Multi Layer Perceptron

EE6132 - Lab 1

Anirudh R

October 7, 2021

Contents

1 Introduction

In this assignment, we look to build the multi-layer perceptron(MLP) from scratch. We write down the `NeuralNetwork()` class where we implement forward and backward propagation and other associated functions to train the network. We then test the MLP built using the MNIST dataset. We also look to try out variations like data augmentation, regularization, various activation functions, various learning rates and feature extractors. We also look to compare the performance of MLP against SVMs and KNN classifier.

The following table shows the main configuration of MLP we will look at. All the other variations will be based on this.

Parameter	**Value**
Dataset	MNIST
Loss Function	Cross Entropy Loss
Optimizer	Mini-Batch Stochastic Gradient Descent
Mini-Batch Size	64
Number of Epochs	15
Number of Hidden Layers	3
Hidden Layer Sizes	500, 250, 100
Hidden Layer Activation	Sigmoid
Output Layer Activation	Softmax
Initialization(biases)	Zeros
Initialization(weights)	Glorot

2 Concepts

2.1 Dataset

We make use of the **MNIST**(Modified National Institute of Standards and Technology) Dataset here. This is a dataset of hand-written digits and the corresponding labels. This contains **60,000** training data points and **10,000** test data points. The digits have been size-normalized and centered in a fixed-size image (28×28 grayscale image).

2.2 Multi Layer Perceptron(MLP)

Multi Layer Perceptron(MLP) is nothing but a neural network made out of *fully connected* layers. We have an input layer of size 784 since the MNIST inputs are 28x28 images which when flattened give 784 features. The output layer will contain 10 neurons corresponding to the 10 classes(each of the single digit numbers).

2.2.1 Activation

The input layer has a **linear** activation and the output activation is usually **softmax** for classification tasks as it provides probabilities for each class. The hidden layer activation can be chosen to be sigmoid, relu, tanh etc.

$$relu(x) = max(x, 0) \tag{1}$$

$$tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2}$$

$$sigmoid(x) = \frac{1}{1 + e^{-x}} \tag{3}$$

2.2.2 Loss Fuction

The loss function used for classification tasks along with the softmax output is the **Cross Entropy Loss**. The following formula defines how this loss is calculated for a single data point.

$$\mathcal{L} = -\sum_{i=1}^{N} y_i log(p_i) \tag{4}$$

where N is the number of classes(here, 10), y is the one-hot vector corresponding to the label of a particular data point and p is the probabilities vector output by the MLP for the same data point. p_i corresponds to the probability of given input being in class i predicted by the MLP.

Therefore, this definition gives **large** loss values when the probability corresponding to the ground truth label is **low** and **small** loss values when the probability corresponding to the ground truth label is **high**.

2.2.3 Structure

Now, let us look at the construction of the MLP. Let us reference the layers by variable l starting from 1 for the input layer and going up to $(L + 2)$ for the output layer where L is the number of hidden layers in the MLP. Let the weights and biases connecting the lth layer and the $(l + 1)$th layer be denoted by W^l and b^l.

Let the pre-activation and activation of the lth layer be denoted by Z^l and A^l respectively. **Pre-activation** is the value obtained by applying the weights and biases on the output of the previous layer. **Activation** is obtained once the pre-activation is passed through the non-linear activation function.

Let the input vector and the output vector of the MLP be denoted by x and p respectively. As we are dealing with *fully connected* layers, every neuron in lth layer will be feeding its output/activation to every neuron in the $(l + 1)$th layer.

2.2.4 Forward Propagation

In forward propagation, we calculate the output probability vector p for a given input x by moving *forward* through the network. Since, we use the outputs of the lth layer in the $(l + 1)$th layer and so on and keep moving forward through the network, the process of finding p for an input x is called **Forward Propagation**.

This can be written down as the following equations for each layer.

$$Z^{l+1} = W^l * A^l + b^l \tag{5}$$

$$A^{l+1} = f(Z^{l+1}) \tag{6}$$

where $f(.)$ denotes the activation function of the $(l + 1)$th layer. For this assignment, we assume a common activation for each of the hidden layers. But, this is not necessary.

2.2.5 Backward Propagation

Backward propagation is the process of calculating the derivative of the loss function with respect to each of the W^ls and b^ls. This is pivotal to the framework of training as we need a way to move towards lower values of loss. Back propagation allows us to iteratively reach lower losses by performing gradient descent with the obatained derivatives. This process is called back propagation because the **local gradients** propagate backwards. The local gradient of $(l+1)$th layer is used in calculation of local gradient of lth layer. The following equation describes how local gradients propagate backwards.

$$\text{Local Gradient: } \frac{\partial \mathcal{L}}{\partial Z^l} = \delta^l = ((W^l)^T \delta^{l+1}) \odot f'(Z^l) \tag{7}$$

where $\odot$ is element-wise multiplication and f is the activation of the lth layer. With cross entropy loss and softmax activation, the local gradient of the output layer will be,

$$\delta = p - y \tag{8}$$

The final equations for derivative of loss with respect to weights and biases will involve local gradients in the following manner.

$$\frac{\partial \mathcal{L}}{\partial W^l} = \delta^{l+1}(A^l)^T \tag{9}$$

$$\frac{\partial \mathcal{L}}{\partial b^l} = \delta^{l+1} \tag{10}$$

2.2.6 Initialization

Here, we initialize the biases with zeros and we use Glorot initialization for the weights. Glorot intialization involves initializing each $w^l_{i,j}$ the following way.

$$w^l_{i,j} \sim \mathcal{U}\left[-d_l, d_l\right] \text{ where } d_l = \sqrt{\frac{6}{fanin + fanout}} \tag{11}$$

where fan-in is the number of input units in the weight tensor and fan-out is the number of output units in the weight tensor.

2.2.7 Optimization

Now that we have the framework ready, we need to perform weight and bias updates in order to approach lower losses and optimize our network. The optimization can be done using a variety of algorithms such as Batch Gradient Descent, Stochastic Gradient Descent(SGD), Mini-Batch SGD, Momentum based GD, Adagrad, Adam, RMSProp etc. Here, we will choose the **Mini-Batch SGD** for the optimization task.

In Mini-Batch SGD,

- We divide the dataset into **Batches**.
- We pick a batch and perform forward, backward propagation to calculate gradients(derivative of loss with respect to weights and biases) with respect to given batch.
- We then perform the weight and bias updates and move to the next batch.

This is sort of the best of both worlds between Batch GD and SGD. Batch GD is too slow as it requires to go through the entire dataset to give gradients. SGD is much faster but leads to very noisy gradients as we use only use 1 data point to calculate each gradient. Mini-Batch SGD is decently fast but the gradient calculated is also not too noisy as we consider a full batch each time we calculate gradients instead of a single data point.

NOTE: Since Mini-Batch SGD calculates gradients with respect to the batch and not the entire data set each iteration, the gradient produced may not be the actual steepest descent direction of the loss of the entire data set. So, the loss curve may not decrease monotonously.

The updates of weights and biases,

$$W^l_{new} = W^l_{old} - \eta \frac{\partial \mathcal{L}}{\partial W^l} \tag{12}$$

$$b^l_{new} = b^l_{old} - \eta \frac{\partial \mathcal{L}}{\partial b^l} \tag{13}$$

where η is the learning rate.

2.3 Regularization

Regularization is a useful tool when we want to deal with over-fitting models. This may happen when we have minimal data. Over-fitting will lead to the MLP fitting the training data too perfectly and not generalizing well to the test dataset. To tackle this we may do something like L2 or L1 regularization. Here, L2 regularization has been implemented. Note that regularization does **not** affect bias updates.

The L2 regularization basically adds an extra term to the loss function to get a modified loss function($\widetilde{\mathcal{L}}$) where β is the quadratic regularization parameter(hyper parameter).

$$\widetilde{\mathcal{L}}(W, b) = \mathcal{L}(W, b) + \frac{\beta}{2}\|W\|^2 \tag{14}$$

This loss modification in turn leads to the following modification in the weight update equation.

$$W^l_{new} = W^l_{old} - \eta \frac{\partial \widetilde{\mathcal{L}}}{\partial W^l} = (1 - \beta\eta) W^l_{old} - \eta \frac{\partial \mathcal{L}}{\partial W^l} \tag{15}$$

The typical values chosen for β are between 0 and 0.1.

3 Experiments & Observations

3.1 Sigmoid Activation for Hidden Layers

With sigmoid activation for the hidden layers and the default specifications mentioned in section 1, MLPs were trained with a variety of learning rates. The following results were obtained.

Table 1: Sigmoid Activation - Hidden Layers

Learning rate	Acc(Train)[in %]	Acc(Test)[in %]
1	99.457	97.98
1e-1	94.265	93.87
1e-2	80.083	80.48
1e-3	11.237	11.35

- Considering that the accuracies for train and test are close, we can say that this network **does not overfit** the MNIST dataset much. This is expected considering the huge number of data points that we have. But, we could still say that the network very slightly overfits the data considering that the test accuracy is lower than the training(even though marginally).
- Therefore, overall, this network does not overfit and generalises well to test data set. We also see that the best training happened with LR = 1.
- The very low accuracy of LR = 1e-3 is due to the fact that the small learning rate is not letting the loss get to the low values that we need, in the 15 epochs that we used. It is very slow in reducing loss.

We now plot the training and test loss plots vs iterations to observe convergence for varying learning rates.

3.1.1 Test Plots:

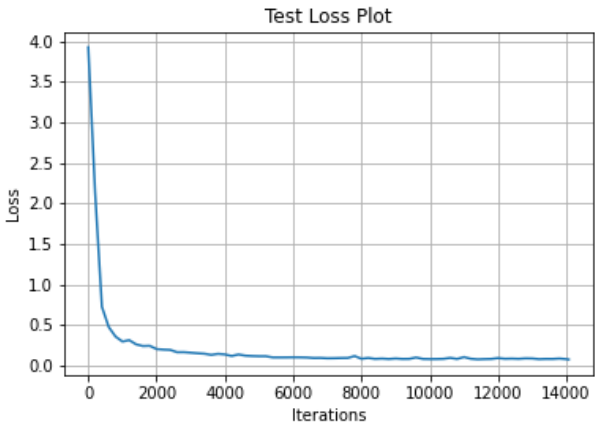

Figure 1: Test Loss - Learning rate = 1

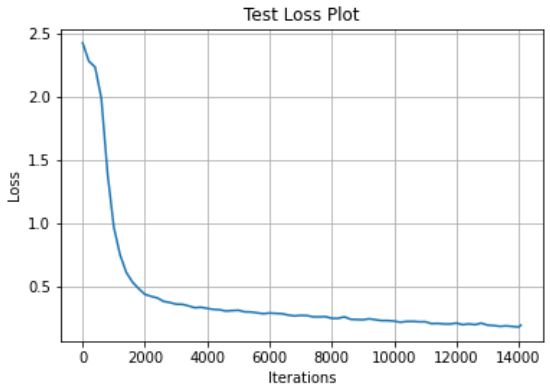

Figure 2: Test Loss - Learning rate = 1e-1

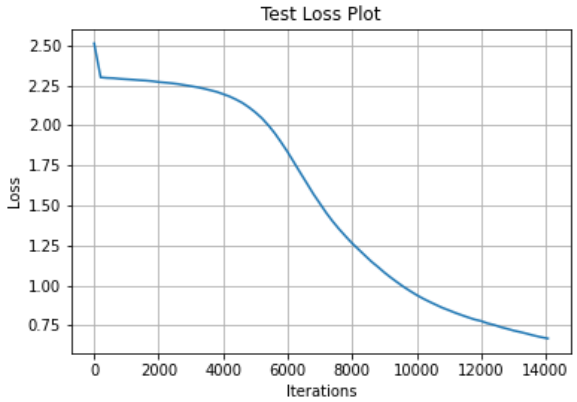

Figure 3: Test Loss - Learning rate = 1e-2

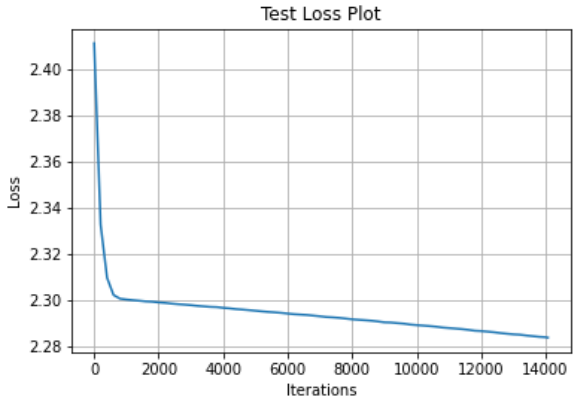

Figure 4: Test Loss - Learning rate = 1e-3

- We see that all the test loss plots decrease monotonously. Only LR = 1 seems to have converged/saturated.
- We see that in 15 epochs, Learning Rate(LR) = 1 has converged to a loss of 0.0731.
- LR = 1e-1 has also sort of saturated but there is still a slight downward slope in the plot. It achieves a loss of 0.19846. Perhaps, the higher loss at the end of training compared to LR = 1 is because it hasn't fully saturated.

- LR = 1e-2, 1e-3 have not saturated yet and reach a test loss of 0.66877 and 2.28401 respectively.
- Overall, the general trend is that higher the LR, more drastically the loss has dropped in fewer iterations. This has led to higher LRs reaching convergence while others not. Also, the small rate of decrease in loss of the smaller LRs has led to higher loss at the end of training and non-convergence as we are training all LRs for the same number of epochs.

3.1.2 Training Plots:

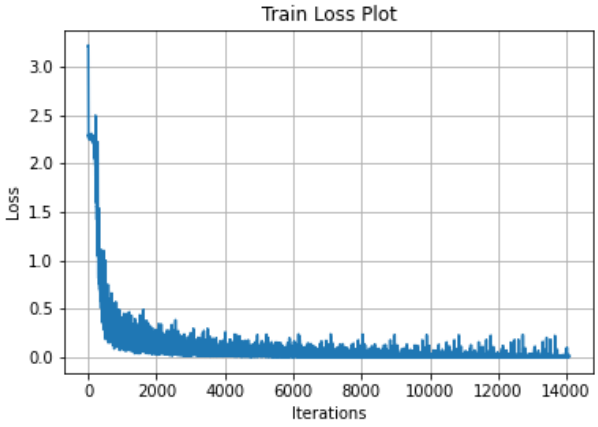

Figure 5: Training Loss - Learning rate = 1

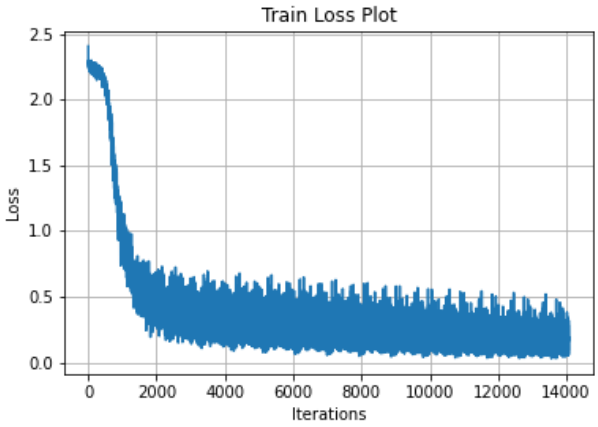

Figure 6: Training Loss - Learning rate = 1e-1

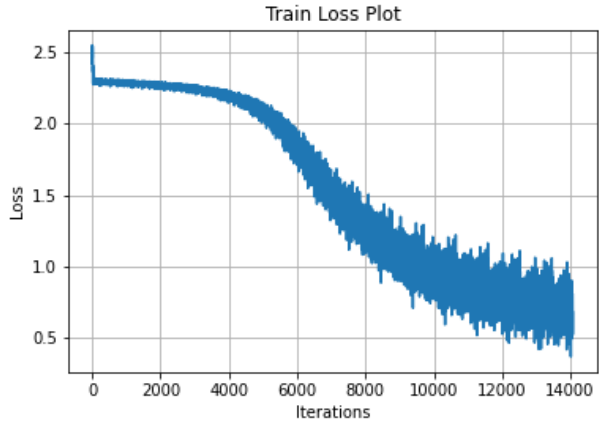

Figure 7: Training Loss - Learning rate = 1e-2

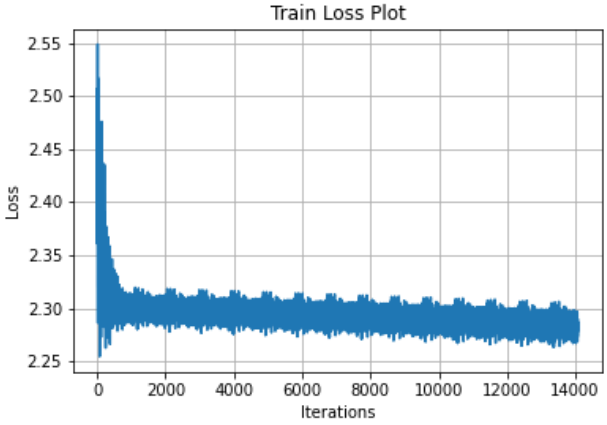

Figure 8: Training Loss - Learning rate = 1e-3

- Similar to the training plots, the general trend in the plots is that the loss is decreasing. But, as we are using Mini-Batch SGD, the gradient calculated at each iteration may not be the direction of steepest descent of the overall loss function. This has led to the noisy training loss plot.
- The fact that we calculate the loss at each iteration purely based on the corresponding batch also contributes to the noisy plot.
- Other general trend observations made in the test plots will also apply here if we take into account the noisy nature of the plot.

3.1.3 Confusion Matrix and Metrics

Confusion matrix and metrics are presented for the LR = 1e-1 case. For the rest of the assignment, LR = 1e-1 will be considered the baseline model.

It is interesting to note that the most prominent off-diagonal values correspond to the digits that look very similar. For example, the way 4 and 9 are written can be very similar and correspondingly this leads to one of the highest off-diagonal elements. This sort of shows that the way we visually process information is very similar to the way the

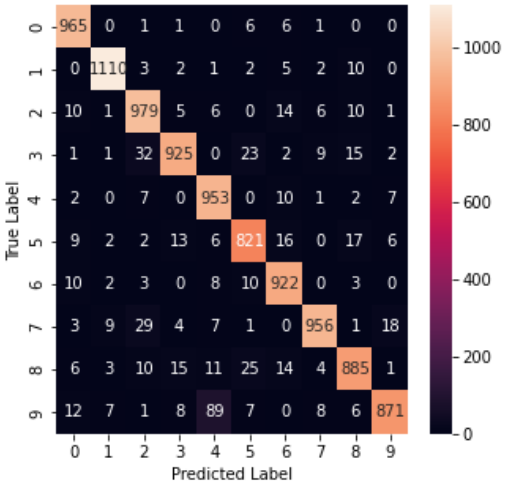

Figure 9: Confusion Matrix - Learning rate = 1e-1

network sees it which leads to similar issues in understanding of the images. What we might find confusing also seems to confuse the network...

Metric	Value
Accuracy(%)	93.87
Misclassification Rate(%)	6.130
Precision	0.93971
Recall	0.9387
F1 score	0.93855

3.2 Other Activations

We now consider the MLP with the exact same configuration but with *tanh* or *relu* for the hidden layer activation. We will consider LR = 1e-1. Sigmoid is also put in the table again for easy reference.

Table 2: Other Activations - Hidden Layers

Activation Function	Acc(Train)[in %]	Acc(Test)[in %]
sigmoid	94.265	93.87
tanh	99.830	98.01
relu	99.992	98.25

It is obvious from these values that the `relu` activation performs much better than `sigmoid` and somewhat better than `tanh`. Both `tanh` and `relu` perform much better than `sigmoid` activation.

Metrics for `tanh` activation:

Metric	Value
Accuracy(%)	98.01
Misclassification Rate(%)	1.99
Precision	0.980117
Recall	0.980100
F1 score	0.980099

Metrics for `relu` activation:

Metric	Value
Accuracy(%)	98.25
Misclassification Rate(%)	1.75
Precision	0.982520
Recall	0.982500
F1 score	0.982501

- We see that the precision, recall and F1 score achieved are very close to each other. They also pretty much match with the accuracy obtained.
- The precision and recall seem very well balanced in both cases. So, we can say that we have a very balanced classifier with respect to the precision-recall trade-off.
- In case of sigmoid metrics presented in the previous page, we see that the difference between precision and recall is much more significant compared to relu or tanh. The exact same MLP with sigmoid activation seems to be giving higher precision than recall by approximately 0.001. So, sigmoid activation seems to make the classifier side more towards precision while tanh and relu maintain the classifier to be very balanced.

3.2.1 Training plots

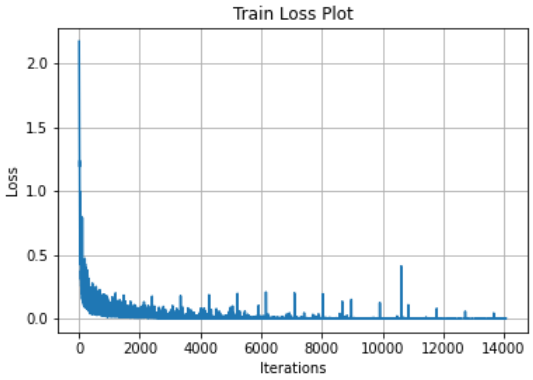

Figure 10: Training Loss - Relu

- We see that tanh and relu converge to 0.00975 and 0.00115 respectively. Both converge to values very close to 0. Sigmoid on the other hand, *sort of* converges to 0.19067 in a very noisy manner. The general trend of the train loss seems to still be decreasing for sigmoid unlike tanh or relu where it has saturated.
- At the end of training, not only is the training loss achieved the lowest for relu, but it also has the least noisy convergence. Sigmoid has the most noisy training plot.
- Relu seems to be the fastest to get a non-noisy stage and also achieves the best loss at the end of training. Therefore, among these 3 activations, from the training plots, we can say that relu performs better than tanh, which both perform much better than sigmoid.

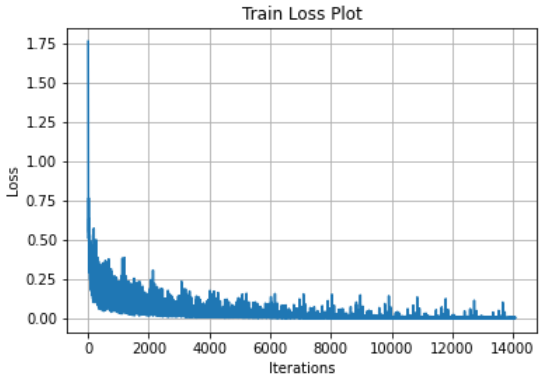

Figure 11: Training Loss - Tanh

3.2.2 Test plots

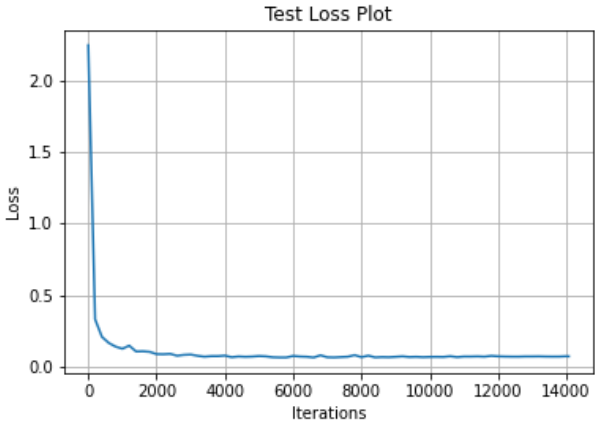

Figure 12: Test Loss - Relu

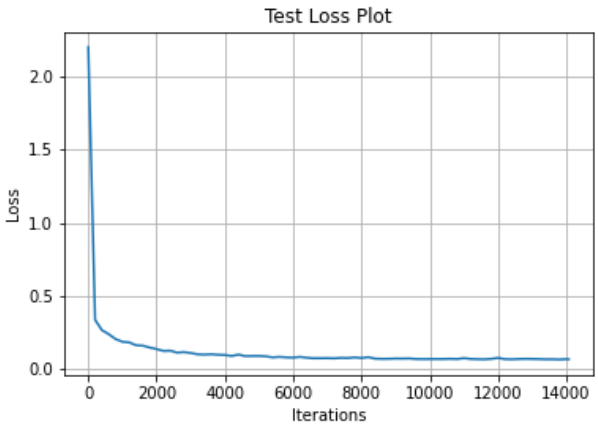

Figure 13: Test Loss - Tanh

- The test loss attained at the end of training are 0.19846, 0.06236 and 0.07572 respectively for sigmoid, tanh and relu. In test, we see that tanh achieves lower loss

at the end of training compared to relu. Sigmoid is much worse compared to relu and tanh. With regards to test loss achieved, tanh performs better than relu which both perform than sigmoid.

- We can also better comment on the rate of convergence here due to non-noisy plots. At 2000 iterations, we see that the test loss achieved by sigmoid is about 0.5. Whereas the test loss achieved by tanh and relu are approximately 0.2 and 0.1. These are approximate numbers gathered from the plot. But, this still shows as to how **fast** relu is in reducing loss. It reduces loss at a very high rate with respect to number of iterations. Tanh is close but a bit slower. Sigmoid is much slower in reducing loss compared to tanh and relu.

- So, overall, tanh seems to give the lower test loss but relu reduces loss at a faster rate. Sigmoid is bad in both aspects.

3.2.3 Percentage of Inactive Neurons

Hidden Layer 1:

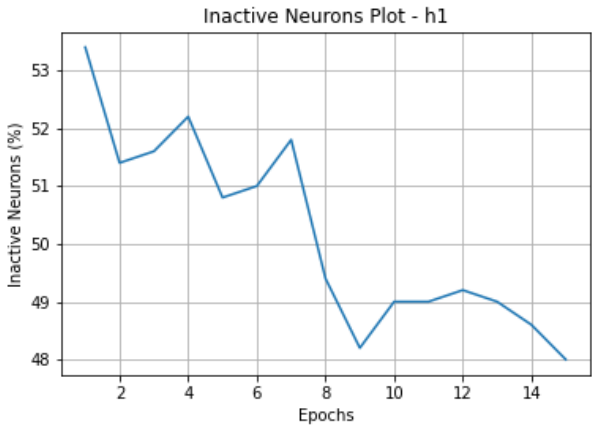

Figure 14: Inactive Neurons - Sigmoid - h1

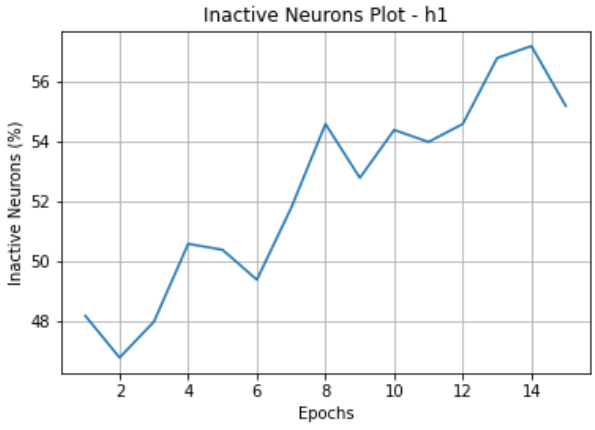

Figure 15: Inactive Neurons - Tanh - h1

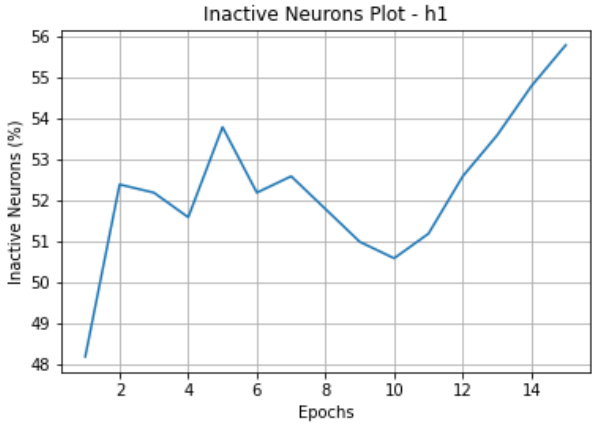

Figure 16: Inactive Neurons - Relu - h1

- We see that for sigmoid, the inactivity reduces from about 53% to about 48%. On the other hand, the inactivity seems to increase overall in relu and tanh and they both reach a peak of about 56%.
- So, tanh and relu lead to increasing inactivity with progression of epochs but sigmoid reduces inactivity.
- Also, the inactivity plot seems rather monotonous in tanh and sigmoid but relu seems to have a rise, a dip and then a rise again.
- On an average, sigmoid has about 50% inactivity while the other 2 have about 52% inactivity. So, the amount of inactivity seems to be a bit lower in h1 for sigmoid.

Hidden Layer 2:

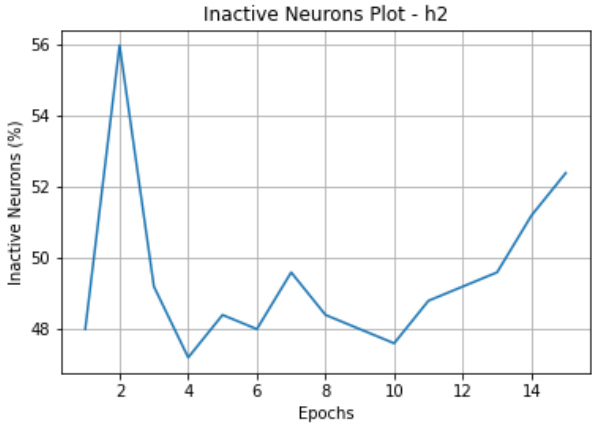

Figure 17: Inactive Neurons - Sigmoid - h2

- Here, sigmoid peaks very abruptly at the start. After the peak, it again seems to slowly climb. Tanh sort of has an abrupt dip near 10 epochs but seems somewhat constant otherwise. Relu sort of has an increasing inactivity.

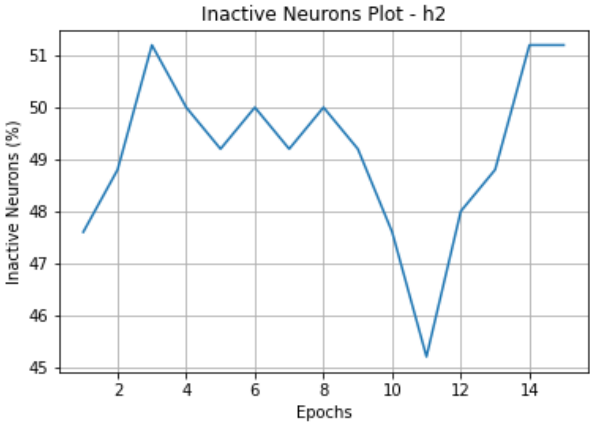

Figure 18: Inactive Neurons - Tanh - h2

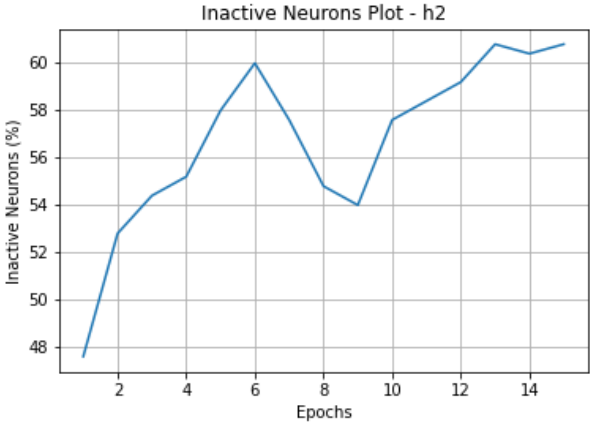

Figure 19: Inactive Neurons - Relu - h2

- From the plots, sigmoid and tanh seems to have an average of about 50%. Relu has an average of about 56% which is much higher compared to tanh and sigmoid. So, it seems like relu has a higher inactivity in h2 compared to sigmoid and tanh.

Hidden Layer 3:

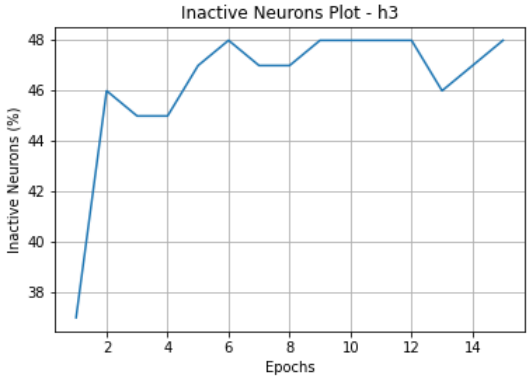

Figure 20: Inactive Neurons - Sigmoid - h3

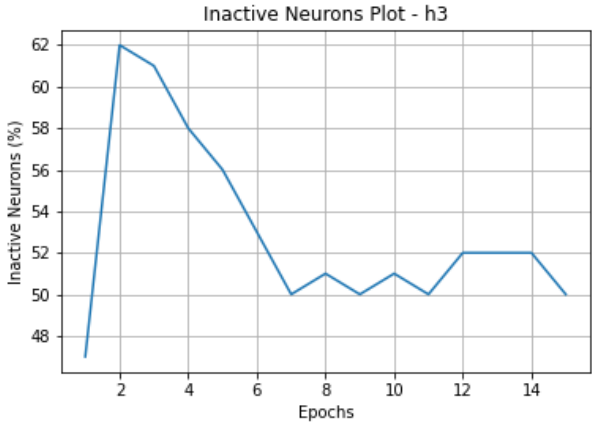

Figure 21: Inactive Neurons - Tanh - h3

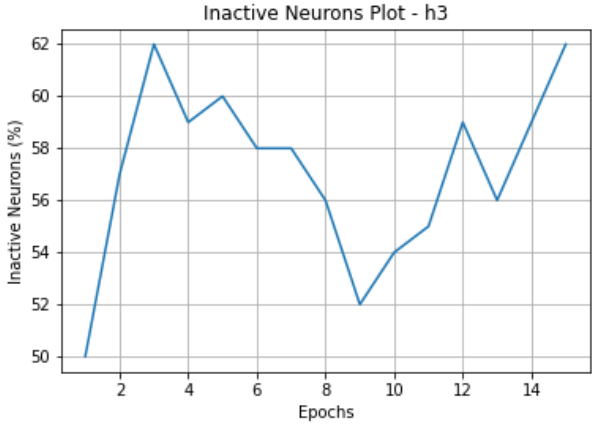

Figure 22: Inactive Neurons - Relu - h3

- Sigmoid inactivity increases and saturates at about 48%. Tanh has a very high peak inactivity of about 62% but then drops and stabilizes at about 51%. Relu too peaks at about 62% but has a rise, a dip and a rise again in h3 as well just like in h1.

- On an average, relu is higher than tanh which is higher than sigmoid in terms of inactivity.

So, overall, on an average, relu has the highest inactivity and sigmoid has the lowest. This is perhaps due to relu saturating/converging very early leading to higher inactivity. On the other hand, sigmoid didn't fully reach saturation even at the end which would have led to lower inactivity. Also, the approximate general trend for the inactivities is increasing barring exceptions. This is understandable considering that the network's loss saturates and converges as we go through more epochs. As we close in on convergence(with more epochs), the loss function gets more "**flatter**"(can be visualised in 2D) which leads to smaller gradients and higher inactivity.

3.3 Regularization

We try 2 methods of regularization here:

- We implement the L2 regularization.
- We add Gaussian noise to the input data.

3.3.1 L2 Regularization

The following table shows how the baseline model compares with the L2 regularized model. Here, we choose Quadratic Regularization parameter, β = 1e-3.

Table 3: L2 Regularization

Model	Acc(Train)[in %]	Acc(Test)[in %]
Baseline($\beta = 0$)	94.265	93.87
Regularized($\beta = 10^{-3}$)	90.04	89.65

- We see that with regularization, the performance of the model decreases. This is due to the fact that the baseline model itself fits very well to the MNIST data set and **doesn't overfit**. So, adding an extra L2 regularization term to the loss only makes the model worse.
- So, it is important to use regularization only when our model overfits the data set. Otherwise, it can prove to be detrimental as we are deliberately **moving away** from the **original goal($\mathcal{L}$) unnecessarily**. In case of over-fitted models, the new loss helps tackle the overfitting.

3.3.2 Data Augmentation

We add Gaussian noise to the training input data of mean 0 and some standard deviations. The results obtained are presented in this table.

Table 4: Data Augmentation

Model	Acc(Train)[in %]	Acc(Test)[in %]
Baseline	94.265	93.87
Noisy($\mu = 0$, $\sigma = 0.5$)	87.934	88.32
Noisy($\mu = 0$, $\sigma = 0.1$)	92.342	92.60

- We see that the higher noise led to much worse performance. This is because we added too much noise and it spoiled the data.
- Even in the lower noise case, it performs better but is still worse compared to the baseline model. This is because the baseline model fits the data very well and generalizes very well too(no overfitting). So, adding noise to the non-overfitted model is only spoiling the performance.
- The noise acts as a regularizer of the model. But, as our model is already very well

fitted, adding noise is only spoiling the performance.

3.4 Feature Extraction

We do an independent feature extraction from the input 28x28 images and then use the output feature vectors of the feature extractor as the input to our model. Here, we make use of **Histogram of Oriented Gradients(HOG)** method for the feature extraction. We use the following parameters for the HOG feature extractor.

Parameter	**Value**
orientations	9
pixels_per_cell	(5,5)
cells_per_block	(2,2)

These specifications take in 28x28 images and gives a feature vector of length 576. We now use this to train our models. We will look at MLP, SVM, KNN classifiers applied on these extracted feature vectors.

We will look at 2 MLP models here.

- **MLP1** - Activation(relu), hidden layer size([300,150]), number of epochs(8), LR(0.1), rest – same as baseline
- **MLP2** - Activation(relu), hidden layer size([400,200,100]), number of epochs(8), LR(0.1), rest – same as baseline

Table 5: Classifiers on Extracted Features

Model	Acc(Train)[in %]	Acc(Test)[in %]
Baseline	94.265	93.87
MLP1	98.433	97.58
MLP2	99.155	97.73
SVM	99.515	98.59
KNN(K=5)	97.080	96.10

- We see that even though the baseline model had 3 hidden layers(sufficient complexity) with more neurons than MLP2, its sigmoid activation and usage of raw 28x28 image has led to significantly lower performance compared to even the KNN classifier.
- These values make it clear how valuable features can be to classification. HOG basically spoon feeds useful features to the classifiers unlike baseline where the network had to use the raw images directly.
- Also, note that MLP2 is actually a leaner version of the baseline model with relu, lesser neurons and lesser epochs. But, with the help of extracted features and relu activation, it is able to get an increase in accuracy of about 4-5% above the baseline.
- According to the given observations, SVM seems to be the clear winner, followed by the MLP2, MLP1 and then the KNN(with much lower accuracy). But, the

rather small gap between SVM and MLP could possibly be bridged by appropriately choosing LR or training for more epochs.

- Among the MLPs, the MLP with more layers and more neurons performed better which is expected.
- From this, we see how feature extraction can be a very important part of image classification tasks. Without feature extraction, it is like we are just directly feeding the images into a MLP classifier. This perhaps can be enough proof to prove the importance of an architecture like CNN where we first extract features using convolutions and then classify the extracted features using MLP.

4 Code

The implementation of all the experiments used in this report can be found in the following colab notebook: Link

5 Conclusion

We have thus built MLP from scratch and analysed it under various scenarios. In the end, we also got further proof as to how important an architecture like CNN is.