Handwritten Character Recognition using Multilayer Perceptron

Suyog Daga, *Graduate Student, University of Florida*

***Abstract*- Neural Networks are computational systems which are represented as simple highly interconnected elements, that are able to capture and represent various complex input/output relationships. It consists of a large number of processing elements operating in parallel and arranged in different layers. The first layer receives the input data, each other layer receives data from the layer preceding it and the last layer produces output for the system. Neural networks are known to be adaptive in nature, in which they modify themselves as they learn from initial training and subsequent runs. They can be used to classify different visual input data obtained from documents. In this paper, I implement a multilayer perceptron / neural network to learn handwritten characters provided in MNIST dataset. We see how a simple and generally implemented architecture of multilayer perceptron can yield good results for recognizing a handwritten character.**

***Index Terms*— Multilayer perceptron(MLP), Neural Networks, MNIST dataset, Character Recognition, Back Propagation.**

I. Introduction

Handwriting recognition is the ability of a computer to understand handwritten text from different sources like paper, photographs, word documents etc. It is of great commercial interest. In daily life examples, we can see post offices using this technique to sort mails, a bank using it to read personal checks. Overall process includes a series of steps, which include preprocessing the data, then identifying useful features relevant to the problem, and then using the data for classification purpose. Each step has its own importance. Preprocessing removes irrelevant information from the given data. We can remove unnecessary outliers from the data which are not relevant to the problem, removing such unwanted elements in initial step can help us getting good results. Moreover, speed and accuracy can be increased. Preprocessing generally involves normalization of data or sampling and smoothing of data. Second important step is feature extraction from input data. We should have a sound intuition about our dataset, and features that might help to solve our problem. Generally, data has high dimensionality where not each of the features is relevant. Selecting most relevant features is of core importance, different dimensionality reduction techniques can be used to extract important components from high dimensional data. The overall purpose of feature extraction is to determine important features of our model. The last step involves classification of data, here different techniques are used to map important features into different classes and thus identifying and classifying different handwritten characters.

We use multilayer perceptron's to learn from a training set handwritten characters (MNIST data). We learn from the data and use it to check our accuracy on test data set. Generally, network learns distinct and unique properties that help to differentiate training data. When an observation is passed to determine its classification, networks check similar properties as relevant with the training data. Neural networks have its set of pros and cons. They are definitely easy to setup, but they can be incorrect and inaccurate if they learn features that are not relevant to the problem. Also, they are easy to conceptualize and different implementations and library support are available for them, but there are different alternatives (support vector machines, trees, regression) which can be faster, easier and provide more accurate results.

There has been significant research in this field. Researchers have used back-propagation to learn handwritten zip code provided by U.S. Postal Service [1]. Also researched have looked into different aspects of back propagation, like training big MLP can be hard as back-propagated gradients quickly vanish exponentially with a large number of layers [2]. Overall there were different constraints related to back propagation, hyperparameters, the number of hidden layers etc. but the objective was the same, to minimize classification error.

II. DATA

The MNIST dataset is developed by three researchers Yann LeCun, Corrina Cortes and Christopher Burges for evaluating various different machine learning models for classifying handwritten digits. Various different images of digits were collected from scanning documents, then normalized with respect to size followed by centering. So most of the work related to preprocessing of data has been done on this dataset allowing researchers to focus on machine learning paradigms. Each image in this dataset is represented as a 28 by 28 pixel square or 784 pixels. In general, the dataset is split into sets, training set which consists of 60000 images and testing set which consists of 10000 images. Furthermore, the training set can be again split in training and validation sets as per requirement. This data consists of handwritten digits, that are

basically number 0 to 9 or we can say 10 different classes to predict. When checking for results we check how good our model learns and predicts the digits, so overall we check the prediction error.

III. NEURAL NETWORKS

1. Perceptron:

Perceptrons are single layer feed-forward networks based on threshold transfer functions [3]. They are the simplest type of neural networks and can classify linearly separable cases with a binary target (0,1).

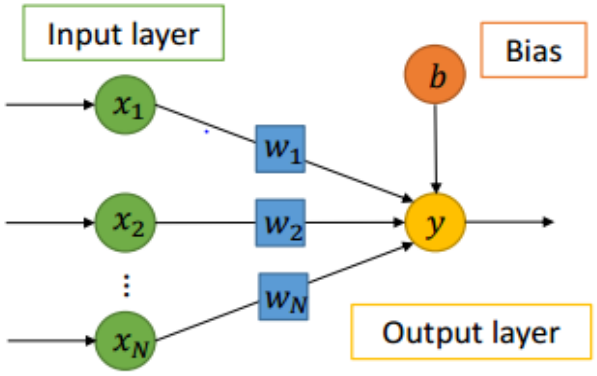

Fig 3.1 Simple Perceptron

Output y is given by:

$$\text{output} = \begin{cases} 0 & \text{if } \sum_j w_j x_j \leq \text{ threshold} \\ 1 & \text{if } \sum_j w_j x_j > \text{ threshold} \end{cases}$$

More specifically for a perceptron,

$$\text{output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases}$$

Here is the working algorithm for Perceptron, which doesn't have any prior knowledge [3]:

a) Initial weights, w, are assigned randomly.
b) At the output, perceptron sums all the weighted inputs and the bias, if the sum is an above-predefined threshold, perceptron is said to activated(y=1).
c) If predicted output matches the desired output, no changes in weight are made, else weights are updated to reduce the error.

Weight adjustments to perceptron is done using following formula,

$$\Delta w = \eta * d * x$$

where, d is the predicted output, η is learning rate and x is the input data.

2. Multilayer Perceptron:

A multi- layer perceptron (MLP) has a similar structure as that of single layer perceptron having one or more hidden layers [3].

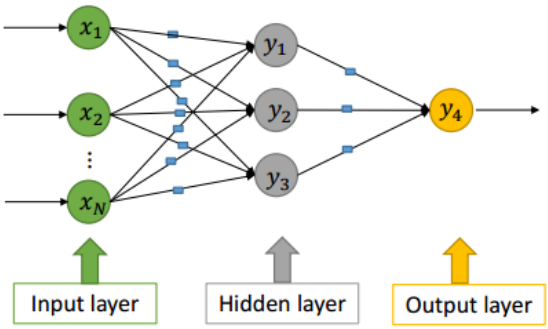

Fig 3.2 Multilayer Perceptron

Its working is similar to that of the perceptron, input layer passes input to first hidden layer neurons, which then calculate their output based on activation function, the output from this layer is then passed ahead to next hidden layer neurons. Finally, we get output at the output neuron. Depending upon the type of problem, a number of hidden layers and the corresponding number of neurons will change.

3. Activation functions

While calculating weighted sum at a neuron, we can use different activation functions. An example could be linear activation functions or non-linear activation functions.
The sigmoid function is often utilized for calculating the weighted sum and it is represented as [3]:

$$f(x) = \frac{1}{1 + e^{-x}}$$

Another example is rectifier function which is given by:

$$f(x) = \max(0, x)$$

4. Back Propagation

The system uses input data to produce its own output and then compares the desired output or target output. If there is no difference, no learning takes place. Otherwise, we need to adjust weights, to reduce the error. We can automate the procedure and can provide our model a feedback. This feedback is defined in terms of the cost function or error criterion which needs to be minimized. For each training pattern, we can define an error (ε_k) between the desired response (d_k) and the actual output (y_k). Here is the equation for mean squared error(MSE) [3]:

$$MSE = J(w) = \frac{1}{2N} \sum_k (d_k - y_k)^2$$

When the error is zero, machine output is equal to desired response. This overall process of passing feedback of training to the model and adjusting weight till we get a minimum error is known as backpropagation. The minimum of MSE cost function is found by following the opposite direction of performance surface gradient at each point. Also for calculating errors at hidden layers, we do following traversals in backward direction, we first calculate the weight change rule for hidden layer from the output weight, then if there are more than one hidden layer, corresponding weight change rule is calculated in backward direction and finally we have weight change rule for an input to the hidden weight.

If we follow the below notation for a neural network,

a) w_{kj} denotes a weight from the hidden to the output layer.
b) w_{ji} denotes a weight from the input to the hidden layer.
c) y_k denotes the output value.
d) net_j denotes the net input at layer j.
e) d_k denotes the target value.

We have following back propagation algorithm steps [3]:

a) Forward Propagation

Initializing weights at random and choosing a learning rate η , we need to forward pass in the network, with fixed weights to produce output(s), in forward direction layer by layer.

For hidden layer we have following equation:

$$net_j = f\left(\sum_i w_{ji} f(net_i) + b_j\right)$$

For output layer we have following equation:

$$y_k = f\left(\sum_j w_{kj}\ f\left(\sum_i w_{ji} f(net_i) + b_j\right) + b_k\right)$$

b) Backpropagation of errors

Having calculated the outputs, one can now evaluate the error from desired response and back propagate the local gradient-based error at each layer. We have following two equations:

$$\Delta w_{kj} = \eta\, \delta_k net_j$$
$$\Delta w_{ji} = \eta\, \delta_j net_i$$

Where $\delta_k = ((d_k - y_k) f(net_k)(1 - f(net_k))$

$\delta_j = [\sum_k \delta_k w_{kj}]\, f(net_j)(1 - f(net_j))$

c) Update weights and observe convergence

We update the weights and repeat step 2 and 3 until the MSE is minimized or convergence is observed. Weight updating is done as follows:

$$w_{kj} = w_{kj} + \Delta w_{kj}$$
$$w_{ji} = w_{ji} + \Delta w_{ji}$$

5. Gradient Descent Methods

Stochastic Gradient Descent and Stochastic Gradient Descent with momentum are optimization algorithms; in which we optimize a set of parameters in iterations to minimize a given error function.

In gradient descent, we can go through all the samples in training set for updating a single parameter in a particular iteration, but in stochastic gradient descent, we use one training sample from training data to update certain parameter in a particular iteration.

Therefore, in general, when training data is very large, we don't prefer gradient descent, as for every iteration where we update a parameter we run through all the dataset. On another hand, the stochastic gradient descent will be faster as compared to gradient descent as it uses only single training sample for updating parameter. Stochastic gradient descent with momentum often converges faster owing to the above fact, but the error function is not well minimized as compared to gradient descent. The closest approximation of values obtained in stochastic gradient descent for parameters is enough as they are near the optimal values for the error function.

6. Dropout in Neural networks

Neural Networks with very large number of hidden layers and parameters is a powerful technique. But as a number of hidden layer's increase, there is more chance of overfitting. Also as the size of network increases, calculation overhead increases, and neural networks tends to become slow, as it is extremely hard to handle overfitting by combining predictions of large networks. Dropping is used to solve this problem. The main idea in dropping is to randomly drop some units, with their connections from the network while training [4]. While training, the dropped samples from an exponential number of "thinned" networks. During testing, it is easy to approximate the effect of averaging this thinned networks by using a single unthinned network with small weights as a substitute. This helps in reducing the overfitting insignificant amount.

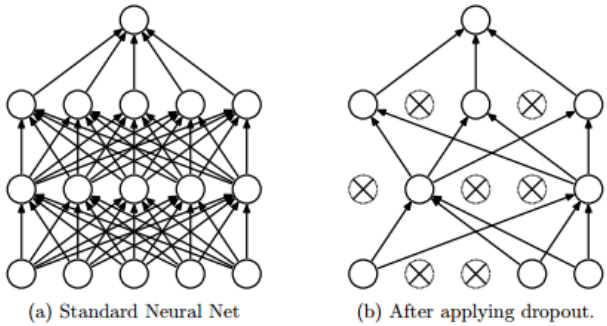

Fig 3.3 Dropping in the neural network.

7. Downsampling

Downsampling or "pooling" layers is a technique that is often used in neural networks, mainly to reduce the dimensionality of features for optimizing computational speed, which can, in turn, improve the overall performance of the network [5]. The main type of pooling is "Max pooling", where the features are

reduced in such a way that, maximum feature response within given data sample is preserved. There are two important features associated with the technique, which make them so prevalent today, it helps in achieving better computational speed and it ensures translational equivariance (output is tolerant to small translational changes in input data).

IV. SIMULATION AND RESULTS

- We have 60000 training images and 10000 testing images. Training images have been further split into 50000 images for training and 10000 images for validation data for early stopping measures
- There is no specific method as to determine the number of input layers and associated neurons with it so using random trial method, I initially start with 784-10-10 neural network, here 784 is the number of neurons, 10 is the number of neurons in single hidden layer, and last 10 is the output neurons associated with each number (0-9).
- Using Stochastic Gradient Descent Algorithm with early stopping, learning rate 0.1, and a number of epochs as 150, running the above configuration of 784-10-10 network I get an accuracy of 87.23% on testing data. But are the number of neurons in hidden layer sufficient enough for learning? Or can I increase the accuracy of a prediction on training data? For this I ran, stochastic gradient descent algorithm with early stopping, variating the number of hidden neurons from 10 to 650,

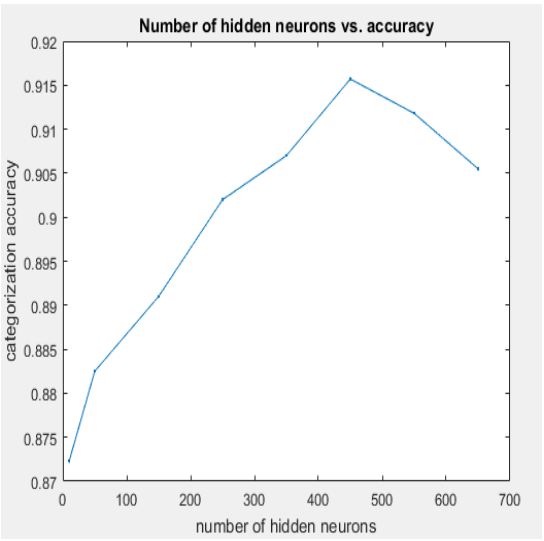

Fig 4.1 Hidden neurons vs accuracy.

- From the above diagram, it is clear that maximum accuracy on test data occurs on approximately 450 neurons in a hidden layer. Here is a comparison of accuracy for different layers:

Hidden Layer Neurons	Accuracy on test data
10	87.23 %
250	91.57 %
450	92.37 %

- From the above table, in general when there are increased the number of neurons, accuracy of prediction increases, not strictly, though.
- Here is the ROC plot for the stochastic gradient descent algorithm with 450 hidden layer neurons, there is a good classification as most of the curves related to different classes lie to the uppermost left corner.

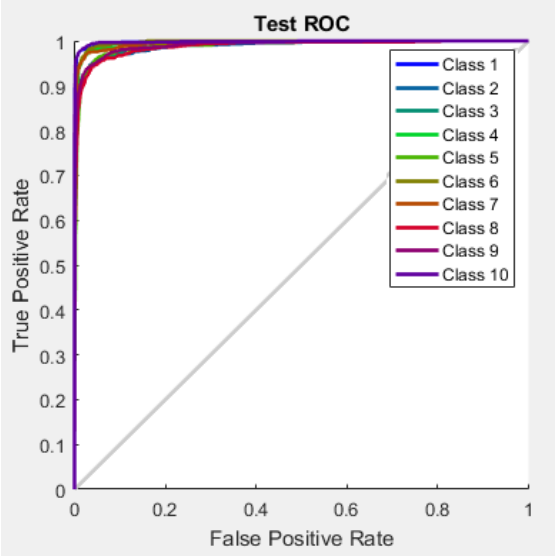

Fig 4.2 ROC curve for test data (SGD)

- Now, we perform similar analysis but will use stochastic gradient descent algorithm with momentum. For comparison with stochastic gradient descent algorithm, I will use similar values of hidden layer neurons, following are the result of analysis. Here is the plot related to varying hidden number of units from 10 to 650,

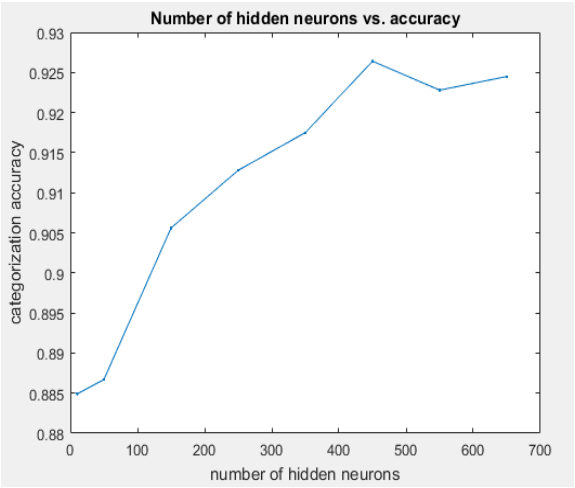

Fig 4.3 Hidden neurons vs accuracy(SGD-momentum).

Hidden Layer Neurons	Accuracy on test data
10	88.49 %
250	91.28 %
450	92.64 %

We see the results are not so different as compared to that of stochastic gradient descent method. Stochastic gradient descent usually achieves a minimum but can take longer time as compared with momentum, if we care for fast convergence for a complex network we can go with stochastic gradient descent method with momentum.

- We now look into the confusion matrix related with both the methods so far dealt, as an overall idea how good was our classification of numbers to respective classes.

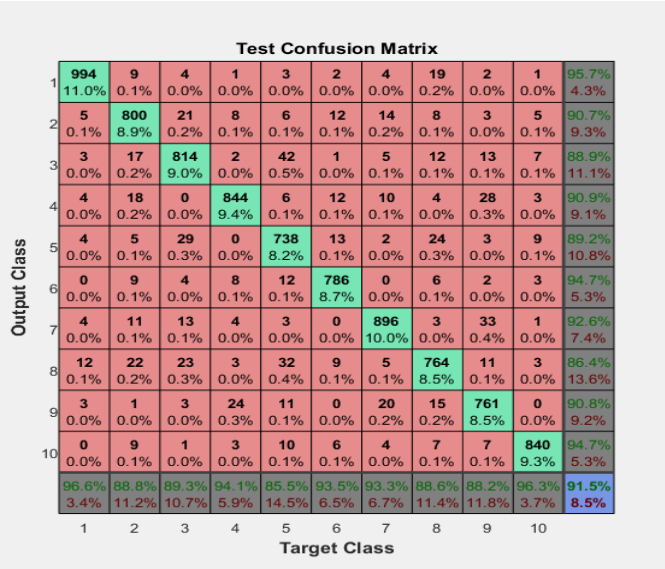

Fig 4.4 Confusion matrix for SGD

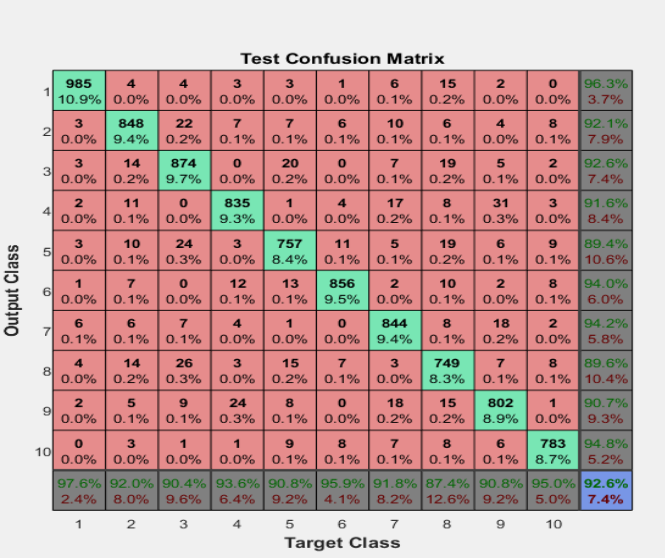

Fig 4.5 Confusion matrix for SGD-momentum

- For both the methods, most of the elements were classified correctly. For SGD method, there was some conflict between output classes 3 and 8, where corresponding target class was given as 5. In SGD-momentum method, we see output class 8 conflicts with target class 3, intuitively due to similar kind of shape of 3 and 8.
- We also look at the performance curve for both the methods, we can see that training curve for SGD-momentum curve has a lower value of MSE than that of SGD algorithm. Also, testing and validation tend to have similar values of MSE approximately.

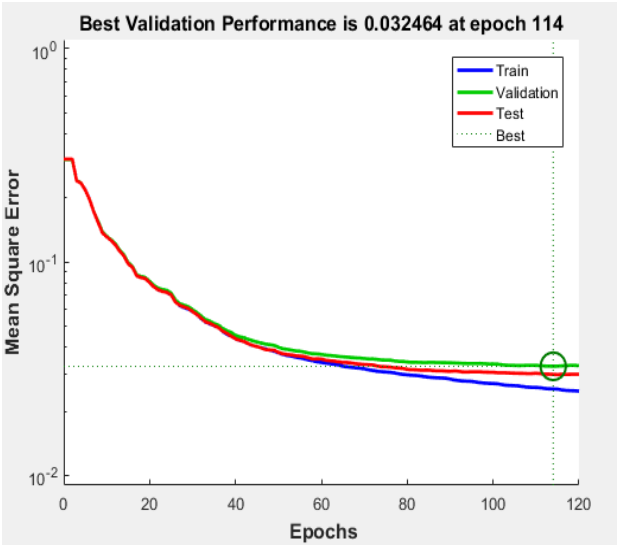

Fig 4.6 Performance curve for SGD

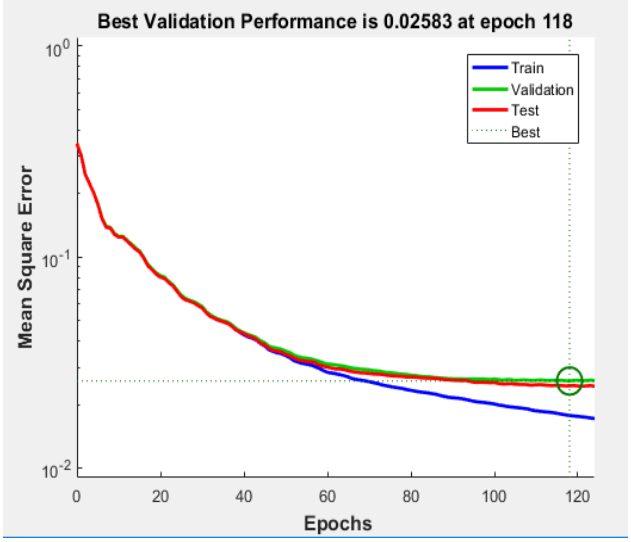

Fig 4.7 Performance curve for SGD-momentum

IV CONCLUSION

I have analyzed the application of two gradient techniques stochastic gradient descent and stochastic gradient descent momentum using neural networks for classification on MNIST dataset. Using a single layer of hidden neurons, we could get approximately 92% classification accuracy. We also got accuracy on the basis of a number of neurons in a hidden layer. Furthermore, we can change learning rate associated with this models and can verify different classification rates. Also, SGD momentum can be used wherever fast convergence is required, but if proper error minimization is looked, we should look into stochastic gradient descent method.

V REFERENCES

[1] Le Cun, B. B., Denker, J. S., Henderson, D., Howard, R. E., Hubbard, W., & Jackel, L. D. (1990). Handwritten digit recognition with a back-propagation network. In Advances in neural information processing systems.

[2] Hochreiter, S. (1991). Untersuchungen zu dynamischen neuronalen Netzen. Diploma, Technische Universität München, 91.

[3] Principe, Jose C., Neil R. Euliano, and W. Curt Lefebvre. Neural and adaptive systems: fundamentals through simulations with CD-ROM. John Wiley & Sons, Inc., 1999.

[4] Srivastava, Nitish, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. "Dropout: a simple way to prevent neural networks from overfitting." Journal of Machine Learning Research 15, no. 1 (2014): 1929-1958.

[5] Schmidhuber, Jürgen. "Deep learning in neural networks: An overview." Neural Networks 61 (2015): 85-117.