ECE 542: Implementing Backpropagation in a Multilayer Perceptron

Shrey Anand
Computer Science
North Carolina State University
Raleigh, NC, United States
sanand3@ncsu.edu

Graedon Martin
Biomedical Engineering
North Carolina State University
Raleigh, NC, United States
gdmarti2@ncsu.edu

Priya Diwakar
Electrical Engineering
North Carolina State University
Raleigh, NC, United States
psdiwaka@ncsu.edu

***Abstract*—The purpose of this report is to examine how backpropogation can be implemented in a multilayer perceptron, and how the hyperparameters of such a network can be tuned to achieve higher testing accuracy.**

I. Introduction

A perceptron is a type of artificial neuron that takes several binary inputs and produces a binary output. Each of the inputs has an associated weight which describes its relative importance; the weighted sum of the products of the inputs of a neuron and their associated weights ($\sum_j w_j x_j$ or $w \cdot j$) is then compared to a threshold that determines the binary output. This threshold, called the *bias*, indicates how easy it is to get the neuron to activate [1].

Sigmoid neurons improve on the "resolution" of perceptrons by allowing for inputs and outputs other than 1 and 0 using the sigmoid function (a specific instance of an activation function). By tuning the weights and biases of a network of these sigmoid neurons, we are able to increase ability of the network to accurately assign input images (from the MNIST dataset) to output classes.

II. Derivation of Backpropagation Code

Before backpropagation, the input data must first propagate forward through the network to the output to generate a set of predictions. A cost function compares the predictions with the actual values; the goal of any optimization problem is to minimize the value of the cost function.

The full backpropagation algorithm is as follows:

1) Input (*lines 5-6*): The activation of the input layer a^1 is set equal to the data input x.
2) Forward Sweep (*lines 8-11*): The vector $z^L = w^L a^{L-1} + b^L$ and the activation $a^l = \sigma(z^l)$ are computed for each layer, starting at layer two.
3) Output Error (*line 13*): The output error equation $\delta^L = \nabla_a C \odot \sigma'(z^L)$ is computed. The cost function used here is the cross entropy function, defined as

$$\sum_{c=1}^{M} (y^0 \log(a^L) + (1 - y^L)\log(1 - a^L))$$

for a system of M classes. Since the partial derivative of this with respect to a^L yields $\frac{a^L - y}{a(1-a^L)}$ and $\sigma'(z^l) = \sigma(z^L)(1 - \sigma(z^L)) = a^L(1 - a^L)$, the output error is given by:

$$\delta^L = \frac{a - y}{a(1-a)} a(1-a) = (a^L - y)$$

4) Backpropagation (*lines 15-20*): The vector $\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l)$ is computed for each layer, starting at the last hidden layer and proceeding backwards. This step is combined with the gradient computation step.
5) Gradient Output (*lines 15-20*): The gradient is computed with $\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j$ and $\frac{\partial C}{\partial b^l_j} = \delta^l_j$. To initialize these values, the computations for the last (i.e output) layer are done prior to the beginning of the loop.

```
1  def backprop(x, y, biases, weights, cost,
    ↪  num_layers):
2  nabla_b = []
3  nabla_w = []
4
5  activations = [x]
6  h = []
7
8  for i in range(num_layers-1):
9      ai = biases[i] + np.dot(weights[i],
        ↪  activations[i])
10     h.append(ai)
11     activations.append(sigmoid(ai))
12
13 g = (cost).delta(activations[-1], y)
14
15 nabla_w=[np.dot(g, np.transpose(activations[-2]))]
16 nabla_b=[g]
17 for i in range(2,num_layers):
18     g = np.dot(weights[-i+1].transpose(), g) *
        ↪  sigmoid_prime(h[-i])
19     nabla_b = [g] + nabla_b
20     nabla_w = [np.dot(g,
        ↪  np.transpose(activations[-i-1]))] +
        ↪  nabla_w
21
22 return(nabla_b, nabla_w)
```

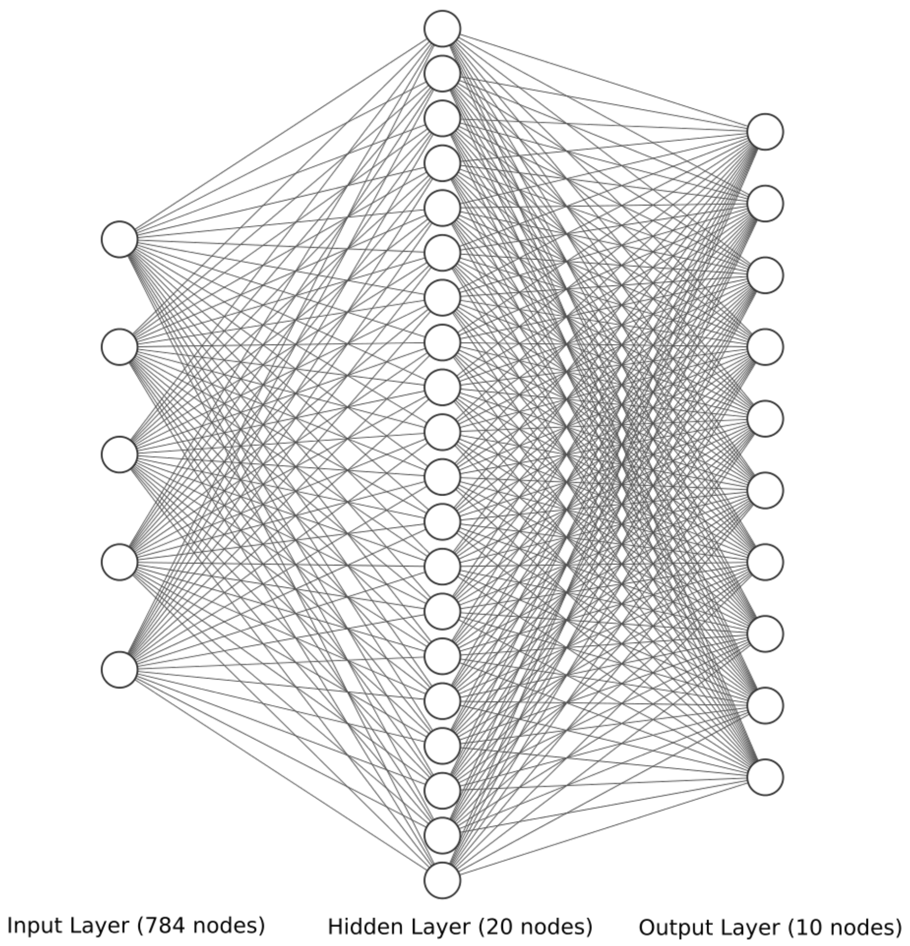


Fig. 1: The structure of the multilayer perceptron.

III. Multilayer Perceptron

A. Network Structure

Figure 1 shows the structure of the multilayer perceptron used. The first layer has 784 nodes, one for each of the pixels in the original 28x28 image. The hidden layer has 20 nodes, and the output has 10 (for the numerals from 0 to 9).

B. Hyperparameters

Epochs	Mini_batch_size	Eta	Validation Data Accuracy
100	64	0.001	0.9084
100	64	0.0001	0.8892
100	128	0.001	0.8752
100	128	0.0001	0.8462
150	64	0.001	0.9073
150	64	0.0001	0.8968
150	128	0.001	0.9068
150	128	0.0001	0.8761
200	64	0.001	0.9057
200	64	0.0001	0.8989
200	128	0.001	0.8972
200	128	0.0001	0.8779

TABLE I: Hyperparmeter Experiments

IV. Discussion

A. Learning Curves

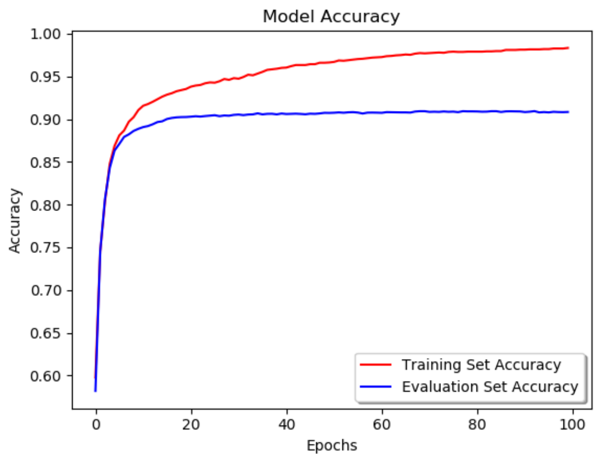


Fig. 2: Model Accuracy for Epochs=100,Batch Size=64,Learning Rate=1e-3

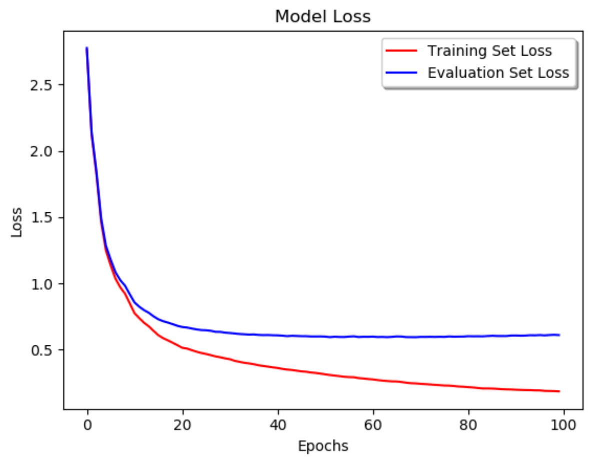


Fig. 3: Model Loss for Epochs=100,Batch Size=64,Learning Rate=1e-3

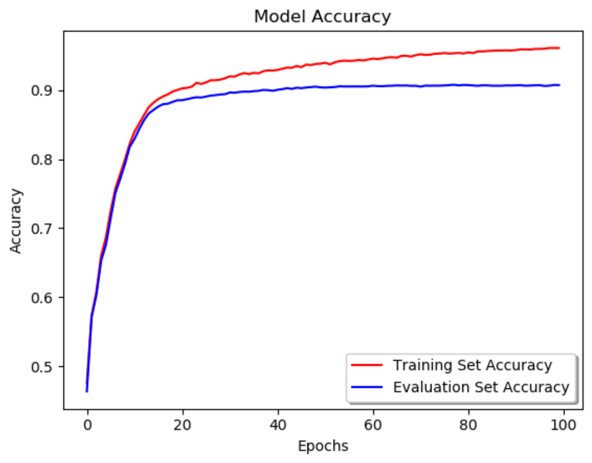


Fig. 6: Model Accuracy for Epochs=100,Batch Size=128,Learning Rate=1e-3

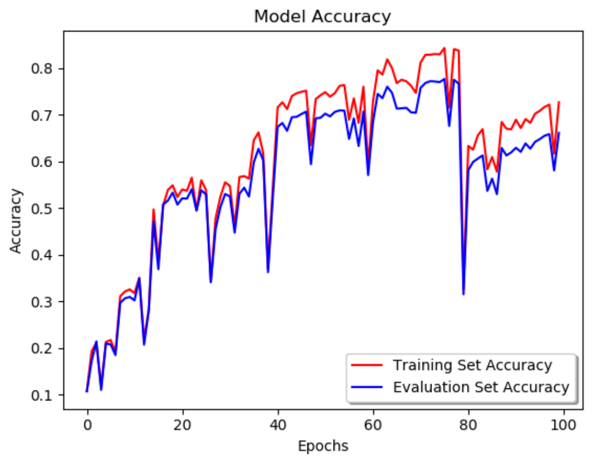


Fig. 4: Model Accuracy for Epochs=100,Batch Size=128,Learning Rate=1e-2

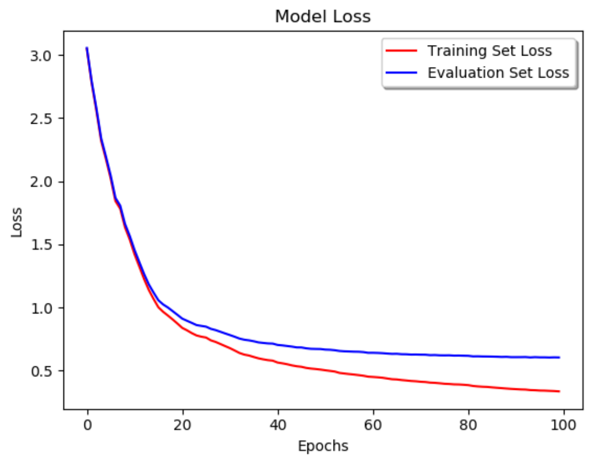


Fig. 7: Model Accuracy for Epochs=100,Batch Size=128,Learning Rate=1e-3

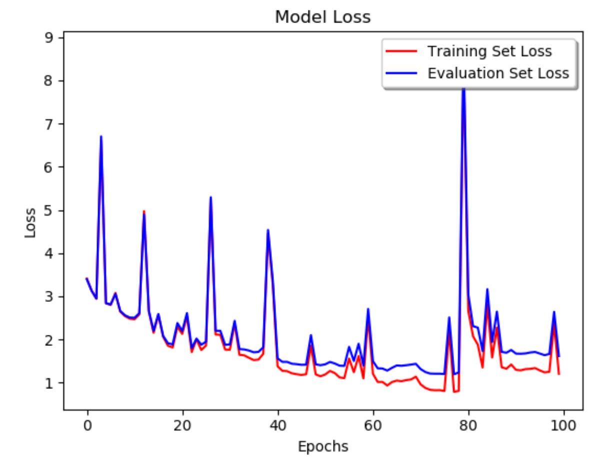
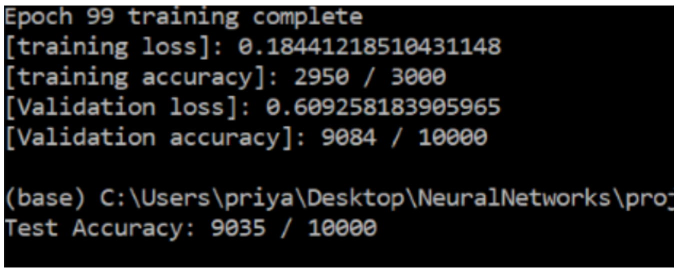


Fig. 5: Model Loss for Epochs=100,Batch Size=128,Learning Rate=1e-2

B. Testing Accuracy

The best accuracy that we got on the testing set was 90.35%.



```
Epoch 99 training complete
[training loss]: 0.18441218510431148
[training accuracy]: 2950 / 3000
[Validation loss]: 0.609258183905965
[Validation accuracy]: 9084 / 10000

(base) C:\Users\priya\Desktop\NeuralNetworks\pro
Test Accuracy: 9035 / 10000
```

Fig. 8: Test Accuracy for Epochs=100,Batch Size=64,Learning Rate=1e-3

V. Conclusion

In this project we have trained a simple Neural Network for MNIST digit classification. We implemented the forward and back propagation step by step. It was observed that the learning rate of 1e-3 gives better results than 1e-2. The number of epochs and batch size, we experimented with, did not have much affect on the model accuracy.

References

[1] Michael A. Nielsen, ”Neural Networks and Deep Learning”, Determination Press, 2015.