# EE6132: Assignment 1
## Multi Layer Perceptron
Date: October 7, 2021

---

## 1 Brief Description of the code

### 1.1 Libraries required:

The libraries required to run the python code are described below:

- wget
- mnist
- os
- PIL
- numpy
- random
- statistics
- tqdm
- matplotlib.pyplot
- sklearn
- cv2

### 1.2 The different modules

The python script can be roughly divided into 10 different modules, namely:

- **Data Processing Module:** This module downloads the training and the testing data from the mnist data base and performs the required manipulations so as to make it compatible with the network designed(i.e., such as converting the training and testing labels into one-hot vectors). A cool trick that has been done here is scaling the training and testing data by 255 which restricts the input values to 1 thereby preventing any sort of explosing while performing the computations.
- **Activation Function Library:** This module hosts all the functions describing the various activation functions and a *library* which helps us access the requisite activation function at will.
  The available activation functions are:
  - **Sigmoid:** $\sigma(x) = \frac{1}{1+e^{-x}}$
  - **ReLU:** ReLU(x) = $\max\{0, x\}$
  - **Tanh:** tanh(x) = $\frac{e^{x}-e^{-x}}{e^{x}+e^{-x}}$
  - **Linear:** lin(x) = x
  - **Softmax:** softmax($x_i$) = $\frac{e^{x_i}}{\sum_j e^{x_j}}$
- **Activation Derivative Library:** This module hosts all the derivatives of the various activation functions described in the activation function library above.
- **Learning Rate Library:** This module hosts two different choice of learning rates, one being a constant learning rate and the other being an exponentially decaying learning rate for epochs after a certain threshold.

- **Gradient Descent Library:** This module hosts a wide array of gradient descent choices which were taught in class. The functions calculate one iteration of the gradient descent for a vector which encompasses the whole minibatch.This vectorization across the mini batch vastly speeds up the training and testing process.
  - Vanilla Gradient Descent
  - Momentum based gradient descent
  - Adagrad
  - RMS prop
- **Loss Function Library:** This module hosts the various loss functions we are incorporating to train our MLP. They are namely,
  - Cross Entropy Loss function: $\sum_i -p_i log(q_i)$ where q are our predictions and p are the true labels.
  - $L_1$ loss function: An additional $L_1$ regularization loss, $\frac{\lambda}{N}\sum_i |w_i|$ (N is the minibatch size) is added to the cross entropy loss function to encourage simpler weights.
  - $L_2$ loss function: An additional $L_2$ regularization loss, $\frac{\lambda}{2N}\sum_i ||w_i||^2$ is added to the cross entropy loss to further penalize larger weights.
- **Scoring Library:** This library hosts functions that helps us compute performance statistics such as the precision,accuracy,etc, given the predictions.
- **The MLP:** The multilayer perceptron is implemented as a class with a wide range of aiding functions such as those for backprop and forward propagation. We feed our choice of parameters along with the training and testing data and we obtain plots and accuracy metrics once the training and testing have ended. It takes roughly 5 minutes to train and test the network across 15 epochs for the given MNIST dataset.
- **Feature Extraction:** The last module performs two feature extraction schemes, the first being PCA based and the second being HOG based.

# 2 Experimenting with learning rate

## 2.1 Across Learning Rates:

**Note:** Experimentation on the learning rates were done on a network with 3 hidden layers with 500, 250 and 100 neurons respectively with the sigmoid activation function. The minibatch size was kept as 64 and the MLP was trained for 15 epochs.The weights were initialized by Glorot's initialization.

Firstly, the convergence of test and training losses and the accuracy metrics were seen for eleven different choices of learning rates, namely $\{0.01, 0.02, 0.025, 0.03, 0.04, 0.05, 0.06, 0.07, 0.08, 0.09, 0.1\}$. The choice of the learning rates were arrived on reading different papers,some preliminary experimentation and experience from other courses.

Shown below are the basic accuracy metrics across the learning rates and their corresponding training and testing loss plots.

**Accuracy,Mean,Standard deviation of error**

| $\eta$ | Accuracy | $\mu_{err}$ | $\sigma_{err}$ |
|---|---|---|---|
| 0.01 | 0.8217 | 0.3661 | 0.2623 |
| 0.02 | 0.8874 | 0.2043 | 0.2563 |
| 0.025 | 0.9007 | 0.1756 | 0.2477 |
| 0.03 | 0.9049 | 0.1660 | 0.2461 |
| 0.04 | 0.9137 | 0.1444 | 0.2384 |
| 0.05 | 0.9195 | 0.1349 | 0.2325 |
| 0.06 | 0.9277 | 0.1227 | 0.2277 |
| 0.07 | 0.9327 | 0.1129 | 0.2206 |
| 0.08 | 0.9373 | 0.1059 | 0.2124 |
| 0.09 | 0.9411 | 0.0969 | 0.2078 |
| 0.10 | 0.9442 | 0.0931 | 0.2052 |

**The Plots:**

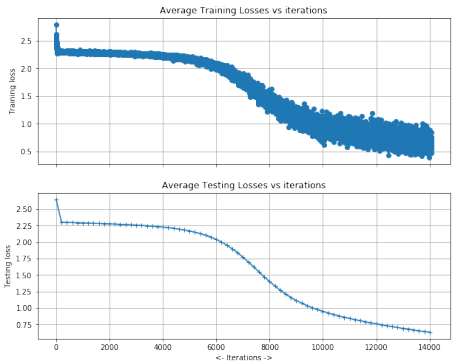



Figure 1: Train and Test loss plot across iterations for $\eta = 0.01$

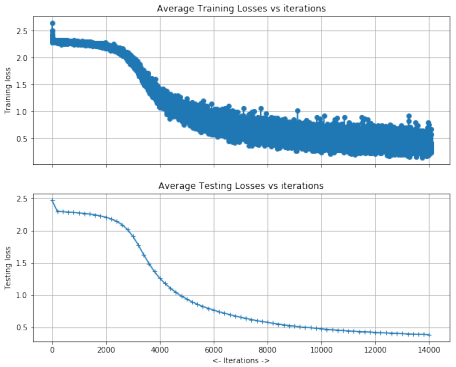



Figure 2: Train and Test loss plot across iterations for $\eta = 0.02$

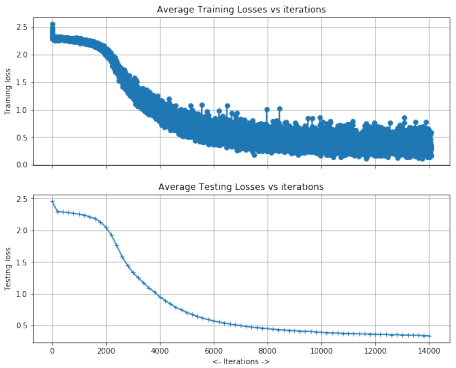



Figure 3: Train and Test loss plot across iterations for $\eta = 0.025$

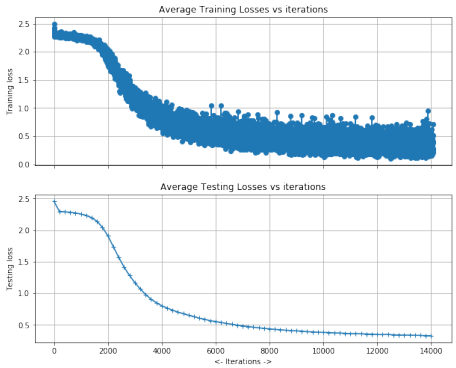



Figure 4: Train and Test loss plot across iterations for $\eta = 0.03$

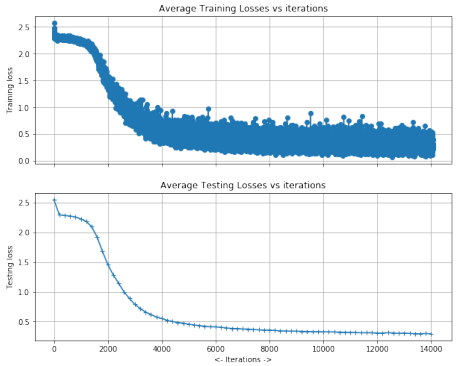



Figure 5: Train and Test loss plot across iterations for $\eta = 0.04$

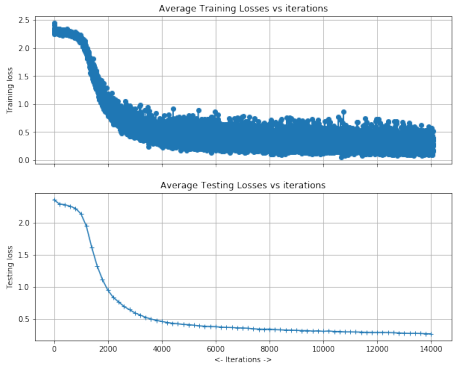



Figure 6: Train and Test loss plot across iterations for $\eta = 0.05$

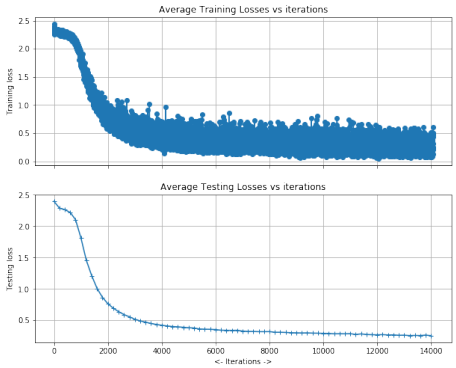



Figure 7: Train and Test loss plot across iterations for $\eta = 0.06$

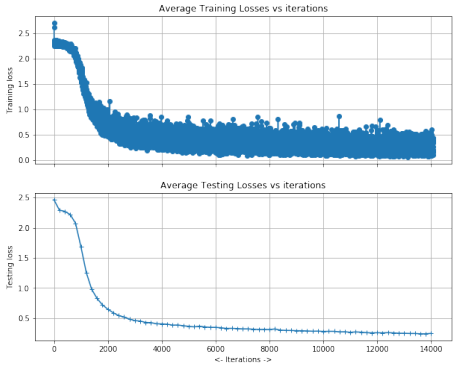



Figure 8: Train and Test loss plot across iterations for $\eta = 0.07$

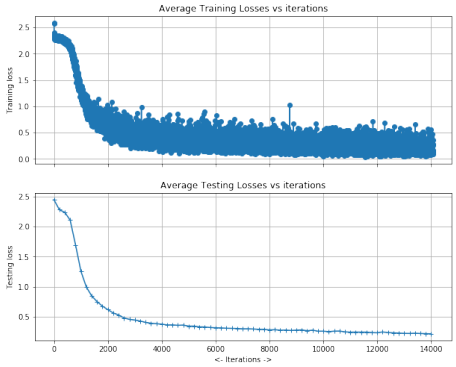



Figure 9: Train and Test loss plot across iterations for $\eta = 0.08$

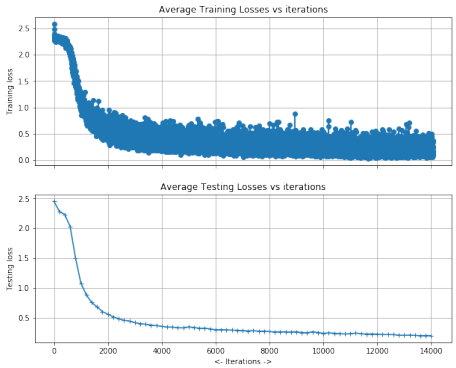



Figure 10: Train and Test loss plot across iterations for $\eta = 0.09$

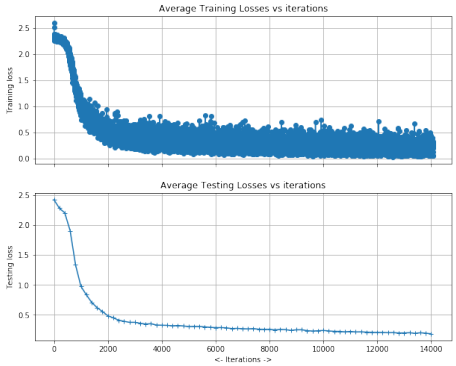



Figure 11: Train and Test loss plot across iterations for $\eta = 0.10$

**Observations:**

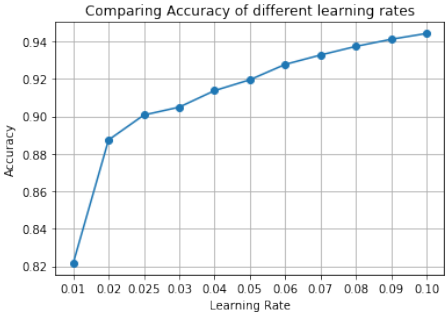



Figure 12: Accuracy vs Learning rate plot

- From the above plot, we see that the learning rate to some extent plays an important role in determining our final accuracy on the test dataset.
- We see that there is a huge jump in accuracy as the learning rate increases from 0.01 to 0.02 but after that the accuracy doesn't improve that much. The mean of errors and standard deviation of errors also follow the same trend with the latter not being as robust as the former.
- On looking at the individual plots for each learning rate, we see that, at first, for larger learning rates, the loss functions drop quickly which is understandable as we move further along the direction of the decreasing gradient of the loss function.
- We also observe that there is a lot of *noisy* oscillation for larger learning rates especially for the higher epochs. This reinforces our theoretical understanding that, larger the learning rate, more likely it is to pass the minima and attempt at coming back to it from the other direction.This results in a lot of oscillation which increases with the increase in the learning rate. This oscillation is represented by the copious amounts of blue region in the graph.

## 2.2 Across different gradient descent algorithms:

Based on the observations seen in the previous part, 0.025 was chosen to be the learning rate for the rest of the experiment, as it gave a very high jump in accuracy and wasn't a large enough learning rate which resulted in a lot of oscillations in the losses.Moreover, this learning rate was also seen to be performing well for the other activation functions.
So, with the learning rate fixed, the optimal Gradient Descent algorithm was required to be found and therefore, the experiment was repeated using all the different Gradient Descent algorithms described in the previous section.
The resulting accuracy metrics and their corresponding training and testing loss plots are shown below.

**Accuracy,Mean,Standard deviation of error**

| Algo | Accuracy | $\mu_{err}$ | $\sigma_{err}$ |
|---|---|---|---|
| $\eta e^{-kt}$ | 0.7928 | 0.4356 | 0.2486 |
| Vanilla GD | 0.9012 | 0.1735 | 0.2476 |
| Momentum GD | 0.9648 | 0.0518 | 0.1642 |
| Adagrad | 0.9387 | 0.1059 | 0.2113 |
| RMS Prop | 0.9794 | 0.0207 | 0.1350 |

**The Plots:**

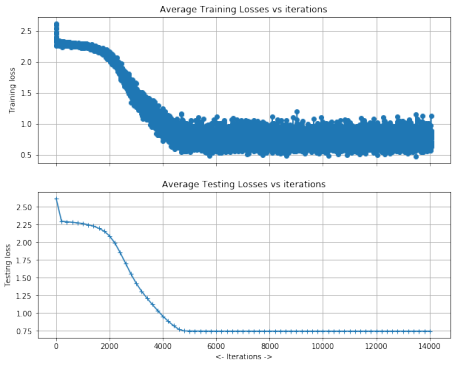



Figure 13: Train and Test loss plot across iterations for exponentially decaying learning rate

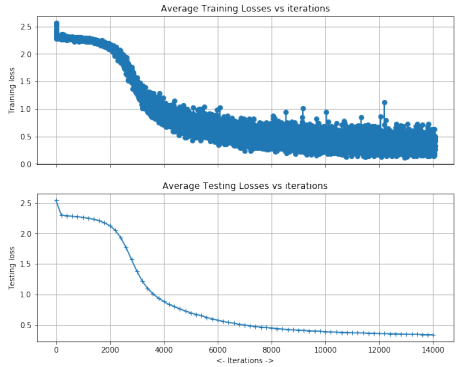



Figure 14: Train and Test loss plot across iterations for Vanilla Gradient Descent

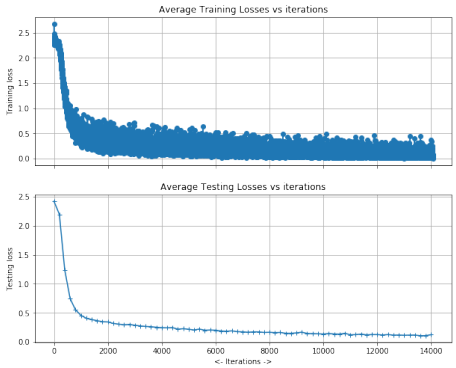



Figure 15: Train and Test loss plot across iterations for Momentum based Gradient Descent

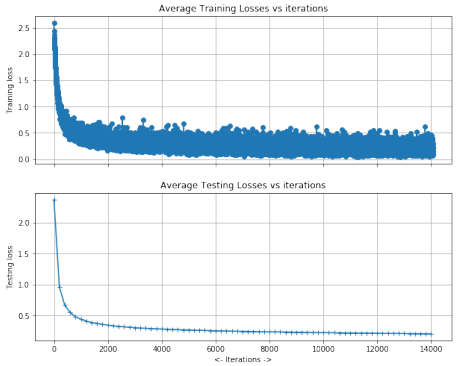



Figure 16: Train and Test loss plot across iterations for the Adagrad algorithm

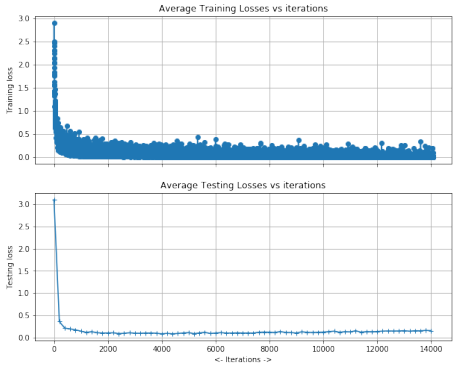



Figure 17: Train and Test loss plot across iterations for the RMS prop algorithm

**Observations**

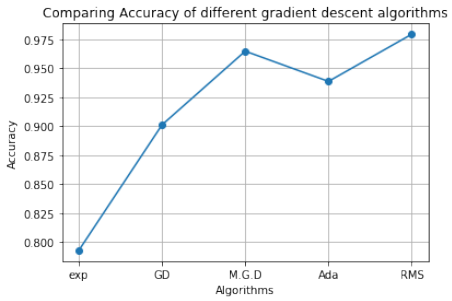



Figure 18: Accuracy vs G.D algorithm plot

- From the above plot, we see that the gradient descent algorithm we choose plays a much more important role than the learning rate when it comes to the accuracy on the test dataset.
- As in the previous case, we see that the error means and the error standard deviations follow similar trends to that of the accuracy.
- We see that the exponentially weighted learning rate performs poorly. This could be because, 15 epochs might not have been enough to train the network. As the learning rate decreases, the gradient descent process also slows down. We however see that once the weighting is applied, the loss function drops in a much smoother manner when compared to the other gradient descent algorithms which is a characteristic of the exponentially weighted learning rate.
- As observed theoretically, the momentum based gradient descent succeeds to a large extent in minimizing the number of oscillations thereby reaching the optima much faster, which is evident by its high accuracy and lower training and test losses.
- We see that Adagrad doesn't really produce any improvement over the momentum based gradient descent. Adagrad accentuates the learning rates of sparse but important features as these could have low gradients.As Adagrad doesn't produce a lot of improvement, we can hypothesize that there does not exist a lot of such sparse and important features. We can lead on from this and make a claim, that pre-processing might not really improve performance by a large extent on this neural network as pre-processing in some sense could potentially highlight such sparse but important features.
- Based on the trends of the good performance of the momentum based gradient descent, it is not surprising that the RMS prop gradient descent performs even better. The RMS prop gradient descent incorporates a momentum based alteration to the learning rate and therefore it outperforms the momentum based gradient descent by converging much faster, reducing the oscillations resulting in a higher accuracy on the test set.

# 3 Results across different activation functions

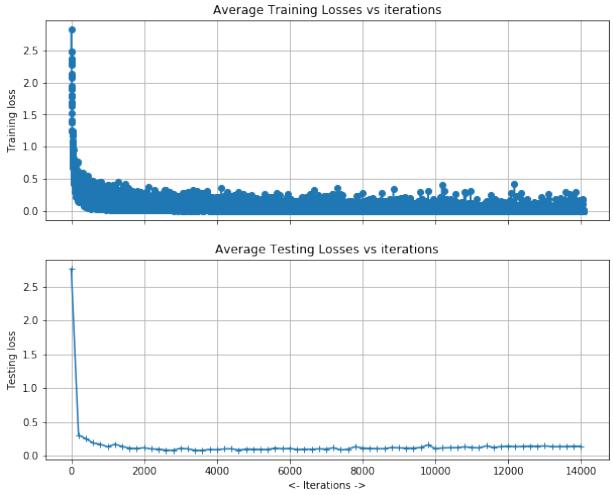
Based on the observations from the previous section, the learning rate has been chosen as 0.025 and the gradient descent algorithm is RMS prop.

## 3.1 Sigmoid activation function

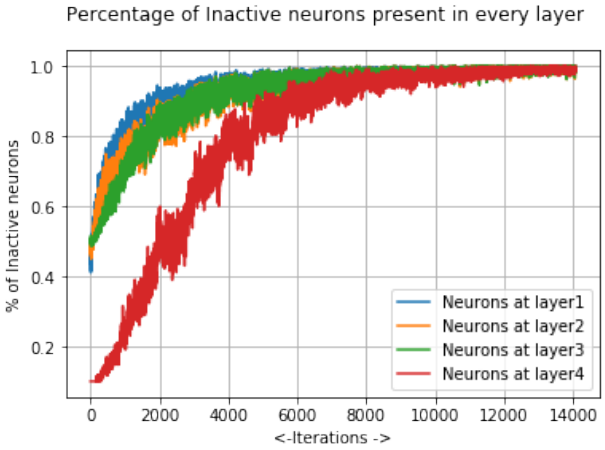
The sigmoid activation function took the longest to train out of the three although not by a lot. The results and plots obtained for the sigmoid activation function are shown below.

**Results:**
Accuracy: 0.9811
Mean of errors: 0.0190
Standard deviation of errors: 0.1290

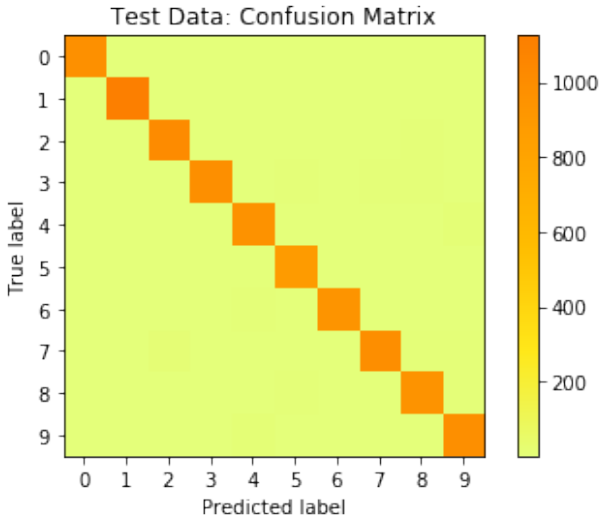
**Plots:**



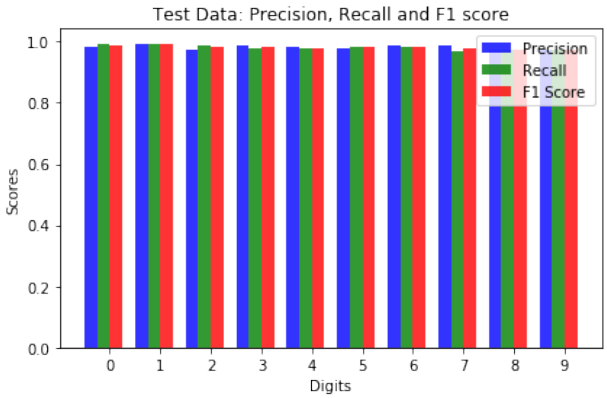

(a)




(b)




(c)




(d)

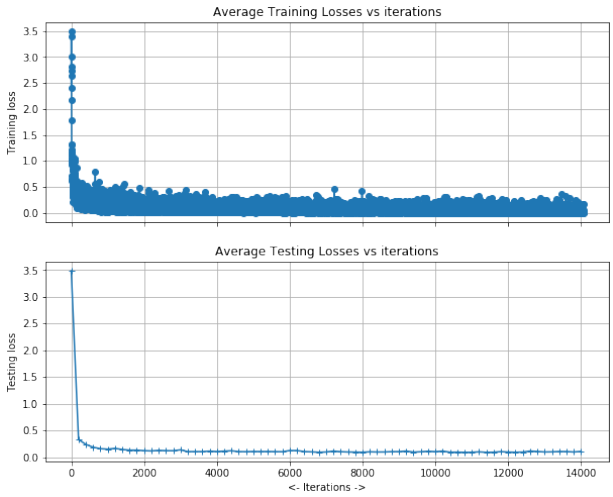
Figure 19: Plots for the sigmoid activation function

## 3.2 Tanh activation function

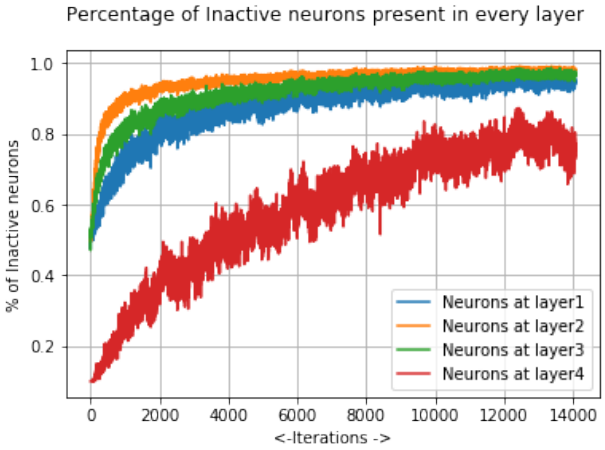
The Tanh activation function took lesser time than the sigmoid to train but more than the ReLU one. The results and plots obtained for the Tanh activation function are shown below.

**Results:**
Accuracy: 0.9769
Mean of errors: 0.0277
Standard deviation of errors: 0.1398

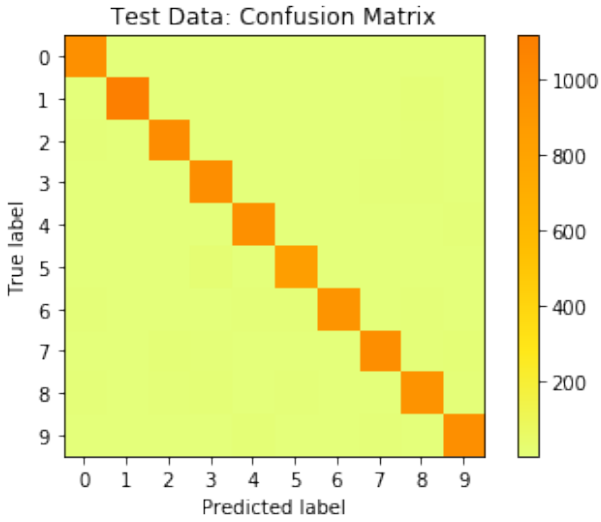
**Plots:**



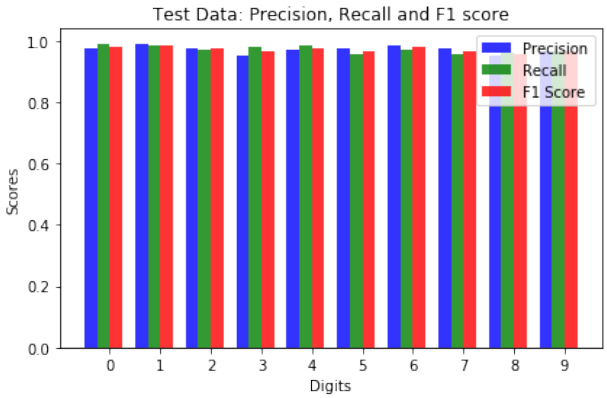

(a)




(b)




(c)




(d)

Figure 20: Plots for the tanh activation function

## 3.3 ReLU activation function

The ReLU activation function took the least time to train out of the three activation functions. Moreover, the ReLU was trained using a Vanilla GD unlike the other activation functions to demonstrate that it performs on par with the other two activation functions without any external enhancements. The results and plots obtained for the ReLU activation function are shown below.

**Results:**
Accuracy: 0.9797
Mean of errors: 0.0298
Standard deviation of errors: 0.1300

**Plots:**

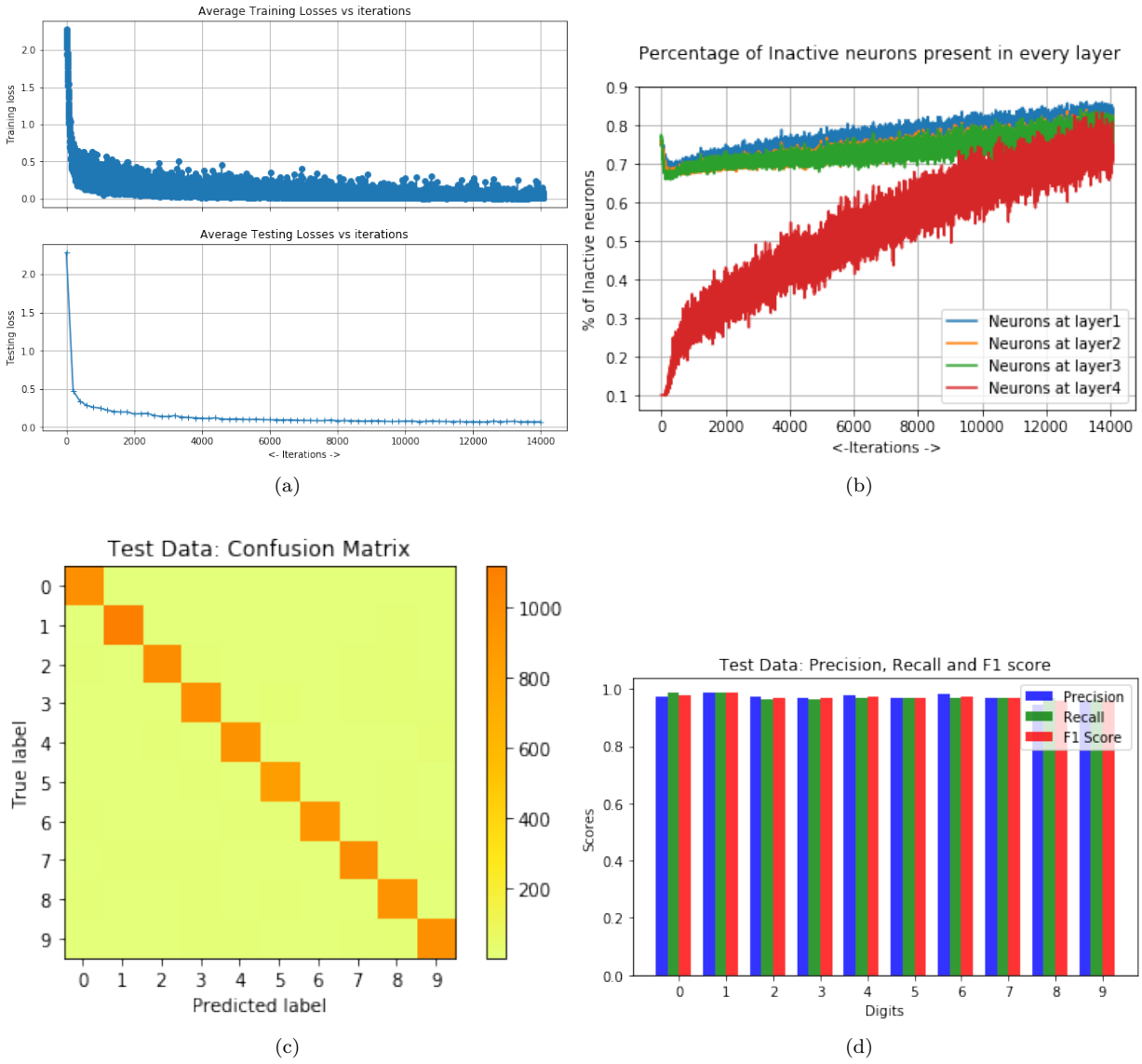



Figure 21: Plots for the ReLU activation function

## 3.4 Observations

- We see that all the three activation functions yield a very similar output in terms of the evaluation metrics. This is a huge testament to the fact that ReLU outperforms the other two activation functions. The ReLU based MLP didn't have the additional aid of the momentum based gradient descent and yet it yielded comparable results to that of the other two activation functions.
- We see that the sigmoid converges to it's optima a bit slower than the tanh activation. This could because of the fact that the sigmoid activation isn't zero centred thereby making gradients across a layer possess the same sign at times.This considerably slows down learning,but we see that the momentum gradient descent compensates for this slowing down thereby resulting in a much smaller difference in the converging speeds.
- Contrary to what we learnt in theory, the sigmoid activation performs marginally better than the tanh one. However, on repeating the experiment a few times, I observed that this isn't a regular occurrence.
- On observing the trends with respect to the inactive neurons. We see that there is a steady increase in the percentage of inactive neurons with increase in iterations and this change is more pronounced for the set of neurons closer to the output layer.

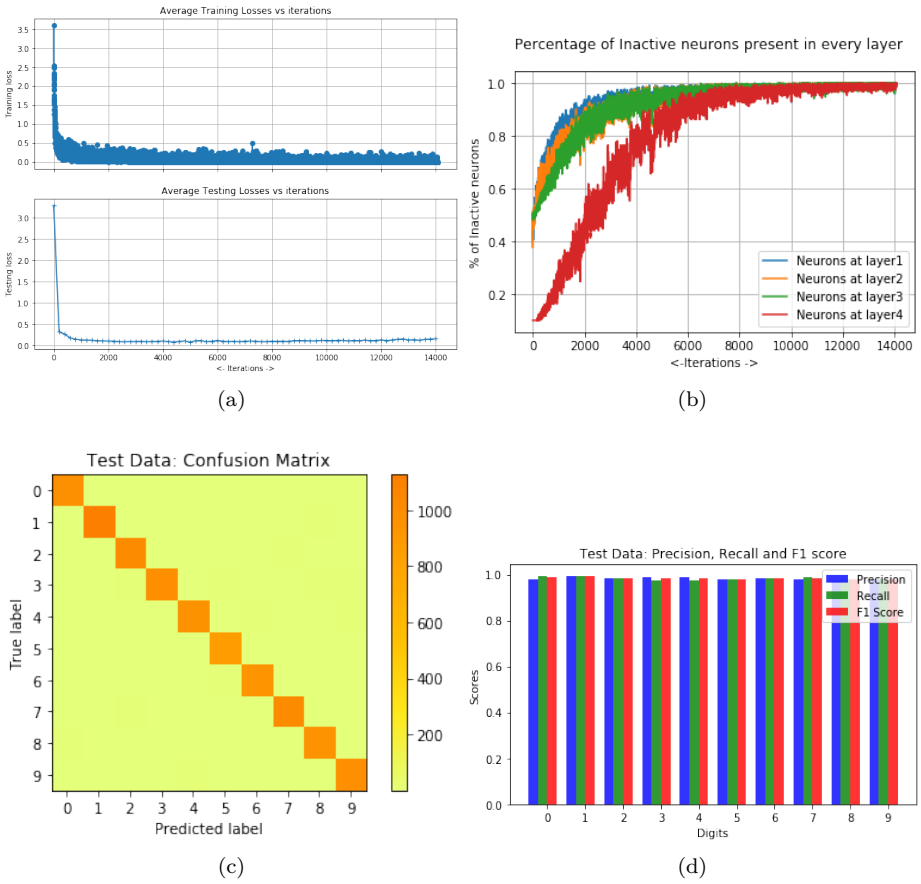
- On running a Vanilla Gradient Descent on the sigmoid activation function, we see that, the percentage of inactive neurons drops by a considerable amount. This could mean that, the more the network learns, the higher the increase in inactive neurons which makes sense for the sigmoid and the tanh activation functions which have the vanishing gradient problem.(i.e., the gradient vanishes for inputs of large magnitude)
- We also see that there is a lot of inactive neurons present for the ReLU activation as well which could be a by product of the dying neuron problem as the gradient drops to zero as soon as the neuron is fed a negative input.
- We see that the ReLU still converges much faster than the other two activation functions despite being handicapped by the Vanilla Gradient Descent.

# 4 Regularization

## 4.1 Data Augmentation with noise:

For every epoch, we add a bit of zero mean guassian noise to the input. This should theoretically contribute to a regularization of the weights which resembles the $L_2$ regularization with $\lambda \propto \sigma_i^2$, the variance of the noise. After experimenting with a few values, the noise standard deviation has been chosen as $0.25 * 10^{-2}$.
Addition of noise yields the below results.
**Results:**
Accuracy: 0.9833
Mean of errors: 0.0169
Standard deviation of errors: 0.1211

**Plots:**




(a) (b)

(c) (d)

Figure 22: Plots for the sigmoid activation function with noise added to the input

## 4.2 $L_1$ regularization

As discussed in the first section, $L_1$ regularization puts a massive penalty on the magnitude of the weights.On implementing the $L_1$ regularization, there is a slight change in the expression for back propagation which is derived below.

$$J(w,b) = \sum_i -p_i log(q_i) + \frac{\lambda}{N}\sum_i |w_i|$$

$$\frac{\partial J}{w_{ij}^{(l)}} = \delta_i^{(l+1)}.a_j^{(l)} + sign(w_{ij}^{(l)}) * \frac{\lambda}{N}$$

This change was incorporated in the gradient descent algorithms should the regularization be chosen. A $\lambda$ of $10^{-3}$ was chosen for this experiment.

**Results:**

Accuracy: 0.9815

Mean of errors: 0.0186

Standard deviation of errors: 0.1281

**Plots:**

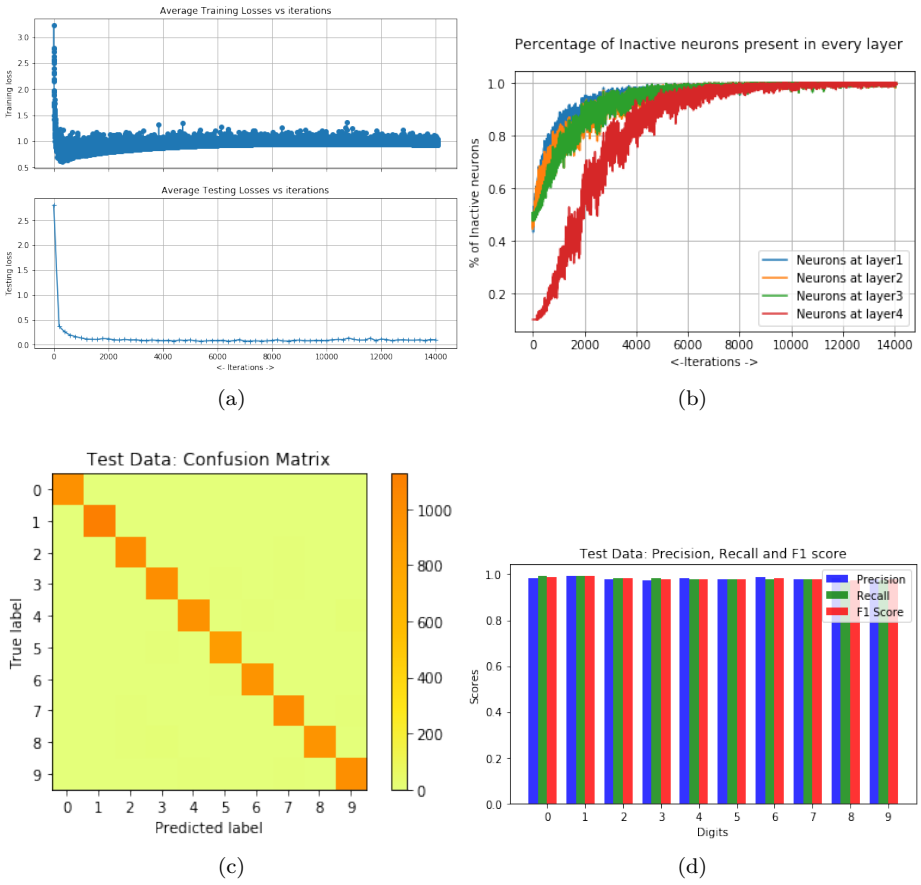



Figure 23: Plots for the sigmoid activation function with $L_1$ regularization

### 4.3 $L_2$ regularization

As discussed in the first section, $L_2$ regularization puts an even massive penalty on the magnitude of the weights.On implementing the $L_2$ regularization, just like the previous case,there is a slight change in the expression for back propagation which is derived below.

$$J(w,b) = \sum_i -p_i log(q_i) + \frac{\lambda}{2N}\sum_i |w_i|^2$$

$$\frac{\partial J}{w_{ij}^{(l)}} = \delta_i^{(l+1)}.a_j^{(l)} + w_{ij}^{(l)} * \frac{\lambda}{N}$$

This change was incorporated in the gradient descent algorithms should the regularization be chosen. A $\lambda$ of $8 * 10^{-4}$ was chosen for this experiment.

**Results:**

Accuracy: 0.9793

Mean of errors: 0.0216

Standard deviation of errors: 0.1347

**Plots:**

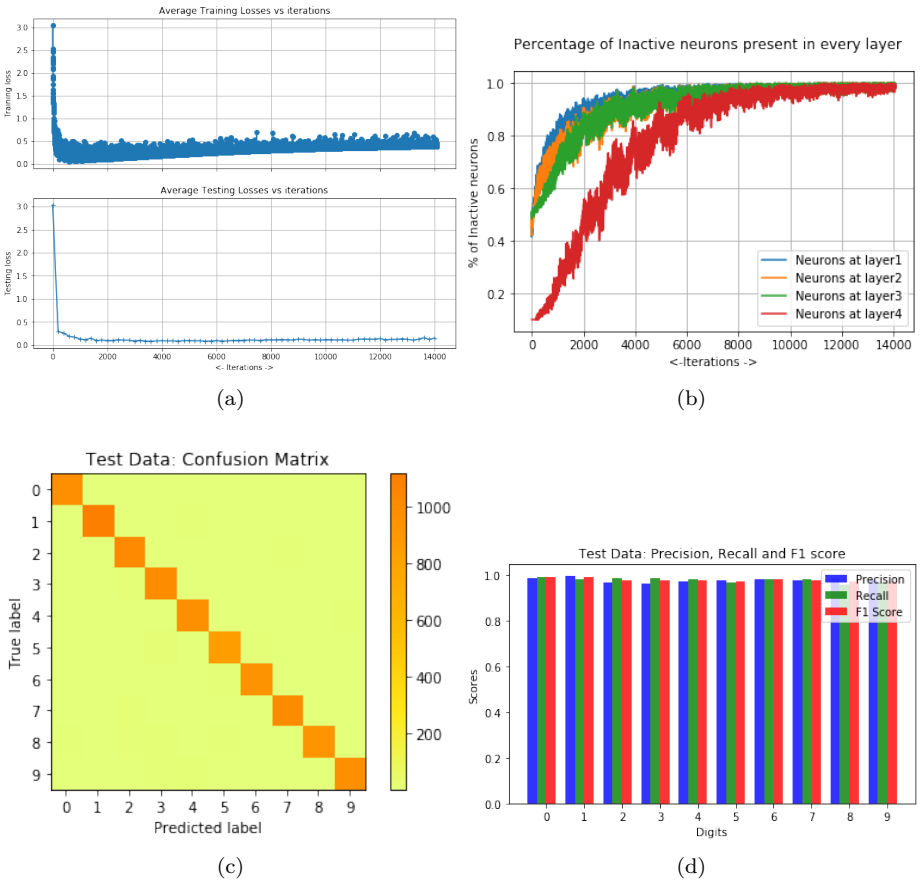



Figure 24: Plots for the sigmoid activation function with $L_2$ regularization

## 4.4 Observations

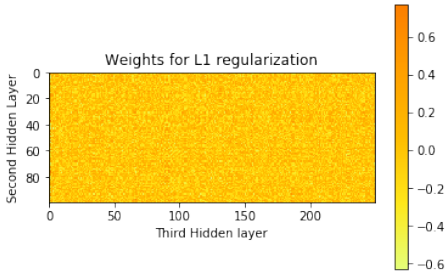



Figure 25: Colorbar plot of the weight matrix obtained through $L_1$ regularization

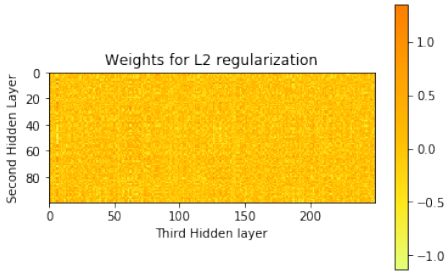



Figure 26: Colorbar plot of the weight matrix obtained through $L_2$ regularization

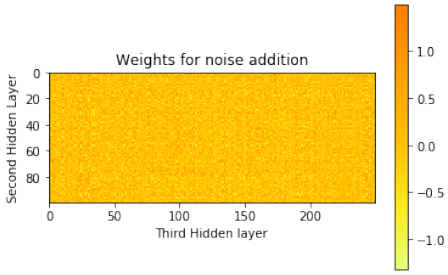



Figure 27: Colorbar plot of the weight matrix obtained through noise addition

- Firstly,we see that the weights in the last two plots are more or less similarly distributed. This is due to the choice of lambda and variance we made, they were of the same order and their impact would be equal if we were to compare the derived expressions.
- We see that the weights obtained using the $L_1$ regularization are much smaller when compared to the other two as a result of higher regularization constant but the distribution of the weights looks somewhat similar.
- In all three cases, we get a similar or slightly higher accuracy when compared to the non-regularization case, but we see that the training loss isn't minimized but increases to a slight extent as the epochs increase. This could be attributed to the fact that the regularization we've implemented isn't a strong one, and hence the cross entropy loss term dominates leading to a slight increase in the complexity of the weights.
- I also tried having a large regularization and witnessed the following.

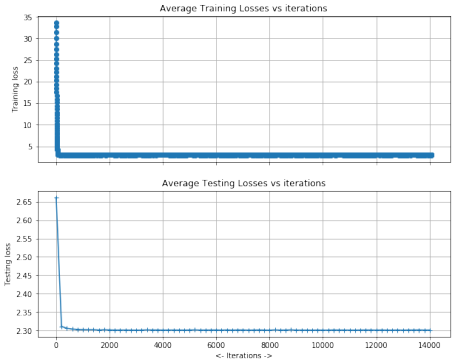



Figure 28: Training and testing losses with high regularization $\lambda = 0.1$

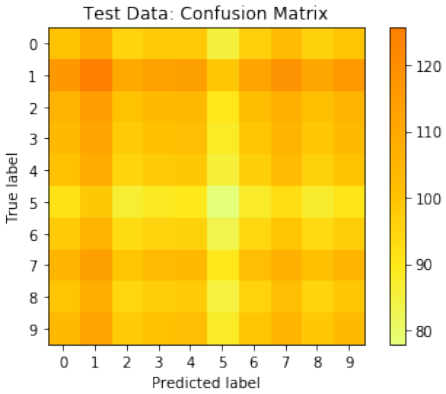



Figure 29: Confusion matrix for high regularization $\lambda = 0.1$

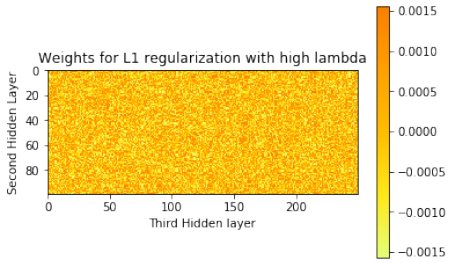



Figure 30: Colorbar plot of the weight matrix for a very high regularization

We see that a very high regularization ensures that the regularization loss dominates over the cross entropy loss and therefore, we don't witness the slight increase in the loss for higher epochs as before.
We also see that a very high regularization kills the accuracy by the sad state of the confusion matrix.However, this ensures that the weights are much smaller than the previous case(roughly by a 1000) as shown by the above plots.

# 5 Using features:

I tried two different features, both of which resulted in an output vector of size 81 after starting with an input vector of size 784.
**Principle Component Analysis:**
I tried obtaining a reduced feature vector by implementing a dimensionality reduction using principle component analysis. These features were later fed into the SVM and the K-NN classifiers.
**HoG:**

This was the non-trivial feature I tried extracting.
Before extracting the hog features, I did some preprocessing based on a few observations I made.

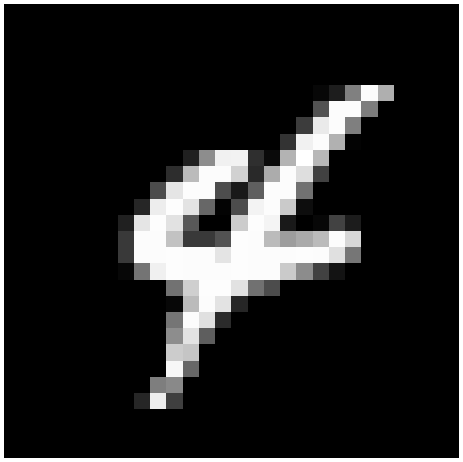


Figure 31: A sample digit which is skewed/rotated

As seen above,the digit isn't *straight*, and therefore, using first and second order moments, I tried correcting the skew/rotation which looks like the below.

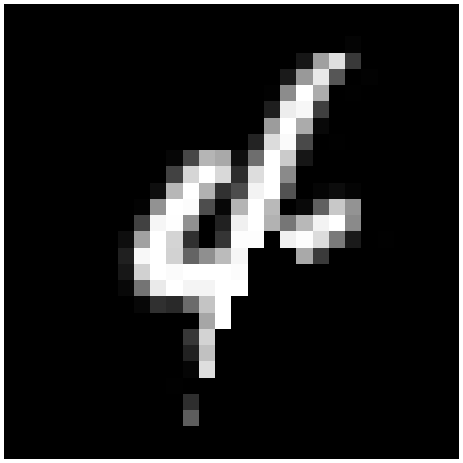


Figure 32: A sample digit with corrected skew

After this, an off the shelf algorithm for implementing hog was used after reading sources for the best set of hyperparameters. HoG or Histogram of Gradients is a dimensionality reduction approach which relies on directional gradients across cells in the image.The output of hog is usually well interpretable as seen in class, however, for the MNIST dataset, I couldn't get any intuition as to the appearance of the "HoG" image shown below.

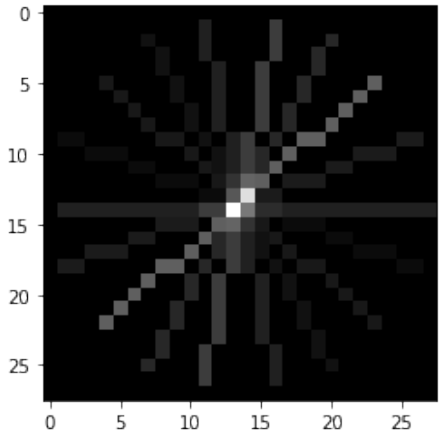


Figure 33: HoG Image

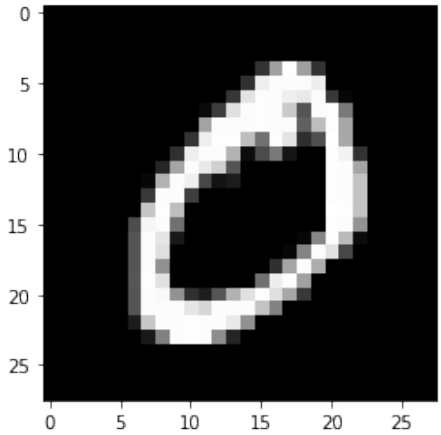


Figure 34: True Image

As in the previous case, this resulted in a feature vector of length 81 for every image and this was fed to the classifiers.

## 5.1 SVM

Two choices of SVM classifiers(one with a radial basis function and the other with a linear basis function) were tried and there wasn't a lot of difference between the two classifiers.
As expected, the classifiers performed much better for the HoG extracted features than the ones extracted using PCA leading to the below results.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.86 | 0.74 | 0.80 | 460 |
| 1 | 0.87 | 0.85 | 0.86 | 571 |
| 2 | 0.53 | 0.68 | 0.60 | 530 |
| 3 | 0.47 | 0.61 | 0.53 | 500 |
| 4 | 0.44 | 0.40 | 0.42 | 500 |
| 5 | 0.36 | 0.30 | 0.33 | 456 |
| 6 | 0.67 | 0.47 | 0.56 | 462 |
| 7 | 0.61 | 0.67 | 0.64 | 512 |
| 8 | 0.44 | 0.40 | 0.42 | 489 |
| 9 | 0.40 | 0.42 | 0.41 | 520 |
| accuracy | | | 0.56 | 5000 |
| macro avg | 0.57 | 0.56 | 0.56 | 5000 |
| weighted avg | 0.57 | 0.56 | 0.56 | 5000 |

Figure 35: SVM results for PCA

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.99 | 1.00 | 0.99 | 980 |
| 1 | 1.00 | 0.99 | 1.00 | 1135 |
| 2 | 0.99 | 0.99 | 0.99 | 1032 |
| 3 | 0.99 | 0.99 | 0.99 | 1010 |
| 4 | 0.99 | 0.99 | 0.99 | 982 |
| 5 | 1.00 | 0.99 | 0.99 | 892 |
| 6 | 0.99 | 0.99 | 0.99 | 958 |
| 7 | 0.98 | 0.99 | 0.99 | 1028 |
| 8 | 0.99 | 0.99 | 0.99 | 974 |
| 9 | 0.99 | 0.98 | 0.99 | 1009 |
| accuracy | | | 0.99 | 10000 |
| macro avg | 0.99 | 0.99 | 0.99 | 10000 |
| weighted avg | 0.99 | 0.99 | 0.99 | 10000 |

Figure 36: SVM results for HoG

Note: The above accuracy values are rounded to the second decimal place, actual accuracy is somewhere around 0.985.

## 5.2 K-NN

The K-NN classifier was implemented using a hyperparameter K of 200. However, on running the classifier across different K, I didn't really get a huge variations in the results. The K-NN classifier also follows the same trends set by the SVM ones.

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.65 | 0.83 | 0.73 | 460 |
| 1 | 0.62 | 0.93 | 0.74 | 571 |
| 2 | 0.82 | 0.41 | 0.55 | 530 |
| 3 | 0.50 | 0.52 | 0.51 | 500 |
| 4 | 0.43 | 0.68 | 0.53 | 500 |
| 5 | 0.54 | 0.27 | 0.36 | 456 |
| 6 | 0.68 | 0.52 | 0.59 | 462 |
| 7 | 0.66 | 0.50 | 0.57 | 512 |
| 8 | 0.42 | 0.49 | 0.45 | 489 |
| 9 | 0.43 | 0.36 | 0.39 | 520 |
| accuracy | | | 0.56 | 5000 |
| macro avg | 0.58 | 0.55 | 0.54 | 5000 |
| weighted avg | 0.58 | 0.56 | 0.54 | 5000 |

Figure 37: K-NN results for PCA

| | precision | recall | f1-score | support |
|---|---|---|---|---|
| 0 | 0.98 | 0.99 | 0.99 | 980 |
| 1 | 1.00 | 0.98 | 0.99 | 1135 |
| 2 | 0.98 | 0.98 | 0.98 | 1032 |
| 3 | 0.95 | 0.99 | 0.97 | 1010 |
| 4 | 0.99 | 0.97 | 0.98 | 982 |
| 5 | 0.99 | 0.97 | 0.98 | 892 |
| 6 | 0.98 | 0.99 | 0.98 | 958 |
| 7 | 0.98 | 0.97 | 0.97 | 1028 |
| 8 | 0.96 | 0.96 | 0.96 | 974 |
| 9 | 0.96 | 0.96 | 0.96 | 1009 |
| accuracy | | | 0.98 | 10000 |
| macro avg | 0.98 | 0.98 | 0.98 | 10000 |
| weighted avg | 0.98 | 0.98 | 0.98 | 10000 |

Figure 38: K-NN results for HoG

## 5.3 MLP

I tried experimenting a lot with the MLP by changing the number of layers, changing the learning rates, implementing drop out, increasing the number of epochs,etc.
Decreasing the number of neurons in the layer didn't really improve accuracy. Dropout didn't really improve accuracy as well, but it improved the performance of the neurons,i.e., there were fewer inactive neurons in each hidden layer as shown in the below plot.

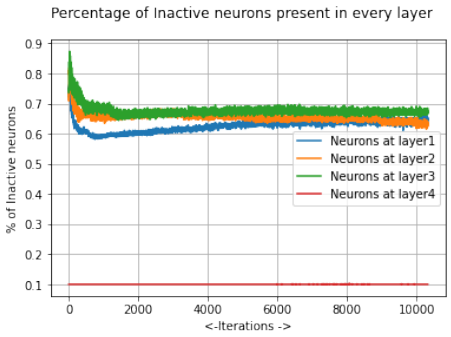
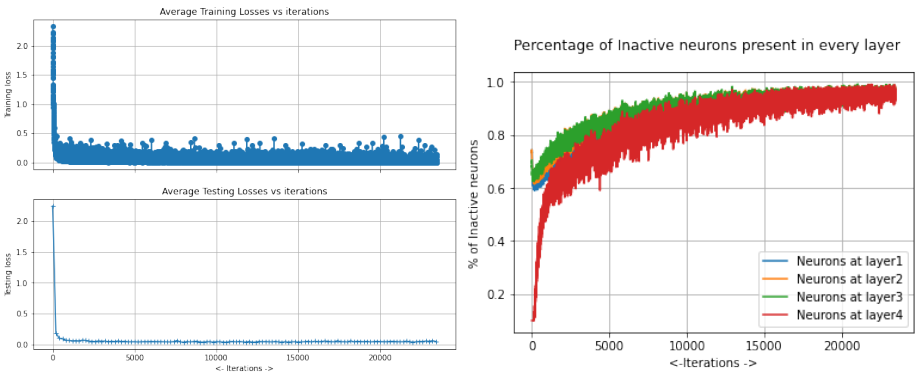


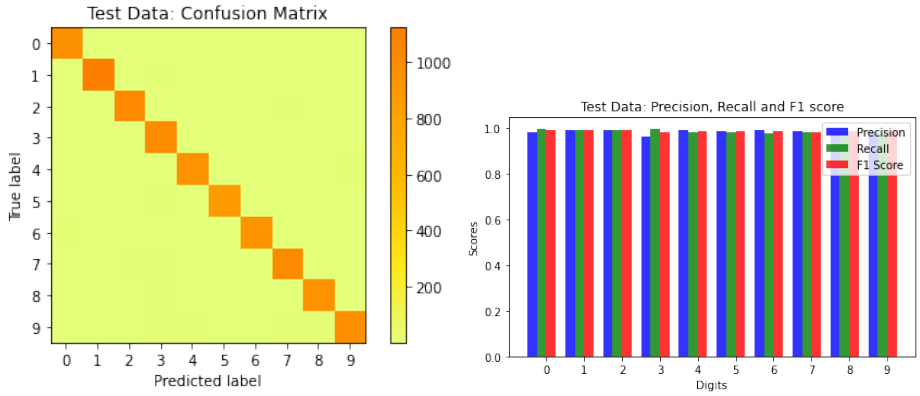

Figure 39: Inactive neurons while implementing dropout

Decreasing the learning rate and increasing the number of epochs improved the accuracy of the neural network.
So I finally went with the same architecture as the rest of the assignment but with a ReLU activation function using an RMS Prop GD with a learning rate of 0.025 running over for 25 epochs. This ended up with the following results for the HoG feature vectors.
**Results:**
Accuracy: 0.989
Mean of errors: 0.0133
Standard deviation of errors: 0.0963

**Plots:**




(a) (b)




(c) (d)

Figure 40: Plots for the best performing MLP

## 5.4 Observations

- We see that our MLP performs almost on par with Le Cun's state of the art neural networks.It makes just 110 mistakes out of the whole test dataset.
- We see that the K-NN and the SVM approaches don't lag behind the MLP by a lot. The additional minimal accuracy obtained by the MLP doesn't justify the extra training time and additional storage space(when compared to SVM) it requires. Therefore, the conventional machine learning classifiers seem to be better suited for this problem.
- As discussed in the first few sections,the feature extraction didn't really improve the MLP's performance by a lot.This could probably be because, the complex fully connected architecture in the MLP would've deduced all the information provided by the features thereby reducing it's use.However, as the feature vector was smaller in size, the MLP trained much faster.
- We saw that the linear SVM classifier performed as well as the rbf SVM classifier. This means, that the classes are well separated in the 784 dimensional space.Let's see how well they are separated when reduced to the 2 dimensional space.

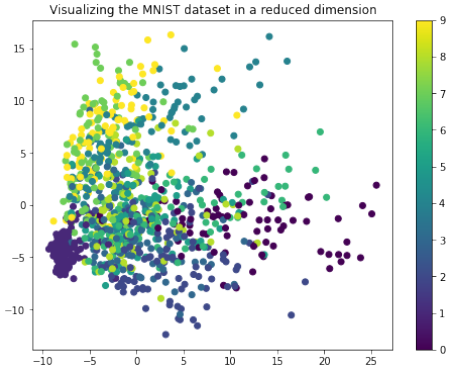



Figure 41: The MNIST images visualized in lower dimensions

We see that the above plot is very messy, which implies that the there is a lot of mixing amongst the classes in the lower dimensions. This tells us why the PCA dimensionality reduction didn't work very well as any digit could be mistaken for the other save 0 as shown in the above plot.
Let us see what happens, when the hog features are brought to a lower dimension.

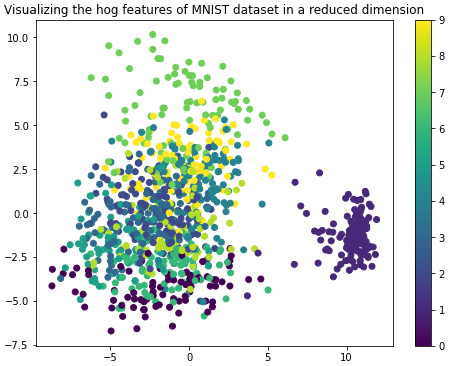



Figure 42: The MNIST hog images visualized in lower dimensions

The HoG unclutters some of the mess in the original image, thereby demonstrating that it is superior when it comes to dimensionality reduction thereby resulting in a better performance when it came to the classifiers.