ECE- 5332-011: Deep Learning for Medical Signal/Image data

ASSIGNMENT - 2: IMPLEMENTATION OF A SIMPLE NEURAL NETWORK

Team Members:

Rifat Zaman
Truong Dinh
Md Raihan Majumder

Contents

Abstract

The primary purpose of this project is the implementation of Neural Network. We implemented several algorithms e.g. perceptron algorithm, single layer neural network, and multi-layer (two hidden layers) neural network. We have done this implementation by steps. First, we have observed how perceptron works. We also recorded that the XOR logic gate did not converge for perceptron. Second, we have implemented Neural Network for 2-bit logic gates and then for Fisher's iris dataset. We used backpropagation algorithm to update weights and train the network. The project helped us have a firm grip on the theory and implementation of a neural network.

Project Description

The project contains 2 part.

1. In the first part, we have demonstrated how the smallest functional unit in any Neural Network, the perceptron, work. We implemented 2-bit AND, OR, NAND, NOR and XOR (did not converge for XOR) gates.

2. In the second part, we have implemented a Neural Network
    - First, we have implemented 2-bit logic gates using a neural network. We have trained a simple neural network with one hidden layer and implemented 2-bit AND, OR, NAND, NOR, and XOR gates.
    - Second, we have trained a model to classify the Fisher's iris dataset. (1 Hidden Layer)
    - Finally, we have approached the same Fisher iris classification problem with a network with two hidden layers. (2 Hidden Layers)

PART 1: THE PERCEPTRON ALGORITHM

1.1 Theory

A perceptron takes weighted inputs from more than one inputs, adds them together and passes that through a non-linear activation function. The structure of a simple perceptron is shown in Figure 1.

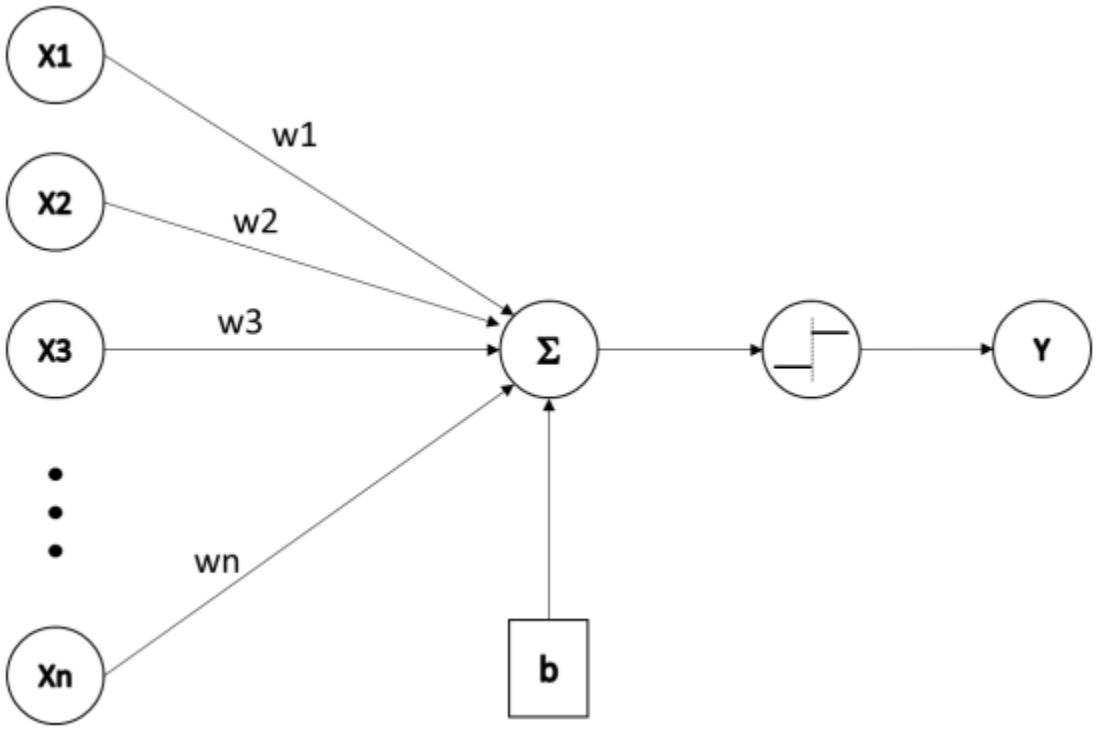


Figure 1: Perceptron

Here X's are the inputs, b is the bias and Y is the output. Mathematically, the output 'y' is computed using:

$$Y = f(X_1 w_1 + X_2 w_2 + \cdots + X_n w_n + b) \quad \text{..... (1.1.1)}$$

Here, $f(z)$ is the activation function.

In this part, we have uses the perceptron for implementing simple 2-bit logic gates. Because the output of a logic gate can take only binary values of either 1 or 0, we have uses a step activation function, such as

$$f(z) = \begin{cases} 1; & z > 0 \\ 0; & otherwise \end{cases} \quad \text{..... (1.1.2)}$$

For training the perceptron, we have used a single example at a time. The weight update equation is as follows:

$$W_{k+1}{}^{(i)} = W_k{}^{(i)} + \rho \times [Y_k - \bar{Y}_k] \times X_k{}^{(i)} \quad \text{….. (1.1.3)}$$

Here 'ρ' is the learning rate, 'k' is the iteration count, and 'i' is the feature number.

The process of training a perceptron for a 2-bit logic gate was as follows:

i. To define inputs and outputs
ii. To set initial weights
iii. To compute output using weights and compute error
iv. If error is not zero, to update weights (For practicality, we have checked if the RMS error is less than a very small value, in the order of 10^{-16}. If the RMS error is greater than this value, then we have updated the weights.)
v. To jump to step (iii) until error is zero

We have uses the above defined perceptron algorithm to implement 2-bit AND, OR, NAND, NOR and XOR gates. For each case, we have varied the learning rate 'ρ' from 0.0001 to 1 in orders of 10.

We have set 10^5 as the maximum number of iterations allowed. We have set initial weights to random numbers from the normal distribution, in the beginning of the iterations.

1.2 Results

Table I: Average No. of Iterations for Convergence

		Logic Gate				
		AND	OR	NAND	NOR	XOR
Learning Rate	0.0001	29679	45483	28126	38134	Did not converge
	0.001	5182	5805	2898	3367	Did not converge
	0.01	455	454	164	526	Did not converge
	0.1	44	62	26	59	Did not converge
	1	8	5	10	6	Did not converge

In Table I, we see that the number of iterations required for AND, OR, NAND and NOR gates decrease with the increase of the value of the learning rate. This happened, because with higher learning rate, the weights went towards the optimal values faster.

For calculating the average, we have used four different values in the rng() functions for random number generation, and calculated the numbers of iterations for them. Then we have taken the average for all of the cases and mentioned the average values in the above table.

The final values of the biases and weights for AND, OR, NAND and NOR gates are mentioned in the following tables for rng(3).

Table II: For $\rho = 0.0001$, rng(3)

	AND	OR	NAND	NOR
b	-0.4584736454	-0.0000079728	0.3463540784	0.0000171146
w_1	0.4584276216	1.1221214941	-0.3364610413	-0.8298834118
w_2	0.0000468168	0.3236835859	-0.3463096893	-0.1261573429

Table III: For $\rho = 0.001$, rng(3)

	AND	OR	NAND	NOR
b	-0.1816404287	-0.0002397575	0.8289042850	0.0000741022
w_1	0.0004240542	0.1279576348	-0.1366454300	-0.0002739312
w_2	0.1814562177	0.0004061777	-0.8288233527	-0.6554862383

Table IV: For $\rho = 0.01$, rng(3)

	AND	OR	NAND	NOR
b	-0.7686539284	-0.0017686184	0.3991966468	0.0001671283
w_1	0.0996315496	0.0039853341	-0.0274859445	-0.3089842361
w_2	0.7685293677	0.0032196585	-0.3736359260	-0.1258363539

Table V: For $\rho = 0.1$, rng(3)

	AND	OR	NAND	NOR
b	-0.1245687529	-0.0114195576	0.7387657891	0.0024673270
w_1	0.1107057579	0.4375810486	-0.0608653942	-0.4334419602
w_2	0.0271519593	0.0147620867	-0.6958665069	-0.0231173111

Table VI: For $\rho = 1$, rng(3)

	AND	OR	NAND	NOR
b	-0.5808639209	-0.1745657918	0.4501619209	0.0142512653
w_1	0.5117361092	0.4508610101	-0.3707625197	-0.1723883316
w_2	0.2340847641	0.6541935235	-0.0945125839	-0.1104675189

For XOR gate, the number of iterations equal to the maximum number of iterations for all values of the learning rate. This means that, for XOR gate, the iterations did not converge for any learning rate.

1.3 Why iterations for XOR do not converge:

Here, we have two inputs, X_1, X_2, and one bias, b.

From (1.1.1) and (1.1.2),

$$Y = f(X_1w_1 + X_2w_2 + b) = \begin{cases} 1; & X_1w_1 + X_2w_2 + b > 0 \\ 0; & otherwise \end{cases}$$

The truth table of XOR is as follows:

X_1	X_2	Y	Required value of $X_1w_1 + X_2w_2 + b$
0	0	0	≤ 0
0	1	1	> 0
1	0	1	> 0
1	1	0	≤ 0

Therefore, for convergence, (b,w_1,w_2) need to be such that,

$$b \leq 0 \qquad \text{..... (1.3.1)}$$

$$w_2 + b > 0 \qquad \text{…… (1.3.2)}$$

$$w_1 + b > 0 \qquad \text{…… (1.3.3)}$$

$$w_1 + w_2 + b \leq 0 \qquad \text{…… (1.3.4)}$$

It can be proved that (1.3.1)-(1.3.4) are NOT simultaneously possible for any combinations of (b,w_1,w_2).

Proof:

Adding (1.3.2) and (1.3.3),

$$w_1 + w_2 + b + b > 0$$

Or, $$w_1 + w_2 + b > -b \qquad \text{…..(1.3.5)}$$

Now, from (1.3.1),

$$b \leq 0$$

Or, $-b \geq 0$ …..(1.3.6)

From (1.3.5) and (1.3.6),

$$w_1 + w_2 + b > -b \geq 0$$

Or, $w_1 + w_2 + b > 0$ ….. (1.3.7)

Now, (1.3.7) contradicts with (1.3.4), which is given.

Therefore, inequalities (1.3.1)-(1.3.4) are not simultaneously possible for any combinations of (b, w_1, w_2).

[Proved]

Therefore, iterations for XOR do not converge for a single perceptron.

PART 2: NEURAL NETWORK

Part 2(a): 2-bit logic gates

Using neural network, we have implemented the 2-input logic gates (AND, OR, NAND, NOR, and XOR) and a 2-bit adder. The neural network treats this as a classification problem and gives 2 outputs; p(0) and p(1). Here p(x) denotes probability of event to be 'x'. The output of this neural network logic gate is the one with the highest probability.

For calculating average number of iterations, we have used four different random initial values of the weights, and calculated the numbers of iterations for each case. Then, the average numbers of iterations for different initial weights are calculated. As learning rate, we have used $\rho = 0.01$.

For convergence, we have set two conditions:

1. Training accuracy = 100%, and
2. Cost function ≤ 0.25

Table VII: Number of Iterations for Convergence for 4 Hidden Layer Nodes

	AND	OR	NAND	NOR	XOR	Adder
rng(10)	4342	3007	4931	4338	17188	14162
rng(20)	4356	2630	2838	2378	11494	9643
rng(30)	Did not converge for 5×10^6 iterations	7448	4527	4447	1233253	19278
rng(40)	5707	3738	4562	2832	18457	19470
Average	**4802 (for 3 cases)**	**4206**	**4215**	**3499**	**320098**	**15638**

Table VIII: Number of Iterations for Convergence for 5 Hidden Layer Nodes

	AND	OR	NAND	NOR	XOR	Adder
rng(10)	5081	3853	5306	3292	18143	21056
rng(20)	6633	3515	5043	2586	10316	13533
rng(30)	8331	7054	2825	3125	1325268	14667
rng(40)	4096	2812	5267	3893	15681	17735
Average	**6035**	**4309**	**4610**	**3224**	**342352**	**16748**

Table IX: Number of Iterations for Convergence for 6 Hidden Layer Nodes

	AND	OR	NAND	NOR	XOR	Adder
rng(10)	7126	10365	3181	2944	Did not converge for 5×10^6 iterations	16998
rng(20)	104444	9736	102290	6893	43200	19925
rng(30)	3770	3932	4413	3903	15332	12064
rng(40)	6597	6215	3918	3968	Did not converge for 5×10^6 iterations	16261
Average	**30484**	**7562**	**28451**	**4427**	**29266 (for 2 cases)**	**16312**

Part 2(b): Classification of Iris dataset

Table X: Accuracy (%)

		No. of Training/Test Samples			
		50/100	75/75	100/50	125/25
rng(10)	Training	100	100	100	99.20
	Test	99	98.67	96	100
rng(20)	Training	98	98.67	99	99.20
	Test	97	93.33	94	96
rng(30)	Training	98	98.67	99	99.20
	Test	98	97.33	100	100
rng(40)	Training	100	100	99	99.20
	Test	95	93.33	96	96
Average	**Training**	**99**	**99.33**	**99.25**	**99.20**
	Test	**97.25**	**95.67**	**96.50**	**98**

In this part, we have trained the neural network with one hidden layer to classify Fisher's iris dataset. The input data has four inputs and three outputs. The output would be 1 for one of three categories and 0 for others. We have tried four different random initializations of the weights. For that, we have used the rng() function of MATLAB with four different seed values, 10:10:40. We have set 10^5 as the maximum number of iterations here. As learning rate, we have used $\rho = 0.005$.

Part 2(c): Deep network

Table XI: Accuracy (%)

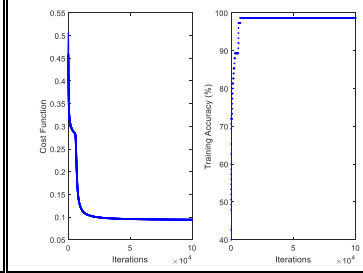
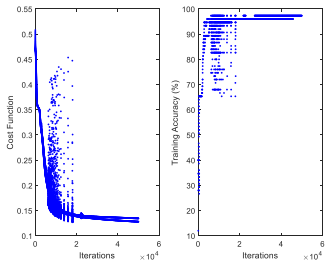
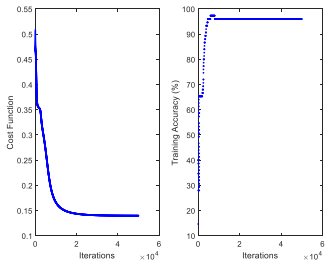
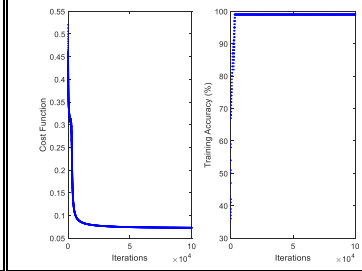
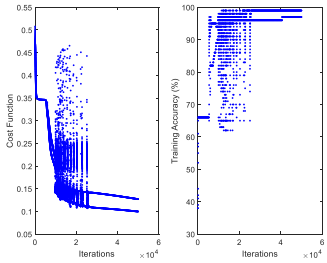
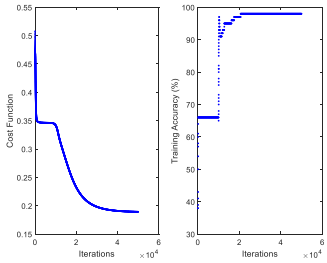
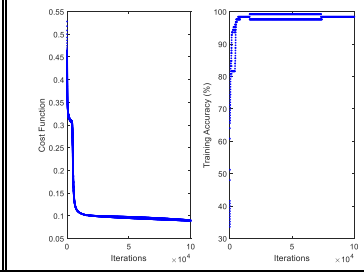
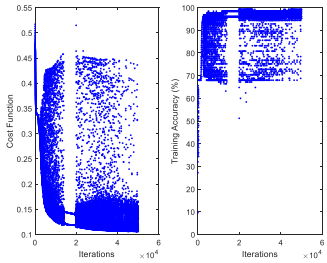
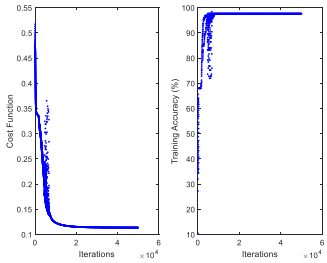
		No. of Training/Test Samples			
		50/100	75/75	100/50	125/25
rng(10)	Training	98	100	100	99.20
	Test	98	96	96	96
rng(20)	Training	98	96	98	97.60
	Test	96	100	96	100
rng(30)	Training	98	98.67	99	99.20
	Test	99	98.67	100	96
rng(40)	Training	100	100	99	99.20
	Test	97	96	96	100
Average	**Training**	**98.50**	**98.67**	**99**	**98.80**
	Test	**97.50**	**97.67**	**97**	**98**

In this part, we have repeated part b with two hidden layers instead of one.

For this part, we have used exponentially decreasing learning rate. The reason of using exponentially decreasing learning rate is discussed in the following section.

Reason of using exponentially decaying learning rate in Part 2(c)

Table XII: Cost function vs iterations for different cases

	For Part 2(b) with $\rho = 0.005$	For Part 2(c) with $\rho = 0.005$	For Part 2(c) with $\rho = 0.005e^{\left(-\frac{k_{iter}-1}{\tau}\right)}$
rng(20), 75/75			
rng(20), 100/50			
rng(20), 125/25			

In Table XII, we see three cases. It is clearly visible that for Part 2(b), the constant learning rate is successfully decreasing the cost function with iterations. However, for Part 2(c), the constant learning rate makes oscillatory cost function (and accuracy) with iterations. We have tried with decreased value of the constant learning rate, but that resulted in much lower accuracy for 10^5 iterations. It seemed that the decreased value of the constant learning rate would achieve good accuracy at last, but with the expense of a lot more number of iterations.

A feasible alternative is, to use decaying learning rate, the result of which has been shown in the rightmost column of the above Table XII. As the value of τ, we have used $\tau = \frac{5000}{\ln(2)}$. Therefore, the learning rate became halved every 5000 iterations. We see that the decaying learning rate, the oscillation of the cost function with iterations is greatly reduced, while the training accuracy is also reasonable (>95%).