Subject

Date

1- $W = \begin{bmatrix} w_{11} & w_{12} \\ w_{21} & w_{22} \end{bmatrix} = \begin{bmatrix} \sqrt{2} & 0 \\ 0 & \sqrt{2} \end{bmatrix}$, $\alpha = \frac{1}{2}$ Q1

w_1 w_2

2- for $P_1 \longrightarrow d(w_1, P_1) = \sqrt{(1-\sqrt{2})^2 + (-1-0)^2} = 1.08$ ☆ wins

$d(w_2, P_1) = 2.61$

calcalutions are done with numpy to be more accurate

$w_1 := w_1 + \alpha \overset{(\alpha - w_1)}{\cancel{\ }} \longrightarrow \cancel{\ } [\cancel{\sqrt{2}}$

$\hookrightarrow w_1 := w_1 + \alpha (P_1 - w_1) = [1.2 , -\frac{1}{2}]$

for $P_2 \longrightarrow d(P_2, w_1) = 1.51$

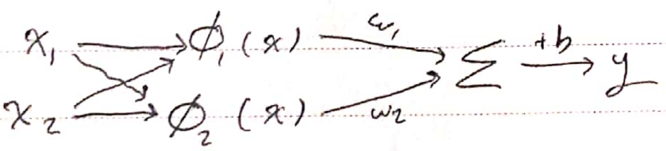
$d(P_2, w_2) = 1.08$ ☆ wins

$w_2 := w_2 + \alpha (P_2 - w_2) = [0.5 , 1.2]$

for $P_3 \longrightarrow d(P_3, w_1) = 2.28$ ☆ wins

$d(P_3, w_2) = 2.66$

$w_1 := w_1 + \alpha (P_3 - w_1) = [0.103 , -0.75]$



Q2

$\phi_k(x) = exp(-\|x - \mu_k\|^2)$

for (1, 0) , (0, 1) responds 1

for (0, 0) , (1, 1) responds 0

$y = \overset{w_1}{\cancel{x_1}} \phi_1(x) + w_2 \phi_2(x) + b$

let $\mu_1 = [0.9, 0.9]$ and $\mu_2 = [0.1, 0.1]$

y	x	$\phi_1(x)$	$\phi_2(x)$
0	(1, 1)	0.98	0.19
1	(0, 1)	0.44	0.44
0	(0, 0)	0.19	0.98
1	(1, 0)	0.44	0.44

calcalated with numpy

$\longrightarrow 0 = w_1 \times 0.98 + w_2 \times 0.19 + b$

$\longrightarrow 1 = w_1 \times 0.44 + w_2 \times 0.44 + b$

$\longrightarrow 0 = w_1 \times .19 + w_2 \times 0.98 + b$

$\longrightarrow 1 = w_1 \times 0.44 + w_2 \times 0.44 + b$

$w_1 = \frac{-100}{29}$

$w_2 = \frac{-100}{29}$

$b = \frac{117}{29}$

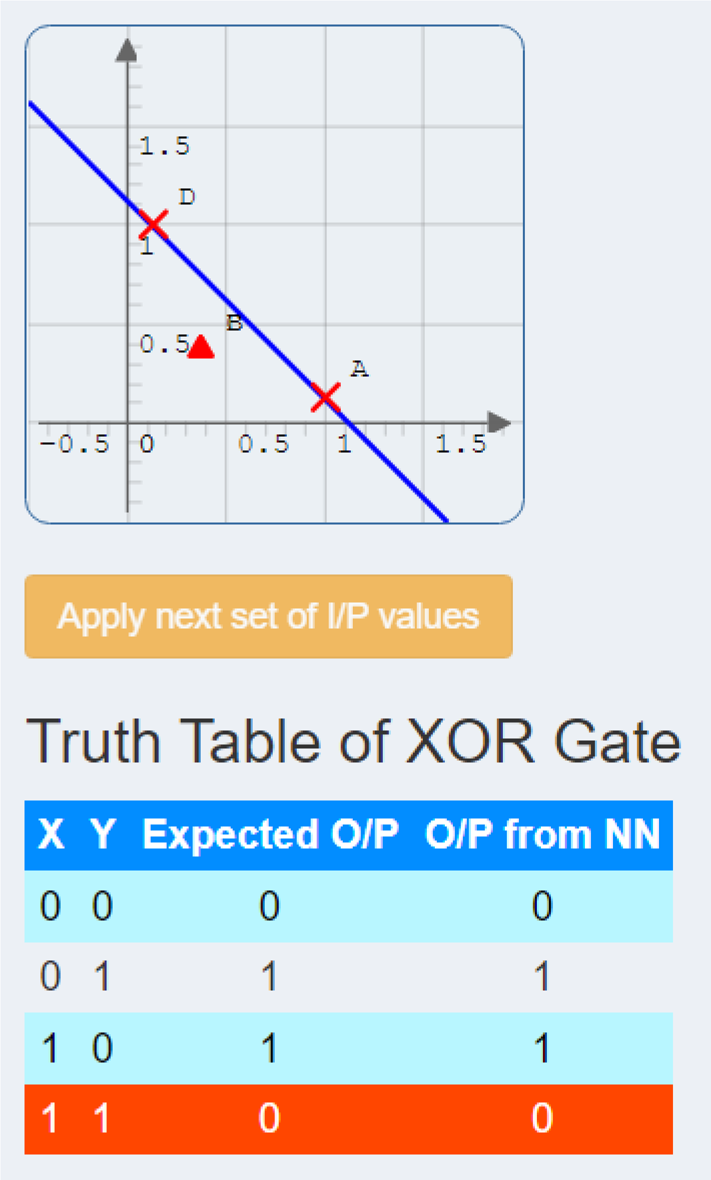
WolframAlpha.com used to solve equations

Scanned by CamScanner

Since I couldn't find out a way to feed my μ_1 and μ_2 to the simulater, I did another test with $\mu_1 = [1, 1]$ and $\mu_2 = [0, 0]$

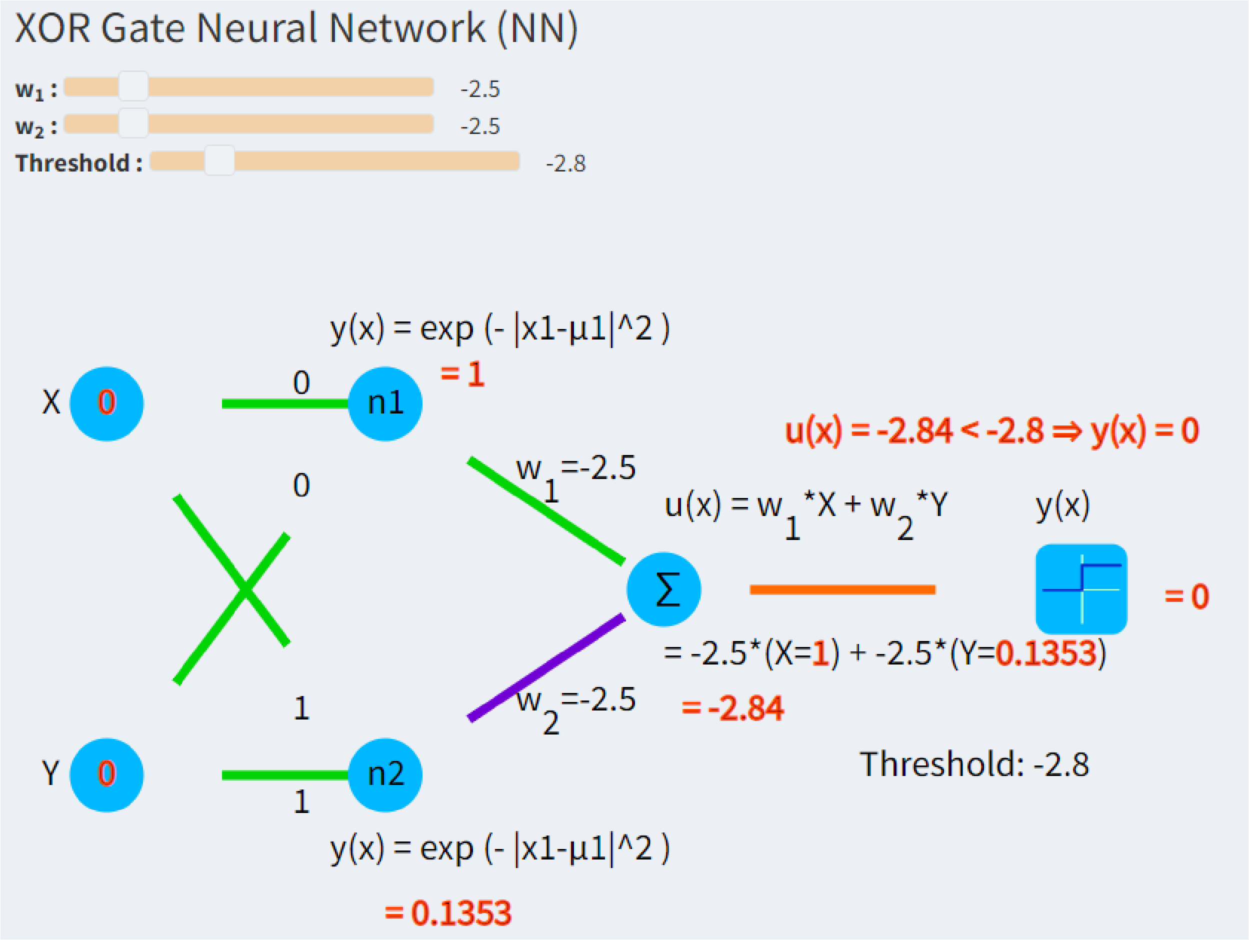
x	$\phi_1(x)$	$\phi_2(x)$
(1, 1)	1	0.13
(0, 1)	0.36	0.36
(0, 0)	0.13	1
(1, 0)	0.36	0.36

$$\underbrace{\left.\begin{aligned} w_1 = w_2 &= -2.5018 \\ b &= -2.8404 \end{aligned}\right\}\text{from solutions}}_{\text{I fed these to the simulation}}$$



Truth Table of XOR Gate

X	Y	Expected O/P	O/P from NN
0	0	0	0
0	1	1	1
1	0	1	1
1	1	0	0



XOR Gate Neural Network (NN)

w_1 : -2.5

w_2 : -2.5

Threshold : -2.8

X
0
Y
0
0
0
1
1
n1
n2
y(x) = exp (- |x1-µ1|^2)
= 1
y(x) = exp (- |x1-µ1|^2)
= 0.1353
w
1
=-2.5
w
2
=-2.5
Σ
u(x) = -2.84 < -2.8 ⇒ y(x) = 0
u(x) = w
1
*X + w
2
*Y
= -2.5*(X=1) + -2.5*(Y=0.1353)
= -2.84
y(x)
= 0
Threshold: -2.8

In the first cell, three library is imported which is used in the next cells.

In the next cell, a radial function is defined, this radial function is a gaussian function.

With the following formula:

$$e^{\gamma||x-\mu||^2}$$

In the third cell, it seems the coder wanted to solve the problem in matrix form but somehow, he changed his\her mind.

Now let’s jump to cell number 6:

The first two lines are the definition of features of the XOR problem. The next line is the appropriate label for each pair of inputs.

The next two lines of codes are the definitions of the centers. These centers are similar to those on the course’s slides.

In the next line, the end_to_end function is called, which is a function to optimize the weights of the rbf network. After that using the optimized weights, we fed those weights and inputs to predict_matrix to get the prediction of the inputs. As we can see, the predictions correct. The weights are different from what I found on paper thou.

Now let’s see what predict_matrix does:

This matrix is implementing the feed forward of RBF network. First it feeds the inputs to the phi functions (which are gaussian radial functions) and then it multiplied it with the trained weights which it received as an input parameter. (dot product)

Then it rounds the number to the decimal values. (Remove the floating points but not setting the round option of round function, so the default value is 0, which means it’s going to round it to decimal point)

At last we can see the end_to_end function:

The first two lines of this function calculates the gaussian radial (both phi functions) for every inputs.

The next 8 lines, simply plot the 4-input point that we have (4 pair of XOR inputs)

In the next 11 lines, it did two things, first it plotted the converted inputs (inputs that are fed to gaussian radial function) and then plotted a sample hyperplane line to demonstrate that now we can linearly separate the inputs.

To find the weights of the network, this function used the matrix inverse technique instead of gradient descent one.

$$\vec{\phi}.\overrightarrow{W} = \vec{d} \qquad \Rightarrow W = \phi^{-1} d$$

This line is the exact implementation of the above formula. It incorporates the linalg sub library of numpy which has inv function that is used to compute the inverse of a matrix. Although to make the matrix rectangular, it multiplied it to itself.

After that with the weights at hand, it calculated the output of the network by multiplying the outputs of phi functions which is stored at matrix A using a for loop to the newly obtained weight. Then it prints out the predicted values against the real values to show result of the model.

Q3:

Same as always in the first we imported the required libraries.

Next we downloaded the dataset and unzip it to the specified folder. Then with numpy loadtxt function we read the data. Since the data stored in csv format, we set ‘,’ as the delimiter.

After that by using the test_train_split function from sklearn we splited the data into test and train.

20 percent of the data kept for testing and 80 percent is going to be used for training.

We also set the random state to 42, to get similar result from every student.

Helper function cell:

minmax_scaler:

here in this function, with the help of MinMaxScaler of sklearn, we scaled the data using min max technique which is to substitute every sample from min and then divide it to the max-min.

e_distance:

this function returns the Euclidean distance of the two x and y inputs.

This function also used the help of the distance.euclidean from sklearn.

(Why didn't we used this: e_distance = distance.euclidean)

m_distance:

this function returns the Manhattan distance of the two x and y inputs.

This function also used the help of the distance.cityblock from sklearn.

(Why didn't we used this: m_distance = distance. cityblock)

Winning_neuron:

This function finds the winning neuron from the map. (Competitive learning)

The algorithm is simple, first of all we initialize the value of the winner score (shortest_distance), then we iterate over every neuron in the map (since the map is 2D, we need 2 loops) for the hope of finding a neuron with smaller distance. In this case by distance, we mean the Euclidean distance.

Decay:

Decay function does two things, first of all it finds out the suitable learning rate for the problem by incorporating the current step and max step into the learning rate calculation.

This means that with every step we take, we are going to use a smaller learning rate. (this is the decay)

The second thing that decay function do is to calculate the neighborhood range.

The range is decreased by each step too.

In the next cell, we defined the values for the hyper parameters of the problem. First one is the number of the rows in map and the second one is the number of columns, which means that the final map will have 100 neurons.

max_m_dsitance is the maximum number of neighoubing neurons to be considered for the weight update, notice in each step we are going to decrease the range in the decay function.

Max_learning_rate is set to 0.5 which is going to be smaller in every step of the learning.

Max_steps demonstrate that the learning algorithm should run for at most 75000 steps.

In the first line of the code in the next cell we normalize the training data using min max scaler.

Then we initialize the map with a fixed seed. (to get similar result in every run)

The next line is the definition of the main loop of the algorithm.

This for loop would run for 75000 steps and print out a report in every thousand iter.

In every iter, we first calculate the learning rate and neighbourhood range since these two are depend on the step.

After that we select a random sample from our training dataset. (selection part)

Find the winner neuron with respect to the selected neuron.

Update the weight of the winner and its neighbours by delta w which is the learning_rate * (x-w).

After that we print out a finish msg.

In the next cell we assign a labels to the every neuron of the map with respect to it's won inputs.

And after that in the next cell we find out the number of mapped input to each neuron, this enables us to plot the heatmap of the map. Then we created a heatmap based on these information to visualize the actual map that we learned from the training input set.

In the final set we normalize the test data. (because we used normalized train data)

Then for each test sample we predict a label and store this prediction. In the last line we compare our predictions over test data with the actual test labels and we can see that we had 100 percent accuracy.

Sorry, the report could have been better, but I caught cold and I couldn’t write a better one.