
Homework 1

# CMSC828L/Deep Learning

February 24, 2022

*Instructors:*
David Jacobs

*Student:*
Siddharth Telang

*Course:*
Deep Learning, Spring 2022

*Course code:*
CMSC828L

# Contents

Table 1: Learning rate Vs Iterations for convergence

| Learning rate | Iterations |
|---|---|
| 0.001 | 4000 |
| 0.01 | 100 |
| 0.05 | 500 |
| 0.1 | 50 |

# 1 No hidden layer

## 1.1 1-D Data

Given the below linear equation **??**, the neural network should be able to learn the weights for the equation.

$$y = 7x + 3 + \epsilon \tag{1}$$

In this part, our network contains just the input layer and output layer, no linearity in the output layer and square loss is used. The weights for the linear layer are initialized by random samples from a Gaussian distribution and the biases are set as zero. As the network trains through the forward and backward passes, these weights get updated. Figure 1 shows the learning phase of the network, the loss per iteration and the final predicted line. The hyper parameters here are the number of iterations and the learning rate. However, I observed that if I initialize my weights by sampling from a uniform distribution, the results were bad. The network was training very slow and the MSE was higher than before. Table 1 shows the learning rate vs the iterations for convergence

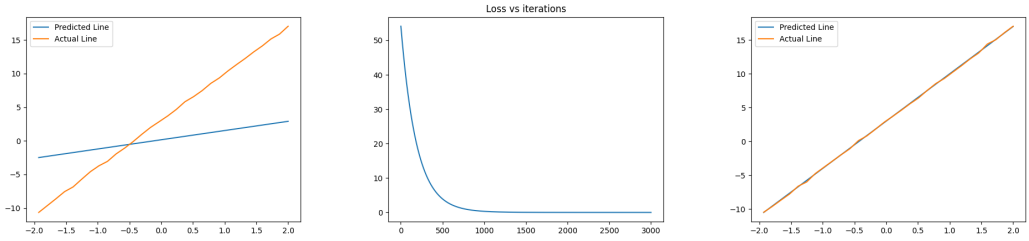


Figure 1: a) Network in learning phase b) Loss per iterations c) Final output

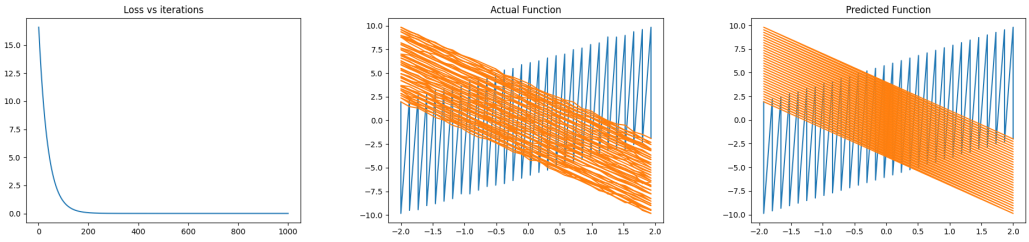


Figure 2: a) Loss per iterations b) Actual Function c) Predicted Function

### 1.2 Higher dimension data

We then test the same network on higher dimension data. The network was able to approximate the function easily with a final MSE of 0.005 on the test data set. Figure 2 shows the plots for this data, and we can see that the simple basic linear network did a very nice job.
The hyper-parameters selection was an easy job, and did not require much experiments. However, the way we initialize the initial weights was a little bit tricky. Sampling from a normal distribution gave better results than other random initializations.

## 2 Shallow Network with ReLU

### 2.1 1-D Data

Now, we build a shallow network with one hidden layer(any number of hidden units) with ReLU and single output unit. Initially when we select any number of hidden units, the network was not learning the function properly, even after changing the learning rate. However, as soon as we re-scale the weights by dividing by a factor - square root of the number of input features, the network starts to learn. This showcases the importance of weights initialization. Figure 3 shows the plots for this, where b) shows the plot when the network is not learning. Figure 4 shows the predicted vs actual function for various hidden units in one layer. Table 2 shows the final MSE for various hidden units in the single hidden layer.

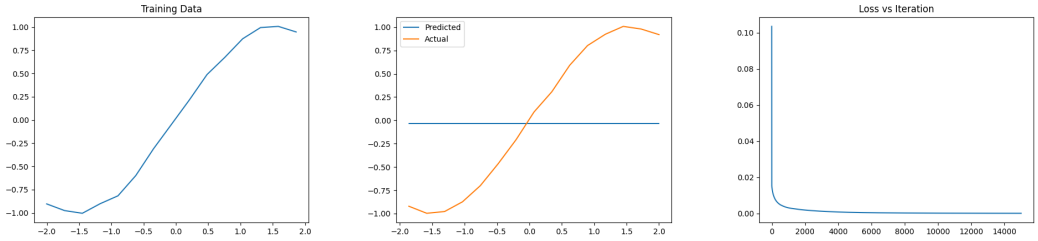


Figure 3: a) Actual Function b) Without scaling the weights c) Loss per iterations after scaling

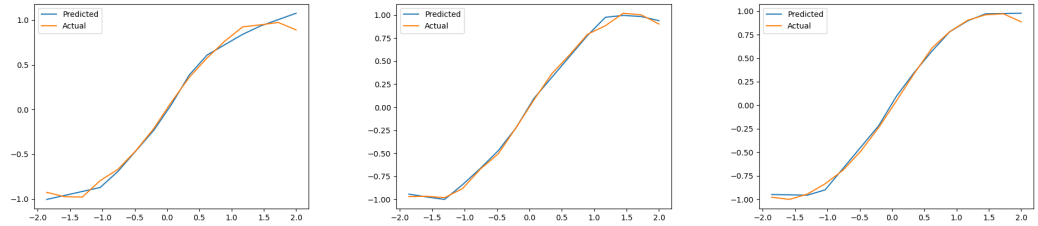


Figure 4: a) 5 units b) 16 units c) 32 units

### 2.2 Higher dimension data

We now test our network on a higher dimension data, and check its accuracy. No hyper-parameter tuning was required in this case as the final MSE(0.00842) was below the threshold limit with 16 units

Table 2: Hidden Units Vs Final MSE

| Hidden Units | Final MSE |
|---|---|
| 5 | 0.0011 |
| 16 | 0.0004 |
| 32 | 0.0009 |

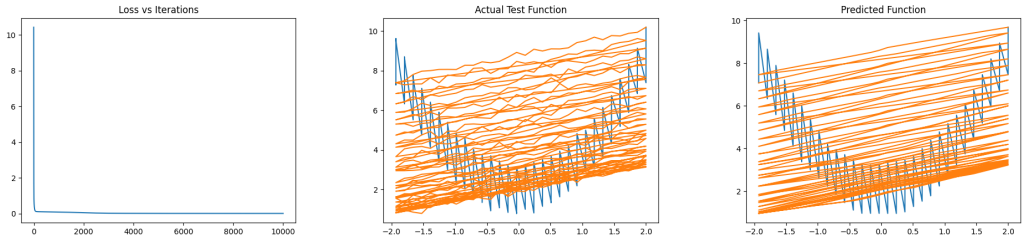


Figure 5: a) Loss vs iteration b) Actual Function c) Predicted Function

in the hidden layer. Figure 5 shows the various plots for the high dimension input.
The most important factor to select in this was the initialization of the weight matrix. If the weight matrix does not have a scaling factor related to the number of input or output features, then we get very poor results. Selecting this factor was difficult and dividing by the square root of the number of input features did the trick!

# 3 Deep Network

## 3.1 1-D Data

Now, we add more layers and hidden units in our network. The deeper we go, the more parameters are to be learnt and hence more iterations are required. Figure 6 shows various loss vs iteration plots for 3 hidden layers and 10 units and various learning rates. We observe that when the learning rate was high (a), the network did not learn at all, and the loss remained almost the same. With decreasing the learning rate, performance was improved, which can be seen in b) and c).
Figure 7 shows the actual vs predicted plots for the above learning rates. Plot c) comes from 20 hidden units and gave a significant lesser MSE than other 3 layered architecture. The predicted curve is smoother than the actual input.

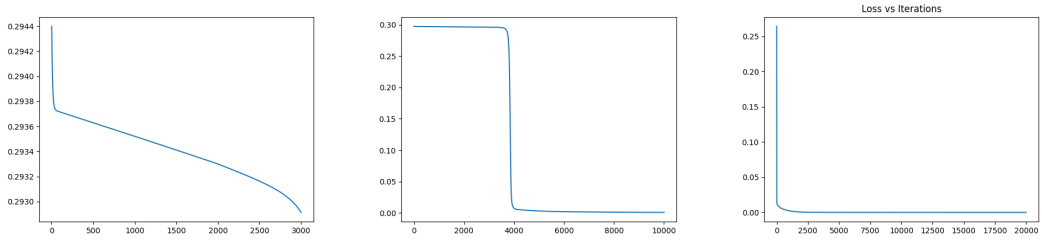


Figure 6: Loss vs Iteration - 3 Hidden Layers, 10 units with learning rates a) 0.5 b) 0.05 c) 0.08

We carry out the same experiment with 5 hidden layers and figure 8 shows the plots. We observe that as we increase the depth, the network requires more training. In a) we can see that the loss kept on fluctuating till 25000 iteration and then settled, and b) was the output and we can see that the

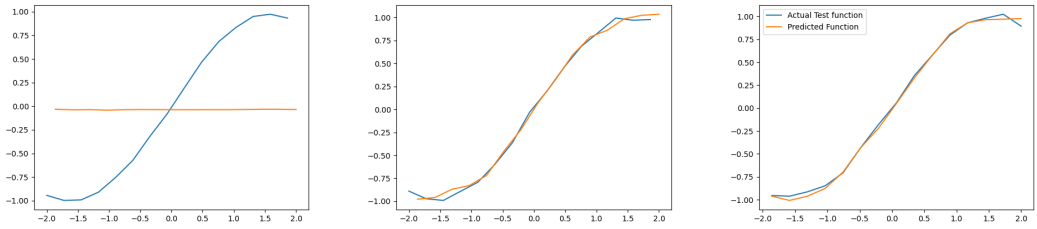


Figure 7: 3 Hidden latyers: Actual vs Predicted a) 0.5 learning rate b) 0.05 c) 0.08 with 20 units

curve is not that smooth. However, if we increase the number of units to 20 with a learning rate of 0.08, then we get a significant improvement and the curve is pretty smooth, which is shown in the c) plot.

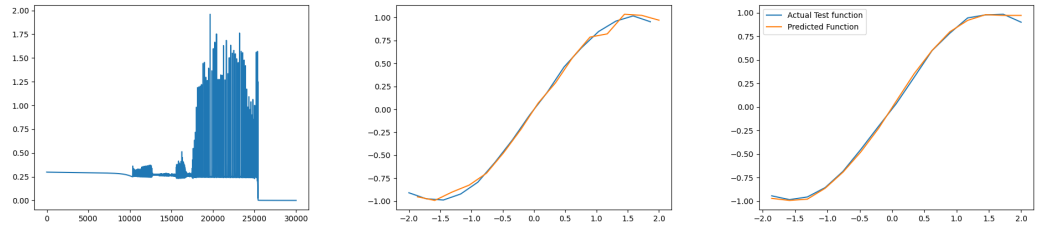


Figure 8: 5 Hidden layersActual vs Predicted a) lr 0.06 10 units b) lr 0.06 10 units c) lr 0.08 with 20 units

## 3.2 Higher dimension data

We now test our deep network on higher dimension data. First, we check for 3 layers and 10 units, figure 9 shows the plots. In this case, we get a final test MSE of 0.01. However, as we increase the number of layers without tuning other parameters, the MSE increased to 0.02. Moreover, the results for 5 hidden layers and 10 and 20 units were almost the same, but the MSE increased to 0.02, figure 10. We also observe that it took more time to converge than the when the network had 3 layers.

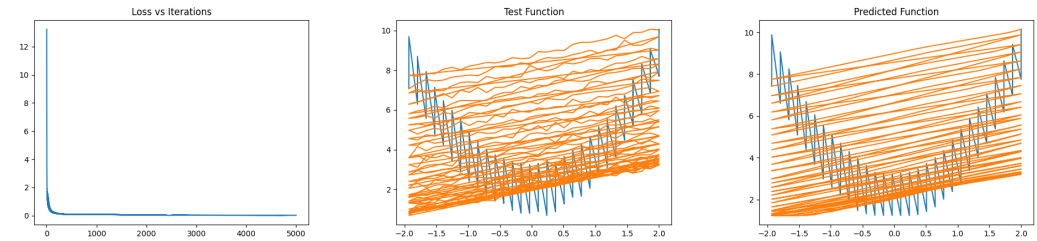


Figure 9: 3 Hidden layers a) Loss vs iteration b) Test function c) Predicted function lr 0.08 3 layers with 10 units

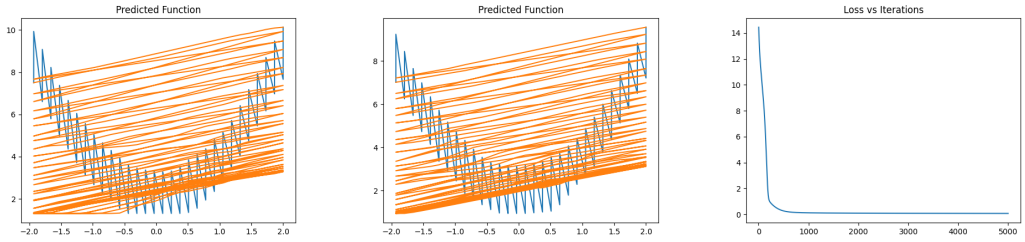


Figure 10: 5 Hidden layers lr = 0.08 a) Predicted function 10 units b) a) Predicted function 20 units c) Loss vs iteration

# 4 Classification

In this section, we are given the task of binary classification, and the output should be either 0 or 1.

## 4.1 Sigmoid

Sigmoid layer gives an output between 0 and 1, but as it's an exponential function, we have to take care that the exponent does not blow up. For this we have two conditions - when the value of value of z is positive and when it's negative as per the below equations

$$\sigma(x) = \frac{1}{1+e^{-x}}, \sigma(x) = \frac{e^x}{1+e^x} \tag{2}$$

For backward pass we have the below when x is positive and negative

$$f(x) = \frac{e^{-x}}{(1+e^{-x})^2}, f(x) = \frac{e^x}{(1+e^x)^2} \tag{3}$$

## 4.2 Cross Entropy

As we have a log function in the cross entropy, we want to make sure that the input is not zero and a very high number. The sigmoid output can be zero, hence we have to ensure this stability. For this, we use the below loss equation, where $\epsilon = 10^{-10}$

$$\sum_{i=1}^{n} (y * log(\epsilon + \sigma(x_i) + (1-y) * log(\epsilon + 1 - \sigma(x_i)) \tag{4}$$

## 4.3 1-D data

We test our network with linear separable data ( 3 hidden layers and 20 units each). Figure 11 shows various plots. The a) part is a basic linear separable data. We also test the effect of increasing the margin between the points which is shown in figure b) and c) with a margin of 0.1 and 0.01. We observe that the loss function gradually decayed when the margin was less, and converged faster when the margin was larger.
We do the same experiment, but now with a 2-d data linear separable with margins, figure 12 Figure 13 shows the loss plots when the margin is a) 0.1 and b) 0.01. Clearly, we see that the when the margin was 0.1, it decayed faster than when the margin was 0.01

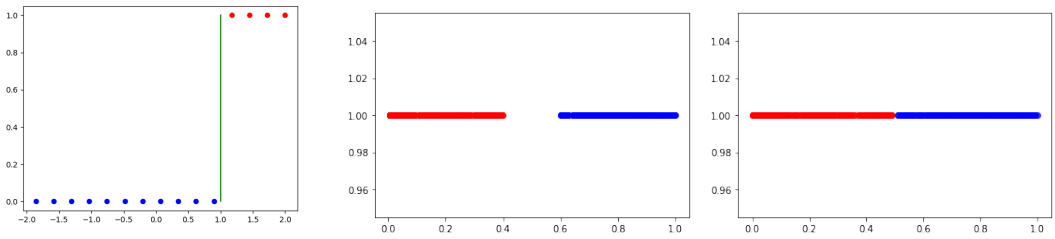

Figure 11: Linear 1-D Separable a)Basic $x > 1$ boundary b) 0.1 margin c) 0.01 margin

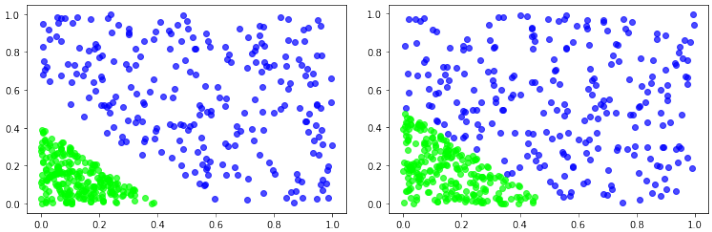

Figure 12: Linear 2-D Separable a) margin of 0.1 b) margin of 0.01

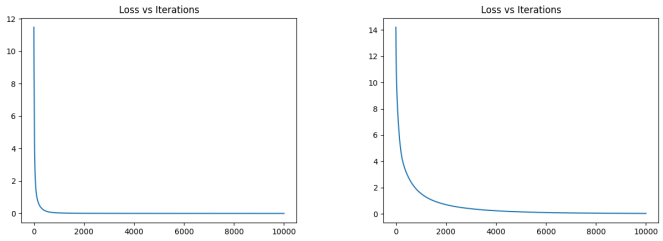


Figure 13: Linear 2-D Separable a) 0.1 margin b) 0.01 margin

### 4.4 Higher dimension data

We test the same network on higher dimension data. We choose 5 layers of 30 units each and perform experiments. Figure 14 shows the plots where b) is the training function and c) is the predicted decision boundary and points, which is almost as similar to the training. We get an accuracy of 97%. When we increase the number of units to 50 in each layer, the accuracy increases to 98%.

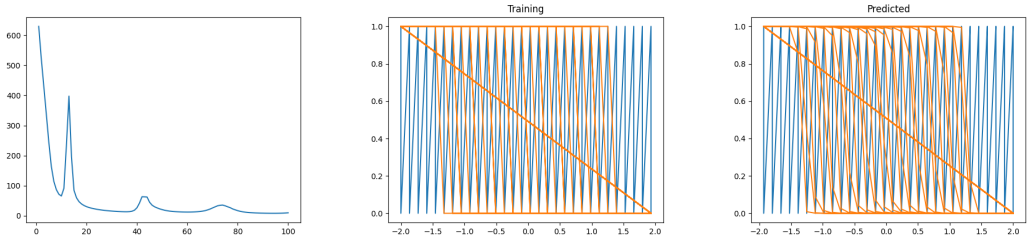


Figure 14: 4b. Higher dimension data a) Loss b) Training function c) Prediction function

Figure 15 shows the results when the data is non-separable.

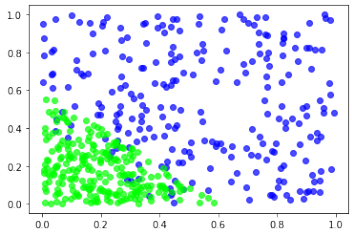

Figure 15: 4b. Higher dimension Non-separable data

Choosing the parameters for this was not difficult, and we were getting almost the same accuracy. We tested with 3 and 5 hidden layers with 30 and 50 units each and the accuracy was almost 98%.

## 5 MNIST

The images are of 784*784 and have pixel value from 0-255 range. To optimize it, we divide each training sample by 255. We carry out experiments which are listed in table 3. The batch size ranged from 128 to 200. The highest test accuracy was obtained with a network of 512, 256, and 128 hidden units with an output of 10 units. Parameters - batch size, number of layers, and number of units per layer, learning rate Figure 16 shows the training loss and the training accuracy on the 512, 256, and 128 architecture.
The training was very slow, and took around 40 minutes for 97% accuracy and the weights file for this was saved.

Table 3: MNIST Testing

| Layers | Units | LR | Test Accuracy |
|---|---|---|---|
| 1 | 784 | 0.8 | 70% |
| 2 | 784, 784 | 0.8 | 76% |
| 3 | 512, 512, 512 | 0.05 | 93% |
| 3 | 512, 256, 128 | 0.05 | 97% |

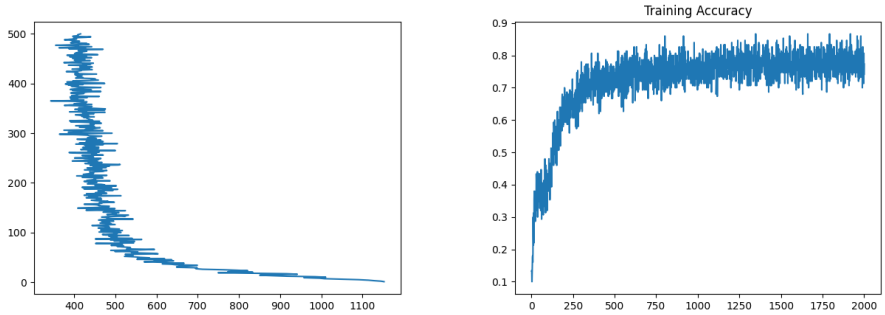


Figure 16: MNIST a) Loss b) Training accuracy