Building a Multilayer Perceptron from scratch

and training it on the MNIST handwritten digits dataset

Introduction/Abstract

In this paper, I will be going through the construction of a Multilayer Perceptron (MLP) and its usage on the MNIST Handwritten Digits dataset to predict the ASCII Value. I have used Numpy and matplot lib for basic structures, functions, and plots. I have used Keras to import the MNIST dataset. The MLP is created from scratch without using any third-party libraries other than Numpy, and Matplotlib.

An MLP is a feedforward version of an Artificial Neural Network - where the network is in the form layers, namely the input layer, the output layer, and if required, with hidden layers between the two. The input is passed through the hidden layers to the output layer. The output values of the nodes in the hidden layer become the input to the output layer, which are scaled by the values of the connection strength as specified by the elements in the weight matrix wo - which is the weight of the matrix between the output layer, and the hidden layer right before it. In my project, I tried a variety of different combinations of hidden layers, their perceptrons, testing their results before selecting an ideal setup. These are explained as we go through the paper.

Additionally, I have also used backpropagation, gradient descent, and regularization to my MLP. Backpropagation is used as a learning algorithm to ensure that the global minimum is achieved. It helps the neural network calculate the gradient descent of a loss function with respect to all the weights in the network. As the name suggests, the output values are passed back through the previous layers to calculate the descent. In mathematical terms, it is the derivate of the cost function as a product of the derivates between each layer in a backward manner [1]. As neural networks are complex, they are prone to overfitting. This is solved by using regularization, which helps modify the learning algorithm to tweak the model for better results by reducing overfitting - making the model more flexible.

Creating the network

I started by creating a class that will contain all the attributes and methods that can be used to create the Multilayer Perceptron. I created a constructor that contains the number of perceptrons in the input layer, a list of the number of perceptrons in each of the hidden layers, and the number of perceptrons in the output layer.

Implementation below:

```
__init__(self, num_nuerons_ip, hidden_list, num_nuerons_op)
```

```
#Define Network
mlp = mlp_Network(784,[512,256,128,64,32,16,8],6)
```

The input layer consists is 784 as when we flatten the images from the MNIST - we get 28x28 which is equal to the size of 784. The output layer consists of 6 perceptrons, as when we convert the numbers 0-9 from decimal to ASCII - they consist of 6 digits i.e. 0 = 48 = [1 0 1 0 1 1].

The layers will be a list that will consist of all the perceptrons i.e. input + hidden + output. I am using NumPy random.rand() to assign random weights to each of the layers. The “for” loop will create 2D arrays (matrices) with random weights - where the number of columns will be the number of neurons in the subsequent layers.

Forward Propagation/Feed Forward

This method will provide us with our output after propagating through all of the layers in the network.

For the first layer - the activations are basically just the inputs. After that, we need to loop through all the weight matrixes i.e. looping through all the layers in the network, where we calculate the net inputs and the net activations.

The net inputs are a NumPy dot product i.e. matrix multiplication between the activation of the previous layer with the weight matrix. For activations, we are using the sigmoid activation. The reason I chose the sigmoid function is that it gives us an output between 0 and 1 which makes it better to use for problems where the required prediction would lie between 0 and 1. In this case - for the ASCII Values - the result will be close to 0 in case of 0s and close to 1 in case of digit 1. Therefore, I am passing through net inputs through the sigmoid function to get the activation for the subsequent layer. For modularity, and clarity purposes, I created a separate sigmoid function.

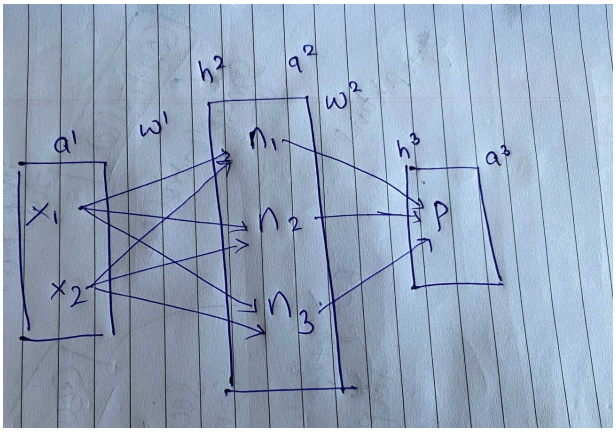
Backward Propagation & Gradient Descent:

While training a neural network, we tweak the weights of the neuron connections such that the prediction accuracy is increased. First, we calculate the error after feeding the input signals forward, and then we backpropagate that error through the layers i.e. we calculate the gradient of the error function over the weights, and then use that information to update the parameters.

The error function is calculated as a function of the parameters - prediction and the expected output. There are a number of ways to calculate the error functions such as cross-entropy, absolute error, and mean squared error. In my implementation, I started with the absolute error,

and while the predictions were good, I noticed that the root-mean-squared error (rmse) method was better as it helps cover the larger errors as well.

Now, calculating the gradient would mean, in mathematical terms, calculating the derivate of the error function with respect to the derivate of the weights. In simple terms, let us say we have one hidden layer, then our gradient descent would look like this:



dE/dW2 = (dE/da3)(da3/dh3)(dh3/dW2)
#Using chain rule from calculus

Where dE = derivate of error fn
a = activations
h = net inputs
W = weight matrix.

The above equation tells us that the derivative of the activations will require us to create a method to create the derivative of the sigmoid function (as activations are calculated via a sigmoid function). Similarly, we go back from here to calculate the derivative over W1.

As we expand these - we can see a pattern forming:

dE/dW2 = (a3 - y) * sig_der(h3) * a2
dE/dW1 = (a3 - y) * sig_der(h3) * W2 * sig_der(h2)*x

Where x = our input vector = dh2/dW1, sig_der = derivative of the sigmoid function, and W2 is simply = dh3/da2. This is the reason, I am saving the activations and derivations in the constructor - to implement backpropagation.

In the implementation of the activation and derivatives in the constructor, I am traversing through all the layers, and creating arrays for activations, and derivations with respect to the neurons in each layer. In derivatives, I am going through all layers -1 as the weight matrices are in between layers and therefore one less than the total number of layers. Therefore, in the backpropagation layer, we loop in reverse to go through all the layers to pass the error back through the layers. In order to go ahead with the dot product, we need to rearrange the current activations and the delta to 2D (similar to our weight matrices), and therefore, I have reshaped them using NumPy reshape. As the delta matrix is a vertical matrix - I have transposed it, to make the dot product possible.

In the case of Gradient Descent, we use the learning rate to tweak our gradient descent to give better predictions i.e. take a step in the opposite direction to the gradient. The size of the step is “Learning Rate”.

For regularization, I have chosen to go forward with L2 regularization [3]. The main difference between L1 and L2 Regularization is the penalty term. Ridge Reg. adds the squared magnitude, whereas L1 = Lasso, adds the absolute value of the magnitude. Regularization in general, helps to make your model better by preventing overfitting. “The key difference between these techniques is that Lasso shrinks the less important feature’s coefficient to zero thus, removing some feature altogether. So, this works well for feature selection in case we have a huge number of features” [4].

Additionally, I used Gaussian Blur[5] to check my output in case of noisy data, I noticed, that in this case, the higher the number of hidden layers - the higher the robustness.

Finally, to train the data - I called the forward propagation method, followed by the backpropagation, and gradient descent with regularization while collecting the errors to plot the accuracy graphs.

Conclusion/Different Variations Tried:

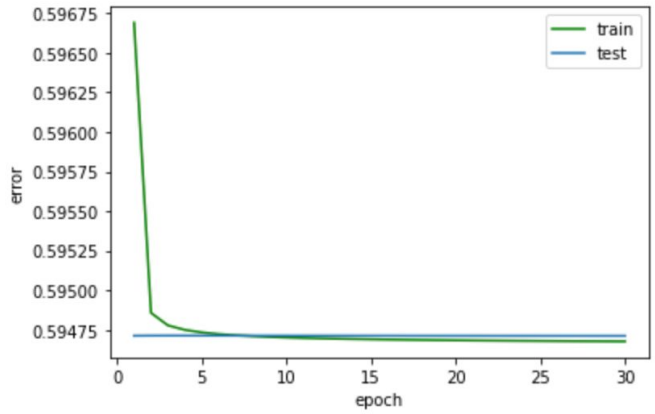


Fig 1.1 - Accuracy with RMSE as the error function

The above figure shows an accuracy plot using the root-mean-squared error function. As we will see ahead, the accuracy is high compared to that while using the absolute error.

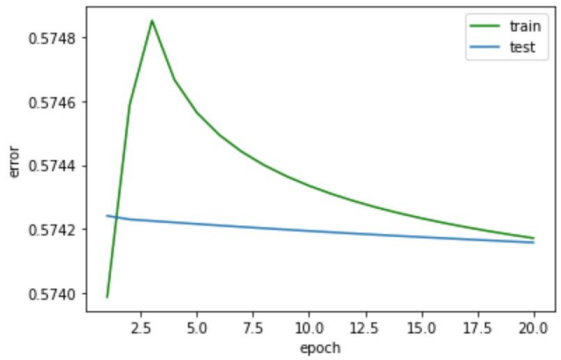


Fig 1.2 - Accuracy with absolute as the error function - epochs 20, training set - 17,000

Fig 1.2 describes the accuracy plot when I took a training set of 17000, with a test set of 5000, and 20 epochs. We can clearly see an evident difference between figures 1.1 and 1.2. This accuracy was, however, increased, when I

increased the training set to 40,000, and the testing set to 10,000.

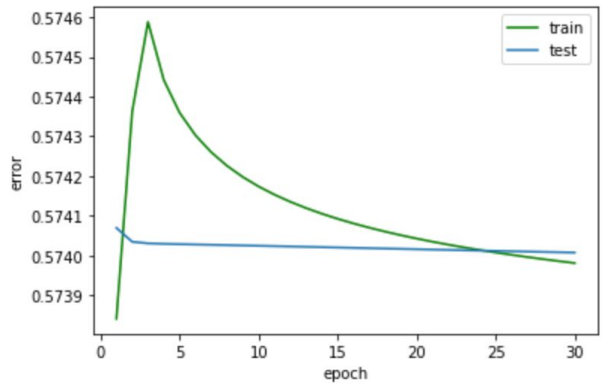


Fig 1.3 - Accuracy with absolute as the error function - epochs 30, training set - 40,000

As we can see here in Fig 1.3, the accuracy increases more than that of 1.2, however, still, 1.1 leads the way.

Additionally, I tried tweaking the learning rate to see the different outcomes, however, I noticed, that as the dataset was large, the learning rate of “1” seemed to produce better accuracy of the error plots compared to the cases where the learning rate was less than one. Similarly in case of regularization, I tried different values of gamma - in which case I noticed that, the larger the training dataset, the lower the value of gamma helped better.

One of the most important factors that mattered in this experiment is the number of hidden layers. Initially, I started with a static program with 3 hidden layers - however, in this case, I noticed that, no matter how much I tuned my other hyper-parameters, my training and test errors changed neglibily after each epoch, in many cases showing no change at all. Therefore, this led me to create a dynamic system, where I could vary the number of hidden layers. In the end, I chose to use 7 hidden layers - which showed me decent changes in the training and test data sets, after that I tuned the learning rate, size of training dataset, and epochs to find a good fit for my program.

References

[1] “Backward Propagation in a Neural Network” - https://www.guru99.com/backpropogation-neural-network.html
[2] “The Sound of AI” - Valerio Velardo - Online, YouTube - https://youtu.be/ScL18goxsSg
[3] “L2 Regularization” - https://www.kaggle.com/mtax687/l2-regularization-of-neural-network-using-numpy
[4] https://towardsdatascience.com/l1-and-l2-regularization-methods-ce25e7fc831c
[5] “Gaussian Blur and Filtering” - https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_imgproc/py_filtering/py_filtering.html