THE UNIVERSITY OF SYDNEY

The University of Sydney

School of Computer Science

Assignment 1 Report

Author:
Enze Ren 520638064
Ziming Zhong 520558795
Jiasheng Chen 520024175

Supervisor:
Linwei Tao

An Assignment submitted for the UoS:

COMP5329 Deep Learning

April 3, 2023

Contents

1 Introduction **3**

1.1 Aim and Objective 3

1.2 Importance 3

2 Methods **3**

2.1 Pre-processing 3

2.2 Modules 3

2.2.1 Multiple Hidden Layers 3

2.2.2 ReLU Activation 4

2.2.3 Weight Decay 4

2.2.4 Momentum in SGD 4

2.2.5 Dropout 5

2.2.6 Softmax and Cross-entropy Loss 5

2.2.7 Mini-batch training 5

2.2.8 Batch Normalization 6

2.2.9 Other Advanced Operations 6

2.3 Design of the Best Model 7

2.3.1 Code Structure 7

2.3.2 Hyper Parameters Set 8

3 Experiments and Results **9**

3.1 Hardware and Software Specification 9

3.2 Performance 10

3.3 Extensive Analysis 11

3.4 Justification 17

4 Discussion and Conclusion **17**

A Code Link **19**

B Instruction for Code **19**

List of Tables

1 Performance of the Best Model 11
2 Evaluation of Different Dropout Rate 12
3 Evaluation of Different Batch Size 13
4 Evaluation of Batch Normalization 14
5 Evaluation of Learning Rate 15
6 Evaluation of Activation Function 15
7 Evaluation of Optimizer 16
8 Combination Evaluation 17

List of Figures

1 Code Structure 8
2 Parameter of the Best Model 9
3 Loss of the Best Model 10
4 Accuracy of the Best Model 10
5 Basic Model 11
6 Comparison of Dropout 12
7 Comparison of Batch Size 13
8 Comparison of Batch Normalization 13
9 Comparison of Learning Rate 14
10 Comparison of Activation Function 15
11 Comparison of Optimizer 16

1 Introduction

1.1 Aim and Objective

Multilayer Neural Network(MNN) is an important part of deep learning. As the one of the basis algorithm of deep learning, studying Multilayer Neural Network help people enhance their understanding of deep learning. In this study, the project aims to apply different modules and hyper parameters for achieving a Multilayer Perceptron (MLP) to observe the modules' effect on accuracy and running time with multi-class classification task. Then, this work will analyzes the optimal combination of MLP and find out the indication for the following study or project.

1.2 Importance

As mentioned above, Multilayer Neural Network is one of the core deep learning algorithm. Its expandability allows people easily to combine it with different modules to solve the multi-class classification problem. During the studying, the impact of different factors will be discovered such as the type of activation function, the number of hidden layers and optimizer methods. These experience will be instructive and enlightening when learning other deep learning algorithm in the following courses.

2 Methods

2.1 Pre-processing

In this work, the dataset is divided into a training set, a validation set, and a test set in a ratio of approximately 8:1:1. The first 40,000 data points in the training data file are selected as the training data, while the remaining 10,000 data points are selected as the validation data. Since the original dataset only contains two types of data (training and test), it is helpful to use a validation set to identify potential issues and optimize parameter settings in advance. The neural network can adjust hyperparameters properly by using the validation set to prevent overfitting and improve the accuracy of the model.

2.2 Modules

2.2.1 Multiple Hidden Layers

Hidden nodes of each layer take the output of the previous layer or input and apply an activation function to get a set of new outputs. Keep doing it until the outputs of the hidden layer arrive at the output layer. Compared to one hidden layer, multiple hidden layers allows the network to learn more complex data from the input because it makes the model non-linear. It can model a more sophisticated and detailed relationship between input data and output results.

More hidden layers can introduce better performance on different tasks, such as regression, classification, and sequence prediction. An increment in the number of hidden layers means more parameters in the training model. This might lead to overfitting if the model is not correctly regularized or the dataset of input is small.

2.2.2 ReLU Activation

ReLU is an activation function commonly used in neural networks. It is a simple non-linear function that outputs zero to positive infinity. The output will be zero if the value is negative. The output will equal the value if the value is positive. The *ReLU* function is defined below,

$$f(x) = \begin{cases} 0, & x < 0 \\ x, & x \geq 0 \end{cases} \tag{1}$$

ReLU is a function easy to implement and it is more computationally efficient compared to other activation functions. *ReLU* is a non-linear function that makes MLP able to process a more complicated dataset. *ReLU* can also help reduce overfitting and improve the model's generalization.

2.2.3 Weight Decay

Weight decay is one of the most used techniques of regularization. It is usually called *L2* regularization as well. By adding a penalty term to the sum of squares of weights in the loss function to punish large weight parameters. It limits the growth of weight parameters during training and makes the model have smaller weight parameters to prevent overfitting,

$$L(w,b) + \frac{\lambda}{2} \|w\|^2 \tag{2}$$

Weight decay constrains the complexity of the model, makes the model smoother, and reduces the influence of noise.

2.2.4 Momentum in SGD

The disadvantage of SGD is that it may cause local optimal or oscillations. To address this problem, momentum in SGD is to reduce oscillation and speed up the convergence. The purpose of momentum is to add the momentum coefficient got from the last gradient to the current gradient. In addition, the value of the coefficient will usually be between 0 and 1. Momentum helps accelerate the gradient vectors to move to the same direction as previous updates. Momentum in SGD updates the parameters based on the current gradient and the previous momentum during each iteration,

$$v = \beta v - \alpha \nabla J(\theta) \theta = \theta + v \tag{3}$$

β is the momentum coefficient, which is usually set to a relatively small value such as 0.9. Momentum in SGD takes into account not only the gradient at the current moment, but also the gradient direction of the previous iteration, and adds momentum with a ratio of β.

2.2.5 Dropout

As a method for suppressing overfitting, dropout is typically used to handle complicated deep learning models when weight decay is difficult to apply. This method randomly chooses neurons in the hidden layer and deletes them during training. After the deletion, these abandoned neurons no longer transmit signals.

To be more specific, these randomly deactivated neurons also change randomly in subsequent training iterations. Therefore, each neuron must be prepared for deactivation at any time, which prevents the MNN from becoming too dependent on any one neuron. It should be noted that the dropout method is only applied during the training phase. Although all neuron signals are transmitted during testing, for the output of each neuron, it needs to be multiplied by the dropout ratio used during training. Through this method, the neural network is able to learn more robust features because it must work effectively under different conditions.

In this work, for each forward propagation, the neurons to be removed are preserved in *self.mask* as $False$ values. After that, *self.mask* generates a random array with the same shape as input, and sets the value of element which larger than dropout ratio to true.

2.2.6 Softmax and Cross-entropy Loss

Softmax and Cross-entropy loss layer, as an output method in deep learning models, allows the model to calculate the difference between the output of MNN and the supervised labels, and backpropagate this difference to the neural network, which adjusts the weight parameters accordingly.

In the output layer of MNN, the softmax function transforms the input data into a probability distribution where the probability of each class is between 0 and 1 and the sum of probabilities is equal to 1. After that, the output of softmax is compared with actual class labels using the cross-entropy loss method.

2.2.7 Mini-batch training

When training the neural network model, the most used method is gradient descent. Vectorization can train the dataset effectively, which allows the processing of the entire training set. Assume that the size of the dataset is m. Then the $train\_data$ and $train\_label$ will be like this,

$$\begin{aligned} X(nx,m) &= \begin{bmatrix} x(1) & x(2) & x(3) & \cdots & x(m) \end{bmatrix} \\ Y(1,m) &= \begin{bmatrix} y(1) & y(2) & y(3) & \cdots & y(m) \end{bmatrix} \end{aligned} \tag{4}$$

Batch gradient descent is meant to process the entire dataset at once, which will be computationally expensive and time-consuming, especially if $batch\_size = m$. It will have to process the whole data and calculate the gradient of the cost function referred to each weight. The gradient is used to update the weights to minimize the cost function after each iteration.

The cost function means the difference between the predicted value and actual value of $train\_data$. The principle of mini-batch refers to the idea of breaking down a large

dataset into smaller subsets (mini-batches), which reduces the time cost of each iteration. It is usually used when the size of the dataset is too huge. Mini-batch not only reduces the time cost of computation, but it also introduces noises into the optimization process to help prevent the model from overfitting to the *train_data*. With that, it will be a better performance on the generalization of the data.

2.2.8 Batch Normalization

The idea of batch normalization is to normalize the data by mini-batch during learning. It normalizes the dataset to have a mean of 0 and a variance of 1. the mathematical formulas are expressed as follows,

$$\begin{aligned} \mu_B(mean) &\leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \\ \sigma_B^2(variance) &\leftarrow \frac{1}{m}\sum_{i=1}^{m}(x_i-\mu_B)^2 \\ \hat{x_i} &\leftarrow \frac{x_i-\mu_B}{\sqrt{\sigma_B^2+\varepsilon}} \end{aligned} \tag{5}$$

The m input data are got from set B and the mean μ_B and variance σ_B^2 are calculated and then normalized. During the normalization, ε is a small value used to prevent division by zero.

Batch normalization can speed up learning, it allows a larger learning rate. It can reduce overfitting because it reduces the need for dropout or any other activation.

2.2.9 Other Advanced Operations

GELU Gaussian error linear unit is an activation function. The input of neural is multiplied by $Bernoulli(\Phi(x))$,

$$GELU(x) = x\Phi(x) \tag{6}$$

$\Phi(x)$ is the cumulative distribution function of the standard normal distribution,

$$\begin{aligned} \Phi(x) &= \int_{-\infty}^{x} \frac{e^{-t^2/2}}{\sqrt{2\pi}} \\ dt &= \frac{1}{2}\left[1+\text{erf}\left(\frac{x}{\sqrt{2}}\right)\right] \\ \text{erf}(x) &= \frac{2}{\sqrt{\pi}}\int_0^x e^{-t^2}dt \end{aligned} \tag{7}$$

So the approximate value of $GELU$ is,

$$\frac{1}{2}x\left[1+\tanh\left(\frac{2}{\pi}\sqrt{x+0.044715x^3}\right)\right] \tag{8}$$

Adam As a combination of AdaGrad and Momentum, Adam gains the advantages of both, allowing it to efficiently search the parameter space and accelerate model convergence.

There are two important concepts in this optimisation method: momentum and adaptive learning rate. Although Momentum is similar to the Momentum method, Adam combines the first-order momentum $\beta 1$ and the second-order momentum $\beta 2$, which represent the mean and variance of the gradient, respectively, used to calculate the update step and the momentum terms,

$$m_t = \beta_1 m_t - 1 + (1 - \beta_1) g_t \tag{9}$$

$$v_t = \beta_2 v_t - 1 + (1 - \beta_2) g_t^2 \tag{10}$$

What make it difference is the feature of bias correction for hyperparameters,

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t} \tag{11}$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t} \tag{12}$$

Finally, update the initial parameter,

$$\theta_{t+1} = \theta_t - \frac{1}{\sqrt{\hat{v}_t} + \varepsilon} \hat{m}_t \tag{13}$$

The above steps ensure a smooth iteration.

2.3 Design of the Best Model

2.3.1 Code Structure

An important premise is that all models are trained under the same code structure and training set, which ensures that the training process of all models are same. The code structure this assignment is shown in Figure 1.

The submission code includes three classes. As shown in Figure 1, the class *Activation* includes all initialized activation function with their derivative functions, which are *ReLU* and *GELU*. These two activation functions can be used throughout the *Activation* class.

In addition, the *HiddenLayer* class creates all functions in each hidden layer in the Multilayer Perceptron (MLP). This class is an object for creating hidden layers in *MLP* class as well as the nodes and the activation function in hidden layer. It shows that there are two essential function which are $forward()$ and $backward()$ indicating the forward propagation and backward propagation in the hidden layer. Additionally, the functions of batch normalization and dropout are both added to the hidden layers. These implement the batch normalization and dropout to reduce the overfitting. Both of them have the forward and backward propagation. Furthermore, an argument $test\_flg$ is used to distinguish the training and testing process in them.

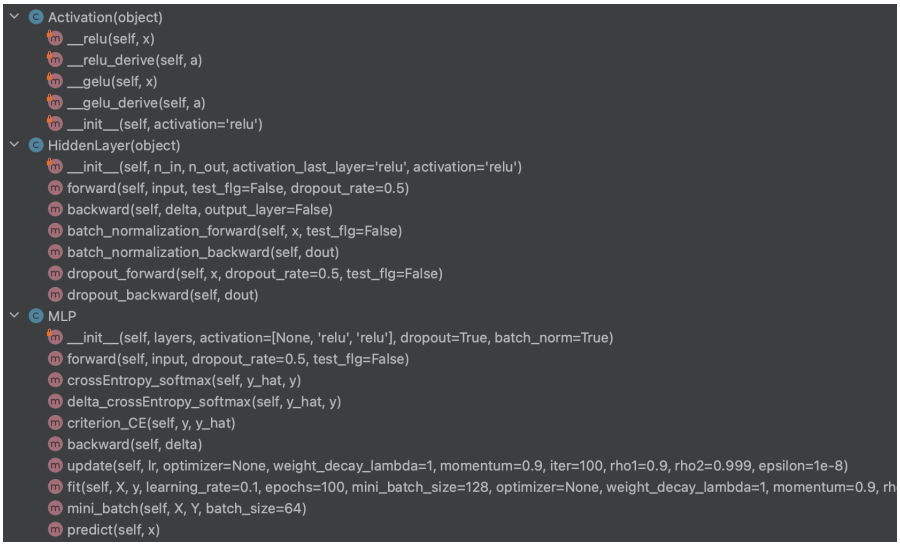


Figure 1: Code Structure

Finally, the main class is the *MLP* that includes all processes in MLP. To initialize it, the parameters can be set to determine the number of the hidden layers, the activation function and whether the batch normalization and dropout are used. It creates the object of hidden layers by the *HiddenLayer* class, which are included in the $forward()$ and $backward()$ for all layers in MLP. The functions $crossEntropy\_softmax()$ and $delta\_crossEntropy\_softmax()$ are used to calculate the loss and derivatives of the outputs created by the forward propagation.

After returning the loss and derivatives using the $criterion\_CE()$, the $backward()$ will update the parameters, such as weights and bias, according to the derivatives. The weights, bias, momentum and all parameters will be updated in the $update()$. Meanwhile, the $fit()$ and $predict()$ control the training and testing based on the mini-batch data training $mini\_batch()$. All the hyper parameters are initialized by using the $fit()$ function.

2.3.2 Hyper Parameters Set

The best performance model is not only consider the highest accuracy but also the minimal overfitting. It is hard to determine the degree of both of them. For example, if using high $dropout\_rate$ for model, the overfitting will be reduced but the accuracy will be small at the same time. Therefore, after the numbers of the experiments, the hyper parameters of the best model are shown below.

With the default nodes in the hidden layer, the $batchnormalization$ and the $dropout$ must be used for mini-batch training and to reduce the overfitting. The *GELU* activation is used due to the better performance than *ReLU*. In addition, the $learning\_rate$ is set to 0.001. The model is trained 200 $epochs$ in the size of 64 $mini\_batch\_size$ with the $adam\ optimizer$. The $weight\_decay\_lambda$ is 0.0001 and the momentum coefficient $momentum$, exponential decay rate $rho1$ $rho2$ are set to default. The $dropout\_rate$ is 0.4 after a lot of experiments. In conclusion, the hyper parameters for the best model are following,

```
nn = MLP([128, 1280, 640, 320, 120, 32, 10], [None, 'gelu', 'gelu', 'gelu', 'gelu', 'gelu', 'gelu'], dropout=True,
        batch_norm=True)

# Try different hyperparameter
# You can set different value of hyperparameter and optimizer.
# optimizer: "momentum", "adam", None
# hyperparameter: learning_rate, epochs, mini_batch_size, weight_decay_lambda, dropout_rate
loss, train_acc, validata_acc = nn.fit(X_train, y_train, learning_rate=0.001, epochs=200, mini_batch_size=64,
                                       optimizer='adam', weight_decay_lambda=0.0001, momentum=0.9, rho1=0.9,
                                       rho2=0.999, dropout_rate=0.4)
```

Figure 2: Parameter of the Best Model

- activation: gelu
- $dropout = True$
- $batch\_norm = True$
- $learning\_rate = 0.001$
- $epochs = 200$
- $mini\_batch\_size = 64$
- $optimizer = "adam"$
- $weight\_decay\_lambda = 0.0001$
- $momentum = 0.9$
- $rho1 = 0.9$
- $rho2 = 0.999$
- $dropout\_rate = 0.4$

The reason for why only using 200 $epochs$ is that the model converges quickly after twenty epochs even with a small learning rate. Because the data set is simple.

Therefore, The training accuracy of this model is 56% and the test accuracy can reach 50%, which will be discussed in the experiments and results part.

3 Experiments and Results

3.1 Hardware and Software Specification

Above all, all the performance of models are evaluated on a laptop with the hardware and software following,

- $Laptop$: MacBook Pro (14inch 2021)
- CPU: Apple M1 Pro
- $CPU\ Core$: 10-Core

- *GPU Core*: 16-Core
- *RAM*: 16GB
- *Compiler*: PyCharm

In addition, the python environment and used python packages are shown below,

- *Python* 3.9.13
- *numpy* 1.21.5
- *matplotlib* 3.5.2
- *sklearn* 1.1.1
- *time*
- *math*

3.2 Performance

Only the performance of the best model will be analyzed in this part because there are too many models for this MLP. Therefore, it is complicated to list all performance of models. This part will illustrate the best model's performance in terms of different evaluation metrics.

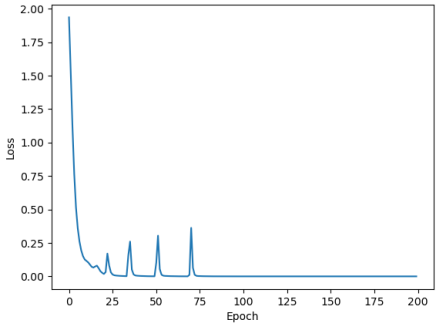


Figure 3: Loss of the Best Model

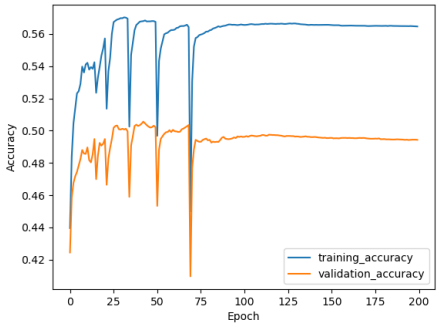


Figure 4: Accuracy of the Best Model

According to the Figure 3, the loss of the best models becomes very stable and approaches zero with training. Meanwhile, the accuracy rate also became stable after the initial shock, and stabilized at around 51%. Although the accuracy of the training set is about 5% higher than that of the test set, the value is within the acceptable range of overfitting as shown in Figure 4.

The hyper parameters of the best model has been discussed in the part 2.3.2. (activation: gelu, $dropout = True$, $batch\_norm = True$, $learning\_rate = 0.001$, $epochs = 200$, $mini\_batch\_size = 64$, $optimizer =$ "$adam$", $weight\_decay\_lambda = 0.0001$, $momentum = 0.9$, $rho1 = 0.9$, $rho2 = 0.999$, $dropout\_rate = 0.4$) The loss and the accuracy figures are shown below,

	Loss	Train_Acc	Val_Acc	Test_Acc	F1	Recall	Precision	Time
Best Model in part 2.3.2	0.000003	56.46%	49.43%	51.03%	48.96%	51.03%	50.38%	116.562 min

Table 1: Performance of the Best Model

As the Table 1 shown, the loss of the best model can reach 0.000003, which is a very low value. But loss cannot represent the accuracy. The test accuracy also achieve 51%, more than the half. If we use a lower dropout rate, the accuracy will be higher. As mentioned in part 2.3.2, we use a 0.4 high dropout rate to control the overfitting. Therefore, it will also reduce the accuracy as expected.

Precision is a measure of how many of the samples predicted to be positive by the classifier are true positives. It means that the classifier was able to correctly distinguish 50% of the positive examples. The recall are same as the test accuracy in all models here. The reason is that the number of predictions for each category in the model's prediction results is equal, that is, the model performs the same on each category. F1 socre is an important evaluation metric. It is the weighted average of precision and recall. In this model, the model performs very well on the two indicators of precision and recall. That is, the model can identify and capture 50% of the positive cases when predicting. However, it takes more time because of using the *Adam* and *GELU* that will be discussed in the following part.

3.3 Extensive Analysis

To have a extensive analysis for each module and hyper parameter, this report achieve the hyper parameter analysis, ablation studies and comparison methods basing on the basic model. The basic model here is considered as the MLP with the *ReLU* activation with the weight decay as shown in Figure 5. All the following experiments are based on it.

The dropout, batch normalization, optimizer, learning rate, mini batch size and the activation function are used to explain the performance of the different modules and different hyper parameters.

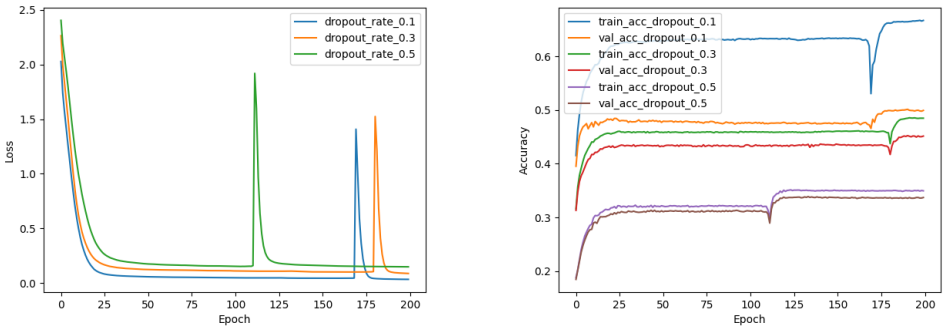
```
nn = MLP([128, 1280, 640, 320, 120, 32, 10], [None, 'relu', 'relu', 'relu', 'relu', 'relu', 'relu'], dropout=True,
        batch_norm=True)

# Try different hyperparameter
# You can set different value of hyperparameter and optimizer.
# optimizer: "momentum", "adam", None
# hyperparameter: learning_rate, epochs, mini_batch_size, weight_decay_lambda, dropout_rate
loss, train_acc, validata_acc = nn.fit(X_train, y_train, learning_rate=0.001, epochs=200, mini_batch_size=64,
                                       optimizer="momentum", weight_decay_lambda=0.0001, momentum=0.9, rho1=0.9,
                                       rho2=0.999, dropout_rate=0.4)
```

Figure 5: Basic Model

Dropout As mentioned before, dropout, which is used to handle complicated deep learning model is the method of suppressing overfitting. It is obvious that a low dropout rate even no dropout will have the high accuracy and severe overfitting, while a high dropout rate will reduce the accuracy and the overfitting. Normally, the dropout rate is

set between the 0.1 and 0.5. Therefore, three dropout rate 0.1, 0.3 and 0.5 is compared here. The results are shown below,



(a) Loss (b) Accuracy

Figure 6: Comparison of Dropout

It is obvious that the low dropout rate (0.1) or none have the highest accuracy but slight overfitting from the Figure 6(b). The reason is that dropout discards several nodes in the hidden layers, which is a functional way to reduce the overfitting. But at the same time, it will also decrease the training ability for model. The higher dropout rate used, the lower accuracy will be. More details of evaluation are shown in Table 2.

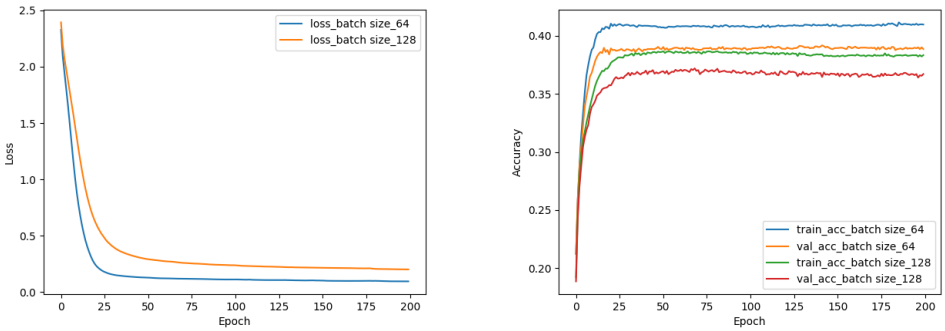
	Loss	**Train_Acc**	**Val_Acc**	**Test_Acc**	**F1**	**Recall**	**Precision**	**Time**
Dropout_rate 0.1	0.0347	66.71%	49.93%	50.06%	49.52%	50.06%	50.21%	50.243 min
Dropout_rate 0.3	0.0886	48.49%	45.15%	45.77%	44.98%	45.77%	47.20%	49.743 min
Dropout_rate 0.5	0.1493	34.96%	33.70%	34.01%	33.21%	34.01%	35.83%	49.268 min

Table 2: Evaluation of Different Dropout Rate

It shows that, as the dropout rate decreases, the accuracy on the test set also increases, but the degree of overfitting is getting bigger and bigger. Therefore, this report selects the 0.4 dropout rate as the best one, which not only ensures accuracy, but also reduces overfitting.

Mini Batch Size Mini-batch training refers to the idea of breaking down a large dataset into smaller subsets, reduces the time cost of each iteration. A large batch size will reduce the training time but also have a lower accuracy than small batch size. This report only consider two batch size, 64 and 128, which are two popular size for small dataset. The results are shown below,

According to the Figure 7, the small batch size will increase the accuracy. Because it means being able to train data in more detail. It will reduce the overfitting slightly with the increase of the batch size, but not much. However, one of the most important function of mini-batch is decreacing the training time.



(a) Loss (b) Accuracy

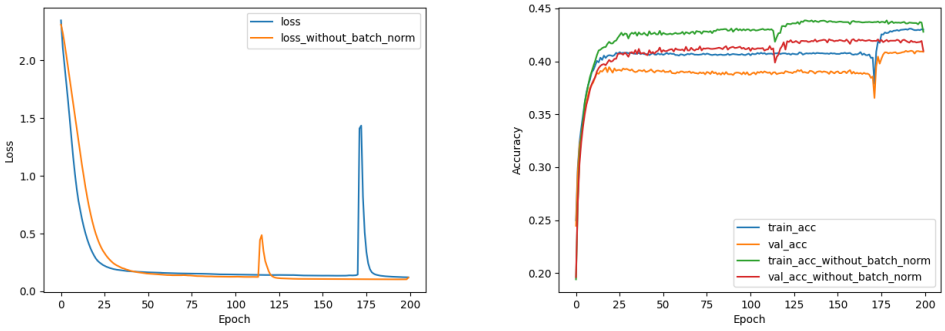
Figure 7: Comparison of Batch Size

	Loss	Train_Acc	Val_Acc	Test_Acc	F1	Recall	Precision	Time
Batch Size 64	0.0952	40.96%	38.85%	40.49%	39.61%	40.49%	42.09%	49.268 min
Batch Size 128	0.2011	38.33%	36.70%	37.50%	35.53%	37.60%	39.20%	36.642 min

Table 3: Evaluation of Different Batch Size

The Table 3 shows that using the large batch size will reduce the training time of the model. But in this program, the better accuracy is preferred. In addition, the extra time is in a acceptable range. Therefore the 64 batch size is used in this code.

Batch Normalization Batch normalization that normalizes the data by mini-batch during learning can speed up learning and reduce overfitting. Due to normalization, the problem of excessive parameter changes caused by different data distributions rarely occurs. In this program, there are two options for batch normalization, *True* and *False*. The results of model with batch normalization and without normalization are shown below,



(a) Loss (b) Accuracy

Figure 8: Comparison of Batch Normalization

As shown in Figure 8(b), it is hard to find the gradient without the batch normalization. Because the mini-batch training is used in this program, it is essential to pull an

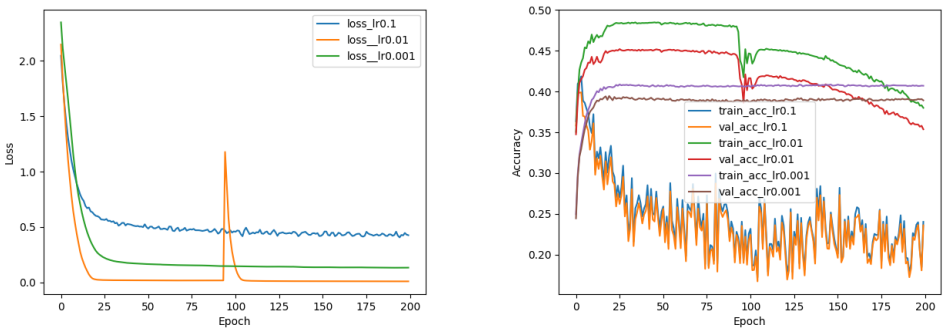
increasingly skewed distribution back to a normalized distribution. That is, batch normalization can make the gradient larger and speed up the learning convergence speed. At the same time, it can avoid the problem of gradient disappearance.

	Loss	Train_Acc	Val_Acc	Test_Acc	F1	Recall	Precision	Time
Batch Normalization	0.1217	43.00%	40.90%	42.14%	40.54%	42.14%	42.86%	49.268 min
Without Batch Normalization	0.1186	43.79%	41.00%	42.02%	40.79%	42.02%	43.40%	51.634 min

Table 4: Evaluation of Batch Normalization

The Table 4 cannot show some difference between them. But according to the Figure 8, it is obviously that using batch normalization can speed up the learning convergence speed. In addition, using batch normalization has a better gradient during training.

Learning Rate If the learning rate is too large, the model cannot converge easily. In addition, shocks may occur near the optimal point, resulting in unstable training. Small learning rate will lead to slow convergence and fall into local optimal solution. The learning rate is usually set between 0.1 and 0.001. Three learning rate, 0.1, 0.01 and 0.001 are compared in this evaluation as the following,



(a) Loss (b) Accuracy

Figure 9: Comparison of Learning Rate

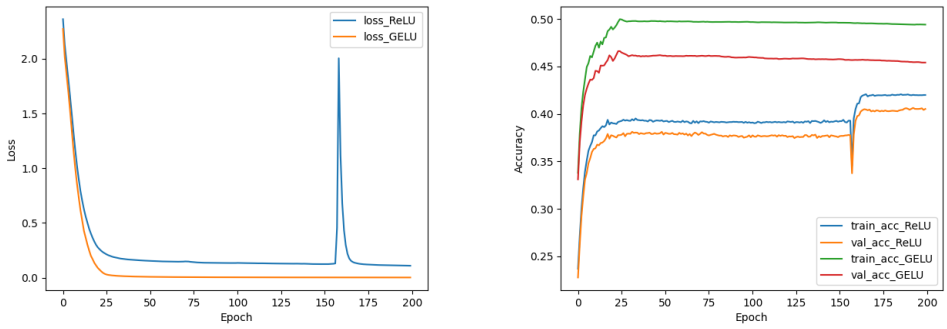
The Figure 9(a) shows that the large learning rate has the highest loss, while the small learning rate has lower loss and convergence after several epochs. In addition, excessive learning rate also leads to the problem of gradient explosion and skipping the optimal solution when updating parameters which will result in lower and lower accuracy of the model. These problems are shown in the Figure 9(b). It is obvious that the accuracy with 0.1 and 0.01 learning rate becomes low because of the gradient explosion. The reason is that the learning rate is too large to skip the optimal solution. On the contrary, a learning rate of 0.001 has a more stable accuracy.

As shown in Table 5, the 0.001 learning rate performs better than others in all of evaluation metrics. Therefore, the 0.001 is selected as the learning rate in this program due to the best accuracy.

	Loss	Train_Acc	Val_Acc	Test_Acc	F1	Recall	Precision	Time
0.1 lr	0.4258	24.04%	23.60%	23.92%	17.89%	23.92%	50.27%	50.832 min
0.01 lr	0.0094	38.00%	35.37%	36.09%	35.65%	36.09%	49.32%	51.325 min
0.001 lr	0.1330	40.70%	38.92%	40.05%	38.47%	40.05%	41.38%	49.268 min

Table 5: Evaluation of Learning Rate

Activation Function The activation function is an important part in MLP. In this report, the *RuLU* and *GELU* are only considered. Both of them are non-linear function. The advantage of *ReLU* is that it is simple and fast, but it will cause the problem of dead neurons. *GELU* is an activation function based on a Gaussian error function, thus avoiding the problem of dead neurons. But its computational complexity is high. The performance of them are shown below,



(a) Loss (b) Accuracy

Figure 10: Comparison of Activation Function

It is obvious that the *GELU* activation function has a better accuracy than the *RuLU* according to the Figure 10. In addition, *GELU* converges more quickly than *RuLU* and *GELU* has a more stable gradient. However, the Figure 10(b) also shows that, *GELU* has a larger degree of overfitting. More evaluations between them are shown in Table 6.

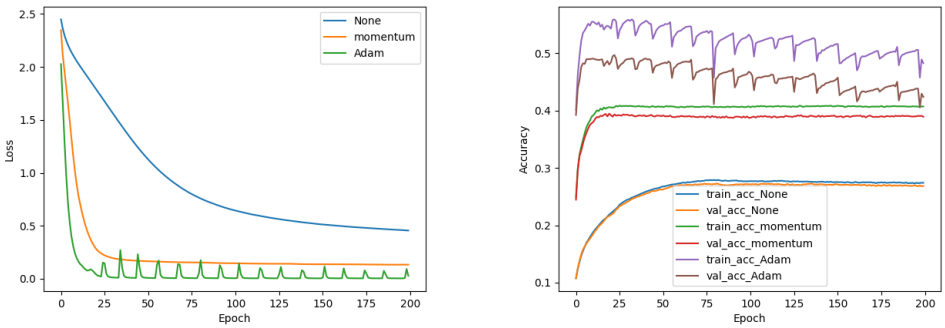
	Loss	Train_Acc	Val_Acc	Test_Acc	F1	Recall	Precision	Time
RuLU	0.1097	42.00%	40.53%	41.25%	40.08%	41.25%	43.77%	49.268 min
GELU	0.0022	49.42%	45.42%	46.49%	45.91%	46.49%	47.99%	79.154 min

Table 6: Evaluation of Activation Function

According to the Table 6, the *GELU* has the lower loss and the higher accuracy and other metric scores. However, the code running time of *GELU* is much longer than *RuLU*. Because the calculation process of *GELU* involves the calculation of the Gaussian error function, it needs to be realized by numerical approximation and other methods, which will increase the complexity and time consumption of the calculation.

Optimizer *momentum* and *Adam* are two normally used optimizer. *momentum* algorithm can effectively reduce the shock of parameter update and speed up the learning speed. The *Adam* algorithm performs well in most cases, and has good convergence

performance and generalization performance. This report considers these two optimizer and compare them with no optimizer to find the influence of them.



(a) Loss (b) Accuracy

Figure 11: Comparison of Optimizer

First of all, these three performance are based on the *ReLU* activation function mentioned in Figure 5. Therefore, according to the Figure 11(b), the accuracy of *Adam* fluctuates and gradually decreases. The reason is that *ReLU* activation function may cause the problem of gradient disappearance or explosion during back propagation and *Adam* may cause problems of overfitting and insufficient generalization ability. However, *Adam* has a better performance with the *GELU* as mentioned in part 3.2.

The Figure 11 also shows that using optimizer has a better performance than not. In addition, the loss will be larger if an optimizer is not used. More details are shown following,

	Loss	**Train_Acc**	**Val_Acc**	**Test_Acc**	**F1**	**Recall**	**Precision**	**Time**
None	0.4552	27.40%	26.84%	27.04%	25.26%	27.04%	38.61%	45.226 min
momentum	0.1330	40.70%	38.92%	40.05%	38.47%	40.05%	41.38%	49.268 min
Adam	0.0271	48.28%	42.38%	44.01%	42.23%	44.01%	46.65%	86.470 min

Table 7: Evaluation of Optimizer

As shown in Table 7, using the optimizer can make the loss of the model smaller and the accuracy higher. And for both optimizer, *Adam* performs better than *momentum*. In addition, because *Adam* calculation is more complicated, it takes more time. Although the combination of *Adam* and *ReLU* does not perform well, *Adam* and *GELU* have relatively better performance in part 3.2. More detail for the optimizer and activation will be discussed following.

The total evaluation of the optimizer with their activation functions are shown in Figure 8. It is obvious that using the optimizer is better than not as the same activation condition. In addition, the *Adam* performs better than the *momentum* under the same activation functions. What's more, *GELU* always has a better accuracy than the *ReLU*, but it also takes more time. The best model discussed in part 3.2 can achieve 0.00000 loss after 200 epochs and it has the highest accuracy.

	Loss	Train_Acc	Val_Acc	Test_Acc	F1	Recall	Precision	Time
None with ReLU	0.4491	31.98%	31.03%	31.56%	29.20%	31.56%	36.06%	45.226 min
None with GELU	0.0238	48.82%	44.86%	45.58%	44.99%	45.58%	46.03%	90.864 min
momentum with ReLU	0.1480	41.65%	39.94%	40.93%	39.57%	40.93%	42.76%	49.268 min
momentum with GELU	0.0015	49.87%	45.66%	47.22%	46.23%	47.22%	47.71%	79.154 min
Adam with ReLU	0.0066	54.91%	48.31%	48.54%	48.39%	48.54%	51.46%	86.470 min
Adam with GELU	0.000003	56.46%	49.43%	51.03%	48.96%	51.03%	50.38%	116.512 min

Table 8: Combination Evaluation

3.4 Justification

After analyzing the different modules and parameters and combining the explanation of part 3.2. It is obvious that among the best models mentioned before, the most accurate modules and hyper parameter values were used. The previous combination of the modules is also analyzed in part 3.2. In other words, the best model is based on the highest accuracy that the modules can obtain. However, due to various considerations, we cannot only focus on improving the accuracy rate. Such as the size of the dropout rate. It is possible to achieve higher accuracy with a small dropout rate or even without dropout. But this will cause severe overfitting. Therefore, the 0.4 dropout rate used by the best model is a choice considering the accuracy and overfitting degree.

According to the Table 2, Figure 3 and Figure 4, the best model can achieve more than half of the accuracy, while the accuracy of most models is basically below 50%. That is to say, the best modules analyzed previously can indeed improve the performance of the model. Generally speaking, a 6% gap is not serious overfitting, which is within the acceptable range. In addition, other evaluation metrics are also the best performance in comparison. Precision and recall rate are difficult to reflect alone as evaluation metrics. However, the F1 score is an important metric that can reflect the quality of the model. The higher the F1 score, the better the model is. Finally, although the best model is the model that takes the longest, this is because it uses *GELU* and *Adam*, which are two more complex models than others. At the same time, 116 minutes is also a relatively fast time interval. But time is not a more accurate metric. Because the instability of computer equipment, such as whether the device is charging or not, whether there are other programs in the background will affect the running time.

4 Discussion and Conclusion

In conclusion, this code implements all required modules and the required modules are effectively reflected in the result. According to the code introduction, different hyper-parameters can be adjusted to achieve different modules. Based on the tutorial code, the report implements a complete code structure and can have excellent performance. On the other hand, this report discusses in detail the principles of the different modules and their role in the program. In addition, the team members spent several days testing different parameters after implementing the code. By adjusting different parameters,

the respective roles of the different modules are discovered. Afterwards, the best combination of parameters achieves the best model. The report analyzes the best models and corresponding modules. Overall, the code implements all modules to the maximum extent and is able to achieve 51% accuracy. After 200 iterations, the best models stabilized.

The report discusses the interaction between the different modules. Dropout can reduce the degree of overfitting, but it will also reduce the accuracy. Similarly, using a large mini batch size will speed up the training, but will also reduce the accuracy. Batch normalization can reduce the degree of overfitting on the basis of mini batch training. Although a large learning rate will speed up the learning speed, it will cause the gradient to drop too fast and cause the gradient to explode. While a small learning rate is stable, it may cause the model to fail to converge. Therefore, an appropriate learning rate is also very important. The choice of optimizer also depends on the needs of the model, *momentum* has a more stable gradient descent and faster speed than *Adam*. But *Adam* performance is better in most cases. For the activation function, *GELU* effectively prevents *DeadReLU* and improves accuracy compared to *ReLU*, although *GELU* requires longer training time. The report draws the above conclusions by analyzing different results and principles.

However, this model still has some room for improvement. First of all, whether there is a way to make *Adam* more stable. Because the *Adam* has a unstable performance although it has a better accuracy. In addition, The accuracy rate of the test set can only reach about 50%. Although the use of dropout reduces the accuracy to a certain extent, it can be compensated for in other ways. For example, dropout rate can be reduced and the batch normalization is used to compensate for the overfitting caused by not using dropout. However, it is found that the effect of batch normalization is not as good as dropout according to the implementation. Therefore, other methods may be needed to solve this problem. On the other hand, it is not possible to achieve hundreds or thousands of epochs due to the hardware equipment. And many models converge after 200 epochs. At the same time, it was found in the experiment that using *GELU* may cause data overflow. This is because the value in the data is too large, which may cause the weight value of the model to become very large during calculation. The value is too large when the *GELU* activation function is in back propagation. Under this situation, the training data can be normalized or standardized in advance to solve it. However, it was found that this method did not improve the accuracy after standardized implementation. Therefore this code does not employ this technique.

Overall though, there are still many issues that have not been resolved, but this program fully implements the required modules and has a good performance.

A Code Link

https://drive.google.com/drive/folders/1DVHNM79By-jZOB_rS5086BC_C-aHrxZ4?usp=share_link

B Instruction for Code

The code is build on the Colab. Therefore, you only need to use the web link above and **Run block by block**. The process of loading the dataset is included in the block.

To successful run the code. The following packages are required.

- *Python*3
- *numpy*
- *matplotlib*
- *sklearn*
- *time*
- *math*

To change the hyper parameters for multiple training. You can change them in the *Learning* block in Colab code. The following options can be changed or selected:

- activation: "relu", "gelu"
- *dropout*: True, False
- *batch_norm*: True, False
- *optimizer*: None, "momentum", "adam"
- *learning_rate*
- *epochs*
- *mini_batch_size*
- *weight_decay_lambda*
- *momentum*
- *rho*1
- *rho*2
- *dropout_rate*