Assignment 1 - MLP

Subhankar Chakraborty

September 4, 2019

1 Introduction

The assignment was on coding a multi layer neural network from scratch and see the effects of various activation functions as well as regularization techniques on it's performance. Performance was measured using parameters like the f1 score, precision, recall and confusion matrix. Effects of data augmentation were also tested. The test accuracy was also compared to traditional Machine Learning techniques like SVM's and basic classifiers like the k-nearest neighbors classifier.

2 The Programming environment

A conda based python 3.6 programming environment was used. The following needs to done to run the code properly.

Required libraries

The following libraries are required for running the code properly.

- numpy
- matplotlib
- python-mnist
- wget
- sklearn
- skimage
- sys
- importlib
- os

Installing the libraries

There is a file names *requirements.txt* which has all the required libraries mentioned. Also, a bash script called *setup.sh* has been provided which can be run to install the given libraries. Just ensure that you have **pip3 preinstalled.**

```
source setup.sh
```

The above line sets up the environment.

3 Modularity of the code

The following files and directories exist.

- **background** - Contains PDF's for the necessary background.
- **Images** - Contains matplotlib plots and terminal screenshots.
- **source-codes** - Contains the source codes. The following files are contained in it.
  - **activations.py** - Contains definitions of the different activation functions.
  - **derivatives.py** - Contains the derivatives of the respective activation functions.
  - **classifiers.py** - Conatins the K-nearest-neighbors and SVM based classifiers.
  - **evaluations.py** - Contains code for finding out various metrics like accuracy, precision, recall and confusion matrix.
  - **downloader.py** - Downloads the MNIST dataset and parses it such that it is accessible to the other codes.
  - **image-transformations.py** - Adds noise to the images for data augmentation and also contains the hog feature extractor.
  - **regularization.py** - Contains code for calculating the loss with L2 regularization as well as adding noise during training to the activations.
  - **neuralnet.py** - Contains the main algorithms like back prop, feed forward, gradient descent etc.
  - **main.py** - Acts like a wrapper giving the user accessibility to whatever section he/she wants to see.
- **extras** - The assignment PDF and the setup scripts.

4 Q1 - MLP Architecture and training.

A multi layer network was implemented using numpy. The layer sizes are [784, 5000, 250, 100, 10] with the first and last being the sizes of the input and output respectively and the other three being the sizes of the hidden layers. The activation function used is sigmoid. The weights were initialized using a glorot initialization. The other parameters included are the precision, recall, accuracy and the confusion matrix.

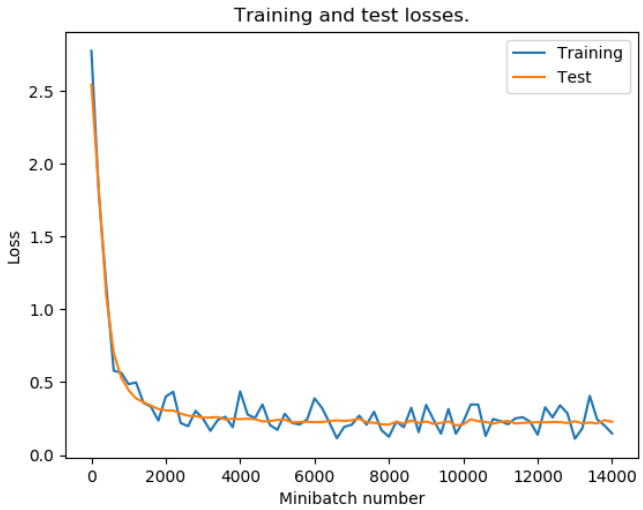

Figure 1: Loss curves

Performance metrics

The results are shown with $\alpha = 0.05$. The performance metrics along with the confusion matrix is given below.

```
precision -- [0.96203796 0.98144876 0.92358491 0.93997965
0.88374291 0.94850299 0.96805112 0.93980583 0.90659898
0.91811668]
recall -- [0.98265306 0.97885463 0.94864341 0.91485149
0.95213849 0.88789238 0.94885177 0.94163424 0.91683778
0.88899901]
f1 score -- [0.97223624 0.98014998 0.93594646 0.92724536
0.91666667 0.91719745 0.9583553  0.94071914 0.91168964
0.90332326]
```

The confusion matrix is shown below.

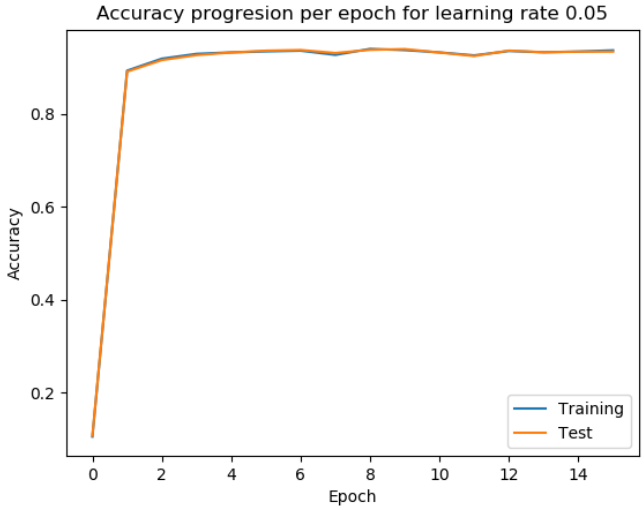

Figure 2: Accuracy curves

```
Confusion Matrix
[[ 963.    0.    2.    1.    1.    6.    2.    2.    2.    1.]
 [   0. 1111.    5.    2.    0.    0.    3.    4.   10.    0.]
 [  10.    0.  979.    7.    6.    0.    4.   13.    9.    4.]
 [   2.    0.   27.  924.    1.   17.    1.   14.   20.    4.]
 [   1.    0.    5.    1.  935.    0.    3.    2.    7.   28.]
 [   7.    5.    2.   26.   17.  792.   11.    4.   19.    9.]
 [  10.    2.    3.    0.   22.    6.  909.    0.    6.    0.]
 [   1.    5.   22.    2.    5.    0.    0.  968.    1.   24.]
 [   3.    4.   14.   14.   14.   10.    5.    7.  893.   10.]
 [   4.    5.    1.    6.   57.    4.    1.   16.   18.  897.]]
```

Figure 3: Confusion Matrix for sigmoid

Inferences

The following inferences can be drawn (some of them are not apparent from the graphs here. The graphs have been plotted for an α value of 0.01)

- The accuracy of the model on the test set is on an average about **0.9371**
- The digits 8 and 4 have low values of precision showing the model is detecting a lot of false positives for those digits.
- The digits 5 and 9 have low recall values showing that the classifier is detecting a lot of false negatives for those digits.
- From the f1 scores we see the model is performing the best for 0,1 and the worst for 9.

- There is no particular trend among the precision or recall being higher than the other, showing that the model is equally susceptible to false positives as well as false negatives.

5 Q2 - Activation Functions

Here we try with other activation functions like **ReLU** and **tanh**.Note that **I had to shrink the pixel values of the data between 0 and 1** for it to work properly with **ReLU**. Such kind of mean normalization was used for **tanh** also. No such mean normalization was used for **sigmoid** however. A comparative training and test loss as well as training and test accuracy plots for different learning rates are plotted below.

Figure 4: Training loss at learning rate 0.005

One such plot for accuracy is also included to see how fast are the activation functions leading to high test accuracy rates.

Performance with ReLU

Test accuracy obtained with ReLU is **0.9721**.The performance metrics with ReLU used as activation are

```
precision -- [0.97076613 0.98169137 0.98129921 0.96582031
0.97948718 0.97291196 0.97696335 0.96862745 0.95736041
 0.965]
recall -- [0.98265306 0.99207048 0.96608527 0.97920792
0.97250509 0.96636771 0.97390397 0.96108949 0.96817248
```

Figure 5: Training loss at learning rate 0.01

Figure 6: Training loss at learning rate 0.05

```
0.95639247]
f1 score -- [0.97667343 0.98685364 0.97363281 0.97246804
0.97598365 0.9696288 0.97543126 0.96484375 0.96273609
0.96067695]
```

The confusion matrix is shown below.

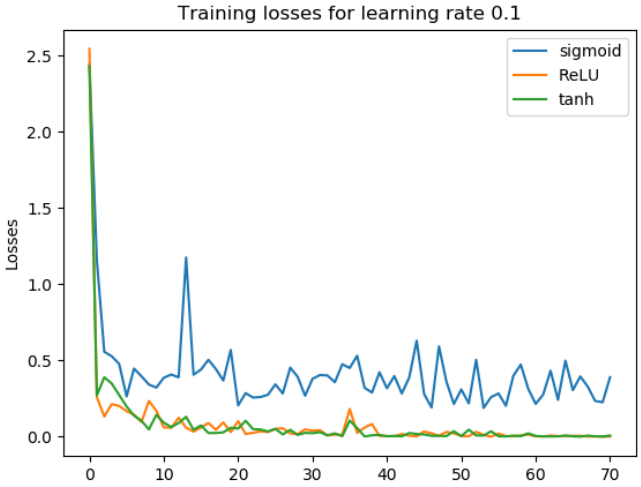

Figure 7: Training loss at learning rate 0.1

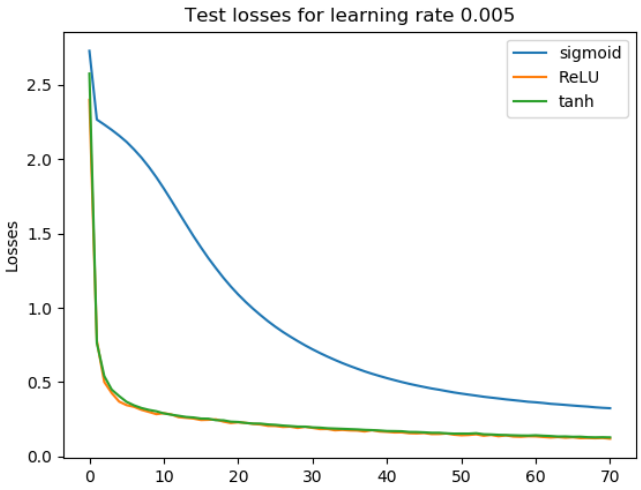

Figure 8: Test loss at learning rate 0.005

Performance with tanh

Test accuracy obtained with tanh is **0.9681**. The performance metrics with tanh used as activation are

```
precision -- [0.97686117 0.98510079 0.98326772 0.98490946
0.98134715 0.97640449 0.98105263 0.95992366 0.95858586
0.96146245]
```

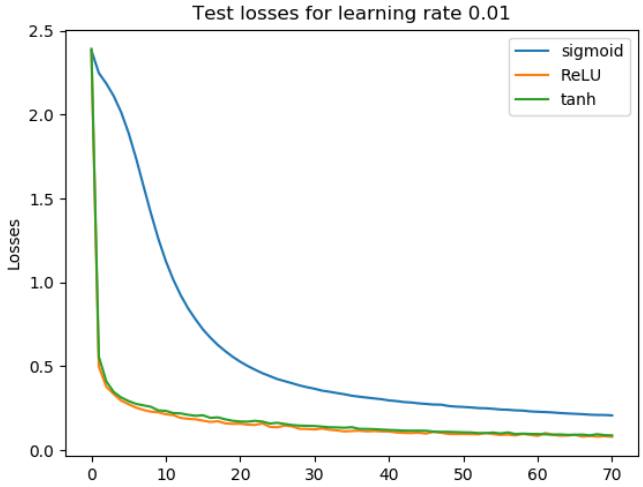

Figure 9: Test loss at learning rate 0.01

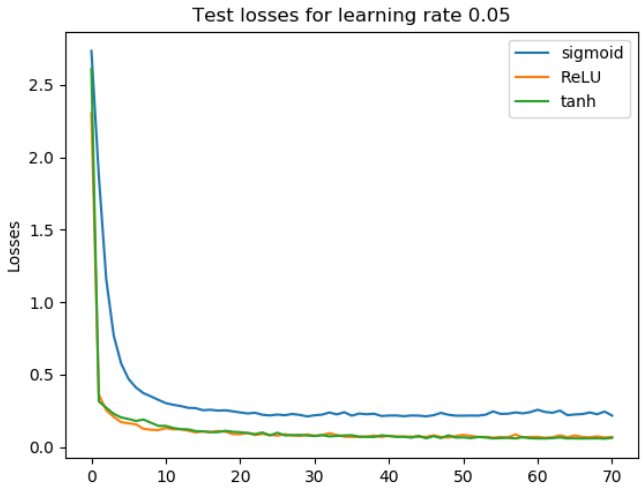

Figure 10: Test loss at learning rate 0.05

```
recall -- [0.99081633 0.99030837 0.96802326 0.96930693
0.96435845 0.97421525 0.97286013 0.97859922 0.97433265
0.96432111]
f1 score -- [0.98378926 0.98769772 0.97558594 0.97704591
0.97277863 0.97530864 0.9769392  0.96917148 0.96639511
0.96288966]
```

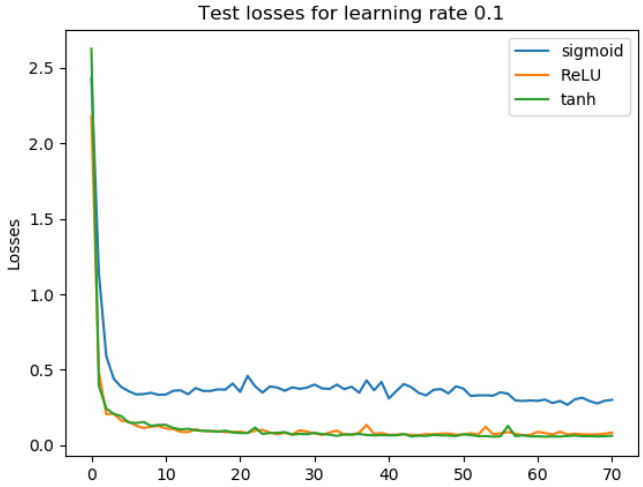

Figure 11: Test loss at learning rate 0.1

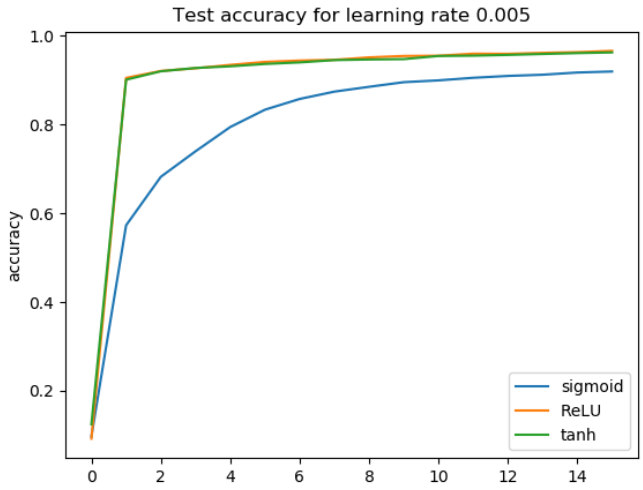

Figure 12: Test accuracy at learning rate 0.005

The confusion matrix is shown below.

Percentage of Inactive Neurons

We define those neurons as inactive where the magnitude of gradient is lesser than 10^{-5}. The percentage of inactive neurons for the three different activation functions as the training progresses is shown below.

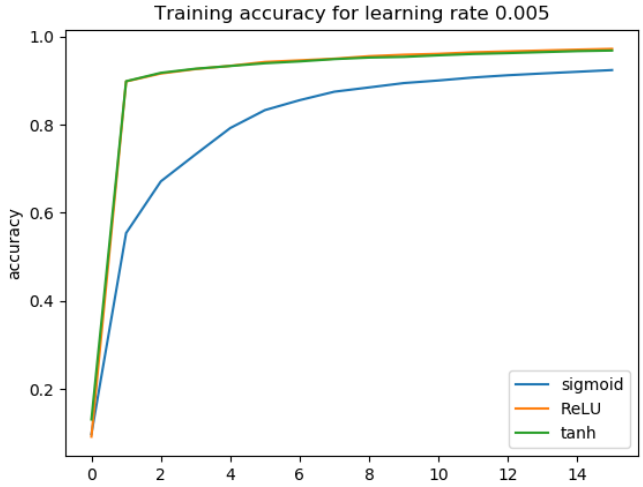

Figure 13: Training accuracy at learning rate 0.05(please note the label on the plot is wrong)

```
Confusion Matrix
[[ 963.    0.    1.    0.    0.    2.    5.    2.    4.    3.]
 [   0. 1126.    3.    1.    0.    1.    2.    0.    2.    0.]
 [   7.    2.  997.    4.    3.    0.    2.    8.    9.    0.]
 [   0.    0.    1.  989.    0.    7.    0.    5.    6.    2.]
 [   2.    0.    3.    1.  955.    0.    3.    2.    1.   15.]
 [   6.    0.    0.    7.    2.  862.    7.    1.    5.    2.]
 [   5.    3.    0.    1.    5.    5.  933.    1.    5.    0.]
 [   1.   11.    9.    4.    0.    0.    0.  988.    4.   11.]
 [   4.    1.    1.    7.    3.    5.    3.    5.  943.    2.]
 [   4.    4.    1.   10.    7.    4.    0.    8.    6.  965.]]
```

Figure 14: Confusion Matrix for ReLU

```
Confusion Matrix
[[ 971.    0.    0.    0.    1.    2.    1.    1.    2.    2.]
 [   0. 1124.    2.    1.    0.    1.    2.    2.    3.    0.]
 [   5.    3.  999.    2.    1.    0.    3.   10.    9.    0.]
 [   0.    0.    4.  979.    0.    5.    0.   10.    7.    5.]
 [   3.    0.    2.    0.  947.    0.    3.    3.    2.   22.]
 [   2.    0.    0.    3.    1.  869.    6.    0.    9.    2.]
 [   6.    3.    0.    1.    5.    6.  932.    1.    4.    0.]
 [   1.    7.    7.    0.    0.    1.    0. 1006.    1.    5.]
 [   4.    0.    2.    3.    4.    3.    2.    4.  949.    3.]
 [   2.    4.    0.    5.    6.    3.    1.   11.    4.  973.]]
```

Figure 15: Confusion Matrix for tanh

Inferences

The following inferences can be drawn from the graphs.

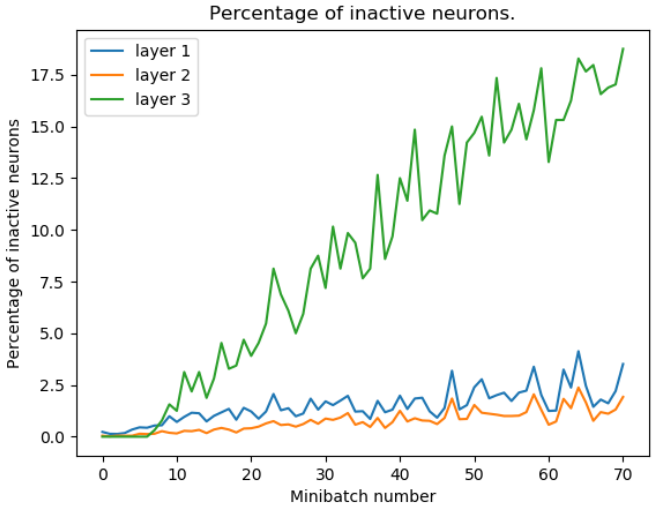

Figure 16: Percentage of inactive neurons for sigmoid.

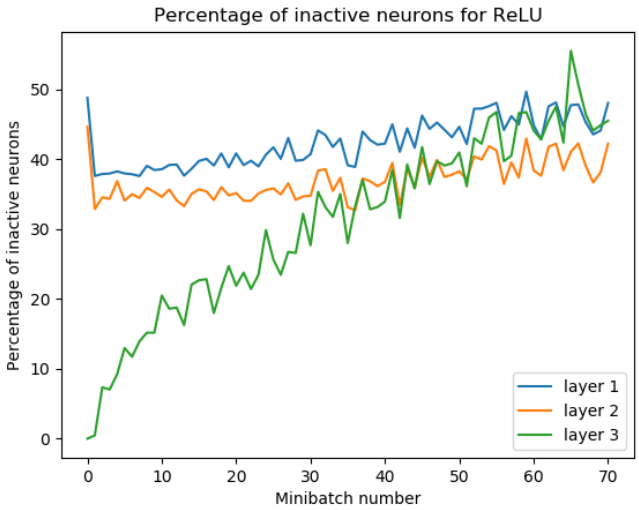

Figure 17: Percentage of inactive neurons for ReLU.

- The performance with ReLU is slightly better to that of tanh, both of which perform markedly better than sigmoid.
- The training losses for ReLU and tanh converge almost at the same rate, which is in general much faster for a given learning rate than that of sigmoid.
- ReLU and tanh start giving 90 percent and above accuracy values from

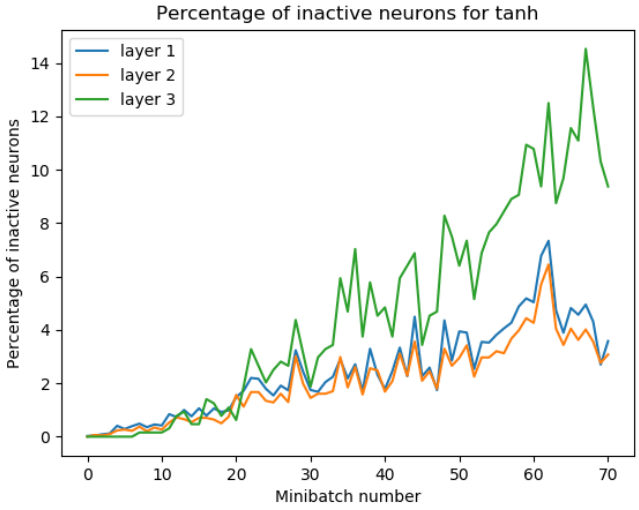

Figure 18: Percentage of inactive neurons for tanh.

the first epoch even for learning rates as low as 0.005 which is faster compared to sigmoid.

- For learning rates below 0.005, the convergence in sigmoid is incomplete.
- For higher learning rates, the training as well as test losses for sigmoid does begin to converge faster.
- If however we increase the learning rate too much, the training loss for sigmoid starts oscillating showing that the learning rate is too high.
- The optimal performance for sigmoid is seen at a learning rate of 0.05 where the convergence is fast enough without showing oscillations.
- ReLU and tanh are much more invariant in their convergence depending on the change of the learning rate, both of which converge in general faster than sigmoid.
- As for the number of inactive neurons, the number of inactive neurons is more for ReLU, which is significantly larger than the other two.
- The higher values for ReLU are expected as its gradient will become zero whenever it encounters a negative sample, forever killing that neuron.

6 Q3 - Regularization

In this section we try different forms of regularization and see its effects on the training and test performance. We use L2 regularization and use noise addition to dataset as a data augmentation method.

Adding noise to hidden layers during training.

The following results were found. A few plots have been mentioned to support the argument.

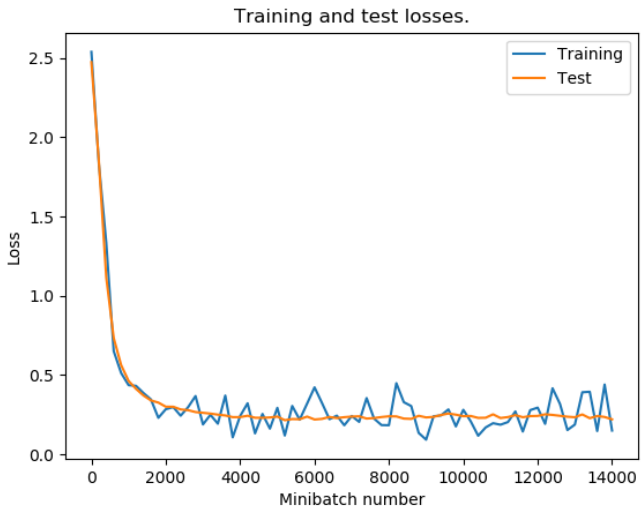

Figure 19: Loss curve with $\sigma = 0.01$ noise added to forward prop.

- As expected, adding noise of very less standard deviation (0.001 or below) does nothing.
- Adding reasonable noise does give a slight improvement in performance. The improvement is rather in faster convergence of the loss function than any noticeable change in accuracy on the test set.
- Although not plotted here, adding noise with a very high standard deviation throws off the neural network.
- No significant difference was noticed in adding noise to forward prop or back prop.

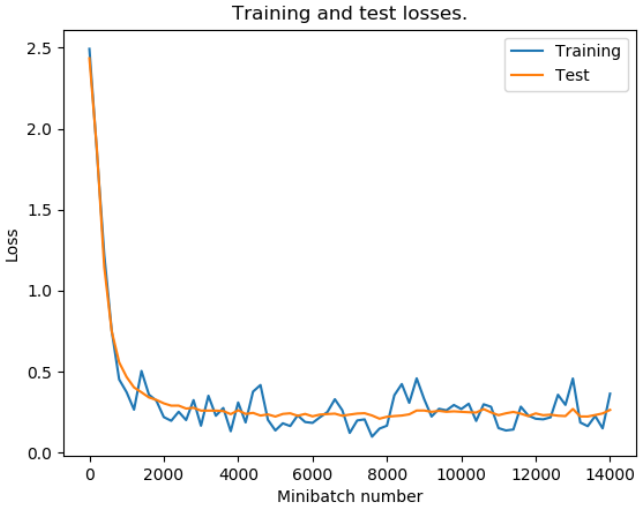

Figure 20: Loss curve with $\sigma = 0.01$ noise added to back prop.

Data augmentation and L2 regularization

Data augmentation

In this section we increase our dataset by by adding some gaussian noise to the training images and using them also for training along with the non noisy images. A few plots which are significant are shown.

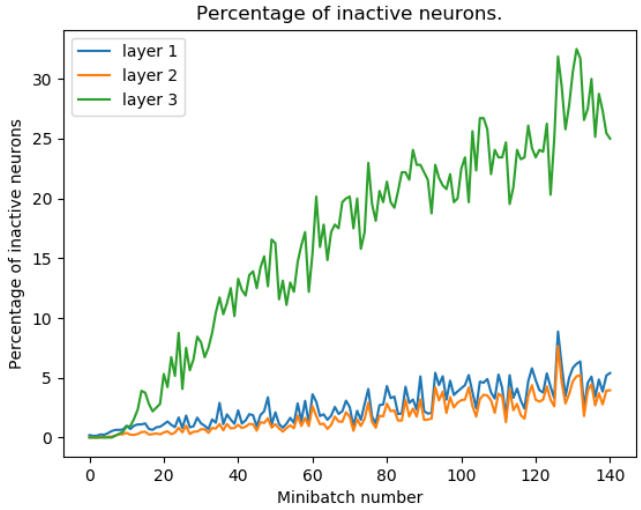

Figure 21: Inactive neurons with $\sigma = 10$ noise added to dataset.

The following inferences can be drawn (some of which are obvious and not plotted to keep the report succinct).

- Adding noise of very less standard deviation (0.1 or lesser) does not do anything. It on the contrary is detrimental as the pixel values range from 0 to 255 and adding such a small noise to them is doing nothing significant. Which basically means, we are augmenting the data with something very similar to what we had originally, leading to overfitting and a drop in performance (mine dropped to 90 percent).
- For reasonable values of sigma (1), we do not see much improvement in performance on the testset over the original model. However as in the case of adding noise to backprop and forward prop, we see a faster convergence.
- Another noticeable change is a significant increase in the number of inactive neurons for the last layer.
- If sigma is increased to a very high value (¿30), the test accuracy begins to take a drop as the training dataset has samples now very different from the test dataset (This again is not plotted as it is an obvious fact).

Using L2 regularization

The plots are shown below.

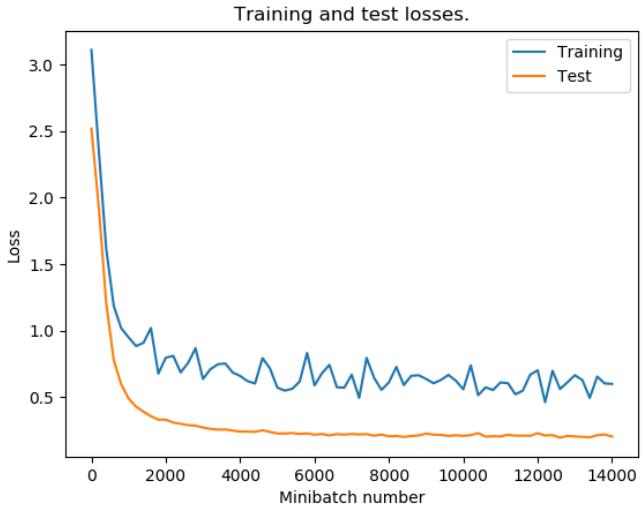

Figure 22: Loss curves for $\lambda = 0.05$

The following inferences can be drawn.

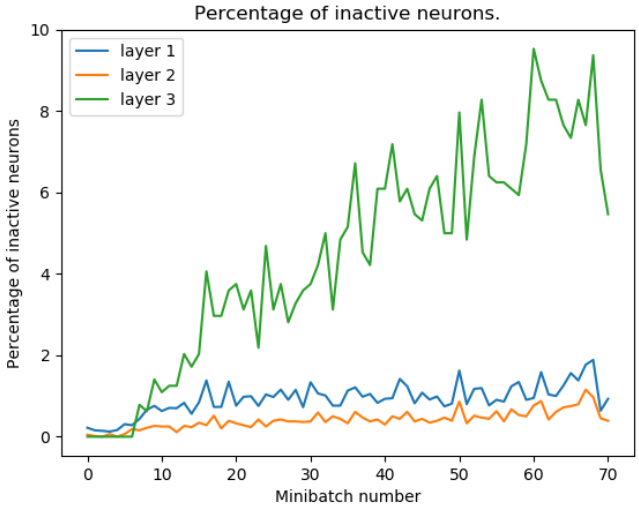

Figure 23: Inactive neurons for $\lambda = 0.05$

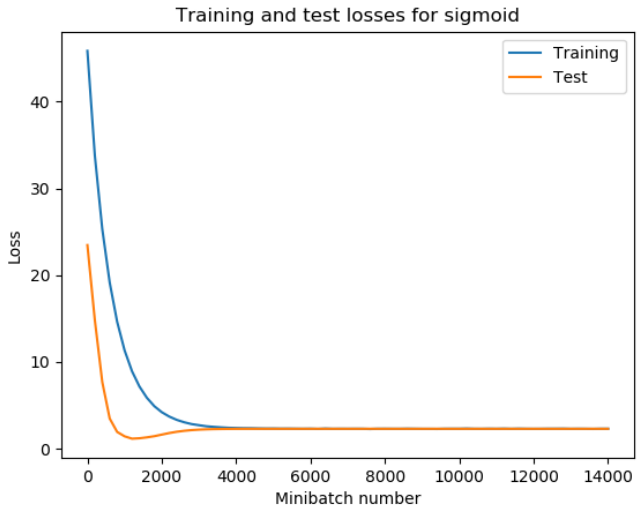

Figure 24: Loss curves for $\lambda = 5$

- For very low values of λ, there is no effects on the training accuracy as expected.
- For very high values of λ we get a very low test accuracy showing that our data has high bias. One such example has been plotted with $\lambda = 5$.
- The optimal performance is observed with λ around 0.05 which gave

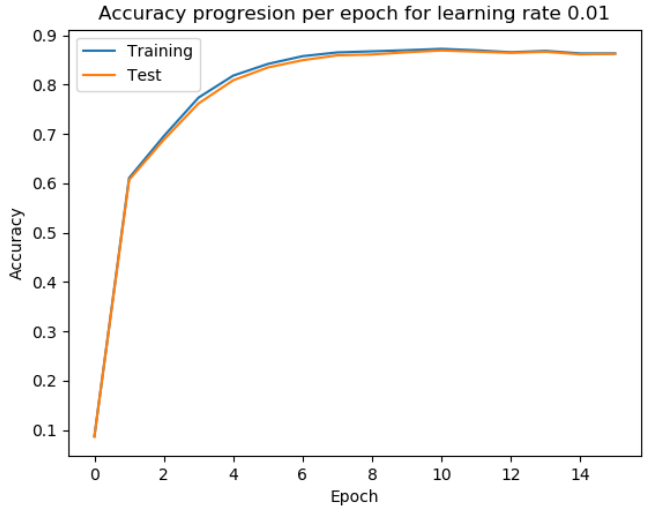

Figure 25: Accuracy plot for λ=5

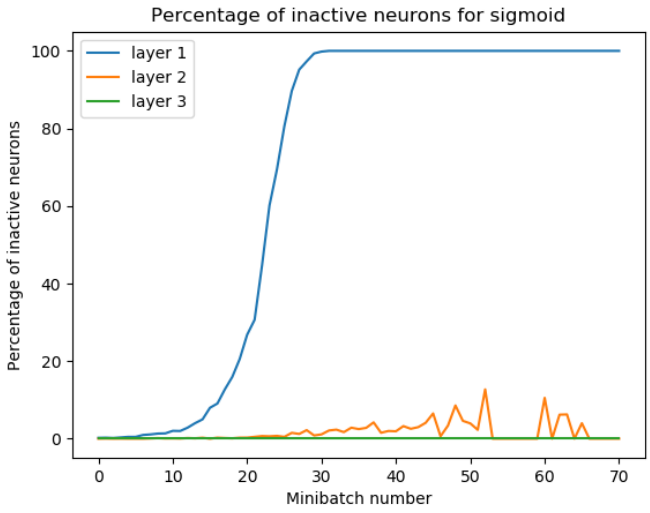

Figure 26: High λ kills gradients at the neurons.

me on an average **94.43** percent test accuracy compared to **93.71** on the non regularized data.

- We also see a decrease in the number of inactive neurons with regularization added.
- The training losses are higher than those without regularization as expected as we have an additional term.

- One interesting observation is that a high value of regularization almost kills all the neurons in one layer.

7 Q4 - Hand crafted features

Here we use hand crafted features for training a custom neural network and compare its performance to traditional machine learning techniques like SVM's and K-nearest-neighbors classifiers. Note that we are using the module **skimage** for this purpose. We extract a 180 length hog feature vector from each image and use it for training the network as well as the classifiers. The choice of HOG is justifies as the MNIST datasets being high contrast images, their gradient representation will contain a significant part of the information.

```
def hog_transform(image, number_of_bins = 10, pixels_per_cell_=(7,8)):
    """
    Returns a hog feature vector of length 180
    """
    return hog(image,orientations=number_of_bins,pixels_per_cell
    =pixels_per_cell_)
```

The neural network

We choose a neural network with layer sizes [180, 120, 60, 25, 10] with the first and last being input and output layers and the ones in the middle being the hidden layers. The activation used is ReLU and learning rate used is 0.01. We are not using any kind of regularization here as the network is relatively simple. The loss plot is given below. The accuracy plot is given below.

The obtained performance metrics are

```
precision -- [0.96582915 0.97623239 0.95717131 0.919201520.
96597938 0.95351474 0.97248677 0.94871795 0.93592437 0.9]
recall -- [0.98061224 0.97709251 0.93120155 0.95742574
0.95417515 0.942825110.95929019 0.93579767 0.91478439
0.93657086]
f1 score -- [0.97316456 0.97666226 0.94400786 0.93792435
0.96004098 0.9481398 0.96584341 0.94221352 0.92523364
0.91792132]
```

The obtained confusion matrix is

The accuracy obtained was **0.9542**.

Figure 27: Training Loss for the neural network trained using HOG classifier.

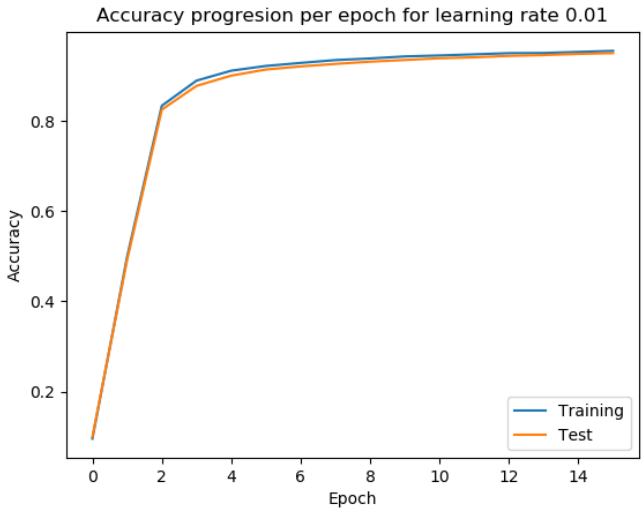

Figure 28: Accuracy for the neural network trained using HOG classifier.

Using SVM

Here we use SVM from sklearn with the default parameters. The code is given below.

```
class svm_classifier(classifier):

    def initiate_classifier(self, kernel_="linear",
```

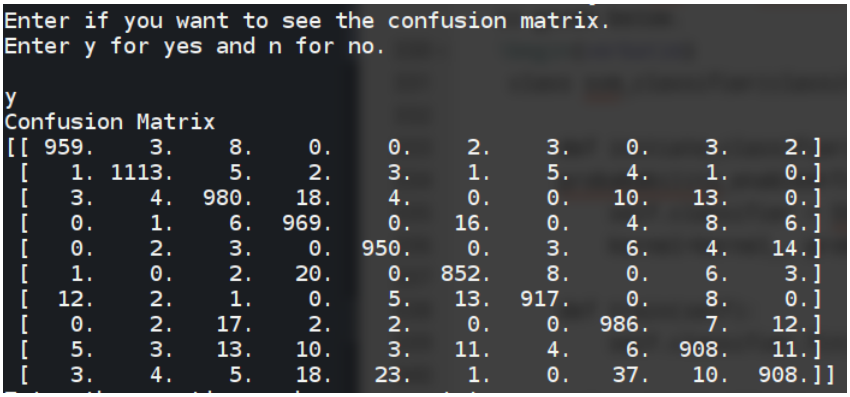

Figure 29: Confusion Matrix for neural network using Hog feature vector

```
        probabability_enabled=True):
            self.classifier = SVC(
            kernel=kernel_, probability=probabability_enabled)

        def train(self):
            self.classifier.fit(self.inputs, self.labels)

        def predict(self, x):
            return self.classifier.predict(x)
```

The test accuracy obtained was around 0.97(0.9698), significantly higher than those attained on the neural network.

Using KNN

Here we use KNN from sklearn with the default parameters (5 neighbors). The code is given below.

```
    class knn_classifier(classifier):

        def initiate_classifier(self, n_neighbors_=5):
            self.classifier = knn(n_neighbors=n_neighbors_)

        def train(self):
            self.classifier.fit(self.inputs, self.labels)

        def predict(self, x):
            return self.classifier.predict(x)
```

The accuracy obtained on the testset was **0.9674**.

Inferences

The following inferences can be drawn.

- The SVM classifier performs the best on the testset, even better than the neural network.
- The KNN classifier does not perform as well as the SVM classifier. But even that outperforms the neural network.
- A main reason for this is that the MNIST dataset is a relatively simple dataset with lesser number of features per image. A feedforward neural network is not the best suited for such a task.
- Adding to the previous point, feed forward networks are better for classifying higher dimensional problems.
- When the feature vectors are chosen carefully, traditional ML classifiers like SVM's are much more efficient showing the fact that they are better suited than neural networks for lower dimensional tasks.