PROGRAMMING ASSIGNMENT 1

Utkarsh Pratap Singh Jadon, jadon.1

CSE 5526: PA-1: Report

1. INTRODUCTION

Programming Assignment 1 required the implementation of two-layer perceptron with the backpropogation algorithm to solve the parity problem. Both the forward and backpropogation paths had to be implemented, including the implementation of a perceptron and the activation function. For a 4-bit input pattern, desired output will be 1 if the input pattern contains odd number of 1's, and 0 otherwise. Developed a neural network with an input layer consisting of 4 neurons, a hidden layer consisting of 4 hidden neurons, and an output layer consisting of 1 output neuron. Number of epochs was unknown, but the learning was stopped when an absolute error of 0.05 was reached for every input pattern. Weights and biases were initialized to random numbers between -1 and 1, and a logistic sigmoid function with a = 1 was used as activation function for all neurons.

2. RESULTS

Table 1. Truth Table

x_3	x_2	x_1	x_0	y
0	0	0	0	0
0	0	0	1	1
0	0	1	0	1
0	0	1	1	0
0	1	0	0	1
0	1	0	1	0
0	1	1	0	0
0	1	1	1	1
1	0	0	0	1
1	0	0	1	0
1	0	1	0	0
1	0	1	1	1
1	1	0	0	0
1	1	0	1	1
1	1	1	0	1
1	1	1	1	0

In Table 1, the first four columns represent the input bits, and the fifth column represents the output bit. Output (y) is 1 when number of 1's in input pattern is odd, and 0 otherwise.

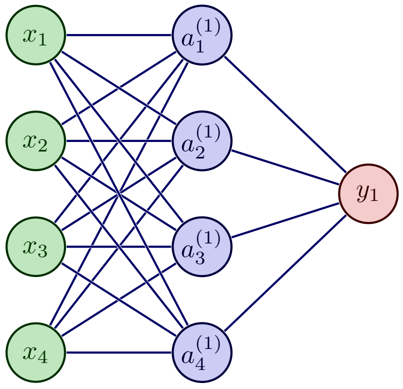


Fig. 1. Neural Network

As shown in Figure 1, a two-layer neural network was designed with 4 input neurons, 4 hidden neurons, and 1 output neuron. Following computation graph was used to calculate the loss, J, using forward pass.

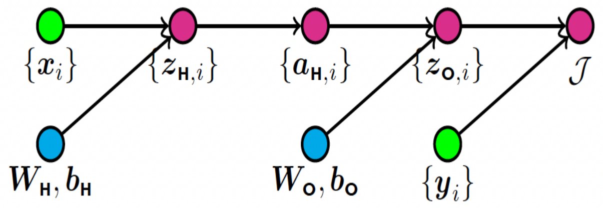


Fig. 2. Computation Graph

From Figure 2, following equations can be derived that are used in forward pass.

$$z_{H,i} = W_H x_i + b_H$$
$$a_{H,i} = h_H(z_{H,i})$$
$$z_{O,i} = W_O a_{H,i} + b_O$$
$$a_{O,i} = h_H(z_{O,i})$$

where a is logistic sigmoid function. Total cost is given by:

$$J = \sum_{i=1}^{N} J(z_O; y)$$

Once the total cost, J, is calculated during forward pass, the error is backpropagated through all layers and their neurons. Gradient descent is applied for all weights and biases in the output and hidden layers, using partial derivative of the cost function. Following are the equations used in gradient descent:

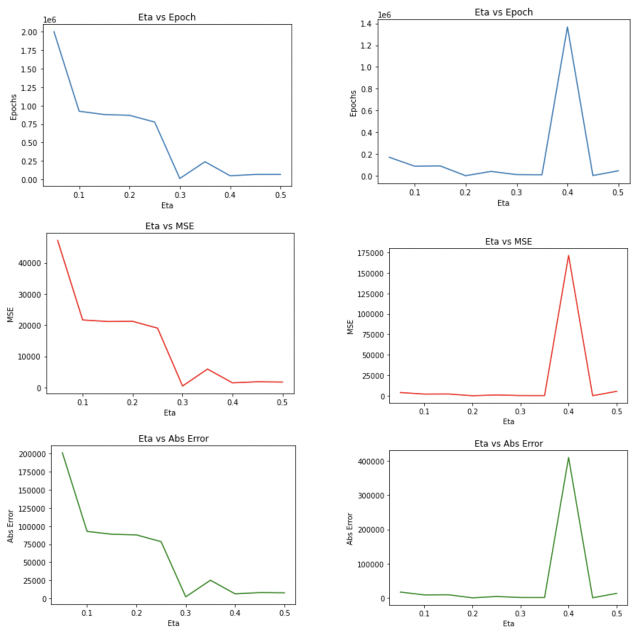
$$W_O = W_O - \eta \frac{\mathrm{d}J}{\mathrm{d}W_O}$$

$$b_O = b_O - \eta \frac{\mathrm{d}J}{\mathrm{d}b_O}$$

$$W_H = W_H - \eta \frac{\mathrm{d}J}{\mathrm{d}W_H}$$

$$b_H = b_H - \eta \frac{\mathrm{d}J}{\mathrm{d}b_H}$$

Momentum term is included in the gradient descent algorithm, so that the gradients accumulated from past iterations will push the cost further to move around a saddle point even when the current gradient is negligible or zero.



(a) Without Momentum (b) With Momentum

Fig. 3. Results

3. CONCLUSION

Neural network without momentum was created by seeding the randomly initialized weight and bias values between -1 and 1, following which the forward and backward propagation were implemented, for different values of learning rate η. Figure 3(a) shows the variation of η vs Epochs, η vs MSE, and η vs Absolute Error, for gradient descent without momentum. Essentially, we can say that as the learning rate increases, the number of epochs required to reach desired result decreases. However, these values still depend on how well the weights and biases were initialized.

Similarly, another neural network was created by seeding and randomly initializing the weights and biases between -1 and 1. This time, along with the learning rate, I incorporated momentum, denoted by β, with a value of 0.9. Figure 3(b) shows the variation of η vs Epochs, η vs MSE, and η vs Absolute Error, for gradient descent with momentum $\beta = 0.9$. We can see that except for $\eta = 0.4$, the speed of training for all other values is much less as compared to without momentum. Thus, we can conclude that the model performs much faster, and in much less epochs, once we incorporate momentum in the gradient descent algorithm.