Implementation of Multi-layer Neural Network

COMP 5329 Assignment 1 Report

Shoudi Huang
Department of Engineering
University of Sydney
Sydney, Australia
500478204

Shi Su
Department of Engineering
University of Sydney
Sydney, Australia
520108095

Jiahe Song
Department of Engineering
University of Sydney
Sydney, Australia
520215083

***Abstract*—Classification is a fundamental problem in the field of machine learning. In this study, a multi-class classification modal was implemented from scratch based on a Multi-layer Neural Network. Apart from the essential modules such as activation module, many optimization and regularization modules such as dropout, mini-batch training, and batch normalization were included for better performance. The data set used contains 128 features and 50,000 instances, classified into 10 classes. This report contains four parts where the first part introduces the aim and importance of this study, second part describes the training methodology, third part demonstrates the experiment results, forth part discusses the findings and conclusions, and the last part introduces the experiment software, hardware, and how to run the code.**

***Index Terms*—Keywords: Multi-class classification, Multi-layer Neural Network**

I. Introduction

A. Aim of this study

The goal of this assignment is to build a multi-class classification model based on a Multi-layer Neural Network. Additionally, the model needs to be implemented based on scientific computing libraries instead of deep learning frameworks. Besides, the data set adapted in this study consists of 128 numerical features and 50,000 instances, which were classified into 10 classes. All required modules will be demonstrated in the next section.

B. Importance of this study

The perceptron concept introduced by Rosenblatt, et al. in 1958 [1] is a milestone of deep learning. But that model does not have much practical value since it can only deal with data that can be linearly divided. Rumelhart et al. then introduced the concept of back propagation and hidden layers in 1986 [2]. Based on the hidden layers concept, more neuron nodes were added between the inputs and outputs, which led to the development of the Multi-layer Neuron Network adapted in this study. Thus, implementing this modal and applying it to a multi-class classification task is meaningful since this model represents an important turning point of deep learning discipline. Furthermore, implementing the network from scratch would help us have a better understanding of essential, optimization, and regularization modules such as activation function and batch normalization.

[Code & Dataset Link]

II. Methodology

A. Pre-processing

The pre-processing techniques used in this assignment are normalization and standardization. Both techniques aim to scale the input features to a smaller range thus improving convergence, accuracy and reducing the impact of outliers. Standardization:

$$x_{\text{stand}} = \frac{x - \text{mean}(x)}{\text{standard deviation } (x)} \tag{1}$$

Normalization:

$$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{2}$$

B. Principle of different modules

1) ReLU activation:

$$f(s) = max(0, s) \tag{3}$$

Rectified linear unit (ReLU) is an activation function that is commonly used in hidden layers. It maps all negative inputs to zero and keeps positive inputs the same. Its simplicity allows lower computational costs and makes it very easy to implement. A potential risk of ReLU activation function refers to Dead ReLUs, which may cause a part of neurons to be inactivated.

*2) * GELU activation:*

$$GELU(x) = x \cdot \frac{1}{2}[1 + erf(x/\sqrt{2})] \tag{4}$$

The Gaussian Error Linear Unit (GELU) was introduced in 2016. [3] It looks very similar to ReLU activation and can be considered as a smooth version of ReLU. Unlike ReLU, GELUs allow both positive and negative values, which helps reduce Dead ReLU problem. Furthermore, according to the original literature, “ReLU gates the input depending upon its sign, while the GELU weights its input depending upon how much greater it is than other inputs.” [3]

3) Weight decay:

$$\underset{\theta}{min}\hat{L}_R(\theta) = \hat{L}(\theta) + \frac{\alpha}{2}\left\|\theta\right\|_2^2 \tag{5}$$

Unexpected high weights in the network is one of the main reasons causing overfitting. Weight decay, also known as L2 regularization, is a popular method that addresses the overfitting issue. It adds an extra term into the loss function, which limits the growth of weights. In other words, the model will be forced to reduce weight as penalties were made by high weight values.

4) Momentum in SGD:

$$v_t = \gamma v_{t-1} + \eta\nabla_\theta J(\theta) \tag{6}$$

$$\theta_t = \theta_{t-1} - v_t \tag{7}$$

Stochastic Gradient Descent (SGD) often suffers from oscillations in loss function and slow convergence to the optimal solution. Momentum algorithm aims to resolve these two problems by adding a fraction of the previous weight update to the current update. Thus, the model will be awarded if it keeps moving towards the optimal result. In other words, the model will be accelerated in the right direction while limited in the wrong directions. Its value is often set around 0.9.

5) Dropout: Dropout is a straightforward algorithm that addresses the issue of overfitting in neural networks. It has two different approaches, randomly cuts off a part of the neurons based on probability p, or generates a mask to drop forward connections based on probability 1-p. By training many smaller sub-models, it allows most of the neurons to learn during the training process instead of few neurons with high weight value. This study adapts the latter approach, also known as Dropconnect.

6) Softmax:

$$\hat{P}(class_k \mid x) = z_k = \frac{e^{net_k}}{\sum_{i=1}^{K} e^{net_i}} \tag{8}$$

Softmax, also known as the normalized exponential function, is commonly applied before the output layer, especially for the multi-class classification. It maps the outputs to a probability distribution in range (0, 1) and probability for each class sums to 1. Predictions could be made based on picking the class with the highest probability.

7) Cross-entropy loss:

$$CrossEntropy(t, z) = Entropy(t) + D_{KL}(t \mid z) \tag{9}$$

Cross-entropy loss is a popular loss function for classification tasks. After the outputs go through the softmax function, the cross-entropy loss function is then adapted to calculate the training error. More specifically, the cross-entropy is calculated by adding entropy of target distribution and the KL (Kullback-Leibler) divergence between output and target.

8) Mini-batch training: Mini-batch training is a compromise approach of batch gradient descent and stochastic gradient descent. It successfully reduces the computational cost by subsetting the training set. However, the size of the mini-batch needs to be considered carefully to avoid noisy and unstable updates problem similar in stochastic gradient descent.

9) Batch Normalization:

$$\mu_B \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \tag{10}$$

$$\sigma_B^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_B)^2 \tag{11}$$

$$\hat{x}_i \leftarrow \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \tag{12}$$

$$y_i \leftarrow \gamma\hat{x}_i + \beta \equiv BN_{\gamma,\beta}(x_i) \tag{13}$$

Batch Normalization (BN) is an important regularization method introduced by Ioffe and Szegedy in 2015 [4], which is designed to reduce the internal covariate shift problem . In order to save compute time, it takes a batch of the whole data set, calculates the variance and mean value, then applies normalization on the input of the next layer. Furthermore, two parameters γ and β were introduced to shift and scale the normalized value as the normalization process will eliminate some information.

C. Design and Justification of our best model

The hyperparameter value for our best model is shown below:

TABLE I
BEST MODEL'S HYPERPARAMETER

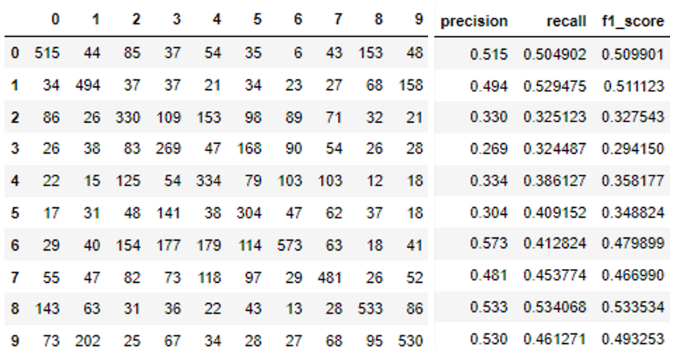
Hyperparameter	Default Value
Layer_sizes	[128, 200, 10]
Dropout_rate	0.1
Weight_decay	0
Momentum	0.95
Activation	"ReLU"
Epochs	50
Batch_size	128
Learning_rate	0.001

The best model chosen based on high test accuracy, feasible running time, best value of hyperparameter chosen by tuning, and best modules chosen by ablation study. Before implementing hyperparameter tuning and ablation study, the dataset is first preprocessed by a standardscaler to make the inputs stay in a smaller range, and thus reducing the impact of outliers. Then, the train dataset is divided into 80% of the train set and 20% of the validation set, the model will do the hyperparameter tuning on the train set and check the train accuracy on the validation set. After hyperparameter tuning, the model will be trained again on the whole dataset by combining the train set and validation set, and checking the test accuracy on the test set. The final model with the best combination of hyperparameters is considered as our best model.

III. Experiments & Results

A. Performance

The following tables describe the confusion matrix and precision, recall, and f1 score of our best model. The plot shows a clear trend of training accuracy and test accuracy after 50 epochs of training of our model and change in cross entropy loss during each epoch.



	0	1	2	3	4	5	6	7	8	9	precision	recall	f1_score
0	515	44	85	37	54	35	6	43	153	48	0.515	0.504902	0.509901
1	34	494	37	37	21	34	23	27	68	158	0.494	0.529475	0.511123
2	86	26	330	109	153	98	89	71	32	21	0.330	0.325123	0.327543
3	26	38	83	269	47	168	90	54	26	28	0.269	0.324487	0.294150
4	22	15	125	54	334	79	103	103	12	18	0.334	0.386127	0.358177
5	17	31	48	141	38	304	47	62	37	18	0.304	0.409152	0.348824
6	29	40	154	177	179	114	573	63	18	41	0.573	0.412824	0.479899
7	55	47	82	73	118	97	29	481	26	52	0.481	0.453774	0.466990
8	143	63	31	36	22	43	13	28	533	86	0.533	0.534068	0.533534
9	73	202	25	67	34	28	27	68	95	530	0.530	0.461271	0.493253

Fig. 1. Confusion Matrix and Evaluation Matrix Table

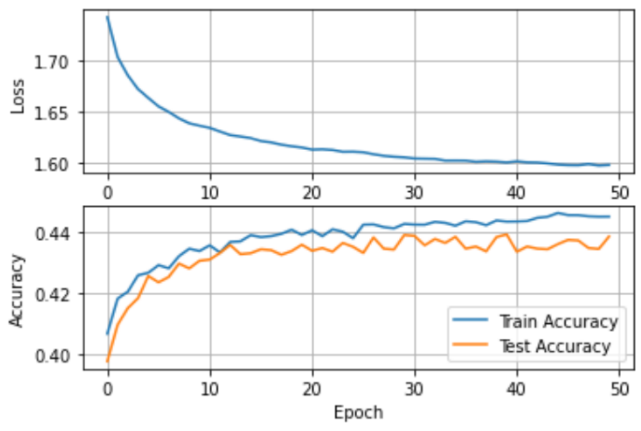


Fig. 2. Upper: Train Accuracy and Test Accuracy in each epoch
Lower: Cross Entropy Loss in each epoch

Our best model reaches a 44.60% train accuracy and 43.63% of test accuracy after 50 epochs of training and hyperparameters tuned. The final model is not overfitting. From the confusion matrix and evaluation matrix, our best model has relatively high ability to predict label 0, 1, 6, 7, 8, 9 and poor performance on predicting label 2, 3, 4, and 5.

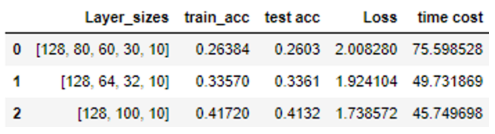
B. Hyperparameter Analysis

The hyperparameters that can be tuned in our model are layer_sizes, dropout_rate, weight_decay, momentum, activation, epochs, batch_size and learning_rate. The default values of these parameters are shown in the Table II. The hyperparameter tuning is done by setting a list of values to a certain parameter and keeping other parameters constant. The preferred value of a certain parameter is determined by the cross entropy loss, train accuracy, test accuracy, and running time.

TABLE II
DEFAULT HYPERPARAMETER

Hyperparameter	Default Value
Layer_sizes	[128, 64, 32, 10]
Dropout_rate	0.5
Weight_decay	0.001
Momentum	0.9
Activation	"ReLU"
Epochs	50
Batch_size	64
Learning_rate	0.001

1) Layer Sizes: The layer_sizes controls the architecture of the MLP model, the default value is [128, 64, 32, 10], which contains two hidden layers with 64 neurons and 32 neurons in them. The hyperparameter tuning for layer size first test the different numbers of hidden layers by setting a list of layer_sizes = [[128, 80, 60, 30, 10], [128, 64, 32, 10], [128, 100, 10]].

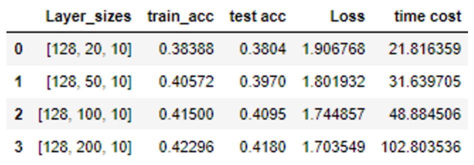


	Layer_sizes	train_acc	test acc	Loss	time cost
0	[128, 80, 60, 30, 10]	0.26384	0.2603	2.008280	75.598528
1	[128, 64, 32, 10]	0.33570	0.3361	1.924104	49.731869
2	[128, 100, 10]	0.41720	0.4132	1.738572	45.749698

Fig. 3. Performance of models with different numbers of hidden layers

The outcome showed that more hidden layers won't improve the performance of our model but make it worse. According to the table above, our best model will only have one hidden layer.

2) Number of neurons: We now keep the model with one hidden layer and then tune the model with different numbers of neurons in each hidden layer. Setting layer_sizes = [[128, 20, 10], [128, 50, 10], [128, 100, 10], [128, 200, 10]]



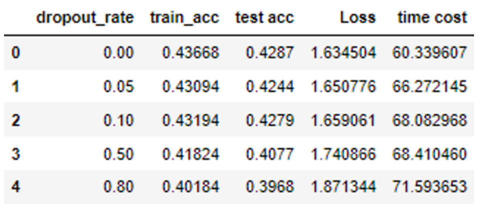
	Layer_sizes	train_acc	test acc	Loss	time cost
0	[128, 20, 10]	0.38388	0.3804	1.906768	21.816359
1	[128, 50, 10]	0.40572	0.3970	1.801932	31.639705
2	[128, 100, 10]	0.41500	0.4095	1.744857	48.884506
3	[128, 200, 10]	0.42296	0.4180	1.703549	102.803536

Fig. 4. Performance of models with different numbers of neurons

The outcome shows that more neurons in the hidden layer will improve the performance of the model but it also increases the running time of the MLP as a more complicated neural network will increase the computing cost generally, so the chosen number of neurons is set to be 200.

3) Dropout Rate: The dropout_rate is a regularization technique used in deep learning models to prevent overfitting. It randomly set a proportion of the activations in a layer to zero and thus forces the model to learn more robust features instead of heavily relying on specific features. The dropout_rate is set to [0, 0.05, 0.1, 0.5, 0.8].

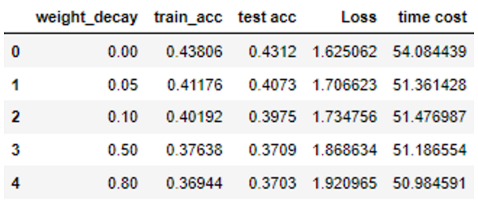
From the above table, the best dropout_rate is set to 0.1 since the train accuracy and test accuracy are relatively low. The dropout rate is unlikely the cause of the poor performance.



	dropout_rate	train_acc	test acc	Loss	time cost
0	0.00	0.43668	0.4287	1.634504	60.339607
1	0.05	0.43094	0.4244	1.650776	66.272145
2	0.10	0.43194	0.4279	1.659061	68.082968
3	0.50	0.41824	0.4077	1.740866	68.410460
4	0.80	0.40184	0.3968	1.871344	71.593653

Fig. 5. Dropout Rate tuning

4) Weight_decay: The weight_decay is also a regularization method to help prevent overfitting in deep learning models. It is a form of L2 regularization which adds a penalty term to the cross entropy loss function and penalizes large weights. The weight_decay is set to $[0, 0.05, 0.1, 0.5, 0.8]$

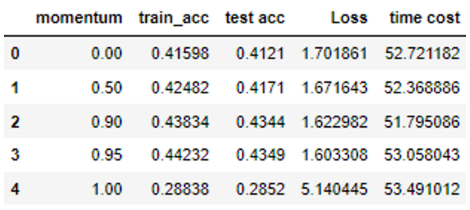


	weight_decay	train_acc	test acc	Loss	time cost
0	0.00	0.43806	0.4312	1.625062	54.084439
1	0.05	0.41176	0.4073	1.706623	51.361428
2	0.10	0.40192	0.3975	1.734756	51.476987
3	0.50	0.37638	0.3709	1.868634	51.186554
4	0.80	0.36944	0.3703	1.920965	50.984591

Fig. 6. Weight decay tuning

Just like the dropout rate regularization, since our model is not overfitting, the usage of weight decay to prevent overfitting is not the key point to improve the poor performance, so the weight decay is set to zero in our best model.

5) Momentum: Momentum is a crucial parameter in stochastic gradient descent (SGD), it can speed up convergence and improve robustness during training. In SGD with momentum, the momentum coefficient is a hyperparameter that controls the contribution of the previous velocity to the current update. The momentum is set to [0, 0.5, 0.9, 0.95, 1].

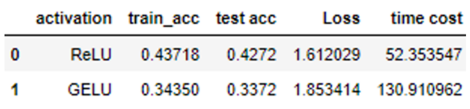


	momentum	train_acc	test acc	Loss	time cost
0	0.00	0.41598	0.4121	1.701861	52.721182
1	0.50	0.42482	0.4171	1.671643	52.368886
2	0.90	0.43834	0.4344	1.622982	51.795086
3	0.95	0.44232	0.4349	1.603308	53.058043
4	1.00	0.28838	0.2852	5.140445	53.491012

Fig. 7. Momentum tuning

The momentum is usually set to 0.9 or similar value, and according to our hyperparameter tuning outcome, the value of momentum equal to 0.95 is a little bit better than the default value, as it has higher train accuracy and lower cross entropy loss.

6) Activation functions: The activation functions are important to make the deep learning capture non-linearity relationship between inputs and outputs. The sigmoid and tanh activation functions are not used here since both suffer from the vanishing gradient problem, and have higher computing cost compared to ReLU. So we include ReLu as our baseline activation function and softmax at the last output layer. Also, we add the advanced activation function GELU into the list for optimization.

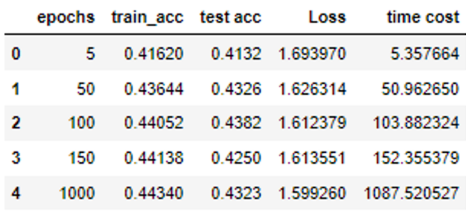


	activation	train_acc	test acc	Loss	time cost
0	ReLU	0.43718	0.4272	1.612029	52.353547
1	GELU	0.34350	0.3372	1.853414	130.910962

Fig. 8. Activation function results

Even though the GELU is expected to be more advanced than ReLU, the hidden layer using ReLU as activation function is outperformed than GELU with with higher train and test accuracy and lower time cost. The activation for our best model is chosen to be ReLU.

7) Epochs: The epoch is a critical hyperparameter in deep learning, an increase in number of epochs will help the MLP learn more about the train dataset but it has potential to lead overfitting and high computing cost if the epochs are set too big.

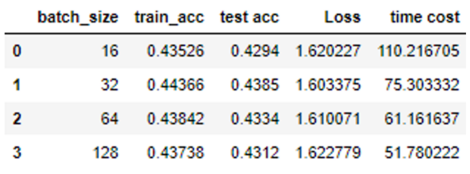


	epochs	train_acc	test acc	Loss	time cost
0	5	0.41620	0.4132	1.693970	5.357664
1	50	0.43644	0.4326	1.626314	50.962650
2	100	0.44052	0.4382	1.612379	103.882324
3	150	0.44138	0.4250	1.613551	152.355379
4	1000	0.44340	0.4323	1.599260	1087.520527

Fig. 9. Epochs results

The epoch is 50 in our final model since increasing the number of epochs does not improve much of performance but dramatically increases the running time of the model.

8) Batch size: The batch_size refers to the number of training examples used in each mini-batch. A larger batch size will result in faster training speed for one epoch and stable gradient, while a small batch size may result in computationally expensive and noisy gradient.



	batch_size	train_acc	test acc	Loss	time cost
0	16	0.43526	0.4294	1.620227	110.216705
1	32	0.44366	0.4385	1.603375	75.303332
2	64	0.43842	0.4334	1.610071	61.161637
3	128	0.43738	0.4312	1.622779	51.780222

Fig. 10. Batch size tuning

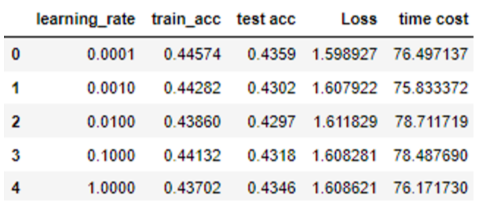
By considering the trade-off between large batch size and small batch size, we choose batch size = 128 as the model has the fastest running time without decreasing too much of the overall performance of the model .

9) Learning Rate: The learning rate controls the pace of learning of the model in each epoch. If the learning rate is too high, the model may overshoot the optimal solution, and if the learning rate is too low, the model may converge very slowly.

The impact of learning rate on model performance is very slight, we choose the learning rate = 0.001 for the best model.

TABLE III
PERFORMANCE COMPARISON

Conditions	Train accuracy	Test accuracy	Loss	Time(s)
With dropout	0.34156	0.3406	1.915508	53.883172
No dropout	0.38844	0.3888	1.723494	51.962972
With weight_decay	0.18924	0.1848	2.204626	52.981601
No weight_decay	0.33344	0.3321	1.929978	55.392453
With momentum	0.34194	0.3420	1.917647	52.736851
No momentum	0.32862	0.3179	1.974109	52.276606
With mini-batch training	0.3254	0.3247	1.934469	58.311987
No mini-batch training	0.1000	0.1000	2.304279	1122.555957



	learning_rate	train_acc	test acc	Loss	time cost
0	0.0001	0.44574	0.4359	1.598927	76.497137
1	0.0010	0.44282	0.4302	1.607922	75.833372
2	0.0100	0.43860	0.4297	1.611829	78.711719
3	0.1000	0.44132	0.4318	1.608281	78.487690
4	1.0000	0.43702	0.4346	1.608621	76.171730

Fig. 11. Learning rate tuning

C. Ablation Study

The baseline model in the ablation study of this assignment is the MLP model with default value of hyperparameters shown in previous section. The ablation study aims to investigate the impact of individual components on the overall performance of the deep learning model, so the study compared 4 pairs of conditions, including changing dropout rate, weight decay, momentum and batch size to evaluate these modules' performance on the MLP model.

Since the performance of our baseline model is relatively poor, the model does not have an overfitting problem. The models with dropout and weight decay regularization have significantly worse results than models without these regularization modules.

The model using SGD with momentum clearly has better performance since the momentum parameter accelerates the convergence, the model with momentum reaches a 34.3% test accuracy but the model without momentum only has 32% of test accuracy.

The mini-batch training helps the model to be more stable and accurate, most importantly, it improves the computational efficiency. The running time of a model without mini-batch training exceeds over 1100 seconds which is about 18 minutes and with only 10% of train accuracy. The model with mini-batch training significantly improves the accuracy of the model and reduces time cost.

In general, the model uses SGD with momentum and mini-batch training will significantly increase performance, the model uses dropout rate and weight decay will have some side effects on the overall performance in this dataset.

D. Comparison Methods

The comparison methods used in this assignment is comparing train accuracy, test accuracy, cross entropy loss, time cost of model training and other evaluation matrix like recall, precision, f1 score. The time cost is a critical comparison method if two models have close performance on accuracy but huge differences between time cost. The model with lower time cost is always much preferred.

From the previous hyperparameter tuning, the overall performance is mostly influenced by the architecture of the MLP model. For comparison, We design a new model by changing the number of hidden layers and neurons, and see if the new model can beat the default model and best model.

The default model uses two hidden layers, ReLU activation function, weight decay, momentum in SGD, dropout, sofmax and cross entropy loss, mini-batch training and batch normalization.

The best model uses one hidden layer, ReLU activation function, momentum in SGD, sofmax and cross entropy loss, mini-batch training and batch normalization.

The new model uses two hidden layers, ReLU activation function, momentum in SGD, sofmax and cross entropy loss, mini-batch training and batch normalization.

The first hidden layer of the new model has 200 neurons, and the second hidden layer has 120 neurons. The values of other hyperparameters are the same as the best model shown before.

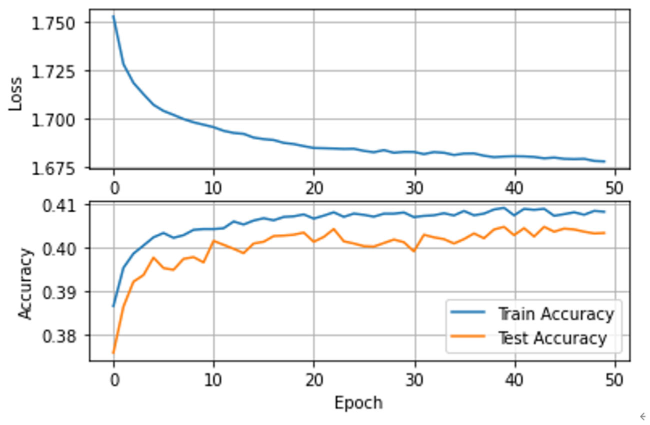


Fig. 12. Upper: Cross-entropy lossLower: Train/Test accuracy

The cross entropy loss and train test accuracy figures are shown above. The new model has 40.65% test accuracy compared to other models.

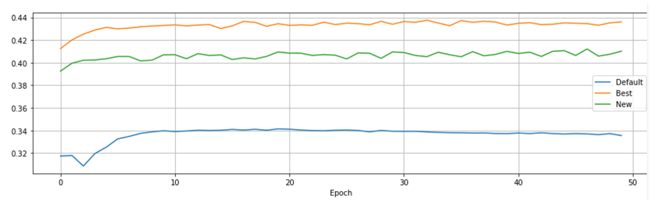


Fig. 13. Accuracy comparison

TABLE IV
PERFORMANCE COMPARISON

Models	Time cost(s)
Best model	72.4364
Default model	58.1136
New model	115.4658

By comparing the test accuracy of three models, the winner is still the best model described before. As the new model is much more complicated than other models, the time cost is higher than others as expected.

IV. Discussion and Conclusion

This aims to classify a ten-class dataset consists of 128 features by producing a Multi-layer Neural Network from scratch. The best model in this study achieved an accuracy of 43% due to architectural limitations, we believe that the accuracy can be improved by adapting to more complex architectures. On the other hand, the main goal of this project was to deeper understanding about optimization modules such as momentum and regularization modules such as batch normalization. Moreover, this study also tuned several hyper-parameters for better accuracy and computational efficiency. Furthermore, this study can be continued in the direction of adding grid search, helps hyper-parameter tuning process.

V. Experiment environment & reproduce

A. How to run the code

Code & Dataset Link

1) Run the first block of code to download the files needed.
2) Run the second block of code to load the MLP model
3) Run the third block of code to load our best model, the code will print epoch number, cross-entropy loss, train accuracy, test accuracy after each epoch and print time cost at the end of epochs.
4) Run the fourth and fifth block of code to show the cross-entropy loss, train accuracy, and test accuracy plots.
5) Run the Last two blocks of code to show the confusion matrix and recall, precision, f1 score.

B. Hardware and Software Specifications

- Jupyter Notebook
- Python 3.8.8
- Numpy 1.21.5
- Matplotlib 3.5.2

TABLE V
HARDWARE SPECIFICATION

Hardware	Model
CPU	Intel(R) Core(TM) i7-10875H CPU
GPU	NVIDIA GeForce RTX 2070 Super
Installed RAM	16.0 GB (15.9 GB usable)
System	64-bit operating system
System Edition	Windows 11 Pro
Version	21H2

References

[1] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. Psychological Review, 65(6), 386–408. https://doi.org/10.1037/h0042519

[2] Rumelhart, D., Hinton, G. & Williams, R. (1986). Learning representations by back-propagating errors. Nature 323, 533–536. https://doi.org/10.1038/323533a0

[3] Hendrycks, D. & Gimpel, K. (2016). Gaussian Error Linear Units (GELUs). arXiv preprint arXiv:1606.08415.

[4] Ioffe, Sergey; Szegedy, Christian (2015). Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. arXiv:1502.03167