Table of Contents

Abstract iii

Lists of Figures iv

Lists of Tables v

1 Introduction 1

2 Methods 2

2.1 Pre-processing . . . 2

2.1.1 Min-Max Nomalisation . . . 2

2.1.2 Standardisation . . . 2

2.2 Principles . . . 3

2.2.1 Forward Propagation . . . 3

2.2.2 Backward Propagation . . . 4

2.2.3 Activation Function . . . 4

2.2.4 Weight Initialisation . . . 7

2.2.5 Loss Criteria . . . 7

2.2.6 Optimisation . . . 8

2.2.7 Regularisation . . . 10

2.2.8 Best Model Architecture . . . 12

3 Experiments and results 14

3.1 Specification . . . 14

3.2 Best Model performance . . . 15

3.2.1 Accuracy . . . 15

3.2.2 Cross-Entropy Loss . . . 15

3.2.3 Confusion matrix . . . 16

3.3 Hyper-parameter Analysis . . . 17

3.3.1 Learning Rate . . . 17

3.3.2 Batch Size . . . 18

3.3.3 Number of Hidden Layers . . . 19

3.3.4 Number of Hidden Units . . . 20

3.3.5 Activation Functions . . . 21

3.3.6 Optimiser . . . 22

3.3.7 Dropout . . . 23

3.3.8 Weight Decay . . . 23

3.4 Ablation Studies 24

4 Discussion and Conclusion **27**

4.1 Discussion 27

4.2 Conclusion 28

References **29**

Appendix A Code Instructions **31**

Appendix B Code Architecture **32**

Appendix C Running Time and Evaluation Details **33**

Abstract

Deep learning has been a trending topic for many years, and it has been widely used across different industries such as Health, Business, Science, and etc. It is important to get a clear idea of the fundamentals of neural network mechanisms. Thus, this project aims to build a multi-layer neural network classifier to investigate the influences of different modules and hyper-parameters on the classification result and eventually find out the optimum configuration of such a model. This paper describes the motivations and methods of constructing a multi-layer network and discuss the experimental results along with the ablation studies.

The multi-layer neuron network model in this project includes several modules, such as ReLU activation function, Momentum optimiser, Weight Decay, Dropout, Batch Normalisation and Cross-entropy Loss. The accuracy of the final model is around 57%, and it outperformed the baseline model by 4.09%.

Keywords: Deep learning, Multiple Layer Perceptron.

List of Figures

2.1 Model Architecture 3
2.2 Forward propagation 4
2.3 Back-propagation 4
2.4 Activation Function 5
2.5 Tanh function 5
2.6 ReLU 6
2.7 Leaky ReLU 6
2.8 Cross-Entropy Loss accompanied with Softmax [1] 8
2.9 Comparison of with and without Momentum [2] 9
2.10 Dropout example [3] 11
2.11 Batch Normalisation 11
2.12 Batch Normalisation Algorithm [4] 12

3.1 Best Model vs Baseline Model - Accuracy 15
3.2 Best Model vs Baseline Model - Loss 15
3.3 BEST MODEL result 16
3.4 Different Learning Rate - Test Set Accuracy 17
3.5 Different Learning Rate - Test Set Loss 17
3.6 Different Batch Size - Test Set Accuracy 19
3.7 Different Learning Rate and Batch Size - Test Set Accuracy . . . 19
3.8 Different Layer Depth - Test Set Accuracy 20
3.9 Different Layer Depth - Test Set Loss 20
3.10 Different Neuron Size - Test Set Accuracy 21
3.11 Different Neuron Size - Test Set Loss 21
3.12 Different Activation - Test Set Accuracy 22
3.13 Different Activation - Test Set Loss 22
3.14 Different Optimiser - Test Set Accuracy 22
3.15 Different Optimiser - Test Set Loss 23
3.16 Different Dropout Rate - No over-fitting 23
3.17 Different Dropout Rate - Over-fitted 24
3.18 Different Weight Decay 24

B.1 Code Architecture 32

List of Tables

2.1 Baseline model vs Best model 13

3.1 Ablation studies 25
3.2 Model comparison 26

Chapter 1

Introduction

More and more Deep Learning applications are put to use in recent years with the rapid growth of Deep Learning technologies. It has proven its capabilities in tackling arduous tasks across multiple fields, including computer vision, computational biology, natural language processing and many more.

Deep learning is essentially a sub-branch in Machine Learning inspired by the structure of the brain. The term ”neural network” is coined as a result of this viewpoint. Since both the brain and deep learning models require many computational units (neurons), deep learning algorithms are similar to the brain in several ways. While these units are not activated, they are not active; but they become intelligent when they interact with one another. This project investigates the architecture of deep learning neural networks and the impacts of incorporating different building blocks.

The given datasets have already been split into the training set and test set with 50000 instances and 10000 instances respectively. Each instance consists of 128 numerical attributes. The given labels in the datasets are 0 to 9, ten classes. Each class have 5000 instances for training. The expected outcome of this project is to successfully construct an efficient multi-layer neural network model that can correctly classify instances with high accuracy.

This project has great significance as it not only sharpens our programming skills and also broadens our knowledge of the structure of a classic multi-layer perceptron neural network by building it from scratch.

Chapter 2

Methods

This section outlines the mechanisms of numerous neural network building blocks and explain the their roles in a deep learning neural network.

2.1 Pre-processing

Since the neuron network use gradient descent as the optimisation technique, it is sensitive to feature scaling. The feature scaling can ensure the gradient descent moves to the minimum point more smoothly and quickly. To achieve the feature scaling, there are two types, min-max normalisation and standardisation.

2.1.1 Min-Max Nomalisation

The min-max normalisation method was used to re-scale the input data to a range of 0 to 1, while keeping the information intact. The normalisation equation is fairly straightforward and depicted below.

$$x_{norm} = \frac{x - \min(x)}{\max(x) - \min(x)}$$

2.1.2 Standardisation

Standardisation is to make the data follows standard distribution. In the following equation, μ is the mean of data, and the σ is the standard deviation of the data.

$$x_{norm} = \frac{x - \mu}{\sigma}$$

2.2 Principles

This section describes the principles of each building blocks used in our neural network model. The model architecture is shown in Figure 2.1.

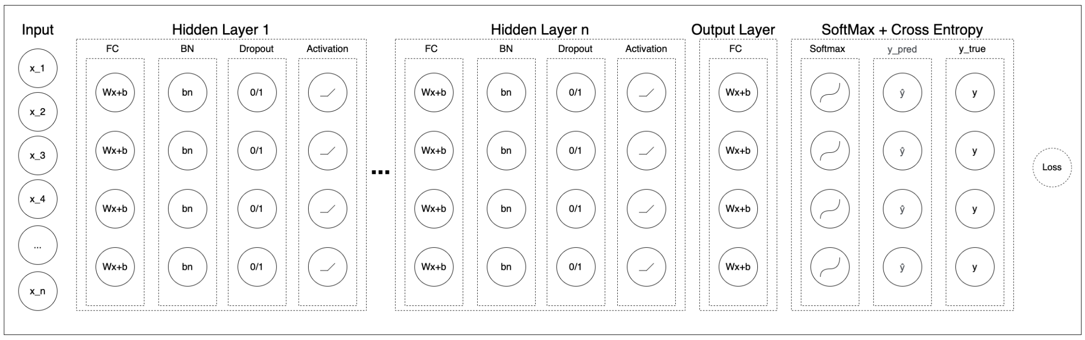


Figure 2.1: Model Architecture

The weights of the model are first initialised with Kaiming initialisation. Then, the input data is passed forward to the Hidden Layers, which contain four parts, including a fully-connected layer, batch-normalisation layer, dropout layer and activation layer, to perform the forward propagation. And this process should be repeated depending on the number of hidden layers. Furthermore, the cross-entropy loss is selected as the loss function for our model and the derivatives are calculated and pass backward in order to calculate gradients. Additionally, the weight decay regulariser is added to prevent the over-fitting problem and the mini-batch training is applied to enhance the efficiency of updating the parameters.

2.2.1 Forward Propagation

The process of feeding the inputs into the neural networks in a forward direction and produce an output is known as forward propagation. Figure 2.4 provides a clear view of the intuition of this process. Every hidden layer receives the inputs, and each hidden units within the hidden layer undergo non-linearity transformation by an activation function and compute a weighted sum and transfer it to the subsequent layer. Where i is the index of units in the input layer;

w_{ji} stands for the weights at the hidden unit j and likewise for w_{kj}.

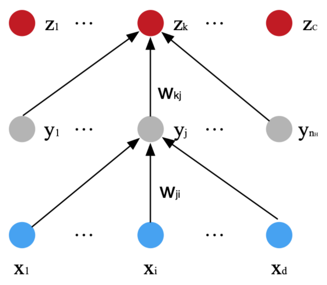


Figure 2.2: Forward propagation

2.2.2 Backward Propagation

Back-propagation plays a vital role in supervised neural network training. The main target of the back-propagation process is to fine-tune the weights based on the loss function. The figure below (2.4) depicts a simple back-propagation example. The mechanism of this fine-tuning is based on gradient descent, which is by calculating the derivatives of the loss function through each layer using the chain rule. It is essentially updating the weights initialised by various initialisation methods. These modified weights will significantly reduce the loss and hence promotes better performance.

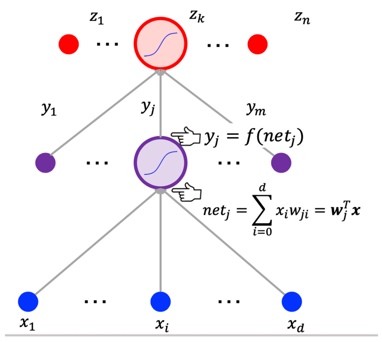


Figure 2.3: Back-propagation

2.2.3 Activation Function

The Activation Functions are used in between the input and output layer for non-linearity transformation (Figure 2.4). If there is no activation transformation, the neuron network will become a simple linear function, which cannot solve the non-linear problem. The activation process ensures the model is able to compute complex calculations and promotes the capabilities of learning [5]. This model includes three popular activation functions, the Tanh, ReLU and Leaky ReLU and will be explained more in details in the following sections.

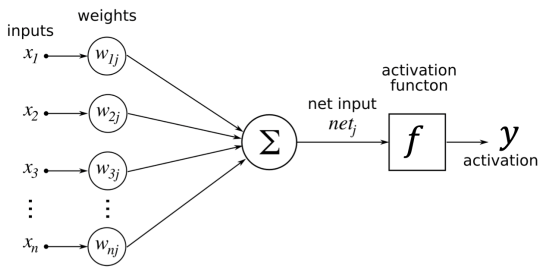


Figure 2.4: Activation Function

Tanh function

The Tanh function is a zero-centred function and could map the output to range -1 to 1. The following equation and figure represent the Tanh function. The zero-centred output of the Tanh function is an advantage that can support the backward propagation, but the disadvantage of the Tanh function is that it cannot prevent the gradient vanishing issue.

$$f(s) = \frac{e^s - e^{-s}}{e^s + e^{-s}}$$

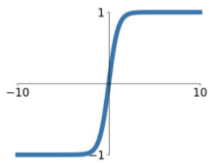


Figure 2.5: Tanh function

Rectified Linear Unit (ReLU)

The Rectified Linear Unit Function or simply ReLU is the most popular activation function nowadays [5]. By looking at the equation and plot (Figure 2.6), it shows that the outputs from ReLU are from 0 to infinity. The main advantage of the ReLu function is it is a lot less prone to gradient vanishing, especially when the output is larger than 1. Furthermore, the calculation is simpler than the Tanh function, and the gradient of ReLU is constant, which is a crucial factor of fast convergence speed. However, the downside of ReLU is also very obvious. ReLU always outputs 0 when values less than 0, which causes the neurons in that 0 value region to fail to activate and be 'dead'. This phenomenon of dead ReLU is less likely to occur with an optimal learning rate.

$$f(s) = \max(0, s)$$

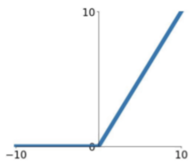


Figure 2.6: ReLU

Leaky ReLU

The Leaky ReLU is similar to the ReLU function, but it converts negative values multiply a small number, such as 0.1, instead of making it equals to zero, therefore the negative values will have a small slope. The output of Leaky ReLU is more balanced than applying simple ReLU.

$$\text{f(x)} = \begin{cases} x, & \text{if } x > 0 \\ \alpha * x, & \text{if } x < 0 \end{cases}$$

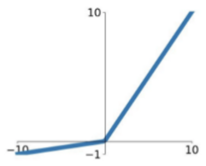


Figure 2.7: Leaky ReLU

2.2.4 Weight Initialisation

Weight initialisation is an important module in neural networks. With proper weight initialisation methods, the chance of slow learning and divergence could be avoided. Random initialisation is a widely-used technique, but for certain activation functions, there are other better initialisation methods. For example, for the Tanh activation function, it is often recommended to use Xavier Initialisation. On the other hand, ReLU or Leaky ReLU are often used in Kaiming Initialisation. In this project, Kaiming Initialisation is selected for the weight initialisation process.

Kaiming Initialisation

Kaiming, he has proposed a robust weight initialisation method, namely the Kaiming Initialisation [6]. This method allows training on deeper neural networks and is very suitable with the ReLU activation function as it solves the gradient explosion issue. As the equation shows below, the Kaiming Initialisation scheme has a mean of 0, and the initial weights are within the specified range.

$$kaiming_{normal} = N(0, \tfrac{2}{n})$$

$$kaiming_{uniform} = u(-\sqrt{\tfrac{6}{n}}, \sqrt{\tfrac{6}{n}})$$

2.2.5 Loss Criteria

The loss functions measure the discrepancies of the model result with the ground truth value. Typically, the loss function should be minimised for neural network optimisation. Such that, the loss of a model would be at a lower level and ultimately boost the performance. The cross-entropy loss would be used as the loss function since it suits the classification model well.

Softmax

In the output layer of the neural network, the Softmax function operates as an activation function that maps the outputs to a probability distribution from

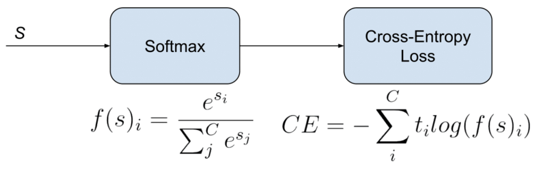
0 to 1 and sums equals to 1. A larger value implies a higher probability of occurrence of such an event.

OneHotEncoding

One-hot encoding is very useful for classifying non-ordinal data, as it transforms the classes into binary values, making it more recognisable for the machine. In this project, one-hot encoding was utilised with softmax and cross-entropy loss.

Cross-Entropy Loss

The cross-entropy loss is used to calculate the discrepancies between the predicted label and ground-truth label, which requires the Softmax function and the One-hot encoding to work collaboratively. As the following figure shows, the outputs from the previous layer are passed to the Softmax layer and converted into probabilities, which is treated as the probabilities of the predicted label, while the One-hot encoding converts the ground truth label into binary values. Then calculate the discrepancy between the two groups of values as the cross-entropy loss.



$$f(s)_i = \frac{e^{s_i}}{\sum_j^C e^{s_j}} \qquad CE = -\sum_i^C t_i log(f(s)_i)$$

Figure 2.8: Cross-Entropy Loss accompanied with Softmax [1]

2.2.6 Optimisation

For a typical training process for multiple layer neural network, it is essential to select proper optimisation methods. These optimisation methods assist the neural network model to compute the optimal parameters through gradient descent. Our optimisation strategies include Mini-Batch Training and Momentum, which will be explained in the below sections.

Mini-Batch Training

The Mini-Batch Training method is essentially a variant of gradient descent. The mechanism is to split the dataset into small batches, which is later used for updating other parameters [7]. It stochastically permutes mini-batches and sequentially take a mini-batch to evaluate the gradient and can be seen as a balance of the stochastic descent and batch gradient descent [8]. This method provides more robust convergence and greatly enhance the efficiency of the updating process as it requires less memory.

Momentum

Momentum is a method that attempts to direct stochastic gradients descent (SGD) in the right directions and suppresses oscillations, thus converge more quickly than without optimisation [2]. As Figure 2.9 suggests, models utilising momentum reaches the local minima faster than without using it, the gradient descent path is significantly shorter. This can be interpreted in the math expression below, where the γ is the momentum term, and it is often set as 0.9.

$$v_t = \gamma v_{t-1} + \eta \nabla_\theta J(\theta)$$
$$\theta_t = \theta_{t-1} - v_t$$

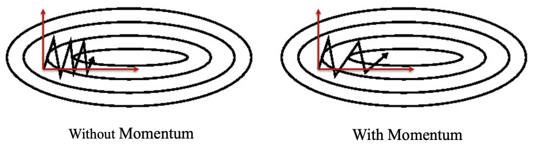


Figure 2.9: Comparison of with and without Momentum [2]

RMSProp

The RMSProp is another optimiser that can be seen as an enhanced version of the Momentum. Compared to Momentum, the RMSProp method utilises an adaptive learning rate, which means the learning rate keeps change during the training process. The learning rate adaptation is determined by taking the square gradient's moving average and normalising the gradient [9]. This optimiser is commonly used because it significantly reduces training time.

$$r \leftarrow \rho r + (1-\rho) g * g$$
$$w \leftarrow w - \frac{\varepsilon}{\sqrt{r+\delta}} g$$

2.2.7 Regularisation

Over-fitting is definitely one of the most encountered problems in deep learning. One commonly used technique is to input more data in the training process, yet the limitation of this technique is conspicuous, as it could be difficult to collect more relevant data. Therefore, regularisation methods are introduced to deal with over-fitting issues in neural networks and promote optimisation. There are many regularisation techniques, and here we explain the ones we used in our project.

Weight Decay

Weight decay is essentially a form of L2 regularisation. Its effect is to restrict the growth of the weights in neural networks in a way that alleviates the large weights associated with the problem of over-fitting. It can be implemented by adding it to the loss function to penalise large weights (1) or simply reduce the weight on each training iteration (2).

$$(1) \qquad min\hat{L}_R(\theta) = \hat{L}(\theta) + \frac{\alpha}{2}||\theta||_2^2$$

$$(2) \qquad \theta \leftarrow \theta - \eta \nabla \hat{L}_R \quad (update)$$
$$\theta \leftarrow \theta - (wd * \theta)$$

Where η is the learning rate, α is the regularisation parameter, and wd is the weight decay parameter.

Dropout

Unlike weight decay, the concept of Dropout is to modify the structure of the neural network rather than adding penalisation on the loss function. In the training phase, numerous hidden neurons are randomly dropped out; an example is shown in Figure 2.10. This solves the over-fitting issue by reducing complex 'co-adaptations' between the neural networks [3]. Meanwhile, the test phase is evaluated by combining the average of all layers by using a standard layer that has fewer weights [3].

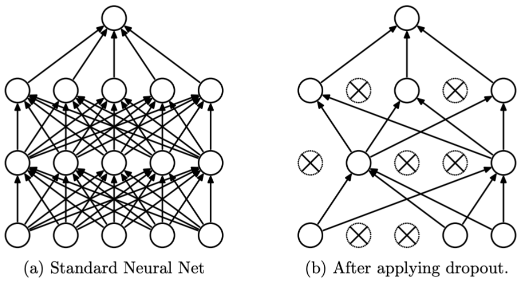


Figure 2.10: Dropout example [3]

Batch Normalisation

Batch normalisation is another regularisation technique we used in our model. The fundamental concept of this regulariser is to mitigate the *Co-variate Shift* by inserting a batch normalisation layer (Figure 2.11) to normalise and standardise the inputs [4]. These inputs are re-scaled either in the range of 0 to 1, or -1 to 1 or with a mean of 0 and variance of 1, so that the latter layer no longer needs to keep adapting the distribution of the prior layer, hence reduce training time.

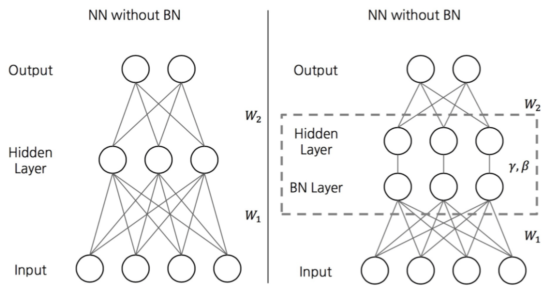


Figure 2.11: Batch Normalisation

Here, the proposed algorithm for Batch Normalisation by Sergey Ioffe [4] is depicted below (Figure 2.12). The algorithm explains the batch normalisation technique in a nutshell, which can be epitomised in three steps. That is, standardisation, normalisation and re-scale, and finally shift the data. Note that there are two important parameters shown in the algorithm, where γ and β are learnable parameters responsible for scale and shift [4].

```
Input: Values of $x$ over a mini-batch: $\mathcal{B} = \{x_{1...m}\}$;
       Parameters to be learned: $\gamma, \beta$
Output: $\{y_i = \text{BN}_{\gamma,\beta}(x_i)\}$

    $\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i$                          // mini-batch mean
    $\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i - \mu_\mathcal{B})^2$          // mini-batch variance
    $\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}}$                    // normalize
    $y_i \leftarrow \gamma\widehat{x}_i + \beta \equiv \text{BN}_{\gamma,\beta}(x_i)$          // scale and shift
```

Figure 2.12: Batch Normalisation Algorithm [4]

Batch normalisation is highly popular in the deep learning field, as it has many advantages over other regularisation methods, such as reducing the occurrence of gradient explosion and gradient vanishing problem, improving training time, reducing the use of other regularisation, allowing for higher learning rates, and providing training with saturating nonlinearities in deep neural networks.

2.2.8 Best Model Architecture

The optimal architecture of our model is obtained by conducting extensive ablation studies, and the architecture with the highest accuracy is selected. According to table 2.1, the models contains 2 hidden layers and has 256 and 128 neurons respectively, utilising Leaky ReLU as the activation function and the learning rate is 0.001. We have discovered that Batch Normalisation and Dropout is unnecessary for our best model if we stop at 48 epochs. The accuracy of the best model excels the baseline model by approximately 4.09%. It significantly converges faster than the baseline model as it only needs 48 epochs, implying more efficient learning. This is achieved by adding a weight decay of 0.001 and modify the batch size to 2. More details of the experiments of obtaining the best architecture will be discussed in the experiment sections.

Models		
Modules	Baseline model	Best model
Weight initialisation	Kaiming	Kaiming
Pre-processing	No	No
Activation	ReLU	ReLU
Optimisation	SGD	SGD
Weight decay	0	0.001
Learning rate	0.001	0.001
Batch normalisation	FALSE	FALSE
Drop out	0	0
Epoch	300	48
Batch size	64	2
Hidden layer	2	2
Hidden units	[128,128]	[256,128]
Accuracy	0.5284	0.5693

Table 2.1: Baseline model vs Best model

Chapter 3

Experiments and results

3.1 Specification

The experiments are conducted on Google Colab Platform and the hardware and software specifications are shown below:

Hardware

OS: Ubuntu 18.04
CPU: Intel(R) Xeon(R) CPU @ 2.20GHz
RAM: 26751716 kB
GPU: Tesla V100 16130 MB

Software

Language: Python 3.7.10
Packages:
1. numpy 1.19.5
2. sklearn.metrics 0.24.1 (for evaluation)
3. seaborn 0.11.1 (for evaluation)
4. matplotlib 3.2.2 (for evaluation)

3.2 Best Model performance

As mentioned earlier in section 2.2.8, our best model utilises weight decay and mini-batch training. The performance of this model is assessed on accuracy, cross-entropy loss, and confusion matrix.

3.2.1 Accuracy

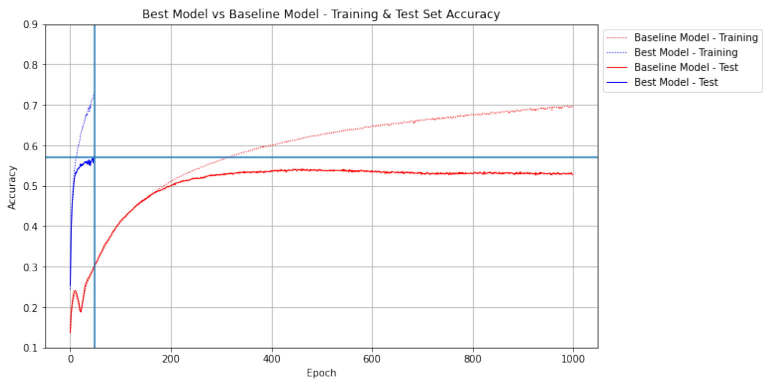


Figure 3.1: Best Model vs Baseline Model - Accuracy

The training accuracy and test accuracy of test two models follow a reasonable trend (Figure 3.1), where over-fitting occurred after convergence. The best model has achieved the highest accuracy at 48 epochs of approximately 57%, which excelled the baseline model by 4.09%.

3.2.2 Cross-Entropy Loss

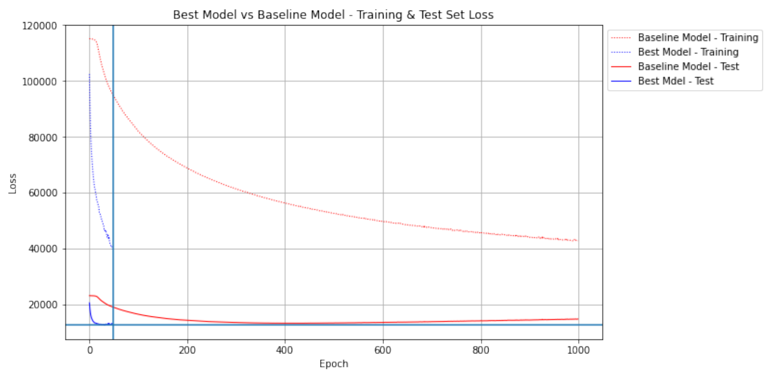


Figure 3.2: Best Model vs Baseline Model - Loss

The cross-entropy loss curves shows that the baseline model training loss and test loss are both decreasing and has gone up after around 400 epoch, which means the model is learning and over-fitting occurred. However, in terms of the best model, it is stopped before over-fitting happening, and the highest accuracy reached at 48 epoch.

3.2.3 Confusion matrix

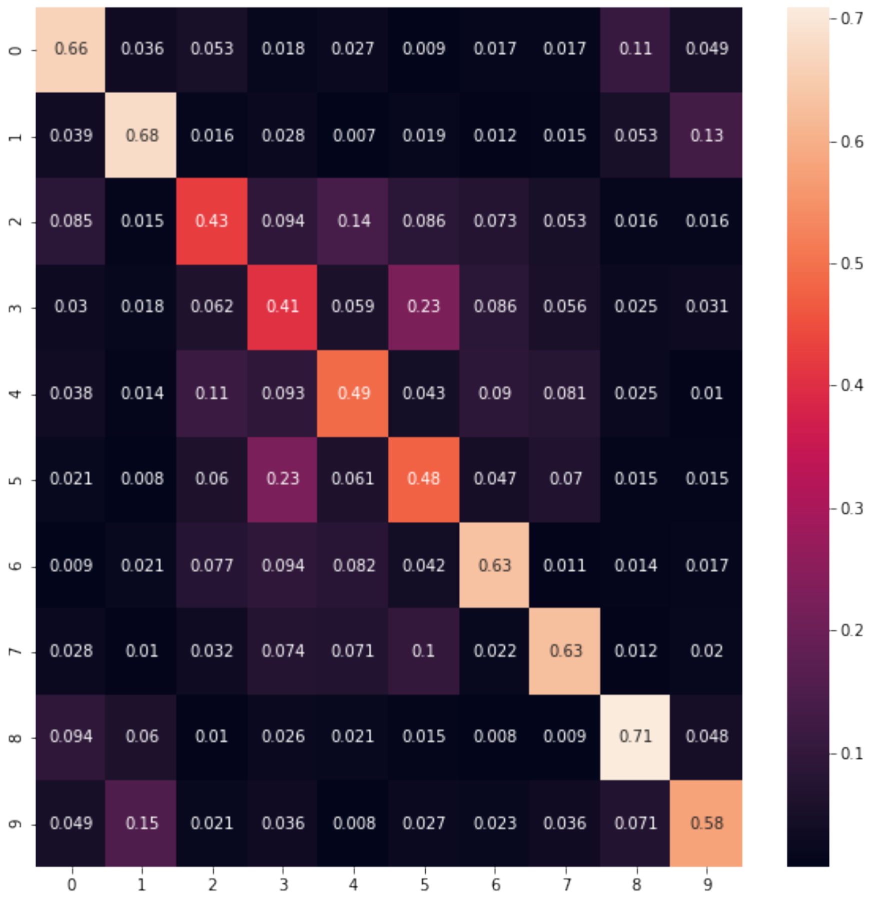

	0	1	2	3	4	5	6	7	8	9
0	0.66	0.036	0.053	0.018	0.027	0.009	0.017	0.017	0.11	0.049
1	0.039	0.68	0.016	0.028	0.007	0.019	0.012	0.015	0.053	0.13
2	0.085	0.015	0.43	0.094	0.14	0.086	0.073	0.053	0.016	0.016
3	0.03	0.018	0.062	0.41	0.059	0.23	0.086	0.056	0.025	0.031
4	0.038	0.014	0.11	0.093	0.49	0.043	0.09	0.081	0.025	0.01
5	0.021	0.008	0.06	0.23	0.061	0.48	0.047	0.07	0.015	0.015
6	0.009	0.021	0.077	0.094	0.082	0.042	0.63	0.011	0.014	0.017
7	0.028	0.01	0.032	0.074	0.071	0.1	0.022	0.63	0.012	0.02
8	0.094	0.06	0.01	0.026	0.021	0.015	0.008	0.009	0.71	0.048
9	0.049	0.15	0.021	0.036	0.008	0.027	0.023	0.036	0.071	0.58

Figure 3.3: BEST MODEL result

The figure above (3.3) is the confusion matrix of our best model; it clearly depicts the classification accuracy of each class. The classification result is around 60% to 70% except for class 3 and its neighbouring classes (2, 4, 5). This could be that class 3 is very similar to its neighbours and is problematic for the model to learn.

3.3 Hyper-parameter Analysis

This sections analyse the effect of using different values for below hyper-parameters and fine tune them. This analysis is based on our baseline model architecture which only incorporated Kaiming initialisation and ReLU activation function.

3.3.1 Learning Rate

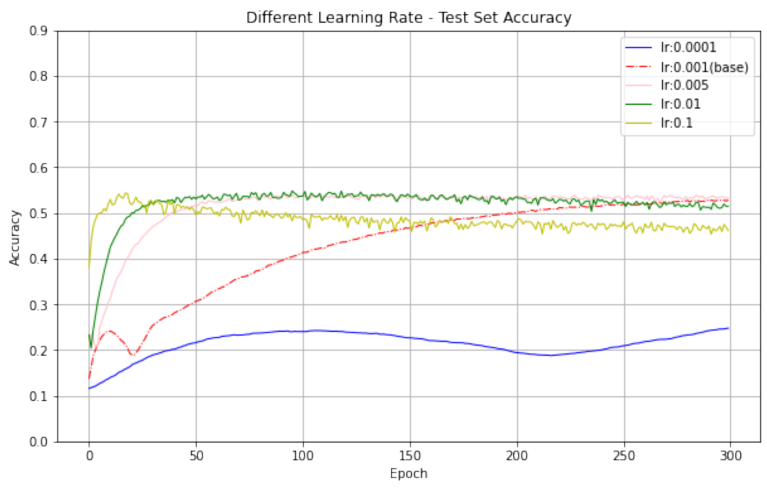


Figure 3.4: Different Learning Rate - Test Set Accuracy

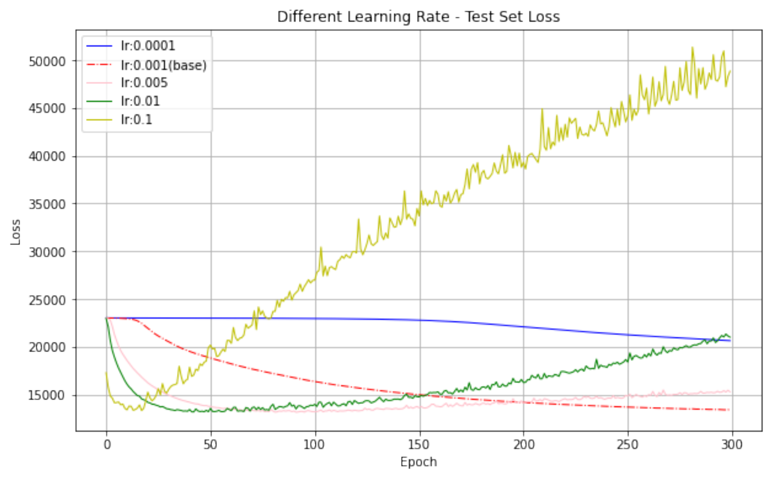


Figure 3.5: Different Learning Rate - Test Set Loss

Learning rate is a hyper-parameter used for updating weights during the training process. According to the comparison plot for different learning rates (Figure 3.4, Figure 3.5), all of the curves have fluctuations to a certain extent, where the curve with the highest learning rate oscillates the most. As expected, the curve with the largest learning rate converges the fastest. However, the loss starts to increase drastically after 10 epochs along with a decrease in accuracy, which means the update step size is too large and missed the local minimum point. On the other hand, the lowest learning rate did not converge after 300 epochs, and this caused the model training process to get stranded. Our baseline model learning rate is 0.001, which generally followed a good trend. It almost reached the optimum accuracy within 300 epochs. 0.01 and 0.005 are very close at first, but at epoch 100, the loss of 0.01 learning rate starts to increase significantly. That being said, the learning rate of 0.01 is the best among these 5 different learning rates when running 300 epochs as it achieved the highest accuracy and convergence speed is also acceptable.

3.3.2 Batch Size

A comprehensive analysis of different batch sizes and selected a few samples to further analyse them with a range of learning rates. Figure 3.6 shows that batch size 4096 and over have a poor accuracy, this implies that the model under these parameters has problems with learning. This could be caused by low updating steps and potential over-fitting issue. The other extreme is the batch size of 1, where it only use one sample from the dataset and updated many times. The accuracy of batch size one is significantly better than those large batch sizes mentioned earlier, but it still needs improvements considering it has a high computation cost. Therefore, the trade-off of batch size and number of updates needs to be taken into account.

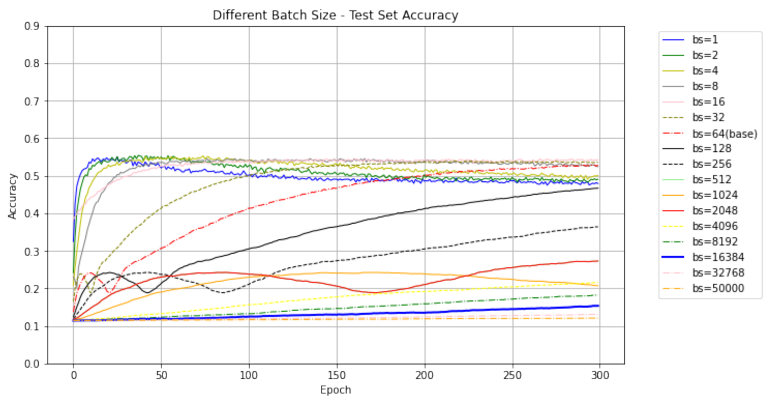


Figure 3.6: Different Batch Size - Test Set Accuracy

Next, two different batch size is selected and tested with 3 different learning rates, 0.1, 0.01, and 0.001. Batch size of 2 to 16 all converged within 300 epochs, where batch size is 2 with learning rate of 0.001 achieved the highest accuracy at approximately 20 epoch as illustrated in the Figure 3.7. This indicates that the larger batch size should match the larger learning rate.

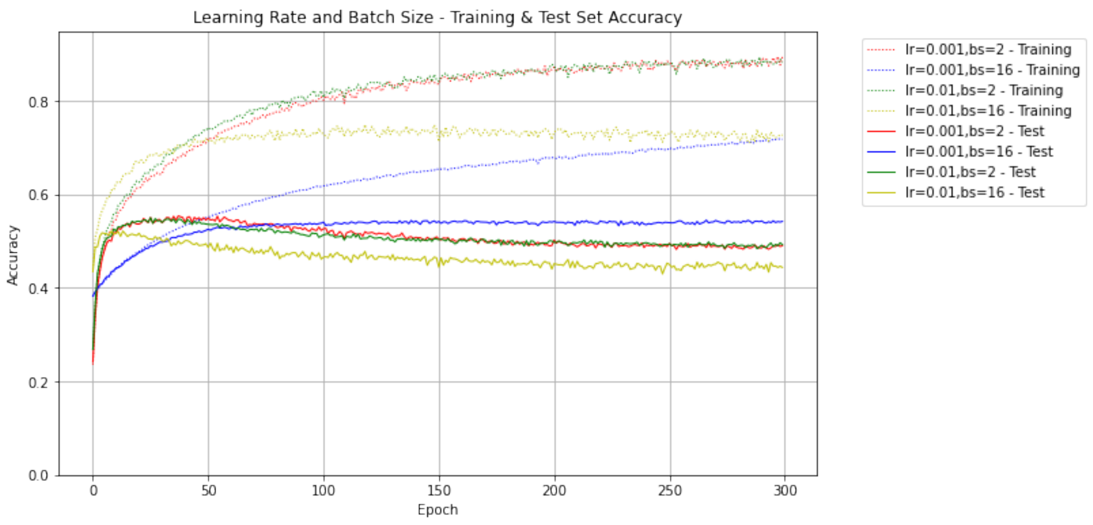


Figure 3.7: Different Learning Rate and Batch Size - Test Set Accuracy

3.3.3 Number of Hidden Layers

The number of hidden layers is tested with the same number of total hidden units (i.e. 256). The result is noticeably straight, shown in Figure 3.8 and Figure 3.9. Initially, a single layer of 256 neurons converge the fastest, but the convergence is nearly the same as two layers of 128 neurons at 250 epochs, and if more epochs were conducted, three layers might even surpass them. That being said, to achieve fast convergence, it is recommended to have the largest hidden layer as the first one, then apply more smaller layers upon needs.

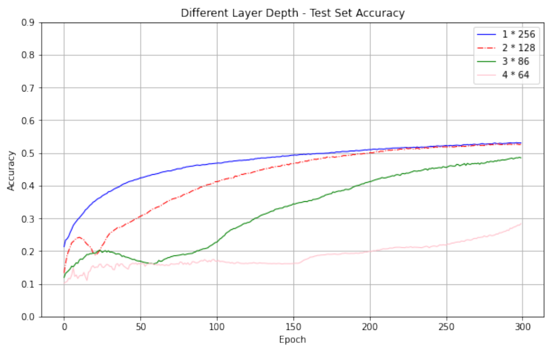


Figure 3.8: Different Layer Depth - Test Set Accuracy

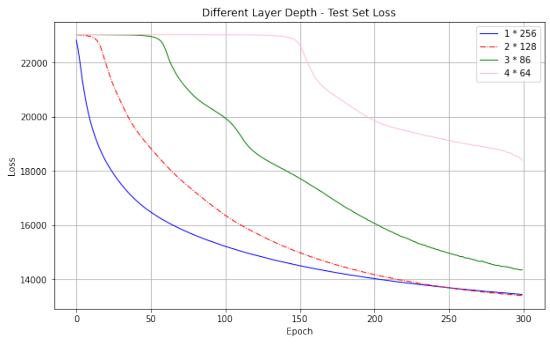


Figure 3.9: Different Layer Depth - Test Set Loss

3.3.4 Number of Hidden Units

The analysis of the impact of a diverse number of hidden units is done by monitor the changes in loss and accuracy with two hidden layers for all cases. Even though the trend of these three curves are very similar, there is still some deviation at the start and after 140 epochs. The plots (Figure 3.10, 3.11) demonstrates that the combination of 64 neurons in the first layer and 32 neurons in the second layer have a nuance compared to 64 neurons in each layer, both of them quickly converges at the beginning but slows down after 140 epochs, where 256 neurons in each layer are the opposite. And the performance of two layers of 128 hidden units each is between these two.

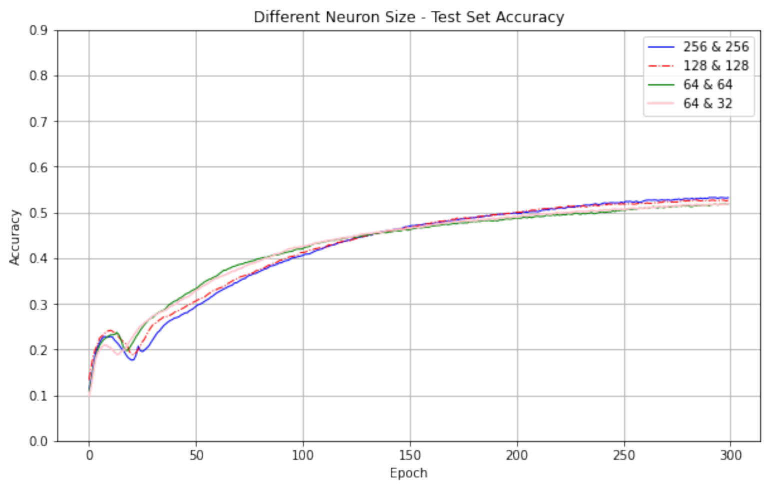


Figure 3.10: Different Neuron Size - Test Set Accuracy

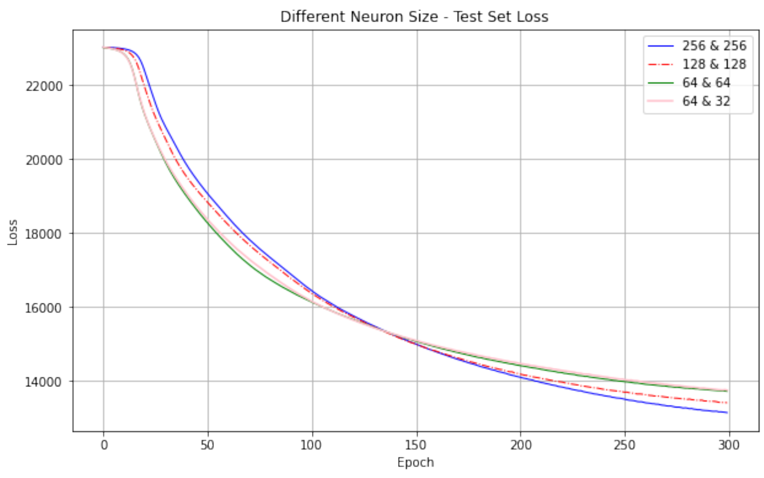


Figure 3.11: Different Neuron Size - Test Set Loss

3.3.5 Activation Functions

Three different activation functions were tested and plotted below (Figure 3.12, Figure 3.13). It can be seen that the ReLU and Leaky ReLU totally overlapped with each other, signifying the Leaky ReLU is unnecessary since ReLU works just as good as Leaky ReLU. Besides, the overall performance of Tanh is significantly lower than the other two functions.

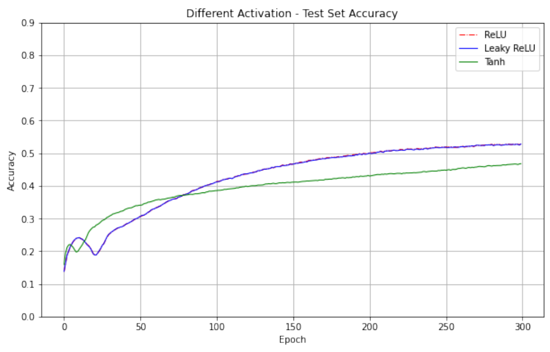


Figure 3.12: Different Activation - Test Set Accuracy

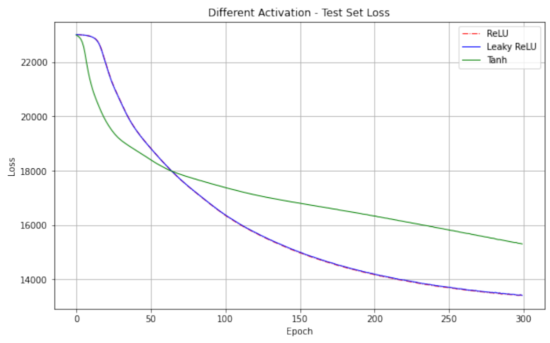


Figure 3.13: Different Activation - Test Set Loss

3.3.6 Optimiser

In this part, we compared several momentum values and include a RMSProp optimiser for reference. The RMSProp optimiser converges the fastest according to the plots (Figure 3.14, Figure 3.15). Also, the plots suggest that momentum optimiser works better with larger momentum parameter as 0.9 is the fastest to reach the minimum point.

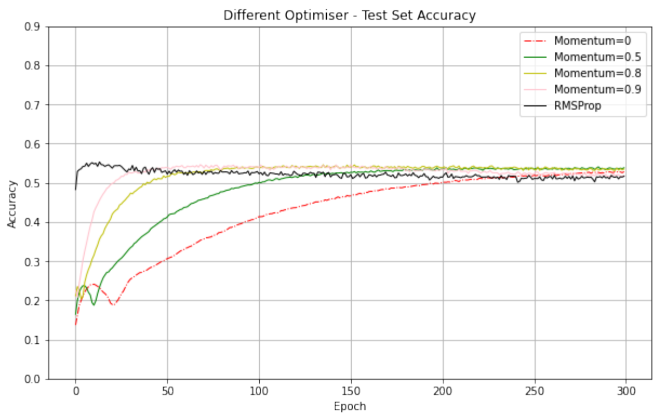


Figure 3.14: Different Optimiser - Test Set Accuracy

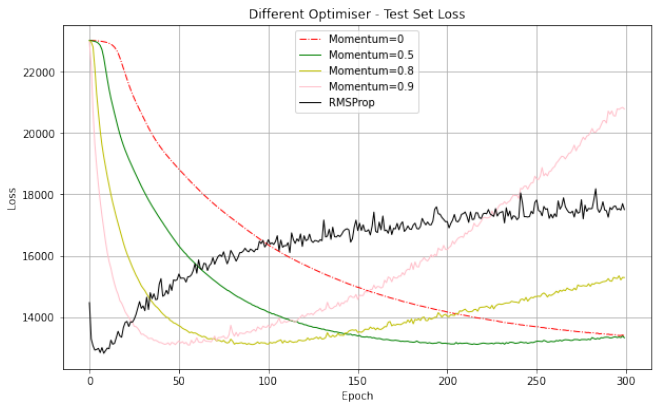


Figure 3.15: Different Optimiser - Test Set Loss

3.3.7 Dropout

The dropout rate represents the probability of dropped neurons, and higher dropout rates indicate stronger regularisation. Figure 3.16 is a plot of a range of dropout rates on a non-over-fitting model, which shows that the accuracy is still rising even then dropout is introduced to the model. Thus, dropout does not work with models with no over-fitting issue, and instead, it will suppress the accuracy. Figure 3.17 reveals the effect of dropout on an over-fitted model, where a larger dropout rate remains higher accuracy than others after it reaches minima. This phenomenon is expected when a regulariser is taking effect.

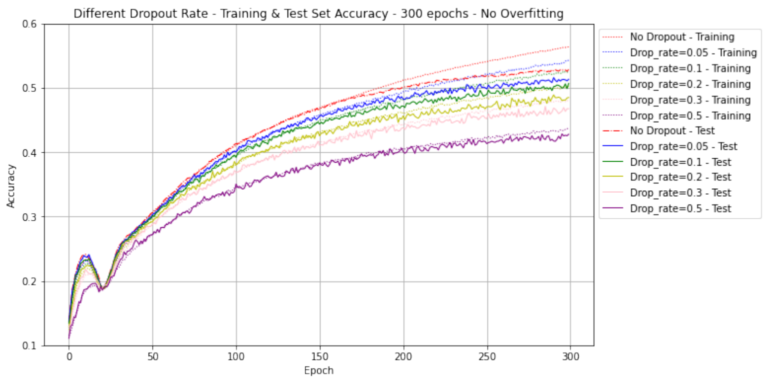


Figure 3.16: Different Dropout Rate - No over-fitting

3.3.8 Weight Decay

Similar to the Dropout, Weight Decay also act as a regulariser to prevent the over-fitting problem. As the figure 3.18 shows, large weight decay value such

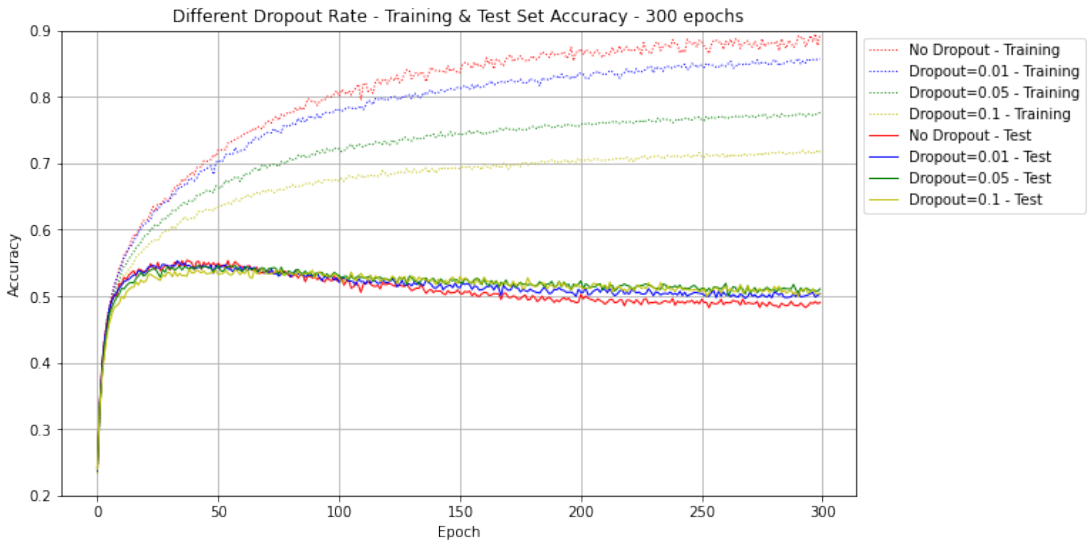


Figure 3.17: Different Dropout Rate - Over-fitted

as 0.1 overly limit the weights resulting lowest test accuracy among others. It is clear to see that the weight decay of 0.001 performs the best as the curve remains constant after convergence, which is opposed to the one without using weight decay.

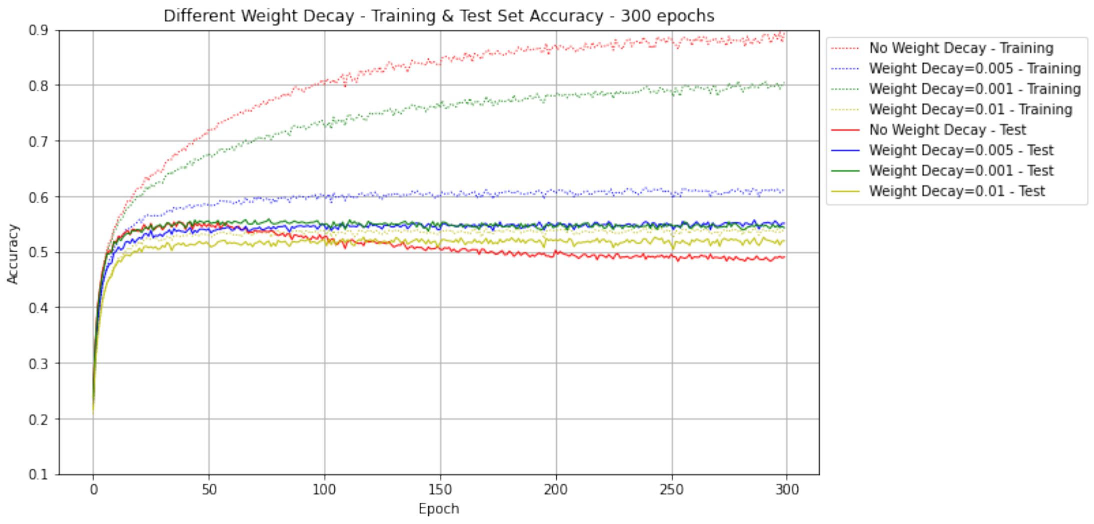


Figure 3.18: Different Weight Decay

3.4 Ablation Studies

The ablation study is a type of experiment that involves training the model and removing one or more of its components. Then investigate the effect of removing those building blocks in a neural network model to understand their impact on overall performance. We analyse different combinations of modules on the baseline model to determine the best model.

Ablation Studies					
Modules	Baseline	Model 1	Mode2	Model 3	Model 4
Initialisation	Kaiming	Kaiming	Kaiming	Kaiming	Kaiming
Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Optimisation	SGD	SGD	**M = 0.9**	**M = 0.9**	**M = 0.9**
Weight decay	0	0	0	**0.001**	**0.001**
Learning rate	0.001	0.001	0.001	0.001	0.001
Batch norm	FALSE	**TRUE**	**TRUE**	**TRUE**	**TRUE**
Drop out	0	0	0	0	**0.01**
Epoch	300	300	300	300	300
Batch size	64	64	64	64	64
Hidden layer	2	2	2	2	2
Hidden units	[128,128]	[128,128]	[128,128]	[128,128]	[128,128]
Accuracy	0.5284	0.5513	0.5457	0.5403	0.5165

Table 3.1: Ablation studies

Our baseline uses Kaiming weight initialisation and ReLU activation function and updated by SGD. We then build upon this model by gradually implement more modules to work out whether these modules are necessary or not. Noted that all the hyper-parameters such as the number of layers and learning rate stays the same for this experiment. The added modules are emphasised in bold letters. From the table 3.1, it can be seen that Model 1 boost up the accuracy by approximately 2% by adding Batch Normalisation. However, when more components are added to the model, the accuracy tends to decrease. This suggests that the batch normalisation is very effective on its own and may not work well with other modules. We then tried many different combinations of models with adjusted hyper-parameters and selected some of the best models, shown in the table below.

As we can see, these models all surpassed the baseline model with batch normalisation as stated above. The best model yields the highest accuracy of 0.5693 by only using weight decay and mini-batch training, the main factor

Model comparison					
Modules	Baseline	Model 1	Mode2	Model 3	Best model
Initialisation	Kaiming	Kaiming	Kaiming	Kaiming	Kaiming
Activation	ReLU	ReLU	ReLU	ReLU	ReLU
Optimisation	SGD	SGD	RMSProp	0	SGD
Weight decay	0	0.001	0	0	0.001
Learning rate	0.001	0.001	0.001	0.001	0.001
Batch norm	FALSE	FALSE	TRUE	FALSE	FALSE
Drop out	0	0	0	0	0
Epoch	300	77	50	38	48
Batch size	64	2	16	64	2
Hidden layer	2	2	2	2	2
Hidden units	[128,128]	[128,128]	[128,128]	[128,128]	[256,128]
Accuracy	0.5284	0.5589	0.5538	0.5541	0.5693

Table 3.2: Model comparison

of improved accuracy might be the number of hidden units in this case. The implication of this comparison is that while selecting appropriate modules certainly give a good result, but combined with fine-tuning we can explore more options in choosing the best architecture.

Chapter 4

Discussion and Conclusion

4.1 Discussion

In this project, we have examined the influences of different components of neural networks and developed an optimal model. The model comprises two hidden layers with 256 and 128 neurons respectively, the accuracy of this model is around 57%. From the experiments, some modules result minor or negative impact on the classification performance, especially the dropout. One of the reason is that our model does not have the problem of over-fitting, consequently, adding the dropout layer will only limit the ability of the model to learn and lower the accuracy. Weight decay is similar to dropout. But using a small weight decay like 0.001 has improved the robustness of our model, and this is due to dropout restrain the weight growth exponentially, and weight decay suppress the growth linearly.

Batch normalisation worked well on its own, quickly fitting the training data. However, the performance dropped when other modules were included. This is unusual but perhaps can be dealt with by increasing the batch size since small batch size induce instability on batch normalisation. Unfortunately, the experiment was not tested due to the time limitation.

In light of the appearance of significant fluctuation in training and test accuracy from the start, it could be potentially caused by using a small batch size (i.e.

2). Because the small batch size is quite noisy, and it is difficult to choose the updating path with the small number of samples. We can definitely run more tests on this project in the future, and it would be better to conduct a grid search for the hyper-parameters tuning.

4.2 Conclusion

In conclusion, we have successfully constructed a multi-layer neural network from scratch and examine the effects of the particular components, such as the optimiser and regulariser. The best neural network model is to apply parts of our existing modules, including Weight Decay and Leaky ReLU, with mini-batch training and accomplished the classification task in a short time with an accuracy of 57%.

The most challenging task of this project is to build the batch normalisation module, as the back-propagation function of the batch normalisation layer is very complex and involves variables storing requirement. We have followed the method stated in the paper [4], which is placing the batch normalisation layer before the activation function for every layer. However, recent research has advised that adding the batch normalisation layer after the activation may provide better results.

The future work of this project includes:
1. Test the batch normalisation with bigger batch size and larger dropout rate.
2. Try to implement an adaptive learning rate, since the model can converge very fast with a high learning rate, while the training in a later stage should scale down the learning rate to achieve a better result.

All in all, this project has broadened our knowledge in the fundamentals of deep learning and provided a remarkable hands-on experience of building a neural network.

References

[1] Additive margin softmax loss (am-softmax) — by fathy rashad — towards data science. https://radiant-brushlands-42789.herokuapp.com/towardsdatascience.com/additive-margin-softmax-loss-am-softmax-912e11ce1c6b. (Accessed on 04/20/2021).

[2] Stochastic gradient descent with momentum — by vitaly bushaev — towards data science. https://towardsdatascience.com/stochastic-gradient-descent-with-momentum-a84097641a5d#:~:text=Momentum%20%5B1%5D%20or%20SGD%20with,models%20are%20trained%20using%20it. (Accessed on 04/20/2021).

[3] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958, 2014.

[4] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. PMLR, 2015.

[5] Sagar Sharma. Activation functions in neural networks. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6.

[6] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[7] Mu Li, Tong Zhang, Yuqiang Chen, and Alexander J Smola. Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670, 2014.

[8] A gentle introduction to mini-batch gradient descent and how to configure batch size. https://machinelearningmastery.com/gentle-introduction-mini-batch-gradient-descent-configure-batch-size/. (Accessed on 04/19/2021).

[9] Rmsprop definition — deepai. `https://deepai.org/machine-learning-glossary-and-terms/rmsprop`. (Accessed on 04/25/2021).

Appendix A

Code Instructions

Our code is running on the Google Colab platform. Please run our code with the following steps:

1. Upload the `Assignment_1_Best_Model.ipynb` file to the Google Colab and open it.
2. Click ”Run all” in the ”Runtime” tab or press ”cmd/Ctrl+F9” to run code blocks from the beginning.
3. In the ”Preparation - Datasets” section, click the link and follow the instruction to sign in to your Google Account, then copy and paste the google verification code, the dataset will be downloaded and imported automatically. **If you want to use other test sets, please replace the file path of the ”test_data” and ”test_label” in the second code block of this section.**
4. The following sections will be run automatically and you will see the results in the last section ”Train and evaluation” **within 30 minutes**.

Appendix B

Code Architecture

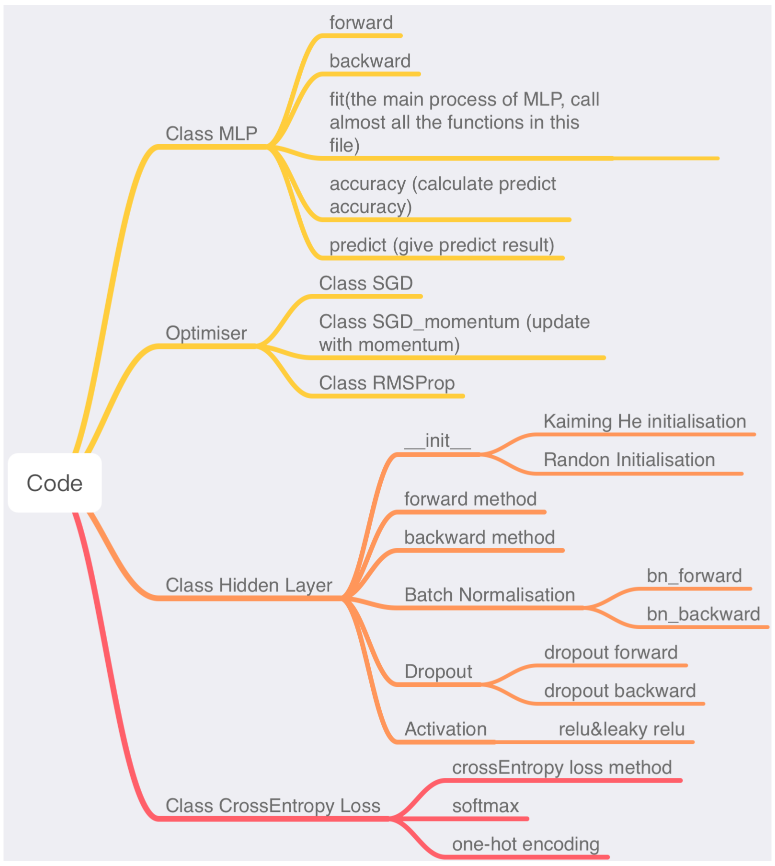


Figure B.1: Code Architecture

Appendix C

Running Time and Evaluation Details

The running time of each epoch for our best model are shown in the following table. The evaluation details of other models are attached in the zip folder.

Evaluation Details of Best Model					
Epoch	training acc	val acc	training loss	val loss	epoch time(s)
1	0.24368	0.254	102358.156	20447.983	40.968
2	0.32838	0.329	91659.56	18240.700	40.853
3	0.39294	0.3959	84219.749	16782.23	40.472
4	0.434	0.4319	79020.837	15834.500	41.035
5	0.46018	0.4598	75129.031	15147.132	40.219
6	0.48352	0.4734	72164.484	14706.142	40.930
7	0.50412	0.4905	69802.750	14305.250	40.642
8	0.51848	0.503	67549.39	13980.329	40.589
9	0.53684	0.5166	65479.173	13765.766	40.180
10	0.54612	0.5235	63936.218	13558.33	41.645
11	0.55738	0.5267	62319.5936	13314.811	40.297
12	0.56682	0.5339	61400.8651	13294.617	40.076
13	0.56862	0.531	60769.7101	13240.879	41.454
14	0.5776	0.5347	59470.24	13163.038	40.072
15	0.58604	0.5406	58631.031	13145.358	39.933
16	0.59178	0.5405	57318.283	12946.877	40.676
17	0.5971	0.5435	56946.787	12993.658	39.761
18	0.60418	0.5427	56098.727	12933.083	39.753
19	0.6068	0.5457	55415.853	12893.49	40.785
20	0.61154	0.5462	54981.71	12915.897	40.309
21	0.62588	0.5539	52970.8160	12680.161	39.998
22	0.62758	0.5487	52891.990	12750.875	40.540
23	0.63316	0.5526	52252.063	12771.360	39.645
24	0.6346	0.5492	51758.984	12833.062	39.692
25	0.64434	0.5528	50544.284	12696.694	40.359
26	0.6433	0.5543	50495.121	12669.537	39.473
27	0.65132	0.5521	49765.160	12696.26	39.561
28	0.65356	0.5556	49119.9495	12722.915	40.560
29	0.66084	0.5566	48682.1169	12708.053	39.518
30	0.6603	0.5593	48409.072	12740.443	39.592
31	0.6714	0.5584	46866.2247	12579.223	40.387
32	0.66948	0.5566	46923.3558	12676.209	39.487
33	0.67838	0.5596	45875.888	12596.107	39.428
34	0.67338	0.5607	46526.1447	12741.520	40.357
35	0.6771	0.5533	45922.746	12845.399	39.421
36	0.6856	0.555	45188.930	12778.475	39.505
37	0.69336	0.5594	44101.1450	12736.535	40.193
38	0.6935	0.5637	43964.567	12736.087	39.699
39	0.68472	0.5521	45081.778	13038.340	39.521
40	0.69858	0.5608	43298.833	12741.163	39.790
41	0.69098	0.5496	44187.3949	13108.56	39.797
42	0.7036	0.5606	42420.458	12879.627	39.713
43	0.71246	0.5602	41773.309	12816.352	40.336
44	0.71276	0.5647	41388.916	12760.082	39.776
45	0.7173	0.5681	40679.471	12752.14	39.888
46	0.71782	0.5645	40639.9811	12842.408	40.321
47	0.71778	0.5571	40777.02	13051.761	39.658
48	0.73004	0.5693	39677.9496	12811.87	39.583