Optimization and Evaluation of Multi-layer Neural Networks: Exploring Regularization, Learning Rates, and Topologies

Linghang Kong
Halıcıoğlu Data Science Institute
University of California San Diego
San Diego, CA 92093
l3kong@ucsd.edu

Weiyue Li
Halıcıoğlu Data Science Institute
University of California San Diego
San Diego, CA 92093
wel019@ucsd.edu

Yi Li
Halıcıoğlu Data Science Institute
University of California San Diego
San Diego, CA 92093
yil115@ucsd.edu

Abstract

In this work, we implement a multi-layer neural network equipped with forward and backward propagation, various regularization techniques, and momentum-based optimization. Our objective was to classify Japanese Hiragana handwritten characters from the KMNIST dataset, employing softmax as the output layer. One-fold cross-validation was utilized to evaluate the model, coupled with the integration of regularization techniques. Our most efficient model leveraged ReLU activations and achieved an accuracy of **0.8688**. Subsequent architecture adjustments, including layer count and hidden unit modifications, yielded a test set accuracy of **0.8626**.

1 Introduction

In this study, we present the design and implementation of a sophisticated multi-layer neural network. This network leverages both forward and backward propagation mechanisms, incorporates a range of regularization techniques, and utilizes momentum-based optimization for enhanced performance. To prepare the dataset for our neural network, we embarked on a preprocessing journey wherein the image data, initially in the form of 2-dimensional arrays with dimensions $(28, 28)$, was transformed into one-dimensional arrays of size 784. Furthermore, we applied z-score normalization to standardize the dataset and ensure better convergence during training. To guarantee a randomized distribution of data, we shuffled the samples prior to the onset of the training process. Our dataset was strategically divided into an 80% segment for training and a 20% segment for validation. In a bid to streamline our evaluation and conserve computational resources, we employed **one-fold cross-validation**. After meticulous iterations and adjustments to the learning rate, our model displayed an impressive accuracy of 0.8546 on the test set, achieved with a learning rate of 0.05. The inclusion of L1 and L2 regularization techniques further fine-tuned our model, culminating in an enhanced test accuracy of 0.8725. This optimal performance was realized with L2 Regularization and a regularization parameter, λ, valued at 1×10^{-3}. As a final exploratory step, we delved into experimenting with a variety of activation functions and pondered over diverse network topologies to discern their impact on the model's efficiency.

2 Back Propagation

We have implemented backward propagation and tested our calculated gradient with both backward propagation and numerical approximation. We have used the cross-entropy as the loss function and the softmax as the output layer activation function.

2.1 Preliminary Math Work

Before implementing, it is essential to understand the math behind the process.

For backward propagation, we need to calculate the derivative of the loss with respect to the weights and bias accordingly.

First, we calculate the gradient with respect to the output of the output layer by using the cross-entropy function $E^n = -\sum_{i=1}^{c} t_i^n \ln y_i^n$ and the definition of $\delta_i^n = -\frac{\partial E^n}{\partial a_i^n}$ where n represents the n^{th} pattern. We get:

$$\delta_k^n = t_k^n - y_k^n \tag{1}$$

Then, we calculate the gradient with respect to the hidden layer with the output of the same hidden layer. According to the derivation, the δ_j should follow the rules of:

$$\delta_j^n = g'(a_j^n) \sum_k \delta_k^n w_{jk}^n \tag{2}$$

In the above equation, j, k symbolized the current layer and previous layer accordingly. Whereas $g(a_j^n)$ symbolized the activation function of the previous layer.

(Note: we have followed the notation used in the lecture where j is for the hidden layers and k is for the output layer).

2.2 Implementation

With equations 1 and 2, we can derive the following equations for the gradient of loss with respect to the weights and the gradient of loss with respect to the bias with some algebraic manipulations:

$$\frac{\partial E^n}{\partial w_j} = \delta_j a_j^n \tag{3}$$

$$\frac{\partial E^n}{\partial b_j} = \delta_j \tag{4}$$

With the equations 3 and 4 above, we have vectorized the above equations for faster computation in our implementations.

2.3 Testing Introduction

Suggested in the write-up, we have used the following equation to test approximate the gradient of loss with respect to the weights:

$$\frac{d}{dw} E^n(w) \approx \frac{E^n(w+\epsilon) - E^n(w-\epsilon)}{2\epsilon}, \ \epsilon > 0, \epsilon \in \mathbb{R} \tag{5}$$

The w here is one weight in the neural network. After setting our $\epsilon = 10^{-2}$, we compared the approximated gradient with the gradient calculated by back propagation. We have randomly chosen one output bias weight, one hidden bias weight, and two hidden to output weights and two input to hidden weights to compared the differences. You can check out our implementation in the `check_gradient()` method in `train.py`. For each selected weight, we have done the following:

Table 1: Architecture of the baseline model

WEIGHTS	$E(W+\epsilon)$	$E(W-\epsilon)$	**NORM OF APPROXIMATED GRADIENT**	**NORM OF BACKWARD PROPAGATION GRADIENT**	**ABSOLUTE ERROR**
Hidden Weight 1	2.915601	2.915669	-0.003394	-0.003394	4.689468e-10
Hidden Weight 2	1.568530	1.568079	0.02255498	0.02255499	6.128687e-9
Output Weight 1	4.945193	4.944855	0.0168997	0.0168995	2.532141e-7
Output Weight 2	2.789656	2.788244	0.0706445	0.0706437	7.989562e-7
Hidden Bias	2.789018	2.788877	0.0070264	0.0070262	1.803573e-7
Output Bias	1.052991	1.052658	0.016604	0.016603	2.630992e-7

1. Add ϵ to one weight of the weights vector, run the forward propagation, and find $E^n(w+\epsilon)$ by using 5.

2. Subtract ϵ to one weight of the weights vector, run the forward propagation, and find $E^n(w-\epsilon)$ by using 5.

3. Run forward and then backward propagation to the unmodified weights vector and get the backward propagation calculated gradient.

4. Compare the values of both approximated and the backward propagation calculated gradient.

2.4 Testing result

The Table 1 shows the result of our test. It is apparent that all the absolute error calculated are within big-O of ϵ^2. Number of decimal points are remained based on whether differences between norm of approximated gradient and norm of backward propagation gradient can be seen.

3 Gradient Descent & Training Procedure

3.1 Minibatch Stochastic Gradient Descent with Momentum

We used minibatch Stochastic Gradient Descent (SGD) throughout all the problems for this programming assignment. SGD in this assignment is similar to programming assignment 1, except that we have also added momentum to our algorithm. A term called velocity is implemented for both weights and bias, denoted as v_w and v_b. The goal of having momentum is to reduce oscillations in the changing gradient, so that loss can be minimized more precisely and efficiently. We set a weight term γ for momentum as 0.9 per assignment instruction. Now the algorithm works as the pseudo code below:

Algorithm 1 Updating rule with momentum

```
1: procedure UPDATING RULE WITH MOMENTUM
2:
3:     Initialization:
4:     $w \leftarrow$ randomized weight
5:     $d_w \leftarrow$ weight gradient
6:     $b \leftarrow$ randomized bias
7:     $d_b \leftarrow$ bias gradient
8:     $\alpha \leftarrow$ learning rate
9:     $\gamma \leftarrow$ momentum gamma
10:
11:    Update velocity:
12:    $v_w \leftarrow \gamma \cdot v_w + (1-\gamma) \cdot d_w$
13:    $v_b \leftarrow \gamma \cdot v_b + (1-\gamma) \cdot d_b$
14:
15:    Update bias and weight:
16:    $w \leftarrow w - \alpha \cdot v_w$
17:    $b \leftarrow b - \alpha \cdot v_b$
```

3.2 Training Procedure

The training procedure in this programming assignment follows a very standard deep learning model training method. The training data passes in as minibatches of fixed batch size 128 per assignment instruction, and were then split into training and validation sets. The training set consists of 80% of the original training data, while the validation set makes up the rest 20%. We were only completing one fold cross validation for this assignment. Batches were forwarded into the network, and their initial loss and accuracy were produced. Then we utilized the loss and accuracy from the input batch to perform backpropagation, thus updating weights and bias. Early stopping is also implemented to prevent overfitting. In general, during training, when validation loss at certain continuous epochs becomes greater than the epochs before, the model must have over-adapted to the training set, thus underfitting the validation set. In such cases, it is justifiable to stop the training process early, rather than receiving a model that fails to recognize general traits in our dataset.

Algorithm 2 Training Procedure with SGD

```
 1: procedure TRAINING PROCEDURE WITH SGD
 2:     train_acc ← []
 3:     val_acc ← []
 4:     train_loss ← []
 5:     val_loss ← []
 6:     best_model ← None
 7:     best_model_loss ← float(”inf”)
 8:     model ← NeuralNetwork
 9:     patience ← 0
10:     for epoch ← range(0,100) do
11:         Shuffle X_train, y_train, X_val, y_val
12:         for batch ← batch_generator do
13:             model.forward(batch[0], batch[1])
14:             model.backward()
15:         train_pred, train_losses ← model.forward(X_train, y_train)
16:         train_accuracy ← accuracy(train_pred, y_train)
17:         val_pred, val_losses ← model.forward(X_val, y_val)
18:         val_accuracy ← accuracy(val_pred, y_val)
19:         if val_losses < best_model_loss then
20:             best_model_loss ← val_losses
21:             best_mode ← copy.deepcopy(model)
22:         if val_losses > best_model_loss then
23:             patience+ = 1
24:             if patience > 5 then
25:                 break
26:             else
27:                 patience ← 0
28:         train_loss.append(train_losses)
29:         train_acc.append(train_accuracy)
30:         val_loss.append(val_losses)
31:         val_acc.append(val_accuracy)
32:     return train_acc, val_acc, train_loss, val_loss, best_model
```

For details of our implementations, please refer to our `train.py`.

3.3 Training Results

By using the default setup with the following parameters in Table 2:

Table 2: Default parameters of our model

PARAMETER	DEFAULT VALUE
layer_specs	[784, 128, 10]
activation	tanh
learning_rate	0.005
batch_size	128
epochs	100
early_stop	False
early_stop_epoch	5
L2_penalty	0.0001
momentum	True
momentum_gamma	0.9

We built our default model. The accuracy and loss on the training and validation set are presented in the line graphs Figure 1 and 2 below.

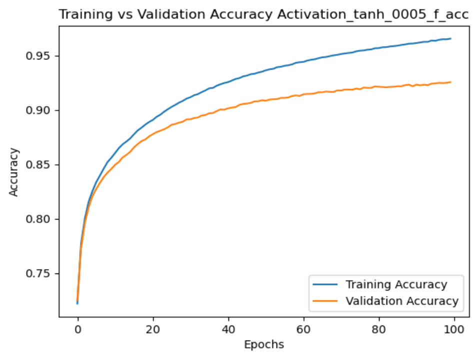

Figure 1: Training and Validation Accuracy For the MLP Using Default Settings

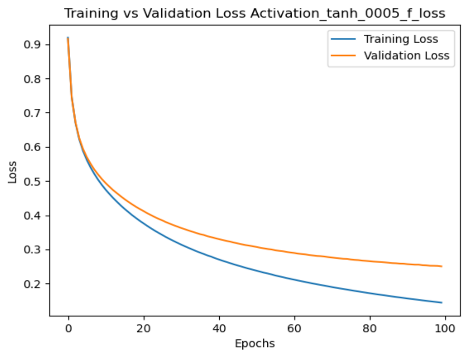

Figure 2: Training and Validation Loss For the MLP Using Default Settings

Afterwards, we tested other learning rates of 0.05, 0.01, and 0.005, and turn on the early stop. Their corresponding losses and accuracy are shown in the line graphs Figure 3 and 4, Figure 5 and 6, and Figure 7 and 8 below:

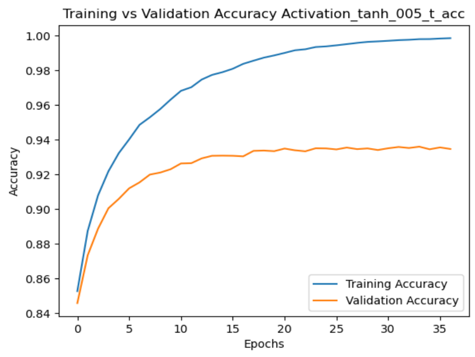

Figure 3: Training and Validation Accuracy For the MLP Using Learning Rate 0.05

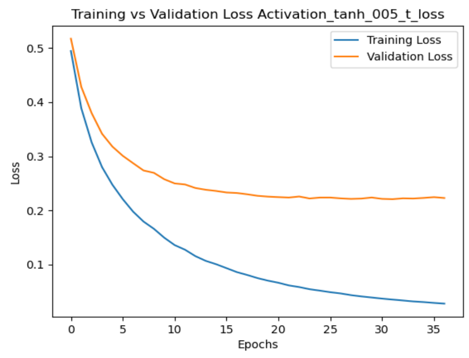

Figure 4: Training and Validation Loss For the MLP Using Learning Rate 0.05

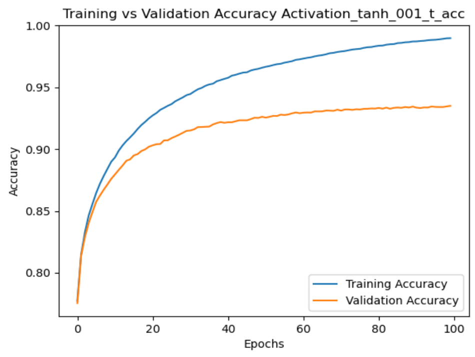

Figure 5: Training and Validation Accuracy For the MLP Using Learning Rate 0.01

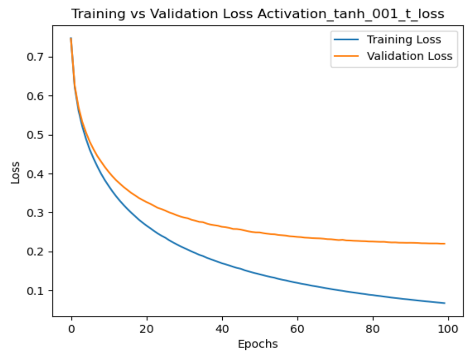

Figure 6: Training and Validation Loss For the MLP Using Learning Rate 0.01

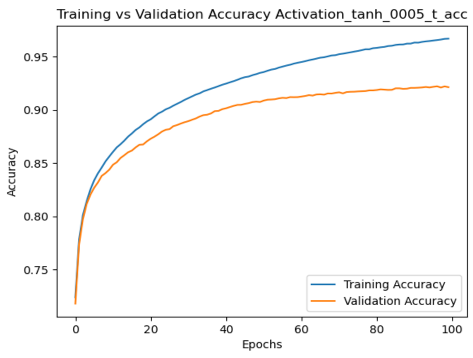

Figure 7: Training and Validation Accuracy For the MLP Using Learning Rate 0.005

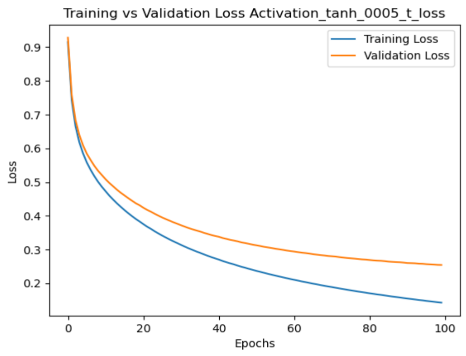

Figure 8: Training and Validation Loss For the MLP Using Learning Rate 0.005

Table 3: Learning rates, accuracy, and loss with default momentum and early stop epoch

LEARNING RATE	**TRAIN ACCURACY**	**TRAIN LOSS**	**VALIDATION ACCURACY**	**VALIDATION LOSS**
0.005	0.965333	0.143583	0.927833	0.253461
0.05	0.997146	0.036258	**0.9345**	**0.222675**
0.01	0.989542	0.067687	0.929833	0.233185

As shown in Table 3 above, we finally chose learning rate 0.05 for testing since its model has the lowest validation loss. Finally, we achieved an accuracy of **0.8546** on the test set.

4 Regularization

In this section, we have experimented with adding different weight decay value to the update rule by using both L1 and L2 regularization starting from the best model we found in part c. After using the equations 6 and 7 for both regularizations, we have aimed to find the optimal amount of regularization λ given the number of epochs increased to 110 and the rest of the hyperparemeters for the best model in part (c) stays unchanged.

4.1 L2 Regularization

$$\text{L2 Regularization} = \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \lambda \sum_{j=1}^{N} w_j^2 \tag{6}$$

Table 4: Accuracy, and loss with 110 epochs and other parameters in part (c) at different L2 λ

PENALTY λ	**TRAIN ACCURACY**	**TRAIN LOSS**	**VALIDATION ACCURACY**	**VALIDATION LOSS**	**TEST ACCURACY**
1e-3	0.9938	0.0690	0.9489	**0.1781**	**0.8725**
1e-6	0.9999	0.0051	0.9333	0.2591	0.8469

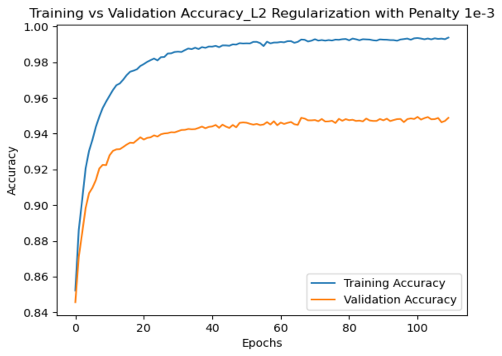

Figure 9: Training and Validation Accuracy For the MLP Using L2 Regularization 1e-3

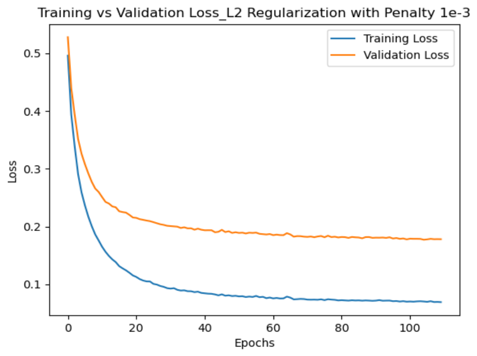

Figure 10: Training and Validation Loss For the MLP Using L2 Regularization 1e-3

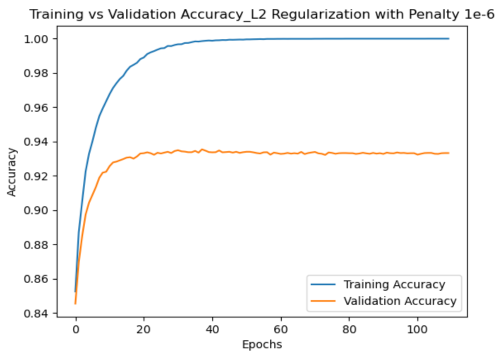

Figure 11: Training and Validation Accuracy For the MLP Using L2 Regularization 1e-6

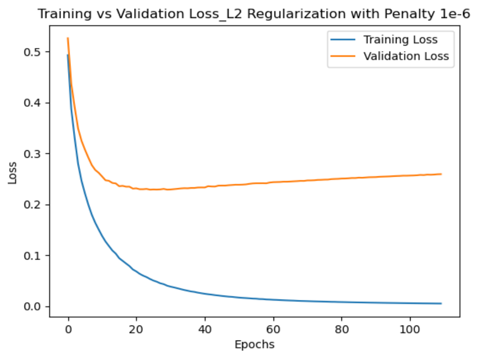

Figure 12: Training and Validation Loss For the MLP Using L2 Regularization 1e-6

4.2 L1 Regularization

$$\text{L1 Regularization} = \sum_{i=1}^{N} l(y_i, \hat{y}_i) + \lambda \sum_{j=1}^{N} |w_j| \tag{7}$$

Table 5: Accuracy, and loss with 110 epochs and other parameters in part (c) at different L1 λ

PENALTY λ	TRAIN ACCURACY	TRAIN LOSS	VALIDATION ACCURACY	VALIDATION LOSS	TEST ACCURACY
1e-3	0.9302	0.2515	0.9091	0.3094	0.8154
1e-6	0.9999	0.0052	0.9337	**0.2574**	**0.8495**

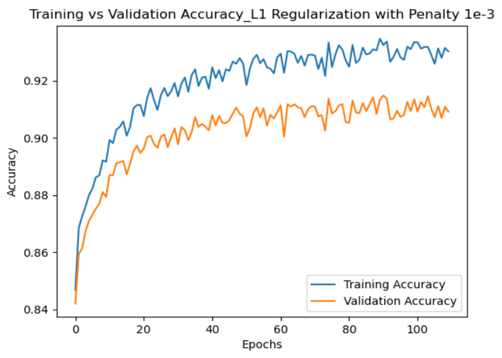

Figure 13: Training and Validation Accuracy For the MLP Using L1 Regularization 1e-3

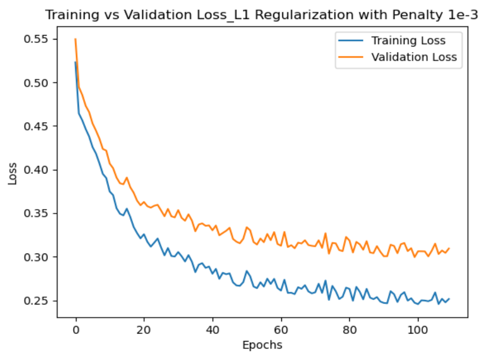

Figure 14: Training and Validation Loss For the MLP Using L1 Regularization 1e-3

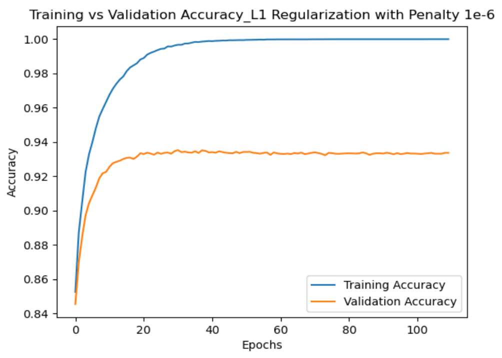

Figure 15: Training and Validation Accuracy For the MLP Using L1 Regularization 1e-6

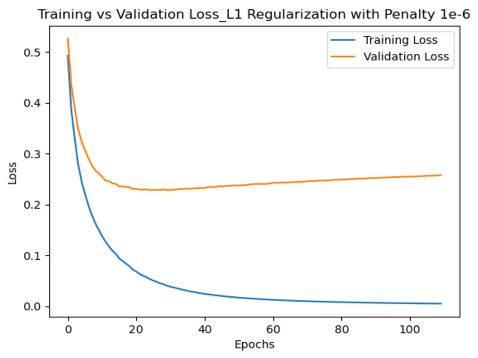

Figure 16: Training and Validation Loss For the MLP Using L1 Regularization 1e-6

4.3 Differences and Conclusion on Regularization

From the Table 4, 5 and graphs above, we have optimized L2 regularization with penalty λ = 1e-3, it produces a minimized validation loss of 0.1781, and a final test accuracy of 0.8725. For L1 regularization, it is optimized at penalty λ = 1e-6 with a validation loss of 0.2574 and a test accuracy of 0.8495. According to the graphs, L2 regularization has very steady loss decreasing and accuracy increasing graphs without too much oscillation. However, much oscillation is shown for L1 regularization with a larger penalty. This might be due to the fact that L1 regularization has a sparse solution, thus would be more likely to miss the minimum with a larger penalty. When penalty is reduced, we can observe significant improvement in model's performance.

5 Activations

In this section, we have experimented different actication functions and compared the differences in terms of the validation accuracy and loss. We applied the best performing model's parameter to test on different activation functions. However, due to different activation functions, some of the models may not converge in 100 epochs, so we adjusted the epochs to 200.

5.1 Sigmoid

The Sigmoid function is given by equation 8:

$$f(z) = \frac{1}{1+e^{-z}} \tag{8}$$

The key characteristic is that the Sigmoid function has a range of 0 to 1. i.e. $f(z) \in (0,1)$. As a result, it has a great vanishing gradient problem when the input is too large, which will affect the effectiveness of updating the weights.

5.2 Tanh

The Tanh function is given by equation 9:

$$f(z) = \frac{e^{z}-e^{-z}}{e^{z}+e^{-z}} \tag{9}$$

The key characteristic is that the Tanh function has a range of -1 to 1. i.e. $f(z) \in (-1,1)$. As a result, it also has a great vanishing gradient problem when the input is too large, which will affect the effectiveness of updating the weights. However, the difference is that this activation function is centered at 0.

5.3 ReLU

The ReLU function is given by equation 10:

$$f(z) = max(0, z) \tag{10}$$

The key characteristic is that the ReLU function has a range of 0 to infinity. i.e. $f(z) \in (0, \infty)$. As a result, it will be able to prevent the vanishing gradient problem in comparison to the previous two activation functions.

5.4 Different Activation Results

Table 6: Accuracy, and loss with different activation functions

ACTIVATION FUNCTION	TRAIN ACCURACY	TRAIN LOSS	VALIDATION ACCURACY	VALIDATION LOSS	TEST ACCURACY
Sigmod	0.993438	0.05249	0.94058	0.20047	0.8587
Tanh	0.998396	0.02781	0.93225	0.22849	0.8516
ReLU	0.997729	0.01941	0.94483	**0.20109**	**0.8688**

Table 6 contains the accuracy and loss of test set trained with different activation functions. It is clear that the ReLU activation function has the best performance because of its ability to resolve the vanishing gradient problem.

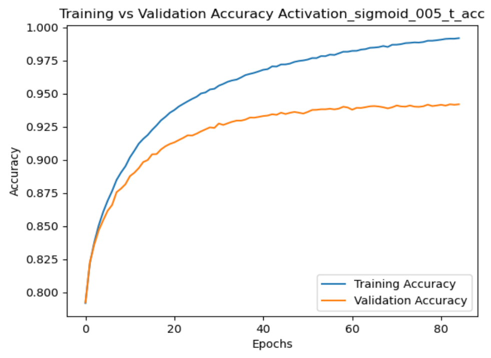

Figure 17: Training and Validation Accuracy For the MLP Using Sigmoid as Activation Function

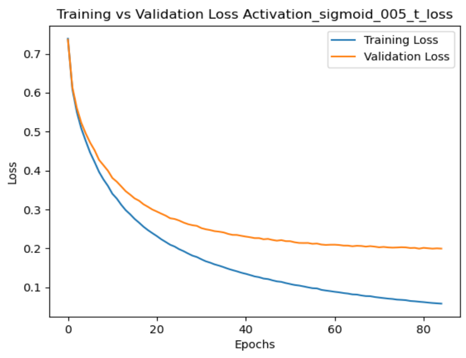

Figure 18: Training and Validation Loss For the MLP Using Sigmoid as Activation Function

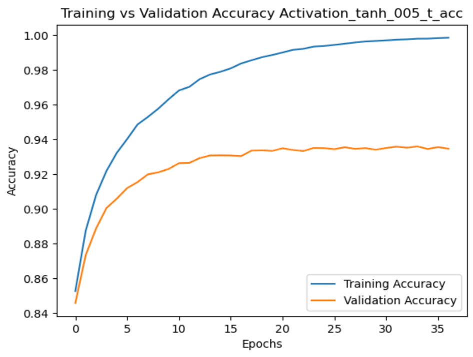

Figure 19: Training and Validation Accuracy For the MLP Using Tanh as Activation Function

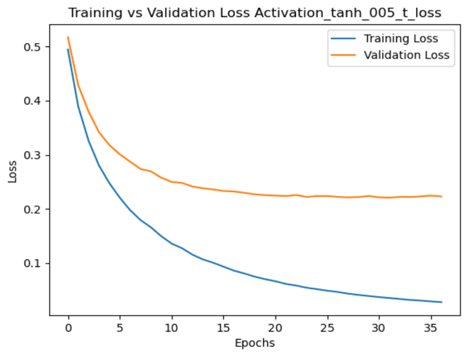

Figure 20: Training and Validation Loss For the MLP Using Tanh as Activation Function

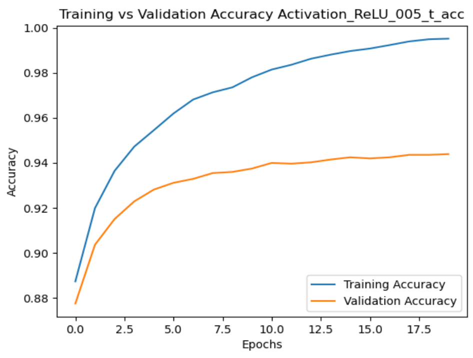

Figure 21: Training and Validation Accuracy For the MLP Using ReLU as Activation Function

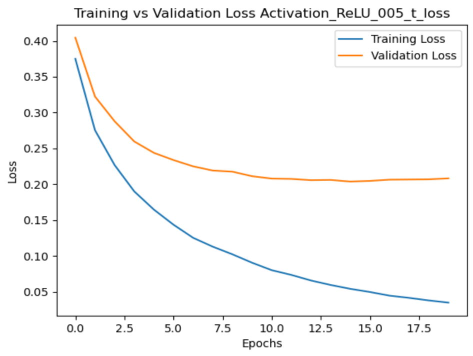

Figure 22: Training and Validation Loss For the MLP Using ReLU as Activation Function

5.5 Conclusion

By using the best performing parameter obtained from assignment part (c), We achieved best test accuracy of **0.8688** with ReLU activation function. According to the graphs, both tanh and ReLu activation functions take approximately 20 to 30 epochs to converge, while Sigmoid activation function requires more than 80 epochs to converge.

6 Network Topology

In this section, we have experimented with different topology of the neural network starting from the best model from part c. Training and validation loss and accuracy, and each topology's accuracy are shown in the graphs below. The topologies include halving the number of units in the hidden layer, doubling the number of hidden units in the hidden layer, and adding a second hidden layer. To make sure that using two hidden layers of equal size and has approximately the same number of parameters for the best model choice in the previous experiment, we set the number of hidden units in both hidden layers to 203.

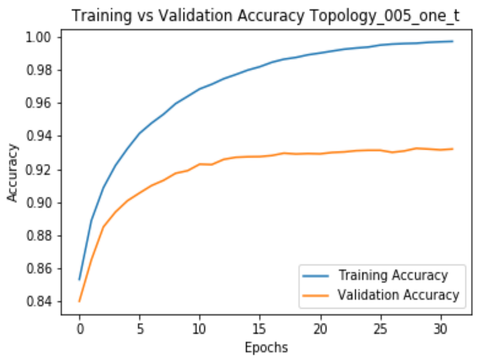

Figure 23: Training and Validation Loss For the MLP Using 128 Hidden Units

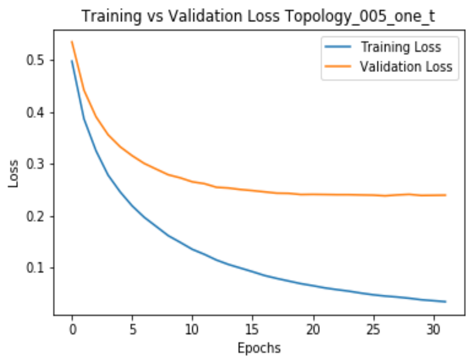

Figure 24: Training and Validation Loss For the MLP Using 128 Hidden Units

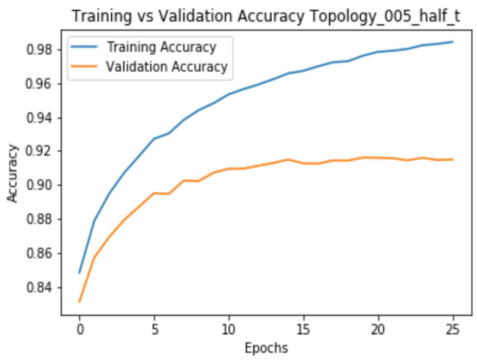

Figure 25: Training and Validation Accuracy For the MLP Using 64 Hidden Units

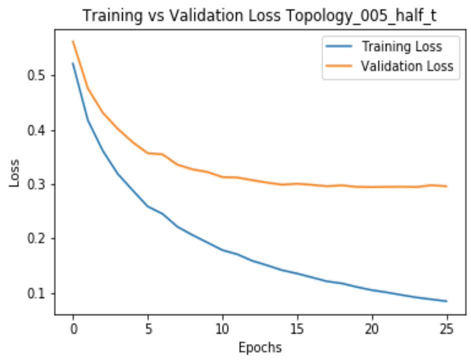

Figure 26: Training and Validation Loss For the MLP Using 64 Hidden Units

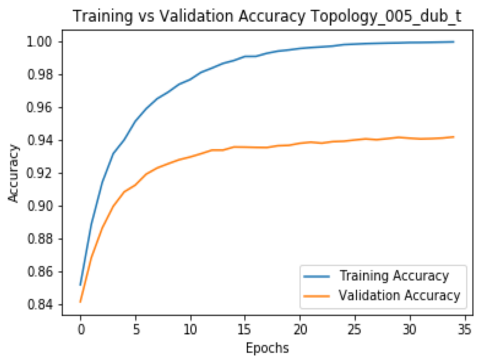

Figure 27: Training and Validation Accuracy For the MLP Using 256 Hidden Units

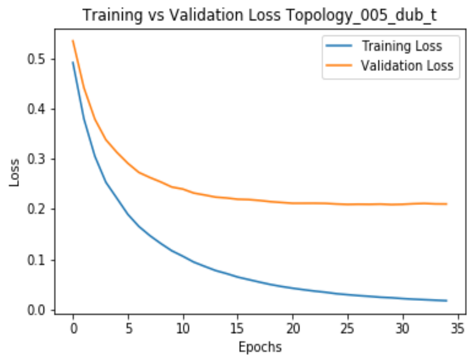

Figure 28: Training and Validation Loss For the MLP Using 256 Hidden Units

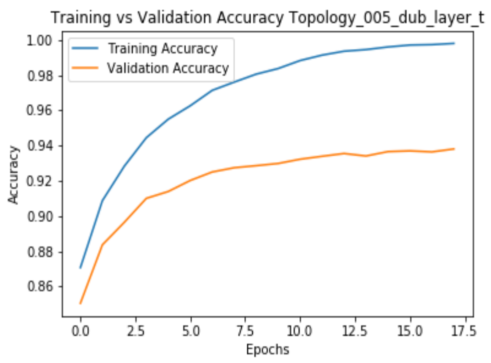

Figure 29: Training and Validation Accuracy For the MLP Using 2 Hidden Layers

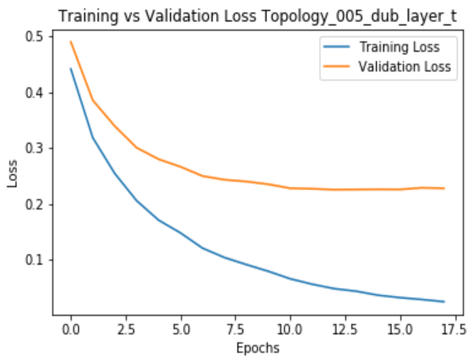

Figure 30: Training and Validation Loss For the MLP Using 2 Hidden Layers

Table 7: Accuracy, and loss with different topologies

TOPOLOGY	**TRAIN ACCURACY**	**TRAIN LOSS**	**VALIDATION ACCURACY**	**VALIDATION LOSS**	**TEST ACCURACY**
64 Units	0.98425	0.085203	0.915	0.296058	0.826
128 Units	0.99725	0.034221	0.932167	0.239340	0.8514
256 Units	0.999542	0.017683	0.941833	**0.210170**	**0.8626**
203x203 Units	0.998083	0.025471	0.938083	0.228035	0.8583

6.1 Conclusion

As shown in Table 7, we had minimum validation loss on the model with 1 hidden layer of 256 hidden units. Per assignment instruction that we should only ”Report final accuracy”, we obtained final accuracy on the test set of **0.8626**. According to the graphs, all of them converged after 20 to 30 epochs. The accuracy increases, and the loss decreases as the number of hidden units in the hidden layer increases. However, adding extra hidden layer, even though improved model’s performance, but started to overfit much earlier much faster than other topologies, which is reflected in the graph as it stopped at less than 20 epochs. Thus we can reach the conclusion that increasing number of hidden units in a hidden layer and increasing number of hidden layers help improve model’s performance, while increasing number of hidden layers may tend to have overfitting issue.