Implementation of Backpropagation

Sreechakra Vasudeva Raju Rachavelpula
Dept. of MAE
North Carolina State University
Raleigh, USA
srachav@ncsu.edu

Sai Srinivasan Chakravarthy
Dept. of ECE
North Carolina State University
Raleigh, USA
schakr23@ncsu.edu

Anusha Gururaja Manur
Dept. of Computer Science
North Carolina State University
Raleigh, USA
amanur@ncsu.edu

***Index Terms*—Loss Function, Gradient Descent, Activation function, HyperParameters, Classification**

I. Abstract

This report discusses the implementation of a simple neural network which can be used to classify hand-written digits. For this, the MNIST dataset consisting of 28 x 28 pixel images of hand-written digits along with their labels was used. The emphasis of this project is on the implementation of the back-propagation algorithm used to find local gradient information. The gradient information is required by the Stochastic Gradient Descent algorithm to learn the optimal weights and biases for the network. The hyperparameters such as network structure, number of epochs and learning rate were also tuned to increase the classification accuracy of the network.

II. Introduction

Neural networks work by tuning the weights and biases of various neurons. This process of tuning is called learning. The learning is accomplished by defining a performance measure such as a cost function which penalizes the network for unsatisfactory performance on the given task. The network learns appropriate weights and biases by trying to minimize this cost function.

Stochastic Gradient Descent (SGD) is one of the common techniques used for minimizing this cost function. SGD is a faster variation of the steepest descent technique which uses only part of the training data, a mini-batch, to get an estimate of the local gradient rather than the entire training data for the exact gradient. This gradient of the cost function is computed by using the back-propagation algorithm.

The back-propagation algorithm gives a strategic technique to calculate the gradient of the cost function with respect to the thousands of parameters of the network. Once the gradient is computed, the cost function can be minimized by optimizing along the direction of the negative gradient with a particular learning rate.

III. Experiment setup

The setup for the project requires the following tools and frameworks:

1) Python3
2) Numpy
3) Matplotlib

IV. Implementation

A. Derivation of Forward Propagation:

Here we calculate the value of $\hat{y}$

$$u_1 = w_1 * x$$

$$u_2 = w_1 * x + b_1$$

$$u_3 = \sigma(u_2)$$

$$u_4 = u_3 * w_3$$

$$u_5 = u_4 + b_2$$

$$\hat{y} = \sigma(u_5)$$

B. Derivation of Back-Propagation:

Calculation of gradient values with respect to the parameters: w, b.

$$\nabla_{w_3} L(\hat{y}) = L'(\hat{y}) * \frac{\partial \hat{y}}{\partial w_3} \tag{1}$$

$$\nabla_{w_3} L(\hat{y}) = L'(\hat{y}) * \sigma'(u_5) * \frac{\partial u_5}{\partial w_3} \tag{2}$$

$$\nabla_{w_1} L(\hat{y}) = L'(\hat{y}) * \frac{\partial \hat{y}}{\partial w_1} \tag{3}$$

$$\nabla_{w_1} L(\hat{y}) = L'(\hat{y}) * \sigma'(u_5) * \frac{\partial u_5}{\partial u_3} * \frac{\partial u_3}{\partial u_2} * \frac{\partial u_2}{\partial w_1} \tag{4}$$

$$\nabla_{b_2} L(\hat{y}) = L'(\hat{y}) * \frac{\partial \hat{y}}{\partial b_2} \tag{5}$$

$$\nabla_{b_2} L(\hat{y}) = L'(\hat{y}) * \sigma'(u_5) * \frac{\partial \hat{u_5}}{\partial b_2} \tag{6}$$

$$\nabla_{b_1} L(\hat{y}) = L'(\hat{y}) * \frac{\partial \hat{y}}{\partial b_1} \tag{7}$$

$$\nabla_{b_1} L(\hat{y}) = L'(\hat{y}) * \sigma'(u_5) * \frac{\partial u_5}{\partial u_3} * \frac{\partial u_3}{\partial u_2} * \frac{\partial u_2}{\partial b_1} \tag{8}$$

C. Activation Function : Sigmoid and Derivative of Sigmoid

The Sigmoid function, given by $\sigma(a)$, is used as the activation for all neurons. It is defined as:

$$\sigma(a) = \frac{1}{1 + e^{-a}} \tag{9}$$

Its derivative is given by:

$$\sigma'(a) = \sigma(a)(1 - \sigma(a)) \tag{10}$$

D. Derivation of $\nabla_a L(\sigma(a))$

Here L is the Cross-Entropy Loss function given by:

$$L(\hat{y}) = -(1-y)\ln(1-\hat{y}) - y\ln\hat{y} \tag{11}$$

then

$$\nabla_a L(\sigma(a)) = \frac{\partial}{\partial a}[-(1-y)\ln(1-\sigma(a)) - y\ln(\sigma(a))] \tag{12}$$

$$\nabla_a L(\sigma(a)) =$$

$$(y-1)\frac{\partial \ln(1-\sigma(a))}{\partial(1-\sigma(a))}\frac{\partial(1-\sigma(a))}{\partial a} - y\frac{\partial \ln(\sigma(a))}{\partial(\sigma(a))}\frac{\partial(\sigma(a))}{\partial a}$$

$$\nabla_a L(\sigma(a)) = (1-y)\frac{\sigma'(a)}{(1-\sigma(a))} - y\frac{\sigma'(a)}{\sigma(a)}$$

$$\nabla_a L(\sigma(a)) = (1-y)\sigma(a) - y(1-\sigma(a))$$

$$\nabla_a L(\sigma(a)) = \sigma(a) - y \tag{13}$$

E. Method

The MNIST dataset was loaded and split as
training dataset : 50,000 entries
validation dataset : 10,000 entries
testing dataset : 10,000 entries
Of the training dataset, 3000 entries were selected as training inputs. These training inputs and their corresponding labels were shuffled and split into mini-batches of size 128 entries. The network of the desired structure was initialized with weights and biases which were randomly generated from a normal distribution.

The network was trained using the Stochastic Gradient Descent algorithm. For this, the gradient of the cross entropy loss function with respect to the weights and biases of the network was determined by implementing the back-propagation algorithm. The gradients were calculated and averaged over all the samples in a mini-batch. This gives an approximation of the local gradient which is used for minimizing the cost with the selected learning rate. This was repeated for all the mini-batches in an epoch. The progression of the accuracy and loss with the number of epochs is plotted for training and validation sets. The trained model was tested to determine its accuracy.
The model was optimized by tuning the hyperparameters – number of hidden layers and neurons, and learning rate.

F. Observations

Networks with larger number of hidden layers and units require more number of epochs. Also, if the weights and biases are initialized with large values, it takes a higher number of epochs for the model to converge and a faster learning rate. On the other hand, if the Learning rate chosen is too high, we observe that there are too many oscillations during the training causing the algorithm to diverge.
Another observation that was made relates to number of hidden layers and number of neurons in a hidden layer.We found that increasing the hidden layers while keeping the learning rate and number of epochs constant causes the model to take too much time to perform one step of SGD. This is due to the almost exponential increase in the number of weights and biases to be computed with increasing number of hidden layers. Hence, our network structure performed poorly when stacked with 4 hidden layers or more.

a) Hyperparameters: Number of Hidden Layers and, Number of Neurons in a layer.

Sr.No	Hidden Layers	Neurons	Testing accuracy(%)
1	4	64,32,32,16	80.92
2	3	110,32,32	83.53
3	2	128,32	88.66
4	1	32	90.75

G. Model

a) Structure of neural network model: The designed neural network model has the following structure:

1) Layer 1 : input layer (784 neurons)
2) Layer 2 : 32 neurons
3) Layer 3 : output layer (10 neurons)

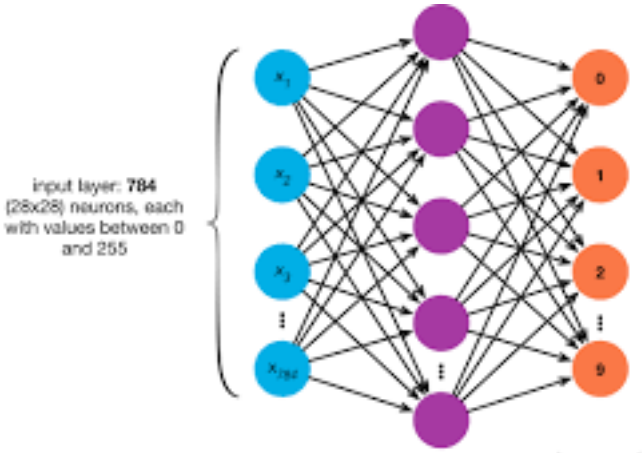

Fig. 1. Neural network model structure

The input layer has 784 neurons corresponding to each pixel of the 28x28 pixel images of the MNIST dataset. The second layer has 32 neurons. The output layer has 10 neurons, where each neuron correspond to one digit observations

b) Hyperparameters: epochs : 60

Sr No	Learning rate	Testing accuracy(%)
1	1e-1	88.92
2	1e-2	89.95
3	1e-3	39.35

c) Hyperparameters: epochs : 100

Sr No	Learning rate	Testing accuracy(%)
1	1e-1	88.72
2	1e-2	90.75
3	1e-3	32.35

V. Plots of Learning Curves

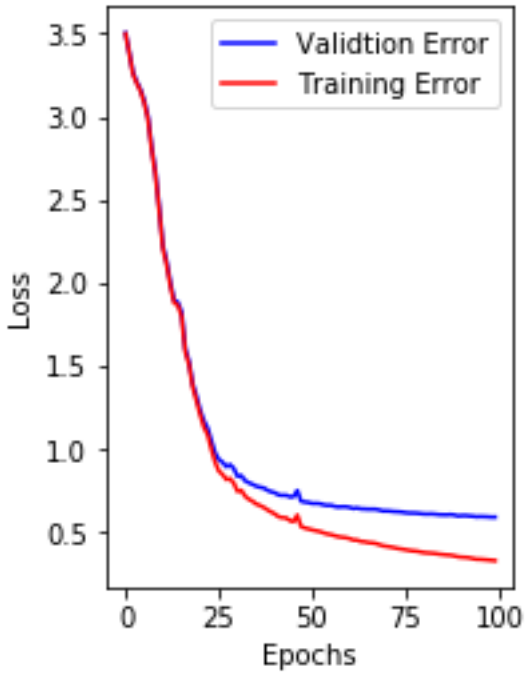

Fig. 2. Learning Curve for Loss on Training and Validation

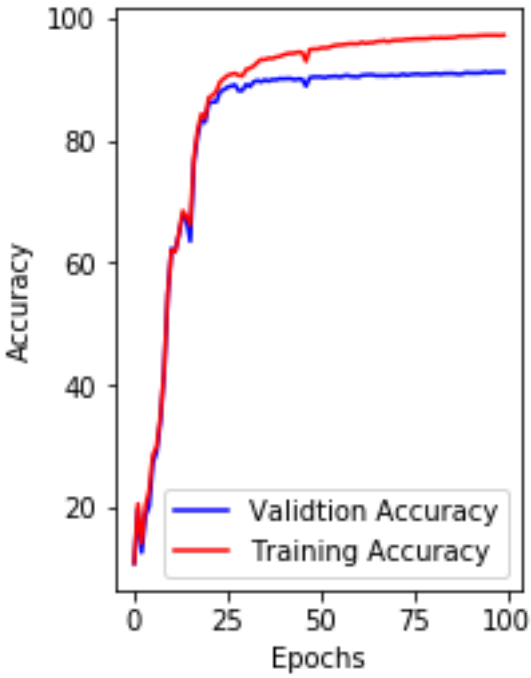

Fig. 3. Learning Curve for Accuracy on Training and Validation

VI. Results

The finalized architecture of the network with one hidden layer and 32 hidden units performed satisfactorily for a learning rate of 10^{-2} with a testing accuracy of 90.75%. The plots show the progression of the training and validation accuracies with the number of epochs. It is apparent that the training accuracy improves linearly during the first few epochs (up to 30 epochs) and much slowly for the remaining as it asymptotically approaches 100%. On the other hand, the validation accuracy increases linearly and seems to saturate at about 90% for the remaining epochs. The training and validation loss curves seem to mirror this behaviour as they decrease with the progression of training.

Another interesting observation that can be made from the learning curves is that they are not smoothly decreasing or increasing but seem to noisy characteristics. This can be attributed to the stochastic nature of the SGD algorithm. As the learning progresses, the network parameters seem to attain certain local optima. But the SGD algorithm allows uphill gradients which forces the network to search for better parameter values.

```
Epoch 79 training complete
[training loss]: 0.3963984461052597
[training accuracy]: 2901 / 3000
[Validation loss]: 0.612415868493412
[Validation accuracy]: 9121 / 10000
======Training Complete======
Training Accuracy is: 0.9670000000000001
Validation Accuracy is: 0.9121000000000001
++++++++Start Test+++++++++
Testing Accuracy is: 90.75
++++++++End Test++++++++
Model Saved Succefully
```

Fig. 4. Output

References

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville. Deep learning, MIT press, 2016.
[2] Michael Nielsen. Neural Networks and Deep Learning.
[3] SGD, towardsdatascience.com
[4] CNN, cs231n.github.io/neural-networks-3/#sgd