Building MLP and Training on MNSIT Data

Tejeswar Reddy Alle
Faculty of Computer Science, Dalhousie University
6050 University Ave, Halifax, Nova Scotia, Canada, B3H 4R2

Abstract

Artificial Neural Networks are powerful technology used to identify the relationships in the underlying data. They are based and inspired by biological neural networks. Multi-Layer Perceptron is one of the feed-forward neural networks. In this paper, Multi-Layer Perceptron is built from scratch using only basic modules like Numpy. The MNIST dataset is the subset of the NSIT Special Database. The training set of the MNIST dataset consists of 60,000 patterns where the testing set contains 10,000 patterns. The MNIST dataset is one of the largest datasets which contains greyscale images of handwritten single digits. This dataset is taken from the famous NIST dataset. The built MLP was trained on the 60,000 patterns of the MNIST dataset and tested on the 10,000 patterns. This paper will discuss how different parameters like adding layers, changing hyperparameters affects the error. It will also discuss how regularization techniques generalize the model.

1. Introduction

The main aim of this paper is to build the Multi-Layer Perceptron from scratch. Building MLP from scratch instead of just using it will help us to understand it better. We will build MLP in python using modules like Numpy. Numpy will make our task of working on n-dimensional arrays easy. We will start by preparing the training and testing data which involves little conversions on the imported MNIST dataset and later proceed with building layers. We will later discuss how feedforward is done in MLP. We will move how the learning process is done in backpropagation and how it impacts the weights between the neurons. We will discuss the error and how it is used in backpropagation and adjusting weights. We will end with generalizing the model using regularization techniques.

2. Preparing the Training and Testing Data

MNIST data is a very famous dataset which contains grayscale images of handwritten digits. These datasets will be available in the Keras module. We can directly import training and testing data from it. Training data contains 60,000 samples where testing data contains 10,000 samples. The input label(X_train and X_test) of both training and testing data contains grayscale images of size 28*28. The output label(Y_train and Y_test) of both training and testing data contains a single digit of size 1

The training data is reshaped into a size of 784 and then each value in the one-dimensional array of size 784 is divided by 255(maximum value of pixel). This is done because we can convert each input value between 0 and 1. Neuron values lie between zeros and ones so dividing these values by 255 will help us achieve this.

We want to convert testing data into their ASCII representation. This can be achieved by adding 48 to each of the values. Since our output values range between 0 and 9, Now they range between 48 and 57. We will now convert this binary representation. This can be achieved by bitwise and between the number and array of two power elements of size 6. Now we have finally converted our output from the size of 1 to 6. We now have training and testing data of input of size 784 and output of size 8.

3. Multilayer Perceptron

The name MultiLayer perceptron tells that it has multi-layers with interconnected perceptrons. The model consists of a hierarchy of layers where each layer consists of a series of neurons. The first layer is the layer where the inputs of the data are applied and this is called the input layer. The last layer is the layer where we get out outputs and this is called the output layer. The layers between the input and output layers are called the hidden layers. Each neuron is connected by all the neurons in the previous layer with the edges where each edge has a weight. As the Input layer is the first layer, Neurons in the input layer has no previous layer neurons to connect. They only connect with hidden layers.

For instance, if there are Input layer I, hidden layer H, output layer O then there are edges with weights W1 connecting neurons in the input layer I and hidden layer H and weights W2 between hidden layer H and output layer O. These weights W1 and W2 are taken randomly at first and later changed by the learning process. We will first create three layers I, H, O with random weights W1 and W2. We will later add layers in between them. We have the input of size 784 and the output of size 6. We will take hidden of size 256. The weights W1 which are between the input layer I and the hidden layer H will be of size 256*784. This is because each neuron in the hidden layer of size 256 will be connected to all the neurons in the input layer of size 784. The weights W2 which are between the hidden layer H and output layer O will be of size 8*256. This is because each neuron in the output layer of the size of 8 will be connected to all the neurons of the hidden layer of size 256. We will now look into FeedForward Network.

3.1 Feed Forward Network

Feed Forward Network is every layer output feed forwards into the next layer of neurons. In other words, calculating the output from the input is done in Feed Forward. In our example, the output of the input layer I feed forwards the neurons of hidden layer H, and the output of the hidden layer H feed forwards the neurons of output layer O. This is done using the formula $\sum_i w_i x_i$

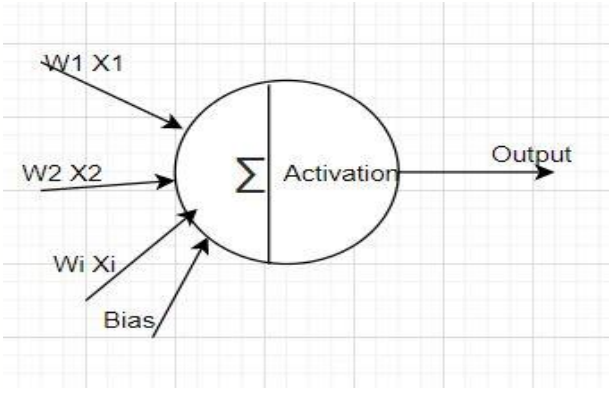

Figure 1Feed Forward Network

Every neuron will get the summation value of the weights and the values of the neurons of the previous layer. An activation function is applied to this summation value. This activation function can be sigmoid, Relu, Tanh, and Linear. Each Activation function has its significance and we will be using a sigmoid function. The sigmoid function is given by formula 1/1+e^-x. Here x is the input of the sigmoid function. In our case, x would be the summation of the weights and output of the neurons of previous layers. The sigmoid function is smooth and yields the values between 0 and 1. One of the reasons for choosing is its simpler derivative which can be differentiable anywhere on the curve. This way Feed Forward Network will finally yield the output of size 6 from the input of size 784. We will see how we will adjust the weights that can yield the correct outputs.

3.2 Back Propagation

The goal of backpropagation is to find the right values for weights in between the layers. So that we can yield the right output. These weights are adjusted based on the error obtained. Error is calculated by two methods which are mean square error and absolute error.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - y_i^p)^2}{n}$$

Figure 2 Mean Square Error

$$MAE = \frac{\sum_{i=1}^{n} |y_i - y_i^p|}{n}$$

Figure 3 Mean Absolute Error

Here in Backpropagation, our goal is to find the minimum points of the derivative in the loss function. We will get the derivative function from the gradient descent that would be output * (1- output). Here the output is the output from the Feedforward. We will multiply this derivative with the error obtained in the previous step. This is called delta output. This will be used to update the weights in the final layer. The dot product between the obtained delta output and the inputs to the last layers will be used as the weights to be added to the existing weights. Since we are trying to find the minimum points in the derivative of the loss function, we may end up in point which causes the graph of the minimum points to take the steep curve. To avoid this and maintain the graph, we will be adding a percentage of the old minimum point. This will be given by the formula dw1 = momentum * dw0 + d0@ X.T. In this formula do is the delta output which we derived from the error and it is multiplied with the input of this layer. We are here deriving the weight value to be added to the existing weights. The weight value to be added is given by dw1. This dw1 is taken from the percentage of the old minimum point which is given by momentum. Finally, this dw1 will be added to our last layer weights W2 by the given formula W2 = W2 + learningrate * dw1. The learning rate will be the hyperparameter that needs to be tuned depending upon the change in error. To tune the hyperparameter, we will perform the feedforward and backpropagation in many epochs and see the change in error in each epoch. We will be changing the learning rate by seeing how it affects the error rate.

We now need to backpropagate the error to the hidden layer to update the weights of the hidden layer. With the help of error, we found the delta output by substituting it in the gradient descent. We will this delta output in the final layer to calculate the delta backpropagate in the next layers. We will find the dot propagate of the delta output and the weights before updated. This dot product will backpropagate the error to this layer and help us find the delt backpropagate in this layer. This is given by the formula dh = h * (1-h) * (w0.T @ d0). In this formula, h is the activation(output) of this layer, d0 is the delta output of the output layer(last layer), w0 is the weight in the last layer before updating. We will calculate the delta backpropagate by substituting the activations and delta output. Similar to the previous backpropagation process in the last layer we will this delta backpropagate to find the weights to be added to correct the existing weights. We will use the momentum as used in the previous step to calculate the change in

weights dwh. This dwh will be added to the existing weights to update them with the learning rate. The below shows the graph of error for 10 epochs with one hidden layer.

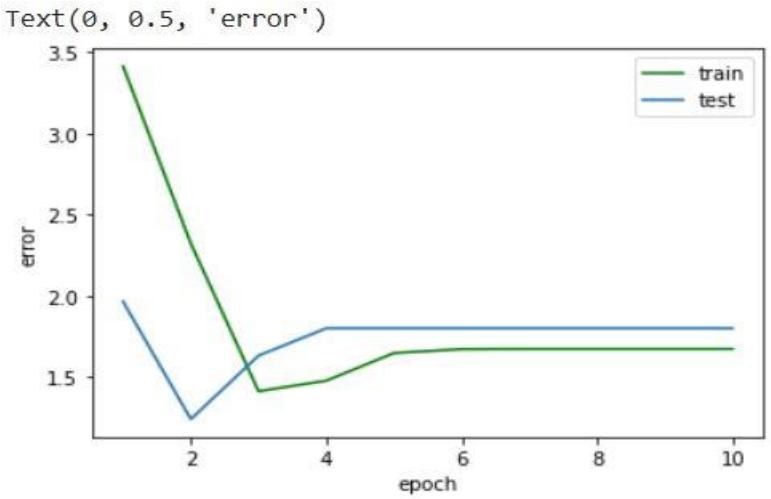

Figure 4 Error Vs Epoch

3.3 Findings and Adding layers

In the above graph, we see the change in error for each epoch. Training error is calculated by the obtained output of the training data obtained during the feedforward. And then backpropagate and then test them on the test data to calculate the test error. This is repeated for the 10 epochs. We observe the training error and the testing error is directly propositional. When there is an increase in train error there is an increase in the test error. When there is a decrease in the training error, there is a decrease in the testing error. We see there is fluctuation in the error rate and sometimes the testing error is greater than the training error. At the beginning test error is small then the training error. This will the common thing you will observe any number of times you run the code.

We will add certain layers in the hidden layers to see if there is any change in the error rate. I have added 6 hidden layers. Adding layers is nothing but adding a set of neurons and connecting them with neurons in the previous and next layer and assigning random weights to them. We will Feedforward and predict the outputs and then calculate the training error and backpropagate. We will again Feedforward to predict the outputs of testing data and then calculate the testing error. I have repeated this process for 10 epochs and plotted the below graph. I have also added regularization to generalize the model. We will further discuss regularization.

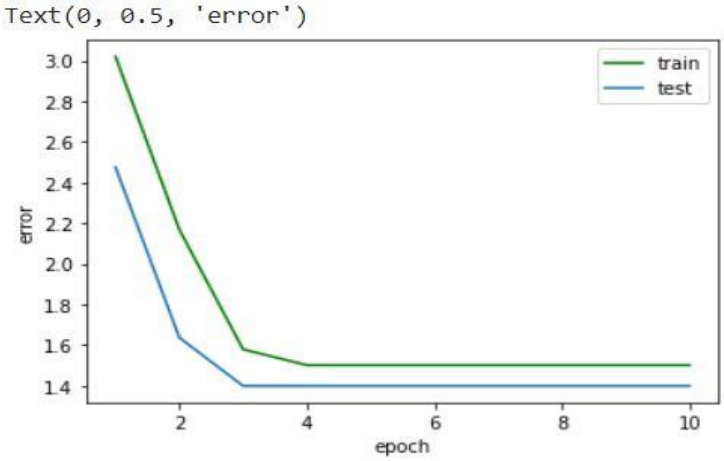

Figure 5 Error Vs Epoch after adding layers and Regularization

We will notice there is a slight change in the graph from the previous graph. Even though there is no much decrease in the error rate. Testing and training errors are constantly decreasing. We also notice that test error is always less and directly proportional to train error.

4. Regularization

In Figure 4, we see there is a sudden rise in the error and we see the model is performing well on the train data and does not perform well on the test data. To avoid this kind of overfitting and generalize the model, We will use Regularization. Regularization is the technique that penalizes the coefficients of the less relative features. In regularization, we will add a regularization term to the loss function. There are 2 famous regularization techniques used. They are Ridge and Lasso. In ridge regression, we are using (y - y_predicted)**2 whereas in lasso we are using the absolute difference between y and y_predicted. The main difference between them Rifge penalizes my reducing the coefficients where loss penalizes my making them to zero. We have a shrinkage parameter denoted by λ. This is also called the tuning parameter. We will change it by seeing how it affects the error rate.

5. Conclusion

To conclude, Building neural networks from scratch helps us to understand the internal workflow of the neural networks. It will help us to solve the problems encountered while using the Keras models of neural networks. While building the MLP, we understood how FeedForward and backpropagation undergoes. This helps us to debug the problems encountered while any of the models throw errors. We learned how adding layers affects the error rate on training and testing data. Finally, we learned how regularization helps in generalizing the model.

References

Introduction to Feedforward Neural Networks - EmilStefanov.net. (2010). Archive.Org. https://web.archive.org/web/20090507210502/http://www.emilstefanov.net/Projects/NeuralNetworks.aspx

Grover, P. (2018, June 5). 5 Regression Loss Functions All Machine Learners Should Know. Medium; Heartbeat. https://heartbeat.fritz.ai/5-regression-loss-functions-all-machine-learners-should-know-4fb140e9d4b0#:~:text=Mean%20Square%20Error%2C%20Quadratic%20loss,target%20variable%20and%20predicted%20values.&text=The%20MSE%20loss%20(Y%2Daxis,range%20is%200%20to%20%E2%88%9