Deep Learning Assignment 1 Report

Siddharth Agrawal 2014ME10688
Harsh Sahu 2014ME10657
Vikas Deep 2014ME10692

ABSTRACT

In this report, we have discussed our overall approach to the problem including the architecture used, the experiments that we have carried on the EMINIST dataset. We have also discussed the reasons behind the various experiments that we carried out and also the findings from these experiments. Some explanations and reasoning for the results obtained from the experiment has also been discussed in this report.

1 Dataset

According to the assignment problem the dataset that is to be used is the EMINIST dataset which has scanned images of hand-written Roman alphabets. However due to limited computational resources we were unable to use our code on this data set as it took too much time for training. Therefore we decided to use the MNIST (digits dataset) for the classification problem. Since the assignment problem was to be done as 9 class classification problem so we chose 0,1,2..8 as our 9 classes for classification problem. Since this was smaller dataset our code was able to work on this dataset. The dataset includes grayscale images of dimension 28 × 28. Therefore, our input feature vector corresponding to each example is of dimension 784. We have used the *MNIST* python library for extracting the training and test data from the ubyte files given in the dataset. We first convert the ubyte files into numpy arrays. We then filter out only those examples which belong to one of the nine classes that we are trying to learn. For the training and test output labels we create a nine dimensional vector corresponding to each example as we are trying to solve a nine class classification problem. Therefore, after pre-processing the data, we create the following numpy arrays:

1. Training data (dim = no. of training examples x 784)
2. Training labels(dim = no. of training examples x 9)
3. Test data(dim = no. of test examples x 784)
4. Test labels(dim = no.of test examples x9)

2 Model

Neural Network: Hidden Layers and Hidden Units

We have used a neural network architecture for this classification problem. We have varied the number of hidden layers and the number of hidden units in each layer of our neural network for studying the effect of these variables on the accuracy of the classifier, and the training and test costs. All the weights in the different layers of the network were initialised to a random value between 0 and 1, by using Python′s random library. We have done a vectorised implementation of the forward and back propagation algorithms therefore we have used the Numpy Library for initialising multidimensional weight matrix. All he biases were initialised to zero initially. Therefore, we initialised our network given the number of hidden layers(L) and number of hidden units in each layer(n) by generating a randomly initialised weight matrix and a bias matrix corresponding to each layer. We represent our ′model′ by two python lists:

1. Weights: List of numpy arrays ([W1, W2,…..])
2. Bias: List of numpy arrays([B1, B2,….])

Activation Functions

We have used two kinds of functions in our experiments:

Sigmoid Activation:

$\sigma(z) = (1/1 + e^{-z})$ $Z = w^T * a + b$

where w is the weight vector, b is the bias value, x is the input to the layer.

Tanh:

$Tanh(z) = (e^z - e^{-z})/(e^{-z} + e^z)$

These two functions were chosen because of the ease of implementation. When using the tanh activation function we used the sigmoid activation for the last forward path as tanh can also give negative values as well which would be inappropriate for output labels.

We have studied the effect of using these two functions on the accuracy of the classifier and the training and test costs.

Cost Functions

We have used the following two objective functions for our problem:

1. MSE Loss:

$$MSE = 1/n * (\sum_{i=1}^{n} \sum_{j=1}^{k} (y_{ij} - \widehat{yij})^2)$$

where n = Total number of training examples

k = Total number of classes of output layer

yij = Actual label of jth feature of ith example

$\widehat{yij}$ = Predicted label of jth feature of ith example

2. Cross entropy loss:

$$\text{Cross entropy loss} = -1/n * (\sum_{i=1}^{n} \sum_{j=1}^{k} y_{ij} log(\widehat{y_{ij}}))$$

During training , we try to minimise the objective function we have chosen by using a suitable algorithm.

Regularisation

To prevent overfitting on the training data we have used following four regularisation techniques studied their effects on the accuracy and training and test errors.

1. L1 regularisation:
2. L2 regularisation:
3. Dropout regularisation:

4. Batch normalisation:

Algorithm used:

We have used the vanilla back propagation algorithm for learning weights and biases of the model. The algorithm calculates the gradients of the cost function with respect to the weights and iteratively update the weights in the opposite of these direction to minimise the cost function. For the minimisation of objective function, we are using batch gradient descent method which takes all the input example at once and update the weights.

Hyperparameters:

In our experiments we have also studied the effects of hyper parameters such as learning rate of gradient descent and decay factor of regularisation.

3 Experiments

Normalisation of Initial training data:

In our initial naive implementation we were directly using the unscaled training feature vectors on doing this we observed that z was taking very large values which required more computational resources for running and hence all the gradients were getting saturated towards zero even in the initial epochs. So, to overcome this we normalised the feature vectors by dividing it by 256 so that each feature vector attains value between 0 and 1.

Effect of number of hidden layers:

Keeping all the parameters and hyperparameters same except for the number of layers , we found out that using 2 hidden layer architecture our model was learning and we got an accuracy of 10% as shown in figure 1 . However with same learning rate it was giving 88 % accuracy for single hidden layer architecture as shown in figure 2. So we decreased the learning rate to 1/10th for double hidden layer architecture and then observed an accuracy of 81% as shown in figure 3.

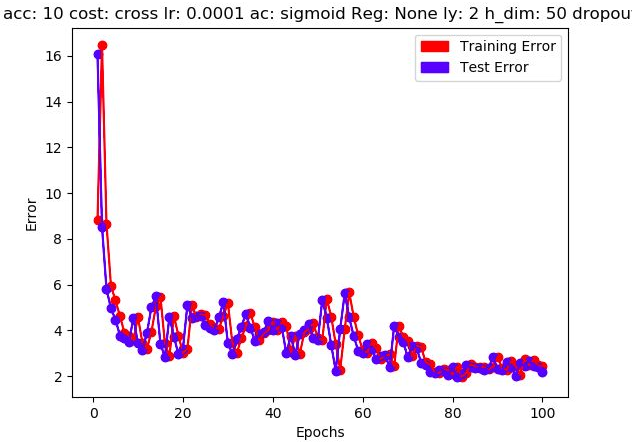

Fig1: 1 hidden layer with .0001 learning rate

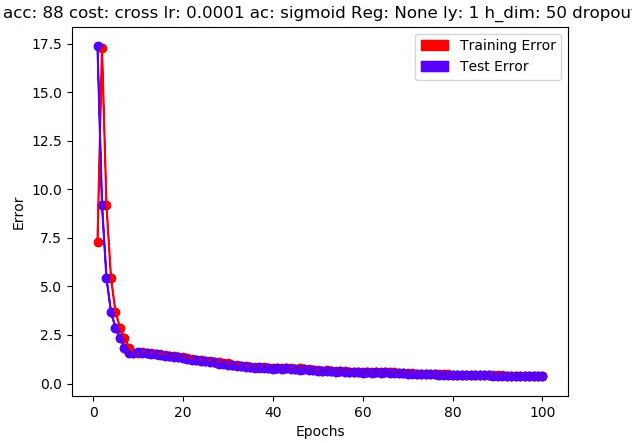

Fig2: 2 hidden layers with .0001 learning rate

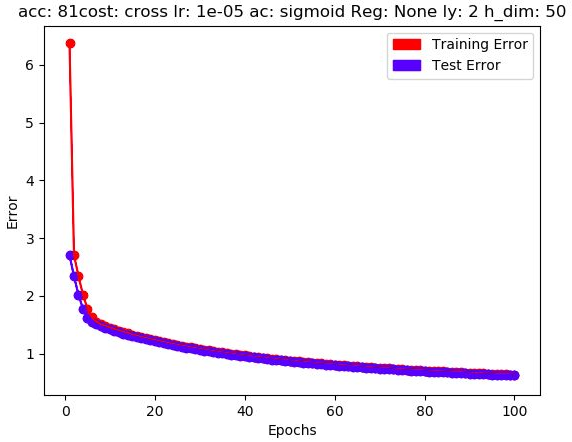

Fig3: 2 hidden layer with lower learning rate

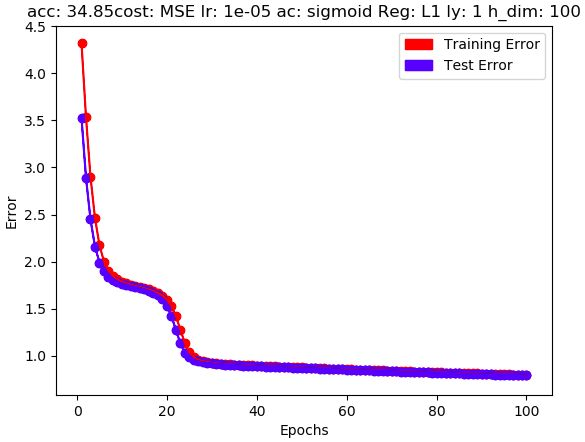

Fig 5: 1 hidden layer with 100 neurons

Effect of number of neurons in a hidden layer:

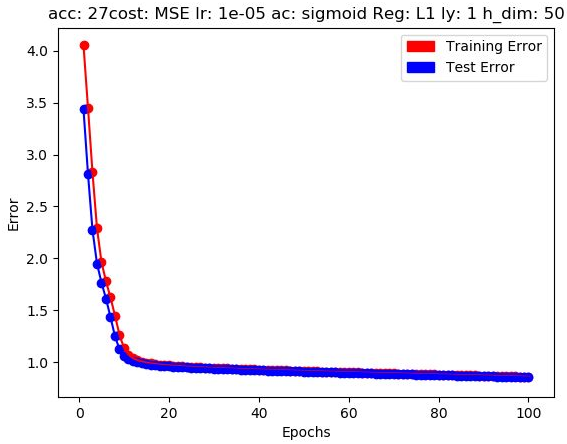

Fig 4: 1 hidden layer with 50 neurons

Various of cost function:

As mentioned above we are considering two cost function - cross entropy and MSE. We can clearly see from the graph that cross entropy is giving better result for our multi classification problem. It can be explained on the basis of convex nature of our objective function. When we are using MSE as our objective function with both activation functions i.e. hyperbolic tan and sigmoid. If we substitute either of activation function in the objective function, it will produce non convex terms while in case of cross entropy function it will produce a convex function and gradient descent method will converge to global optima in case of convex objective function while in other case it may lead to local optima. It can also explained by probabilistic interpretation. MSE gives the same result as maximum likelihood estimator of linear regression problem with normal distribution assumption of predicted output while Cross entropy gives the same result as maximum likelihood estimator of classification with bernoulli distribution assumption of predicted output. As we can see in the figure 6 and figure 7 in the case of MSE accuracy of the model is 55 % at the end of 300 epochs while in case of cross entropy accuracy of the model is 73 % at the end of 50th iterations. As shown in figure 7 in the case of MSE accuracy of model was only 34.85% till 100th iterations hence we had

to train it for more iterations to achieve comparable accuracy with cross entropy function.

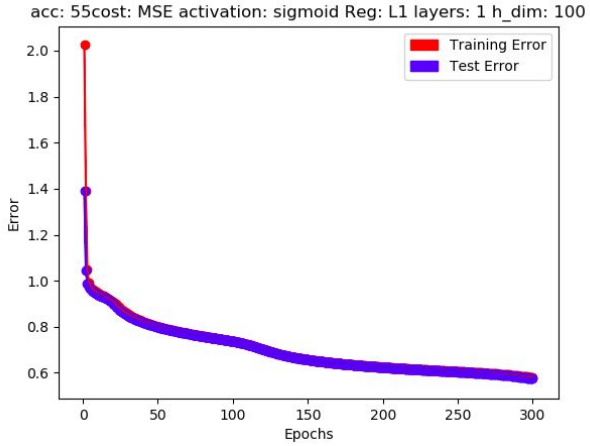

Fig6: MSE function

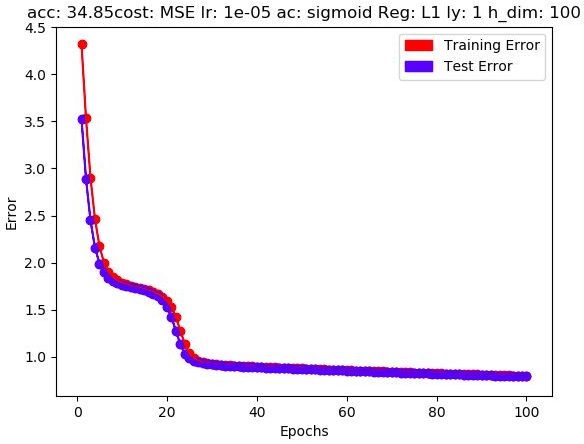

Fig7: MSE function with less number of iterations

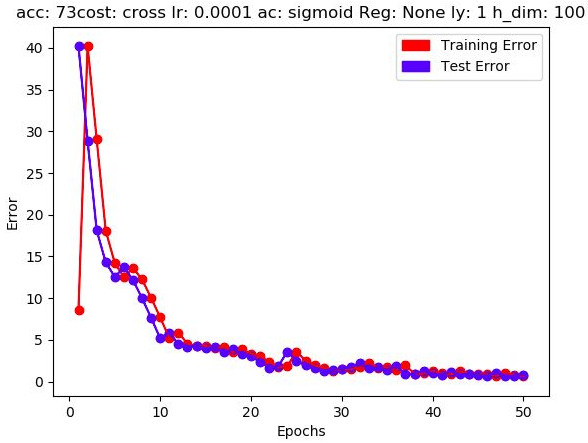

Fig8: Cross entropy function

Various of activation function:

We are considering 2 activation functions in our analysis, hyperbolic tan and sigmoid function. Since we are initially normalizing the training data such that it remains between 0 to 1 and if we closely look into the derivatives of both function in this range we observe that derivative of hyperbolic tan function is greater than the derivative of sigmoid function. Learning rate of neurons depends upon the value of derivative and hence it will be faster for hyperbolic tan function as compared to sigmoid function. As we can see from the graph that it is giving better result for hyperbolic tan function for the same parameters. In the case of tanh function , accuracy of the model is 52% as shown in figure 9 while in the case of sigmoid function accuracy of the model is 48% as shown in the figure 10.

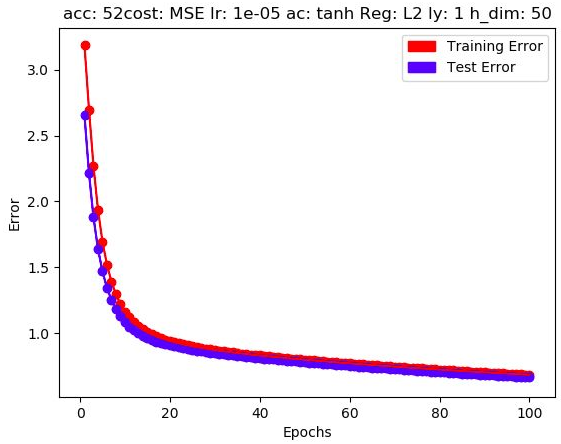

Fig9: MSE with tanh

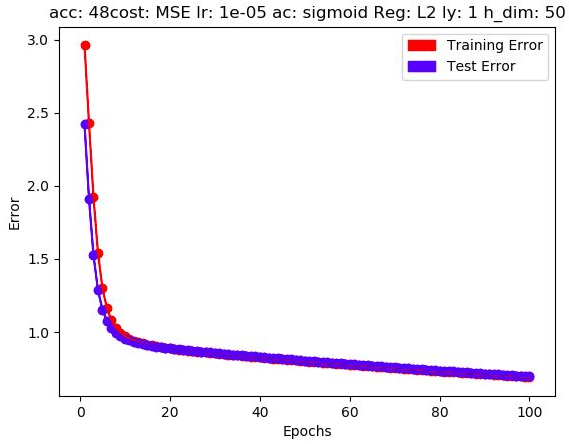

Fig10: MSE with sigmoid

Effect of parameters

1- Effect of learning rate-

(i)- NN with cross entropy cost function-

As shown in the figure 11 and 12 accuracy was drastically decreased by keeping very high learning rate. This can be explained by the concept of overshooting. In figure 11 we are crossing the minima as the learning rate is very high and the it is oscillating around minima. However as we decreased the learning rate , it increased the accuracy and it is converging toward the minima.

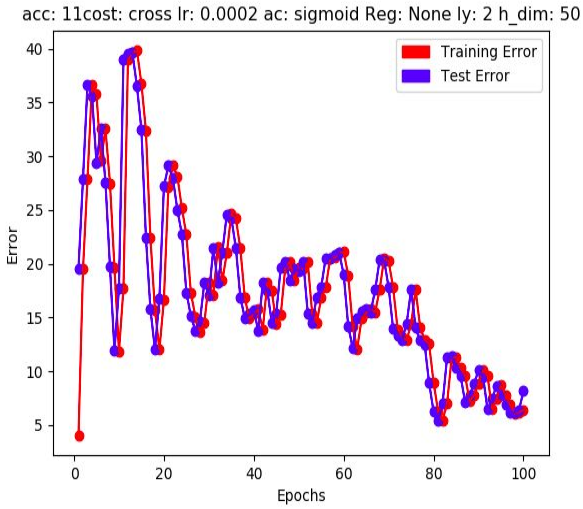

Fig11-(High learning rate)

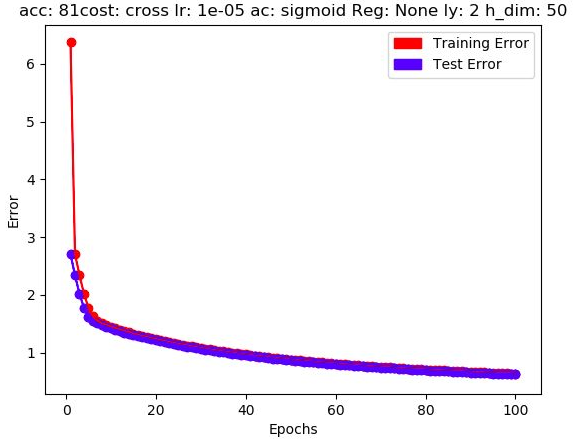

Fig12: low learning rate

(ii)- NN with MSE cost function-

In the case of MSE we are getting the same result as discussed above as accuracy is 10 % for high learning rate while in case of small learning rate it is 34%.

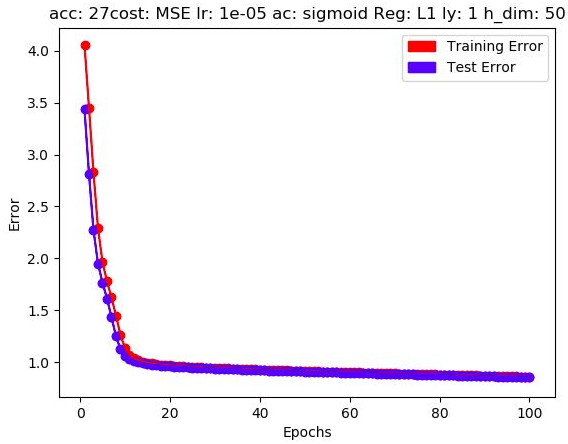

Fig13: (Low learning rate)

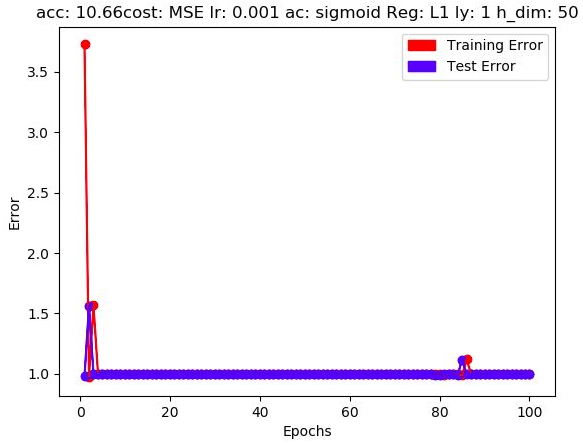

Fig14-(High learning rate)

In this experiment we are fixing all the parameters and running the same model twice . Since initial weights are obtained through randomized process hence we wanted to study the effect of initial weights on the final training and test error. In the figure shown below we can clearly see that both graphs are converging to the same errors hence it proves that initial weights would not affect the accuracy unless we change other parameters.

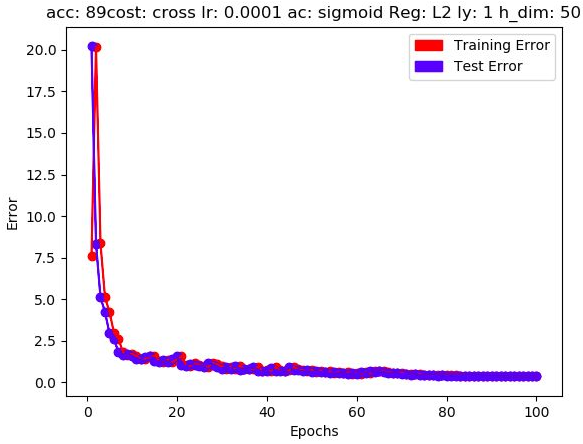

Fig15)

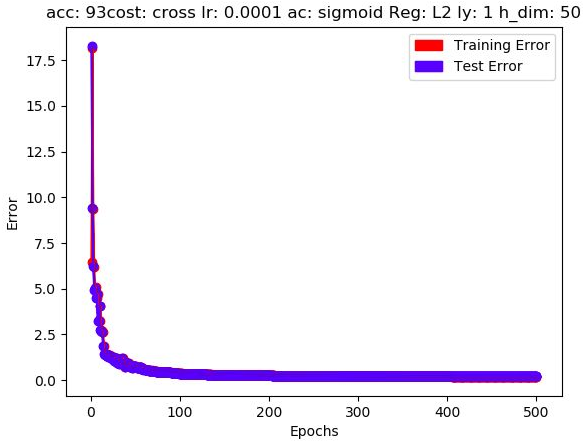

Fig: 16)

Random initialization of weights-

Comparison of different regularisation technique-

To change the scale of graph, we did not consider 20 initial iterations for plotting.

If we compare these graphs on the basis of final test and training error, we are getting minimum training and test error for the case of no regularisation technique but that is because of 500 number of iterations. While other regularisation technique are giving fair enough results at the end of 100 epochs. Accuracy for the case of dropout is a bit small as compared to other regularisation technique this is because we are getting a bit high test error as compared to others. In the case of no regularisation graph(fig 19) we closely observe the test error it seems that it is going to increase after some more epochs which indicates low bias and high variance as initially both graphs are going together and after 400 epochs test errors is differing from training error which is going downward as usual. While in the case of regularisation technique both training and test errors are going downwards.

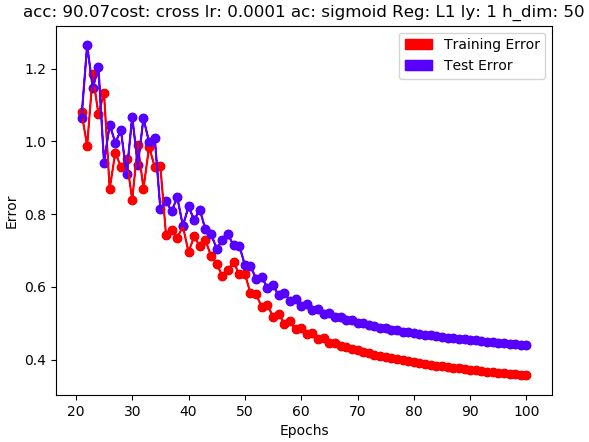

Fig17-L1

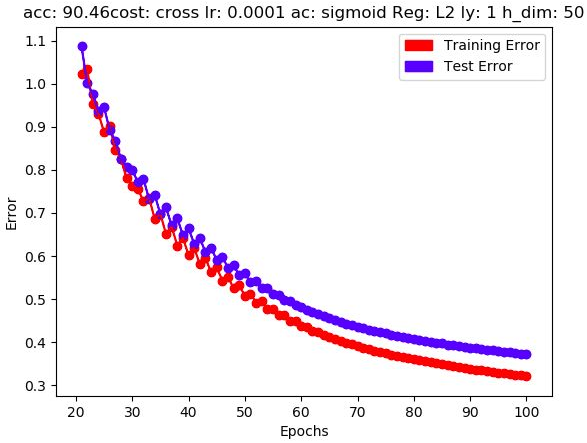

Fig18 -L2

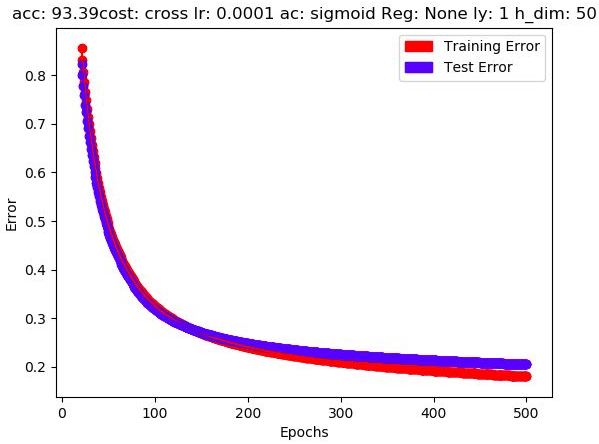

Fig19-No regularisation technique

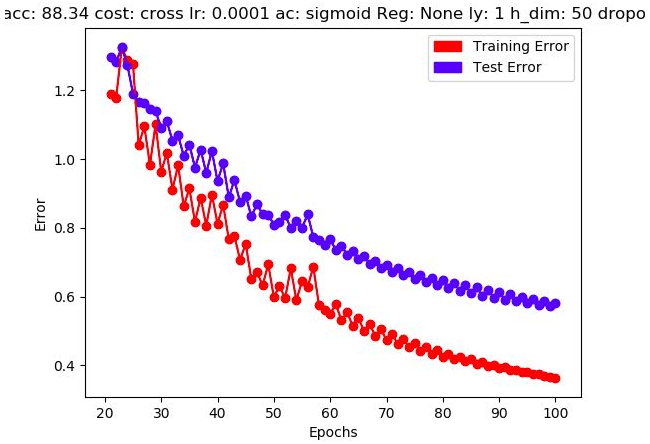

Fig20-Dropout

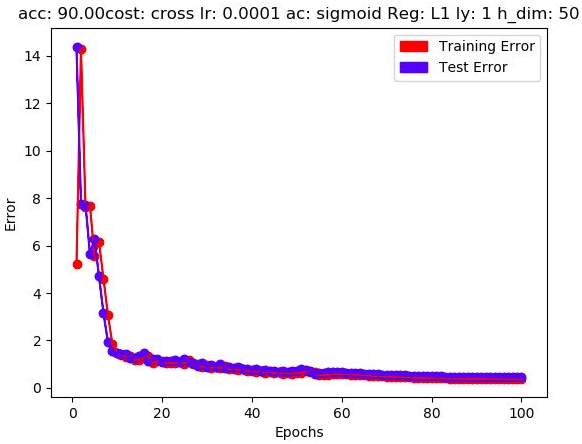

Fig22(decay factor=.01))

Decay factor of regularisation technique-

For the three different decay factors we are not getting any big differences as we are not taking too many hidden layers or no of neurons so model is not that complex .

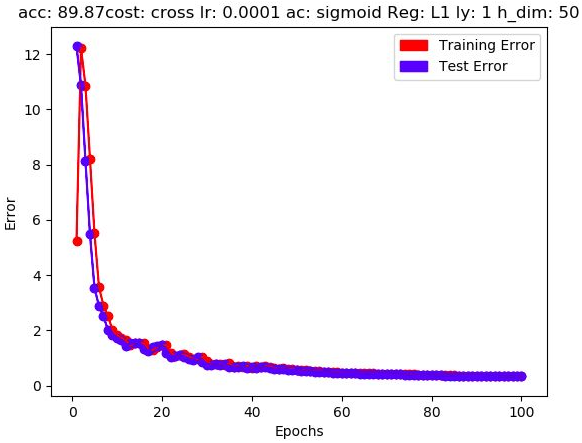

Fig21(decay factor=.0001))

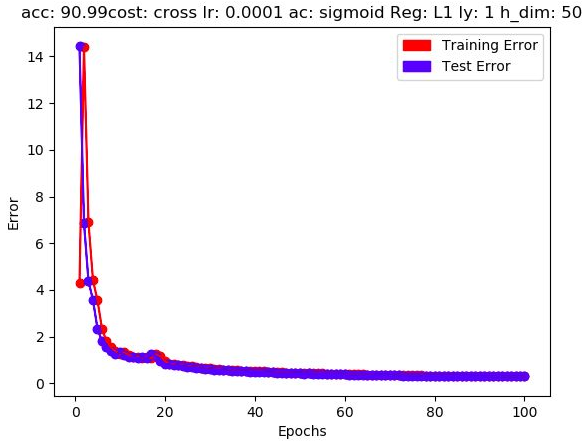

Fig21(decay factor=.001))