Implementation of MLP from scratch with regularization techniques

Sumith Sai Rachakonda

* Faculty of Computer Science, Dalhousie University
6050 University Ave, Halifax, Nova Scotia, Canada, B3H 4R2
sumith@dal.ca

***Abstract*-** The idea behind neural networks is introduced 60 years ago by Frank Rosenblatt a psychologist. Neural networks provide powerful models in solving image classification problems. However, there exists many different neural net architectures. This paper contains an analysis and usage of a simple neural network implementation with backpropagation using gradient descent and lasso, ridge regularization techniques on MNIST handwritten digits dataset. The goal is to analyze the predictions when noise is injected into the images and to determine the performance levels when regularization techniques are implemented. As a result, this simple network can generalize well with help of regularization.

I. INTRODUCTION

In machine learning, the MNIST dataset is a popular catalogue for training and testing models. The MNIST dataset consists of 60 000 samples of handwritten digits in the training set and 10 000 in the testing set. This paper compares the performance of a multi-layer perceptron (MLP) with normal network implementation, noised data, lasso regularization, ridge regularization on 1024 training images from the MNIST dataset. We know that neural networks are the core of deep learning. We have great modules which implements neural networks but in this paper a network is implemented from scratch using python and numpy to understand what exactly is going on while implementing a neural network. This is a minimalistic implementation without much complexity. In general, the models we use are more complex and accurate. The main idea behind machine learning is to let the algorithm to be trained and predict the output without any human intervention such as explicitly telling the rules to classify the data.

II. DATA/ANALYSIS

2.1 Multi-layered Perceptron (MLP)

In Machine Learning, MLP is a basic or common model for supervised learning. These networks use backpropagation with fully connected layers to learn and optimize weights for each individual neuron. In simple terms a neuron will process the data with the help of predefined calculations. Each layer can have multiple neurons. The input layer will have neurons equal to the number of input features. The hidden layers can have different neurons. The output layer will have exactly the same amount of neurons we are going to predict. For example if we want to predict the sales of a product then the output neuron will be one because we are only predicting the number. If we want to predict whether a dog or cat or bird in an image, then we need 3 neurons one for each type in the output layer. The data enters from input layer into the algorithm, all the calculations that is training will happen till output layer and finally the output will be given from the output layer.

This paper will predict the binary ascii value of a numbers from MNIST database. The MNIST database images are 28x28 sized images. Only the first 1024 images are used for training purpose. Here the labels for images must be converted into their ascii representation. With the help of ord() function in python we can get the ascii value for a number. Then we have to convert the ascii value to its binary representation. This representation will have 6 bits because to store the ascii value we have to store it in 6 bits. But in this case, we are only using 10 numbers in ascii values, the first two bits are same for all the labels. So, we can remove these two bits which results in 4 bits that are used to train and predict. The train images have to be flattened in order to train the model. Which mean we have 784 inputs to the network.

This MLP model consists of one input layer, two hidden layers, one output layer. The input layer consists of 784 neurons because we have 784-pixel inputs. The first hidden layer has 392 neurons, and second hidden layer has 196 neurons. The output layer has 4 neurons because we have to predict last four bits of binary asci value of a number.

2.1.1 Implementation

1. Importing packages required such as matplotlib to visualize the data and numpy for data manipulation and mnist for loading images.
2. Loading data from mnist module and normalizing them by dividing train images with 255 so they are in 0 to 1 range which helps the model to learn easily. Flatten the images to feed to the network.
3. Converting the labels from number to its ascii value and then that ascii value to its binary representation and then dividing each bit as a column to predict.
4. Initialize the weights arbitrarily and the derivates of weights to zeros.
5. Iterate through n number of times to train the model until we get small error.

a. In each iteration apply the dot product to weight and input and then apply activation function.
b. Propagating forward till last layer.
c. For back propagation we need to calculate the delta for each layer by finding the derivates of activation function.
d. Now back propagate the delta terms throughout the network.
e. Updating the weight matrix by adding momentum to the weights.

6. For each iteration calculate the error and save it.

2.1.2 Result for basic implementation

Finally, for MLP, we implemented the net using mean square error as our loss function and recorded the loss for 100 epochs in Figure 1. We can clearly see that the loss curve is gradually decreasing.

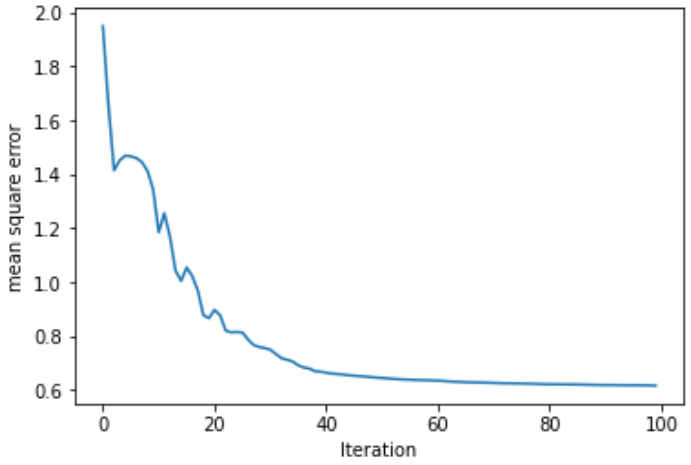

Figure 1 Loss graph for MLP using means square error function for 100 iterations

2.1.3 Adding noise to the data

The above model shows us that it is able to predict using training data. But we want to measure its performance by introducing some noise to the images we used to train and then again try predicting them.

To introduce the noise, we have to first generate it. There are different ways to generate noise for an image using fancy packages. But we will stick to the basic concept of introducing random numbers to the train images. We can generate random distribution matrix using numpy.random.randn() function. Then add this matrix to the training images so it will be filled with noise.

Now we have to do a forward propagation to predict the noised data to see verify the performance of our model. After successful forward propagation, the model is predicting worst to the noised data. Every time it is predicting one bit different when compared to original binary sequence. This tells us that this model is clearly overfitted to training data. To overcome this we can use regularization techniques such as lasso, ridge regression or gradient to our model.

2.1.3 Regularization

Test error is the most important factor for our model to say how well the model performs on unseen data. Test error can be divided into Variance, Squared Bias, Irreducible Error. The following figure 2 will tell us how these effects the model.

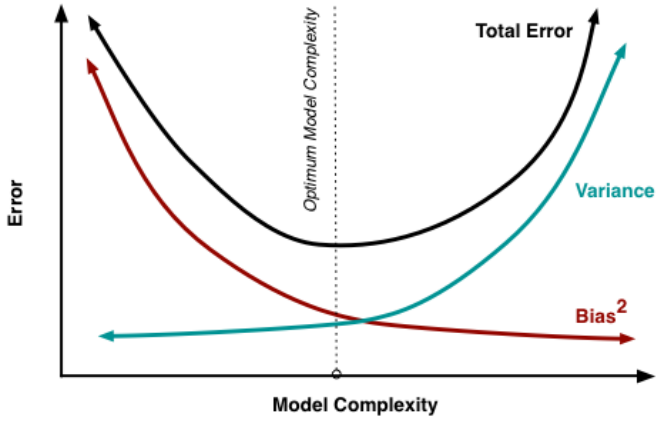

Figure 2 Error vs Model complexity for a model using variance and bias.

We can deduct that models with high bias are simple and underfit which happened in our case. Model with high variance are complex and overfits the data. So, we have to find the point where we get the model just right. We can do this by controlling the variance and bias but not irreducible error. This is where regularization comes into place.

Regularization adds stability to the model by making it less sensitive towards training data. If a model is overfitting, then we need to regularize it. This will result in better test accuracy but reduces the training accuracy. If a complex model is overfitting, then we can increase the bias and reduce the variance to regularize it. We have different techniques to do this, they are L1 Parameter Regularization, L2 Parameter Regularization, Dropout, Data Augmentation, and Early Stopping. We will only discuss about Ridge regression (L1) and Lasso regression (L2) regularization because we have implemented it.

2.1.4 Ridge Regression Implementation

In ridge regression, the cost function is altered by adding a penalty equivalent to square of the magnitude of the coefficients. So we have to update the weights as
wo=wo + (1/N)*alpha*dwo instead of wo=wo+0.1*dwo.

The loss curve will be smooth as shown in figure 3.

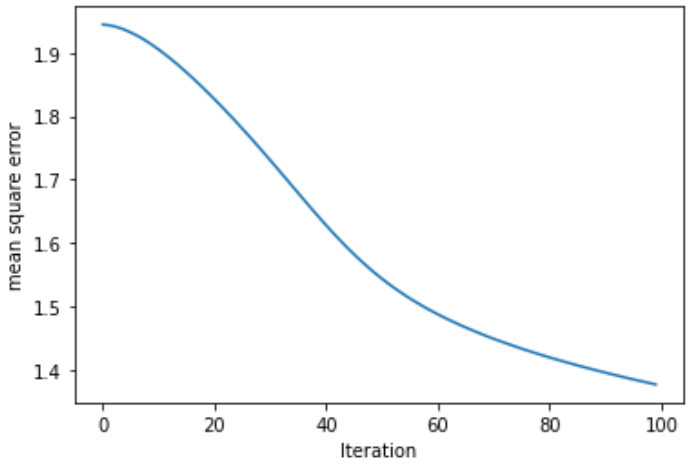

Figure 3 Ridge Regression Loss Curve

2.1.5 Lasso Regression

The cost function for Lasso (least absolute shrinkage and selection operator) regression.

So we have to update the weights as

wo=wo + (2*lambd*np.sign(wo)) + alpha*dwo instead of

wo=wo+0.1*dwo.

The loss curve will be as follows figure 4.

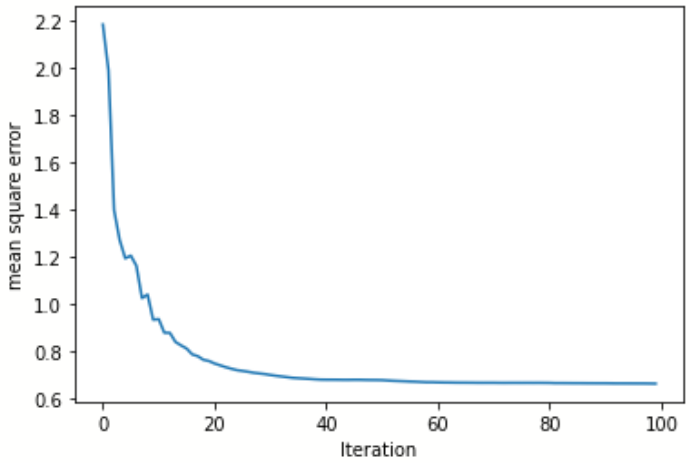

Figure 4 Lasso Regression Loss Curve

III. CONCLUSION

With our valuations on a basic MLP model and then testing with noised data and then doing regularization for better performance. We are able to regularize the model but most of the cases the predictions are varied by a single bit which is not the ideal case. This leads to represent the wrong number. It is therefore concluded that the binary classification of ascii values of labels is worst with basic implementation of a neural network. This can be overcome by using more complex model.

IV. REFERENCES

[1] Understanding the Bias-Variance Tradeoff. (2012). Fortmann-Roe.Com. http://scott.fortmann-roe.com/docs/BiasVariance.html

[2] Imad Dabbura. (2018, May 8). Coding Neural Network — Regularization - Towards Data Science. Medium; Towards Data Science. https://towardsdatascience.com/coding-neural-network-regularization-43d26655982d