Multilayer Perceptron (MLP) - MNIST Dataset

Karan Kumar Vasnani

***Abstract*— In this project, we will use a Multilayer Perceptron network for classifying the MNIST dataset and will analyze and experiment with its performance considering various factors and techniques like the number of hidden layers in the Multilayer Network and units in each layer, dropout, down-sampling of images or using Principal Component Analysis (PCA). We will also use a cross validation set by partitioning our training dataset into two sets. The performance of the model on the training set will be shown as a confusion matrix. Finally, we will compare the performance of our network with the performance of Stochastic Gradient Descent (SGD) and SGD-momentum.**

Keywords: MNIST Dataset, Multilayer Perceptron, Back-propagation, PCA, Dropout, Stochastic Gradient Descent, Confusion Matrix.

I. INTRODUCTION

Handwritten digits recognition is one of the active research topics in digital image processing. Applications of digit recognition involves reading zipcodes on mails, reading luggage tags on the bags at airport, etc.

MNIST dataset is a database of handwritten digits that is commonly used for testing various image processing systems. MNIST is a dataset of 60000 handwritten digits along with their corresponding labels for training and another 10000 handwritten digits for testing. Each image has dimensions of 28x28 pixels as shown in Figure 1. Therefore, we will write each image as a vector of dimensions 784x1.

Fig. 1. A sample of handwritten digits in the MNIST dataset.

We will be using **backpropagation algorithm** to train our network consisting of one or two hidden layers. The backpropagation algorithm trains a given feed-forward multilayer neural network for a given set of input patterns with known classifications. When each entry of the sample set is presented to the network, the network examines its output response to the sample input pattern. The output response is then compared to the known and desired output and the error value is calculated. Based on the error, the connection weights are adjusted. The backpropagation algorithm is based on Widrow-Hoff delta learning rule in which the weight adjustment is done through mean square error of the output response to the sample input.[1] To check the performance improvement, well try using both the two layer and three layer network having one and two hidden layers respectively.

To improve the generalization of our model we will use **Dropout** in our algorithm. This involves randomly dropping some of the neurons in each iteration and then computing the output.[2]

Each image in the MNIST dataset set has a dimension of 28x28 pixels and therefore, each image in our dataset is represented by a 784-dimensional vector. Note that it is not necessary that all these 784 dimensions may represent important information of our image and we may be able to obtain a comparable accuracy of the model even by using a lower sized image. The size of the images can be reduced by either **down-sampling** the images to a reduced scale or using Principal Component Analysis.

To avoid overfitting of the model on the training dataset, we will partition our training data into training and validation set. Validation set is used to evaluate the performance of our network and check for overfitting.

II. PROJECT APPROACH

A. Multi Layer Perceptron

A general diagram of a 2 layer perceptron network is as shown in Figure 2.

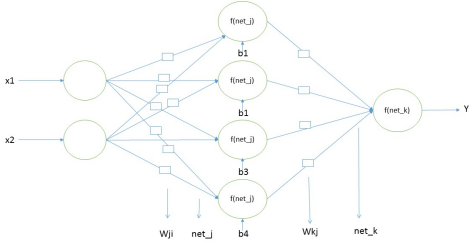

Fig. 2. A general 2 layer neural network.

The equations used for performing backpropagation on this network are as follows:

W_{ji} matrix is weight between input and the hidden layer.

W_{kj} vector is weight between hidden and output layer.

net_j is the net input to the hidden layer and is given as:

$$net\_j = \sum W_{ji} * X_i \tag{1}$$

net_k is the net input to the hidden layer and is given as:

$$net\_k = \sum W_{kj} * X_j \tag{2}$$

f(net) represents the activation function of a neuron. Here, as suggested I have used **Rectified Linear Unit (ReLU)** function as the activation function. Its equation is given as:

$$f(net) = \begin{cases} 0 & net \leq 0 \\ net & net \geq 0 \end{cases}$$

ReLU gives better results and faster convergence over sigmoid activation function.

The output, $Y = f(net\_k)$.

Weight Update equations used during backpropagation are as follows: δ_k is considered as the local error at the output layer and is given as:

$$\delta_k = (d - Y) * f(net\_k) * (1 - f(net\_k)) \tag{3}$$

The weight between the hidden and output layer is then updated using the following equation:

$$\Delta W_{kj} = \eta \delta_k X_j \tag{4}$$

$$W_{kj}(n+1) = W_{kj}(n) + \Delta W_{kj} \tag{5}$$

Here, η is the learning rate.

δ_j is considered as the local error at the hidden layer and is calculated as:

$$\delta_j = [\sum_k \delta_k W_{kj}] f(net\_j)(1 - f(net\_j)) \tag{6}$$

The weight between input and hidden layer is then updated using following equation:

$$\Delta W_{ji} = \eta \delta_j X_i \tag{7}$$

$$W_{ji}(n+1) = W_{ji}(n) + \Delta W_{ji} \tag{8}$$

B. Early Stopping and Cross Validation

Because of such huge size of the training data set, there are chances that our model may overfit to the training set and may not actually perform well on the test set. To avoid this issue of overfitting, we use a technique called Early Stopping. We divide our training dataset into two sets, a training set and a validation set. Then we train the model on the training set and evaluate its performance on the validation set to get a hint of how will the model perform on the test set. All the curves and results presented in this paper are computed using this technique.

C. Applying Backpropagation on the dataset

First, we'll experiment with a 2 layer network i.e., my network will have an input layer, an output layer and exactly one hidden layer. The number of units in the hidden layer as well as the learning rate to be used can only be decided by experimenting with different sets of values of both and plotting the learning curve.

Because of the size of the dataset, I have taken data in batches and I am performing batch training by considering a batch of 1000 at a time for calculating cost function and have taken 500 such epochs to cover the entire training dataset. This way I'm using mini-batch training procedure.

The plots for the different values of learning rate and number of hidden units are as shown in Fig 3 to 7.

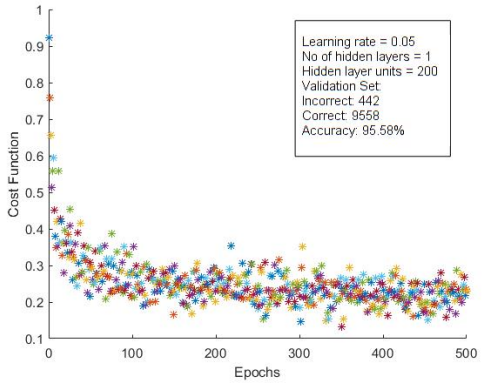

Fig. 3. Cost function.

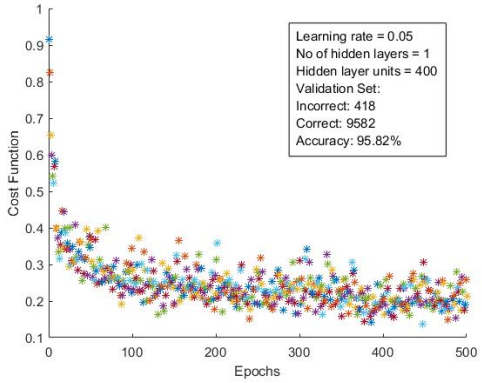

Fig. 4. Cost function

I have also mentioned the result on the validation set and the accuracy of results observed on the plots. As we see there is an improvement in the accuracy while increasing the number of units in the hidden layer from 200 to 700 while there is not any improvement in efficiency after that. So, I will use 700 units in my neural network. Also, the network performs better for a learning rate of 0.01 as compared to 0.05 as shown in the figure. This gives us an accuracy of 96.84% on the validation set which is very satisfying.

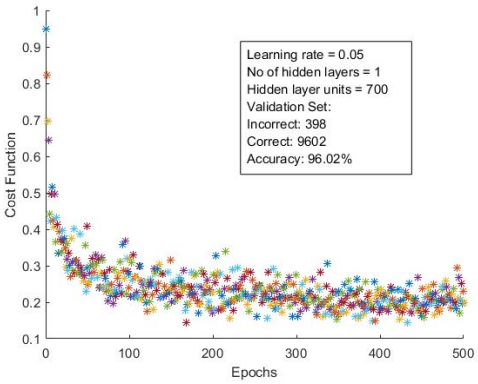

Fig. 5. Cost function

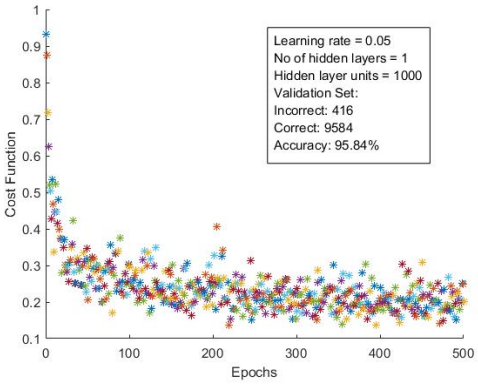

Fig. 6. Cost function

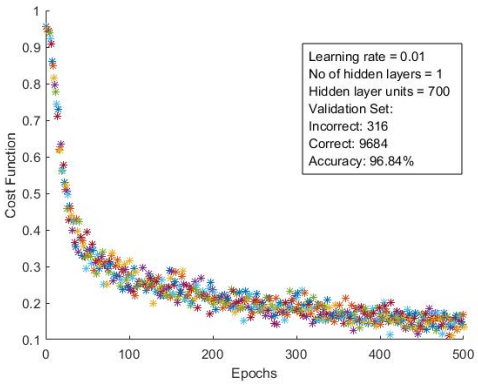

Fig. 7. Cost function

Similarly, we can use 2 hidden layers in our network and experiment with number of units in each layer as shown in fig 8 to 11.

For the case where we have 2 hidden layers, we can observe that there is not much difference in the accuracy on the validation set. Also, we can note that there is an initial delay that can be observed in the curve for cost function which means that the network with more units in the hidden layers is taking more iterations to start converging.

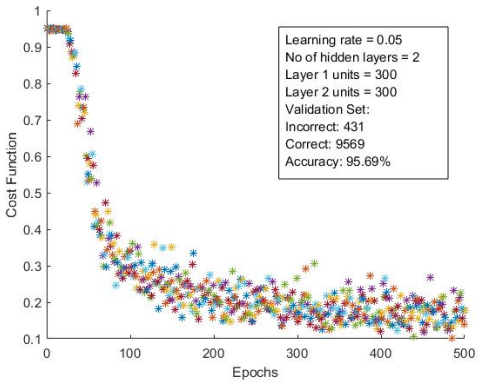

Fig. 8. Cost function.

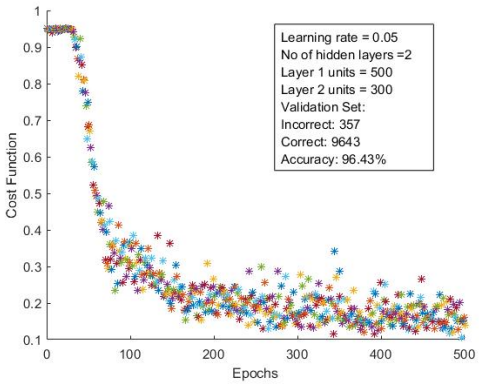

Fig. 9. Cost function

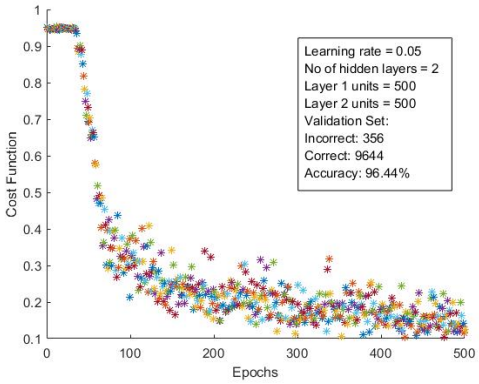

Fig. 10. Cost function

D. Using Dropout

Dropout is a technique where randomly selected neurons are ignored or dropped during training of our Multilayer Perceptron. This means that their contribution to the activation of downstream neurons is temporally removed on the forward pass and any weight updates are not applied to the neuron on the backward pass. As a neural network learns, the weights in the network settle into their context within the network. These weights are tuned for some specific features providing some specialization. Neighboring neurons become to rely on

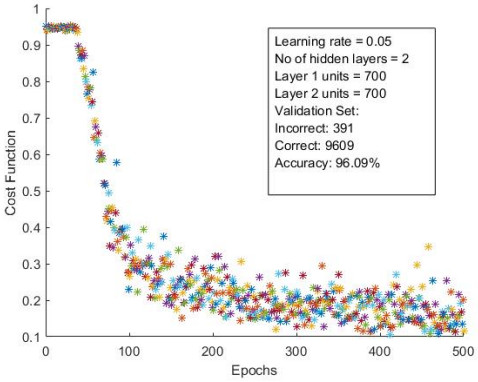

Fig. 11. Cost function

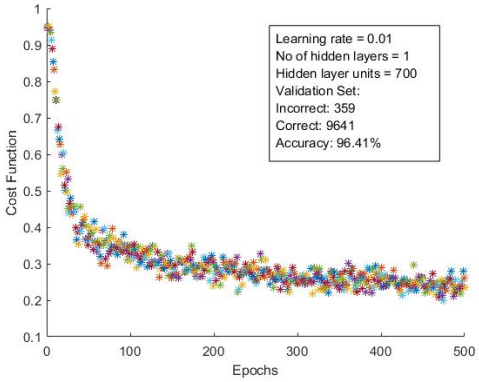

Fig. 13. Cost Function

this specialization, which if done for a large dataset and for many iterations will result the neurons to co-adapt to each other.[2]

Randomly dropping around 20% neurons in the hidden layer in each iteration may help avoiding this situation.

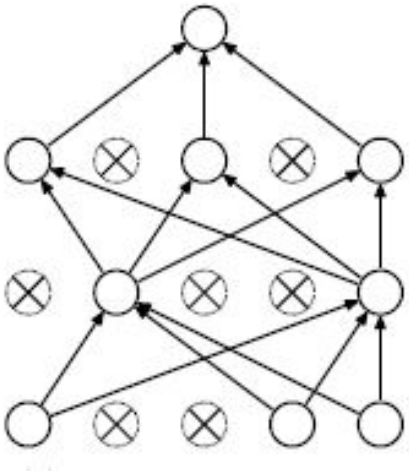

Fig. 12. Neural Network after Dropout.[2]

To apply dropout, the outputs/activations of layer 2 are multiplied elementwise with a binary mask where the probability of each element of the mask being 1 is 0.5 (zero otherwise).

$y = f(net) * mask$

During testing and validation, Weights of the output layer are multiplied by the probability or the dropout coefficient(i.e.,0.5) of the hidden layer.

The learning curve of 1 hidden layer MLP with 700 hidden units is as shown in fig 13.

From the figure 13, we can see the difference in the learning curve for the case without using dropout(Fig 7). The values of the cost function are higher in the case of using dropout. Also, the accuracy on the validation set is 96.41% which is less than the 96.84% in the case of no dropout. This observation validates the generalization concept of the network as explained above. When there was no dropout, our weights were co-adaptive and gave better results on the validation set. Without dropout the result is more generalized and avoids the co-adaptation of the weights.

2 Hidden layer network using Dropout: The dropout in case of 2 hidden layers MLP network can also be computed on similar grounds as done for the 1 layer case. The difference is that now we have to randomly drop the 20% units of both the hidden layers. So, there are two separate masks maintained for the two different layers and generated randomly for every iteration.

The activation functions of both the layers are multiplied by their respective binary masks in the feed forward pass. For testing and validation, the weights between hidden layers as well as those between output and hidden layer are to be carefully updated. The weights between hidden layers have to be carefully multiplied by the correct probability value (although I have chosen 0.5 probability for both the masks). The learning curve along with the performance on validation set are shown in Fig 14.

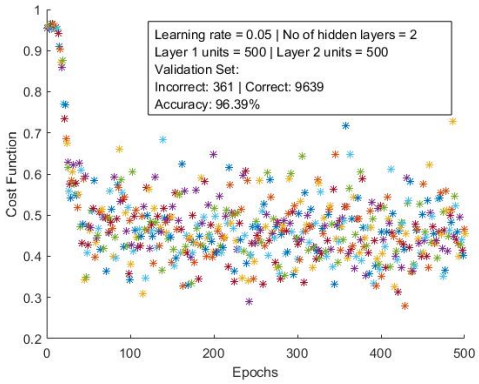

Fig. 14. Cost Function

Again, from the Figure 14, the learning curve seems to be inline with our previous explanation regarding co-adaptation of the neurons. The cost function values are more than the case without dropout (Figure 10). Also, the accuracy is less on the validation set.

E. Down-sampling and Principal Component Analysis

Each image in the MNIST dataset set has a dimension of 28x28 pixels and therefore, each image in our dataset is represented by a 784-dimensional vector. Note that it is not necessary that all these 784 dimensions may represent important information of our image and we may be able to obtain a comparable accuracy of the model even by using a lower sized image.

Down-sampling: The size of the images can be reduced by **Down-Sampling** the images to a reduced scale. This way our image will be down sized to much lower dimensions. I have used the 'imresize' function available in MATLAB for image resizing to reduce the number of dimensions of the input. The plots of the learning curve and the accuracy on the validation set for different values of down-sampling are shown in the figures 15-17.

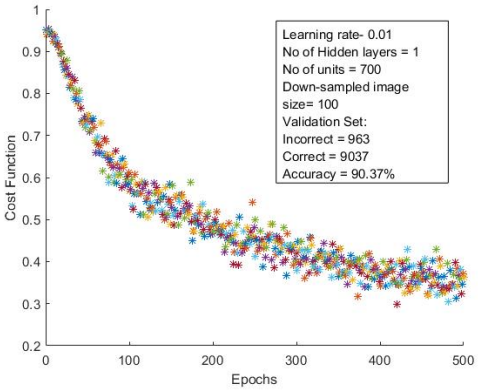

Fig. 15. Cost Function

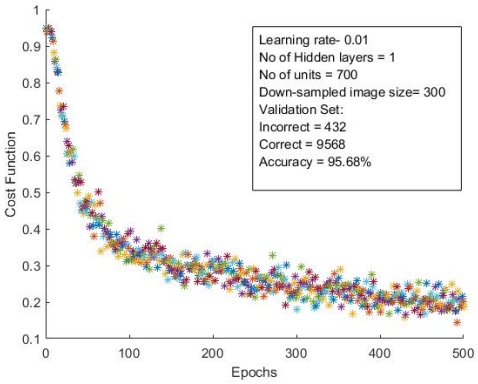

Fig. 16. Cost Function

From the figures, we see that the curve for 100 dimensional image input doesn't give as good performance on the learning curve and the accuracy on validation set. The accuracy comes out to be just 90.37%. On the other hand, 300 and 500 dimensional input image perform much better giving an accuracy of 95.68% and 96.49%. Therefore, we can just use down-sampling function 'imresize' and reduce the size of each image to be a 500 dimensional vector to get

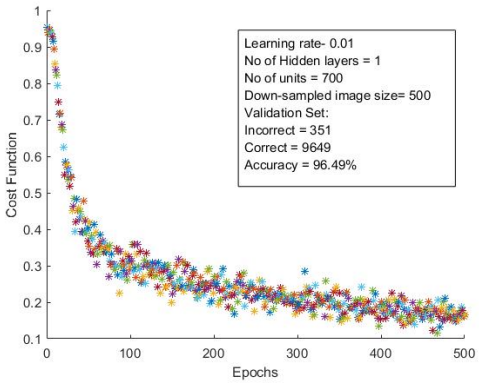

Fig. 17. Cost Function

equally good performance as the original 784 dimensional images.

Principal Component Analysis: An alternative way of down-sizing an image is by using Principal Component Analysis (PCA). The key idea of using PCA is to transform the handwritten digit images into a small set of characteristics feature images, called eigenvectors, which are the principal components of the initial training set of the digit images. All the images in our dataset will then be projected onto these eigen vectors and the projected images will now be fed to our Multilayer Perceptron for training. The test set of images will also be projected on the eigenvectors obtained from training images. The projected images will now be classified by the network.[3]

The first principal component obtained represents the direction of maximum variance for the data. The other principal components represent the directions of subsequent levels of variances. The learning curve using different number of eigen vectors to be considered are as shown in figure 18-21.

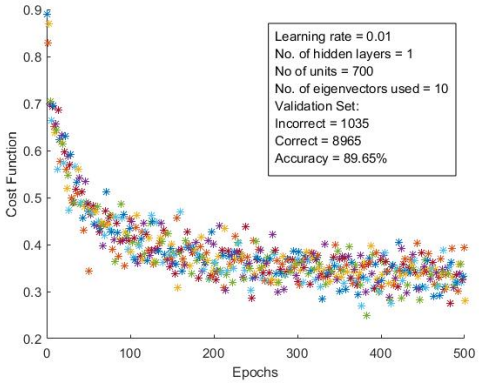

Fig. 18. Cost Function

From the figures, when the number of Eigen vectors used is 10, the performance isn't very good and the accuracy on the validation set is just 89.65%. When we increase the number of Eigen vectors to be 30, there is a significant increase in the performance on the learning curve and the accuracy, giving 96.18% accuracy. After that, increasing the

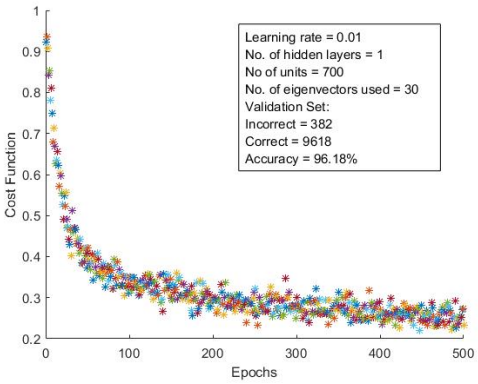

Fig. 19. Cost Function

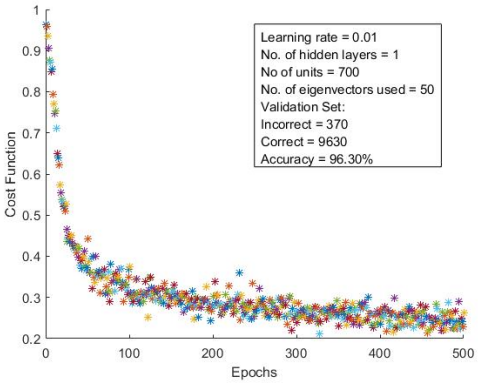

Fig. 20. Cost Function

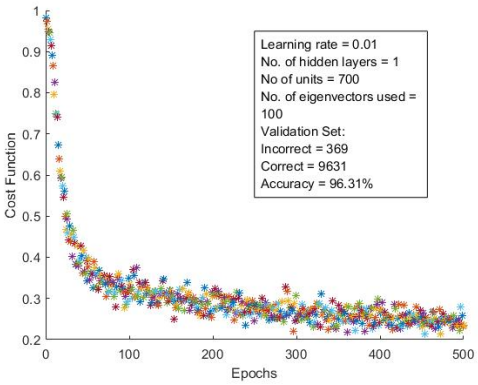

Fig. 21. Cost Function

number of eigen vectors to 50 doesn't give a significant rise in the performance and accuracy and the accuracy remains almost equal when 100 eigen vectors are used. Therefore, we may use not more than 50 eigen vectors to represent our training images and still get as good performance as the case of 784 inputs dimensions.

Hence, using PCA on the training set will significantly reduce the dimensionality of input making computation faster and simpler.

F. Confusion Matrix

A confusion matrix is a table used to describe the performance of a classifier on a test data for which the true values are already known. The confusion matrix for MNIST dataset would represent the number of times a digit is classified as any other digit by the classifier model.

The confusion matrix for the classifier with following characteristics is as shown in the figure. Learning rate = 0.01, Number of hidden layers = 1, Hidden layer units = 700

				Classified Output						
	972	0	2	2	0	0	1	1	2	0
	0	1121	4	2	0	0	2	0	6	0
	6	1	1004	1	3	0	2	5	10	0
	1	0	12	971	0	4	0	7	13	2
	1	0	6	0	936	0	3	1	2	33
True Labels	7	1	1	14	0	849	7	1	10	2
	9	3	0	1	4	4	931	0	6	0
	2	8	21	1	1	0	1	984	1	9
	2	2	3	6	4	0	5	3	948	1
	4	5	1	9	10	3	2	6	15	954

Fig. 22. Confusion Matrix

In a confusion matrix, the each row represents a digit in the actual label dataset while each column for that digit represents the number of times that true digit image has been classified as all the other digits. Ideally, all the values in the confusion matrix should be diagonal, therefore, all the off-diagonal entries represent classification error.

For my classifier model, the confusion matrix has maximum values on the diagonal as expected, only few values are off-diagonal. The off-diagonal values also make some sense as these images are actual human handwritten images and we all have different handwriting, so it is sometimes difficult to classify what someone has actually written, therefore, in those cases the classifier makes a guess to the closest appearing image.

For example, the true label 3 is sometimes classified a 8 which is possible as sometimes people do write 3 to be similar to 8. Similarly, 4 is classified to be 9 for exactly 33 times. This result is logical as we do sometimes write 4 to be similar to 9. Similar, is the case with 7 being classified as 2, 9 being classified as 4 and 8, and so on. These observations validate our model for classification. Overall, a total of 9670 digits have been classified correctly and 330 are incorrect giving an accuracy of 96.70%.

G. Learning Curve and Accuracy for Training and Validation

The learning curves for both training and the validation set are as shown in figure 23-24.

The figure 23 shows the learning curve on training set and it is as expected decreasing to a lower cost function as we train our network. On the other hand, the during validation (figure 24) the network is already trained and should give a

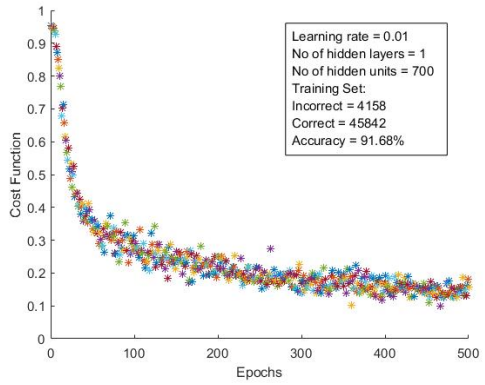

Fig. 23. Cost Function

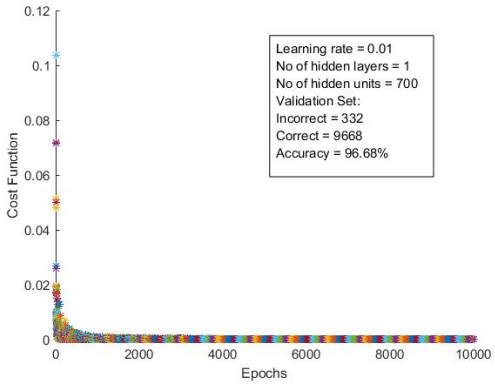

Fig. 24. Cost Function

low error at the output and consequently a low cost function as compared to the training set. The learning curve represents the Mean Square Error calculated at the output which is already very low.

The accuracy obtained for both cases is as follows:

$TrainingSet$:
$CorrectlyClassified = 45842$
$IncorrectlyClassified = 4158$
$Accuracy = (45842/50000) * 100 = 91.68\%$

$ValidationSet$:
$CorrectlyClassified = 332$
$IncorrectlyClassified = 9668$
$Accuracy = (9668/10000) * 100 = 96.68\%$

The accuracy of classification of validation set is obviously more than the accuracy of training set. Also, the accuracy of training set would be very low for the initial iterations and will improve with the iterations.

H. Stochastic Gradient Descent

The Multilayer perceptron we used, can either be trained in a batch or online mode. The term **Stochastic Gradient Descent (SGD)** is used to represent the case when online training is performed. In Gradient Descent, you have to run through all the data samples in your training set and only then a single update is made in the parameter of our network. Therefore, one epoch consists of all the data samples. To evaluate the performance, we can even divide the iterations in batches, so we may take a fix number of images in a batch to iterate and then update the weight several times in an epoch.

On the other hand, you in Stochastic Gradient Descent, only one training sample is used at once to compute output and update the weight parameters based on the error at the output. Therefore, each iteration of weight update involves training on exactly one image from the training dataset.

Therefore, if for our training set we update our weights we get the following learning curve as shown in figure 23.

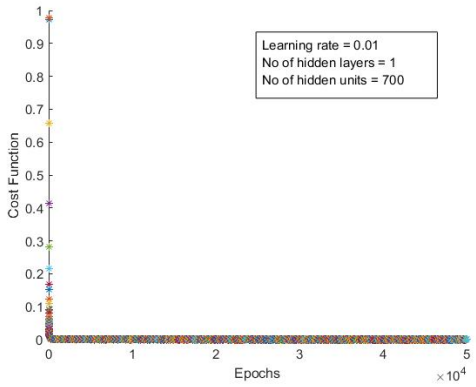

Fig. 25. Cost Function

From the figure, we note that the learning curve converges much faster than the case of batch procedure we discussed before. This is due to the following reasoning:

Batch gradient descent has to scan through the entire training set before taking a single step- a costly operation if m is large. Stochastic gradient descent on the other hand, can start making progress right away, and continues to make progress with each example it looks at. Often, stochastic gradient descent gets cost close to the minimum much faster than batch gradient descent. It may however, never converge to the minimum, and the parameters of the network will keep oscillating around the minimum cost; but in practice most of the values near the minimum will be reasonably good approximations to the true minimum. Therefore, for these reasons, particularly when the training set is large, stochastic gradient descent is ofter preferred over batch gradient descent.

I. Momentum

If the cost function has form of a long shallow ravine leading to the optimum and steep walls on the sides, standard SGD will tend to oscillate across the narrow ravine since the negative gradient will point down one of the steep sides rather than along the ravine towards the optimum. The objectives of deep architectures have this form near local optima and thus standard SGD can lead to very slow convergence particularly after the initial steep gains. Momentum is one

method for pushing the objective more quickly along the shallow ravine.[5]

III. CONCLUSIONS

The objective of the project has been achieved by implementing a Multilayer perceptron network to classify MNIST dataset of handwritten digits. The factors and techniques that have been analyzed and experimented are: number of hidden layers, number of hidden units in each layer, dropout, downsampling images and using PCA, early stopping and using Stochastic Gradient Descent along with momentum.

REFERENCES

[1] Velasquez, Guillermo, "A Distributed Approach to a Neural Network Simulation Program". Master's thesis, The University of Texas at El Paso, El Paso, TX, 1998

[2] G. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. Journal of Machine Learning Research 15 (2014) 1929-1958.

[3] S. Thakur, J. K. Sing, D. K. Basu, M. Nasipuri, M. Kundu, Face Recognition using Principal Component Analysis and RBF Neural Networks.

[4] Andrew Ng, CSE229 Lecture Notes, unpublished.

[5] UFLDL Tutorial. "Optimization: Stochastic Gradient Descent". http://ufldl.stanford.edu/tutorial/supervised/OptimizationStochasticGradientDescent/