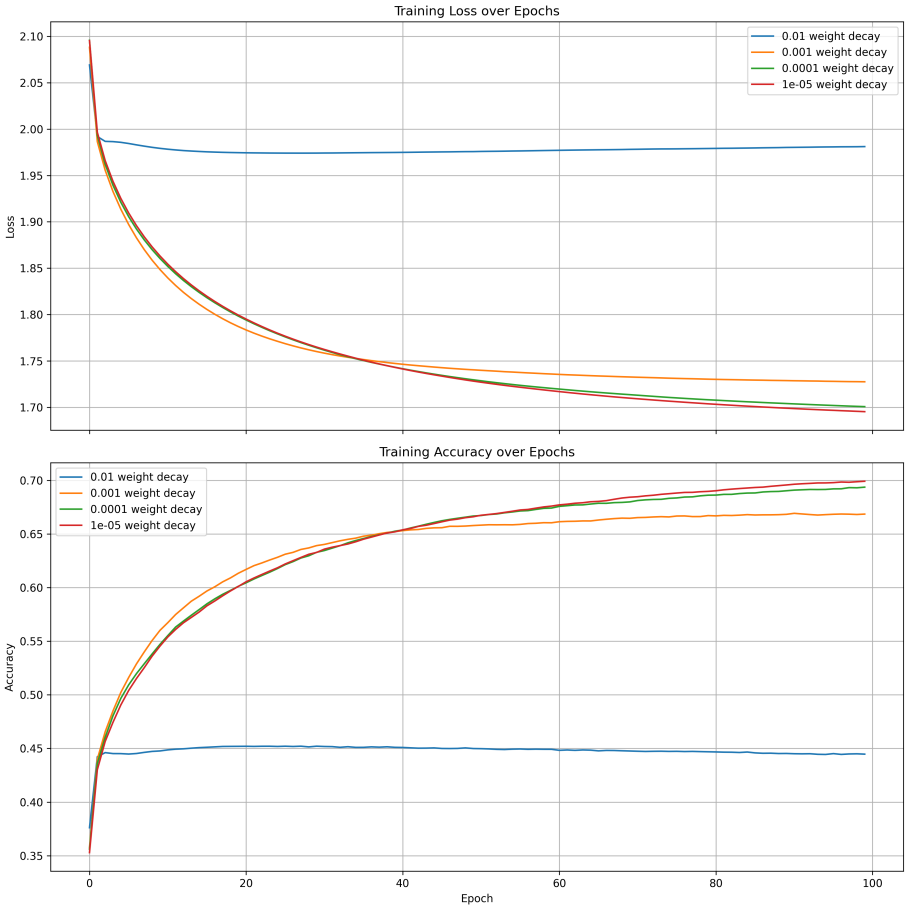
Optimiser	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
Adam Optimiser	1.729	66.69%	1.865	53.86%	53.7	0.0233
SGD Optimiser	1.971	45.12%	1.984	43.69%	38.6	0.0214



The above results show that the Adam optimiser performed better than the SGD optimiser by a significant margin of 10.17% in terms of test accuracy, albeit with a training time 15.1 seconds longer than the SGD optimiser. While the Adam optimiser is clearly the better optimiser and produces much better results, the above numbers indicate that the Adam optimiser might produce a significantly higher amount of overfitting compared to the SGD optimiser, due to the gains in training accuracy being much higher than the gains in test accuracy. This can indicate that while Adam should be the preferred optimiser in classification problems, researchers should be careful when combining Adam and other overfitting-inducing tools and hyperparameters. The increased training time is most likely due to the increased number of computations that Adam does - while SGD just applies the pre-calculated gradient to each parameter, Adam also includes calculations for first and second moments, as well as bias correction [3]. Thus, researchers should take care if there are any other elements in the model that increase the training time by a significant margin, as Adam can add to this.

3.3.6 Weight Decay Results

Weight Decay	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
0.01	1.981	44.47%	1.970	45.58%	51.9	0.0212
0.001	1.727	66.86%	1.863	54.38%	52.4	0.0200
0.0001	1.701	69.38%	1.918	51.76%	53.1	0.0245
1e-05	1.695	69.93%	1.925	51.71%	53.4	0.0209

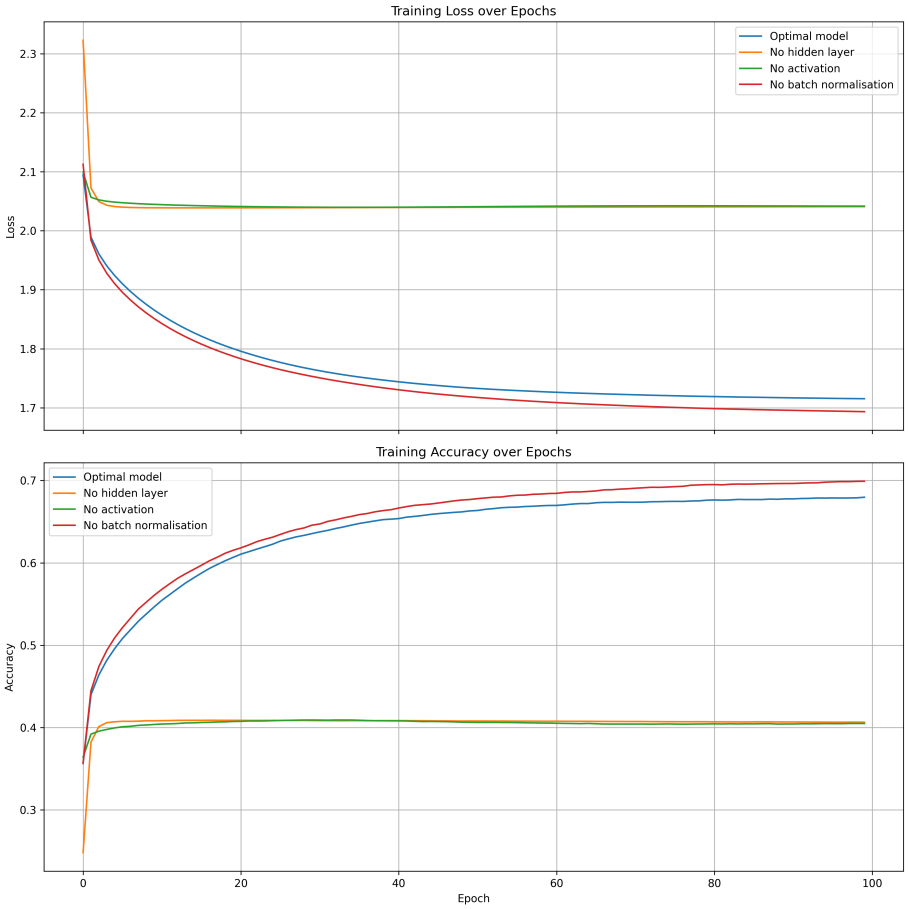


Several different weight decay values were tested to see if they could provide any regularisation benefits to the model. The above results indicate that a weight decay of 0.001 is optimal, providing the highest test accuracy of 54.38%. This weight decay gives the best trade-off between model complexity allowing accurate predictions, and regularisation preventing overfitting. This can be seen through the results of the other values - the weight decays of 0.0001 and $1e^{-05}$ were too small, and thus did not have much of a regularisation effect, as seen by their higher training accuracies than 0.001, but lower test accuracies, indicating overfitting. Meanwhile, the weight decay of 0.01 was too strict, limiting the ability of the model to learn complex relationships, as seen by the relatively low training and test accuracies of 44.47% and 45.58% respectively. Training times seemed to increase as weight decay decreased - this could potentially be due to the regularising effect of weight decay causing some parameters to move closer to zero, meaning calculations relating to those parameters becoming more simple (especially in the case of those parameters being regularised to zero, where any multiplication by 0 would be very quick).

3.4 Ablation Study

After identifying the optimal hyperparameters for the optimal model, we proceeded to conduct an ablation study to understand how the different components of the model affected accuracy. This study was conducted using the following hyperparameters found from the above hyperparameter tests: 100 epochs, batch size of 64, standard scaling normalisation, learning rate of 0.001, weight decay of 0.001, and using the Adam optimiser. Since the optimal model we found was already quite simple, there weren't that many different sections to remove from the model, however even with this limited number of components we were still able to identify some trends and interesting patterns.

Model	Training Loss	Training Accuracy	Test Loss	Test Accuracy	Training Time (s)	Inference Time (s)
Optimal model	1.715	67.96%	1.861	54.45%	37.14	0.0304
No hidden layer	2.041	40.64%	2.042	40.20%	5.60	0.00224
No activation	2.042	40.49%	2.044	40.08%	30.90	0.0207
No batch normalisation	1.693	69.91%	1.869	53.98%	24.2	0.0206



As expected, the optimal model with all the components performed the best with a test accuracy of 54.45%. The next best performing model was the one without batch normalisation at 53.98%. What is interesting about the batch normalisation layer is that it seems to add a very marginal improvement to test accuracy (in the case of these experiments, only 0.47%), which is probably due to the fact that the optimal model is very shallow, while batch normalisation tends to work better for deeper models. This in turn is because batch normalisation attempts to correct for covariate shift, which is most pronounced when the output of a layer goes through several more layers and is constantly shifted, while in a shallow model like ours there is only one hidden layer meaning that the input values

are not shifted that drastically. Nevertheless, the batch normalisation layer does have a positive impact on test accuracy, and also seemingly helps correct for overfitting, as the model without batch normalisation has a higher training accuracy but a lower test accuracy.

Another interesting result is the fact that the model without a hidden layer performed better than the model that contained a hidden layer, but without an activation function, at 40.20% and 40.08% test accuracies respectively. This may be due to the fact that the input data already has a lot of information regarding which class a data point belongs to even without manipulating the data through a hidden layer, which leads to the accuracy produced by the no hidden layer model, while the lower accuracy of the no activation model indicates that activating the input is very vital for that input data to transmit its information regarding the correct class to the next layer, leading to a lower accuracy if that activation is not present. Regardless of these differences, both models show the importance of including both a hidden layer and an activation function in order to learn more complex patterns, as shown by the significantly lower test accuracy compared to the optimal model.

The training times largely reflect the complexity of the models, with more complex models taking longer to train. The optimal model is the most complex model with the most parameters, and thus takes the longest to train at 37.14 seconds, followed by the model without an activation function at 30.9 seconds. The model without an activation function still contains the batch normalisation layer, which is a fairly complex layer causing its longer training time compared to the other models. Following this, the layer containing no batch normalisation drops the training time down to 24.2 seconds, and from this decrease in training time we can see the complexity that the batch normalisation layer introduces. Finally, the model without any hidden layer is the least complex, and thus has a very quick training time of 5.4 seconds. The inference times largely follow this pattern as well, except in the case of the no activation and no batch normalisation models, which have very similar inference times.

3.5 Summary of Optimal Model

In summary, the results above show that the optimal model for this problem and its hyperparameters are as follows:

- 128-Node input layer
- 192-Node hidden layer
- Batch-Normalisation layer
- LeakyReLU activation layer
- 10-Node output layer with Softmax activation

Hyperparameters:

- Number of epochs: 100
- Batch size: 64
- Normalisation: Standard scaling
- Learning rate: 0.001
- Weight decay: 0.001
- Optimiser: Adam Optimiser
- Loss function: Cross Entropy

3.6 Hardware & Software

All of the code was tested on an Apple M1 Max CPU using NumPy 2.2.3 and Python 3.13 (see 6.1 on how to replicate the environment).

4 Discussion

Overall, we successfully built a neural network which was able to classify samples into 10 unique classes. We achieved accuracies of approximately 54% in our best models, which is better than a random classifier with an expected accuracy of 10%.

We also successfully implemented our own deep learning framework with built-in gradient descent and a composable high level design. This has greatly strengthened our understanding of the inner workings of existing established frameworks such as PyTorch.

In our extensive experimentation, we discovered that relatively shallow models perform best for this task at 2 layers deep (including 1 input layer, so 1 hidden layer total), most likely due to the increased complexity from learned relationships in deeper models leading to overfitting. Reasonably wide models of around 192 neurons per hidden layer tended to perform best, with wider models producing lower test accuracy, likely due to increased complexity as well. The LeakyReLU function produced the best accuracy among all activation function variations, likely due to its expressive nature in signalling which neurons are responding to inputs both in the positive direction and in the negative direction by producing slight negative values when the input is negative. The results of using dropout were a bit more mixed - it did not benefit the simple 2-layer model, and this was likely due to the simplicity of the model. A simpler model means that having information not transmit through a node due to dropout leads to information being barred from moving to the output layer, thus minimising the patterns that the model can learn. However, it was theorised that deeper models would benefit from dropout, and this was confirmed by testing on a 3-layer model, which showed that a dropout layer with a 20% chance of dropping values could decently improve accuracy, and as could dropout layers with a 40% or 60% chance. Batch normalisation provided a slight improvement to model accuracy, as shown by both the architecture study and ablation study, however its improvement effects were minimal, most likely due to the shallow nature of our model. Additionally, more complex models (either through more layers, nodes, or complex features such as batch normalisation or dropout) caused both inference and training times to be higher, likely due to the need to optimise more parameters and conduct more calculations.

Our experimentation also determined that a batch size of 64 was optimal for this task. As predicted by some prior research [4], larger batch sizes could actually decrease accuracy, which was seen in our results by the decreasing accuracy obtained from batch sizes of 128. Our research also indicated that larger numbers of epochs were better for test accuracy, which was expected given that this means the model has more time to train. It also confirmed the diminishing returns of increasing the number of epochs, as the gains obtained from moving closer to the local minimum of the loss function becomes smaller as the adjustments needed to the parameters of the model become smaller. We also discovered that a learning rate of approximately 0.001 was best for this task, however lower learning rates such as 0.0001 could be used if a researcher was willing to spend more time to let their model obtain higher accuracy results. The normalisation tests showed that both min-max scaling and standard scaling were viable and produced a largely equivalent test accuracy. Additionally, our tests showed that the Adam optimiser and its momentum adjustments were much better suited to the task of optimisation than the traditional SGD optimiser, although the Adam optimiser may introduce an element of overfitting due to significantly increasing training accuracy by a lot more than test accuracy compared to the SGD optimiser. Finally, we found that a weight decay rate of 0.001 was best for this task, and provided the best mix between generalisation and accuracy, where smaller rates such as 0.0001 sacrificed generality, and higher rates such as 0.01 sacrificed on accuracy to a significant degree. There was a general bias of test accuracy being higher with not enough regularisation rather than too much, as shown by the lower regularisation rates providing similar accuracy levels to the optimal value (e.g. the optimal value of 0.001 giving an accuracy of 54.38% while a weight decay of 0.0001 provided an accuracy of 51.76%), compared to high regularisation rates providing significantly lower accuracy (a weight decay of 0.01 produced 45.58% accuracy). This indicates that if a researcher is unsure about what degree of weight decay to use, they should bias themselves towards lower weight decay values rather than higher ones. In addition to all of this, training and inference times were largely affected by the level of complexity or number of calculations these parameters would induce. For example, lower weight decay rates would produce higher training times, likely due to leaving more non-zero parameters to do calculations with, or higher batch sizes causing lower training times, due to less back propagation passes.

The ablation study we performed showed how different elements of our optimal model interacted to produce the results we achieved. Batch normalisation provided a small boost to accuracy and potentially improved the generalisation ability of the model, while having both a hidden layer and an activation function were vital to learning the relationship of the data. This indicates that the nature of the relationship between the data and the classes is non-linear, as the inclusion of a hidden layer and hence the ability to model non-linear relationships significantly boosted accuracy.

5 Conclusion

Throughout this experiment, we developed a simple deep learning library to address the given multi-classification task. Several different modules were created for this task, including standard linear layers, dropout layers, batch normalisation layers, sigmoid activations, ReLU activations, and LeakyReLU activations. In addition, we implemented several different tools to assist with this task, such as an SGD optimiser, Adam optimiser, cross-entropy loss function, and a back-propagation framework. Using our custom library, we conducted several experiments to identify the best network architecture and hyperparameters for the classification task, and then used this model to achieve an accuracy of approximately 54% on the classification task. This indicates that our model is able to learn the relationship between the data and the required classes, as it outperforms the expected value of a random classifier on this task, which would achieve an accuracy of 10%.

6 Appendix

6.1 Instructions

The code for this assignment can be found at this Github repository here: https://github.com/Edward-Ji/minute-grad/tree/main

Clone the code locally to your device, and check the readme for details.

There are 4 main files that are intended to be run:

- `main.py`, which contains a training loop and test for the optimal model
- `experiments/architecture_experiments.py`, which contains the code used to run the architecture experiments
- `experiments/hyperparameter_experiments.py`, which contains the code used to run hyperparameter testing
- `experiments/ablation_test.py`, which contains the code used to conduct the ablation testing

All other files are modules used by the above scripts to run.

The automatic differentiation framework only depends on NumPy for computation. For experiments, there are optional dependencies `matplotlib`, `tqdm`, and `wandb` for visualisation, a progress meter, and tracking results respectively.

The project uses `uv` for managing dependencies, so you need to install uv first. Then, run the following command to update your local environment:

```
uv sync --no-dev
```

For example, to run the `experiments/architecture_experiments.py` script, simply execute:

```
uv run python experiments/architecture_experiments.py
```

6.2 Individual Contribution

Student ID	**Contribution**
510477226	Introduction and Method sections, some tensor methods and layers
510459679	Parts of Tensor Operations, Experiments, Editing Report
500499357	Batch normalisation, softmax, batch generator, and early Adam optimiser code. Architecture, hyperparameter testing, and results for testing. Results, discussion, and conclusion for the report.

Table 3: Team Members and Contributions

References

[1] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019, pp. 8024–8035. [Online]. Available: http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[2] K. B. Petersen and M. S. Pedersen, "The matrix cookbook," Oct. 2008, version 20081110. [Online]. Available: http://www2.imm.dtu.dk/pubdb/p.php?3274

[3] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *Computer Science*, 2014.

[4] D. Masters and C. Luschi, "Revisiting small batch training for deep neural networks," *ArXiv*, vol. abs/1804.07612, 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:5032969