27/4/2017

Joseph Tran

Neural Networks and Fuzzy Logic

Assignment 1

Table of Contents

[Question 1.1. Discrete Perceptron training 3](#_Toc481007518)

[a) Calculate the final weight 4](#_Toc481007519)

[b) Show that the final weight provides the correct classification of the entire training set 5](#_Toc481007520)

[c) Plot the pattern error curve 7](#_Toc481007521)

[d) Plot the cycle error curve 8](#_Toc481007522)

[Question 1.2. Continuous Perceptron training 9](#_Toc481007523)

[a) Calculate w7 9](#_Toc481007524)

[b) Calculate the weight vector w301 after 50 cycles 9](#_Toc481007525)

[c) Plot the cycle error curve 10](#_Toc481007526)

[d) How would w7 and w301 classify the entire training set? 11](#_Toc481007527)

[Question 2.1. Flight simulation 12](#_Toc481007528)

[2.1.1. Find R = M X A 12](#_Toc481007529)

[2.1.2. Find A1a = M1 ο R via max-min composition 13](#_Toc481007530)

[2.1.3. Find A1b = M1 ο R via sum-product composition 14](#_Toc481007531)

[Question 2.2. Laser Beam Alignment 15](#_Toc481007532)

[2.2.1. Calculate the defuzzified voltage output via Mean of Maximum (MOM) 15](#_Toc481007533)

[a) Fuzzification of E 15](#_Toc481007534)

[b) Fuzzification of CE 15](#_Toc481007535)

[c) Defuzzified output voltage 15](#_Toc481007536)

[2.2.2. Calculate the defuzzified voltage output via Centre of Area (COA) 15](#_Toc481007537)

[a) Fuzzification of E and CE 15](#_Toc481007538)

[b) Total Area 15](#_Toc481007539)

[c) Total Moment 15](#_Toc481007540)

[d) Defuzzified output voltage 15](#_Toc481007541)

[Appendix 16](#_Toc481007542)

[Question 1.1.c Pattern error table 16](#_Toc481007543)

[Question 1.1.c and 1.1.d error plotting code 18](#_Toc481007544)

[Question 1.2.a 1.2.b continuous perceptron weight generation code 20](#_Toc481007545)

[Question 1.2.a and 1.2.b weight generation 22](#_Toc481007546)

[Question 1.2.d validating the training set code 24](#_Toc481007547)

# Question 1.1. Discrete Perceptron training

Code behind Question 1.1. is located in the Appendix under Question 1.1. Discrete Perceptron training code.

## Calculate the final weight

The code in the Appendix provides the following output:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 0.2309 | 0.3087 | 0.2150 | 0.3923 | 0.3892 | 0.3892 | 0.3892 | 0.4886 | 0.4886 | 0.4886 |
| 0.5839 | 0.6325 | 0.5857 | 0.7236 | 0.7128 | 0.7128 | 0.7128 | 0.7749 | 0.7749 | 0.7749 |
| 0.8436 | 0.8436 | 0.2345 | 0.2936 | 0.2812 | 0.2812 | 0.2812 | 0.2812 | 0.2812 | 0.2812 |
| 0.4764 | 0.4861 | 0.0644 | 0.1235 | 0.1204 | 0.1204 | 0.1204 | 0.1329 | 0.1329 | 0.1329 |
| -0.6475 | -0.5502 | -1.0188 | -0.8218 | -0.8372 | -0.8372 | -0.8372 | -0.7129 | -0.7129 | -0.7129 |

Note that by the 8th weight (which is essentially the 2nd pattern on its second time trip into the neural network) is providing a weight that does not change. This is seen on the 9th and 10th weight.

Hence the 10th weight is:

|  |
| --- |
| 10 |
| 0.4886 |
| 0.7749 |
| 0.2812 |
| 0.1329 |
| -0.7129 |

## Show that the final weight provides the correct classification of the entire training set

To determine if the final weight provides the correct classification for the training set above, additional code was established to run the final weight against all the inputs and the actual output is measured against the expected output via subtraction.

Anything that does not fill will not be zero.

The validation code is below:

clc;

clear;

finalWeight = [0.4886, 0.7749, 0.2812, 0.1329, -0.7129];

% copied from q11

% augmented input vectors

x1 = [0.8, 0.5, 0, 0.1, 1];

x2 = [0.2, 0.1, 1.3, 0.9, 1];

x3 = [0.9, 0.7, 0.3, 0.3, 1];

x4 = [0.2, 0.7, 0.8, 0.2, 1];

x5 = [1, 0.8, 0.5, 0.7, 1];

x6 = [0, 0.2, 0.3, 0.6, 1];

% each of the augmented vectors are placed into a single vector to churn

% through

inputs = [x1; x2; x3; x4; x5; x6]';

% associated outputs

outputs = [1, -1, 1, -1, 1, -1];

patternErrors = validation(inputs, outputs, finalWeight);

disp(patternErrors);

function validation(inputs, outputs, weight)

[~, iCols] = size(inputs);

patternErrors = zeros(1, iCols);

for index = 1:iCols

error = calculateVariation(outputs(:, index), weight, inputs(:, index));

patternErrors(:, index) = error;

end

end

function error = calculateVariation(expectedOutput, weight, input)

actualOutput = sign(weight \* input);

error = (expectedOutput - actualOutput);

end

The results are as follows:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Inputs | 1 | 2 | 3 | 4 | 5 | 6 |
| Errors | 0 | 0 | 0 | 0 | 0 | 0 |

As noted that when placed back into a section of a system (one that does not have a feedforward mechanism) it meets all the expected outputs with zero errors. This shows that the final weight as mentioned above provides the correct classification of the training data.

## Plot the pattern error curve

Calculating the pattern error curve is simply determining the error at which the system is evolving at.

This can be calculated via the equation of:

Where:

* Ep is the p-th error step, where p is the pattern
* dp is the expected output of the p-th pattern
* zp is the actual output from the activation function from the p-th pattern

In essence it is the variation of the actual output in comparison to the expected output.

As such some modification were made to the code to capture the pattern error run against an activation function and the expected output.

The following is the plotted pattern error. For the full set of plots, please refer to the appendix of the equivalent name.

It is observed that the error fluctuates between the desired output and the actual output. Though it is also noted that as more cycles are progressively made, the less the errors become.

For the code that produced the stated output, please refer to the appendix for Question 1.1.c and 1.1.d Matlab code.

## Plot the cycle error curve

In accordance to the equation of:

Where:

* Ec is the error cycle
* P is the patterns that are processed
* dp is the expected output of the p-th pattern
* zp is the actual output from the p-th pattern
* Ep is the pattern cycle

In essence cycle error curve is assessing how for each “evolved” weight how well it fits the expected output of each input.

The progression is that as the cycles continue, the weights converge on a function that meets the desired output.

|  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Cycles | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| Summed Error | 8 | 6 | 6 | 4 | 2 | 2 | 2 | 2 | 2 | 2 |

The summed error as noted goes down, it arrives at its point where the error for each weight per cycle is reduced.

For the code that produced the stated output, please refer to the appendix for Question 1.1.c and 1.1.d Matlab code.

# Question 1.2. Continuous Perceptron training

The sigmoid activation function found in a discrete perception is replaced with a bipolar logistic function which is as follows:

An alternative representation of the function is:

## Calculate w7

The structure of the code was similar to the code listed in Question 1.1. except with a few changes to the core of the code to accept a continuous activation function (as noted above, a bipolar logistic function was used).

Below is the calculated weight at the 7th cycle of the program.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Cycle | Weight | | | | |
| 7 | 0.4421 | 0.6366 | 0.5754 | 0.3443 | -0.7502 |

For the full set of code and output generated, please refer to the Appendix for further details on the implementation under the heading of: Question 1.2.a and 1.2.b.

## Calculate the weight vector w301 after 50 cycles

Continuing the feedforward process for approximately 50 times, the weight evolution process is as follows:

| Cycles | Weight | | | | |
| --- | --- | --- | --- | --- | --- |
| 50 | 1.7325 | 1.0295 | -0.6459 | -0.1066 | -0.9328 |

For the full set of code and output generated, please refer to the Appendix for further details on the implementation under the heading of: Question 1.2.a and 1.2.b.

## Plot the cycle error curve

Given the equation of the error cycle:

The reference code can be found in the Appendix under the Question 1.2.a and 1.2.b code. The listed output can be found also in the Appendix under the Question 1.2.c cycle error.

Using Appendix Question 1.2.c cycle error data, the following is a plot of the errors across 54 cycles.

Note that the errors approach zero but never reach it due to the continuous activation function in contrast to that of the

## How would w7 and w301 classify the entire training set?

The weights have been modelled after the current set of data. As data is passed through the neural network, the weights would slowly converge on a function that approximates the generalised features of each given input.

Given the nature of a continuous activation function, that the resultant weights that are produced by the neural network converge on an extremely accurate result, the error is therefore minimised to a point that any further classification on the training data will result in overfitting.

Passing both weights back into the inputs to see their classification (see the appendix on the full code that was used to validate this claim), the classification is a lot different to that of a discrete perceptron and that mainly is derived from the fact that a continuous activation function is used, meaning that the outputs will always approach but never arrive at an exact result.

This means that the subsequent “classifications” will approach the desired values and this is shown in the data collected below from the code in the Appendix:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| Desired variation from output | 0 | 0 | 0 | 0 | 0 | 0 |
| w7 variation | 1.0219 | -1.2260 | 0.8174 | -1.1553 | 0.6504 | -0.8788 |
| w301 variation | 0.5548 | -0.3897 | 0.4915 | -0.8009 | 0.4538 | -0.5440 |

As noted that the variations are way off in the first 7 cycles (as the closer the variation is to 0, the more accurate it is). Though as shown below in the 301st cycle, the variation has decreased and is a lot closer to 0 than when it was initially.

# Question 2.1. Flight simulation

## Find R = M X A

The code that was used is as follows:

M = [0, 0.25, 0.75, 1, 0.75, 0.25, 0];

A = [0, 0.3, 0.6, 1, 0.6, 0.3, 0];

R = getRelation(M', A);

function output = getRelation(a, b)

output = min(a, b);

end

R is calculated to be via reading the minimum of both M’ and A:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0.725 | 0.73 | 0.735 | 0.74 | 0.745 | 0.75 | 0.755 |
| 8350 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 8400 | 0 | 0.25 | 0.25 | 0.25 | 0.25 | 0.25 | 0 |
| 8450 | 0 | 0.3 | 0.6 | 0.75 | 0.6 | 0.3 | 0 |
| 8500 | 0 | 0.3 | 0.6 | 1 | 0.6 | 0.3 | 0 |
| 8550 | 0 | 0.3 | 0.6 | 0.75 | 0.6 | 0.3 | 0 |
| 8600 | 0 | 0.25 | 0.25 | 0.25 | 0.25 | 0.25 | 0 |
| 8650 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

## Find A1a = M1 ο R via max-min composition

Using the R matrix that was calculated from the previous question, it is reused to calculate A1a. The code is as follows.

M1 = [0, 0.5, 0.8, 1, 0.6, 0.2, 0];

A1a = maxMinComposition(M1', R);

function output = maxMinComposition(a, b)

output = max(min(a,[],2),... % read the rows of a

max(b,[],1)); % read the columns of b

end

A1a is calculated to be:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | 0.725 | 0.73 | 0.735 | 0.74 | 0.745 | 0.75 | 0.755 |
| 8350 | 0 | 0.3 | 0.6 | 1 | 0.6 | 0.3 | 0 |
| 8400 | 0.5 | 0.5 | 0.6 | 1 | 0.6 | 0.5 | 0.5 |
| 8450 | 0.8 | 0.8 | 0.8 | 1 | 0.8 | 0.8 | 0.8 |
| 8500 | 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 8550 | 0.6 | 0.6 | 0.6 | 1 | 0.6 | 0.6 | 0.6 |
| 8600 | 0.2 | 0.3 | 0.6 | 1 | 0.6 | 0.3 | 0.2 |
| 8650 | 0 | 0.3 | 0.6 | 1 | 0.6 | 0.3 | 0 |

The first column are the various altitudes and the first row are the various “machs” or speeds near the speed of sound.

Each row of M1’ and R have their minimum values calculated and then combined via a max comparison between the two respective rows to form each cell of this matrix.

## Find A1b = M1 ο R via sum-product composition

Each column of a transposed M1 matrix is summed up and each column of the R matrix is summed up and then multiplied together to form A2.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Mach | 0.725 | 0.73 | 0.735 | 0.74 | 0.745 | 0.75 | 0.755 |
|  | 0 | 4.34 | 7.13 | 9.3 | 7.13 | 4.34 | 0 |

# Question 2.2. Laser Beam Alignment

## Calculate the defuzzified voltage output via Mean of Maximum (MOM)

Given e = 3.2 and ce = -0.47

### Fuzzification of E

### Fuzzification of CE

### Defuzzified output voltage

## Calculate the defuzzified voltage output via Centre of Area (COA)

Given e = 3.2 and ce = -0.47

### Fuzzification of E and CE

### Total Area

### Total Moment

### Defuzzified output voltage

# Appendix

## Question 1.1. Discrete Perceptron Training code

clc;

clear;

% augmented input vectors

x1 = [0.8, 0.5, 0, 0.1, 1];

x2 = [0.2, 0.1, 1.3, 0.9, 1];

x3 = [0.9, 0.7, 0.3, 0.3, 1];

x4 = [0.2, 0.7, 0.8, 0.2, 1];

x5 = [1, 0.8, 0.5, 0.7, 1];

x6 = [0, 0.2, 0.3, 0.6, 1];

% each of the augmented vectors are placed into a single vector to churn

% through

y = [x1; x2; x3; x4; x5; x6]';

% associated outputs

d = [1, -1, 1, -1, 1, -1];

% given lambda

lambda = 1.5;

% given cycles

cycles = 10;

% starting weight

w = [0.2309, 0.5839, 0.8436, 0.4764, -0.6475]';

% a counter for the cycles to be measure against

inputCounter = 1;

% setting up the output matrix

[dRows, dCols] = size(x1');

output = zeros(dRows, cycles);

cycleErrors = zeros(1, (cycles - mod(cycles, 6))/6 \* 2);

cycleErrorIndex = 1;

for index = 1:cycles

output(:, index) = w;

[w, cycleError] = variablecorrection(w, lambda, y(:, inputCounter), d(:, inputCounter));

cycleErrors(:, cycleErrorIndex) = cycleErrors(:, cycleErrorIndex) + (cycleError)^2;

inputCounter = inputCounter + 1;

if inputCounter > size(d)

inputCounter = 1;

cycleErrorIndex = cycleErrorIndex + 1;

end

end

disp(output);

disp(cycleErrors);

% weight correction formula given

function [output, error] = variablecorrection(w, lambda, y, d)

error = (d - sign(w' \* y));

output = w + 0.5 \* (lambda \* abs(w' \* y) / (y' \* y))...

\* error \* y;

end

## Question 1.1.c Pattern error table

| Steps | Pattern | Variation |
| --- | --- | --- |
| 1 | 1 | 2 |
| 2 | 2 | 2 |
| 3 | 3 | 0 |
| 4 | 4 | 2 |
| 5 | 5 | 0 |
| 6 | 6 | 2 |
| 7 | 1 | 0 |
| 8 | 2 | 2 |
| 9 | 3 | 0 |
| 10 | 4 | 2 |
| 11 | 5 | 0 |
| 12 | 6 | 2 |
| 13 | 1 | 2 |
| 14 | 2 | 0 |
| 15 | 3 | 2 |
| 16 | 4 | 0 |
| 17 | 5 | 2 |
| 18 | 6 | 0 |
| 19 | 1 | 2 |
| 20 | 2 | 0 |
| 21 | 3 | 0 |
| 22 | 4 | 2 |
| 23 | 5 | 0 |
| 24 | 6 | 0 |
| 25 | 1 | 2 |
| 26 | 2 | 0 |
| 27 | 3 | 0 |
| 28 | 4 | 0 |
| 29 | 5 | 0 |
| 30 | 6 | 0 |
| 31 | 1 | 2 |
| 32 | 2 | 0 |
| 33 | 3 | 0 |
| 34 | 4 | 0 |
| 35 | 5 | 0 |
| 36 | 6 | 0 |
| 37 | 1 | 2 |
| 38 | 2 | 0 |
| 39 | 3 | 0 |
| 40 | 4 | 0 |
| 41 | 5 | 0 |
| 42 | 6 | 0 |
| 43 | 1 | 0 |
| 44 | 2 | 0 |
| 45 | 3 | 0 |
| 46 | 4 | 2 |
| 47 | 5 | 0 |
| 48 | 6 | 0 |
| 49 | 1 | 0 |
| 50 | 2 | 0 |
| 51 | 3 | 0 |
| 52 | 4 | 2 |
| 53 | 5 | 0 |
| 54 | 6 | 0 |
| 55 | 1 | 0 |
| 56 | 2 | 0 |
| 57 | 3 | 0 |
| 58 | 4 | 2 |
| 59 | 5 | 0 |
| 60 | 6 | 0 |

## Question 1.1.c and 1.1.d error plotting code

clc;

clear;

allWeights = [0.230900000000000,0.308704210526316,0.214990815418829,0.392280039358605,0.389201861254868,0.389201861254868,0.389201861254868,0.488622034833465,0.488622034833465,0.488622034833465;0.583900000000000,0.632527631578947,0.585670934025204,0.723562552645030,0.712788929281949,0.712788929281949,0.712788929281949,0.774926537768572,0.774926537768572,0.774926537768572;0.843600000000000,0.843600000000000,0.234462931801334,0.293559339781260,0.281246627366310,0.281246627366310,0.281246627366310,0.281246627366310,0.281246627366310,0.281246627366310;0.476400000000000,0.486125526315790,0.0644152483320978,0.123511656312023,0.120433478208286,0.120433478208286,0.120433478208286,0.132860999905611,0.132860999905611,0.132860999905611;-0.647500000000000,-0.550244736842105,-1.01881171237954,-0.821823685779789,-0.837214576298476,-0.837214576298476,-0.837214576298476,-0.712939359325229,-0.712939359325229,-0.712939359325229];

% copied from q11

% augmented input vectors

x1 = [0.8, 0.5, 0, 0.1, 1];

x2 = [0.2, 0.1, 1.3, 0.9, 1];

x3 = [0.9, 0.7, 0.3, 0.3, 1];

x4 = [0.2, 0.7, 0.8, 0.2, 1];

x5 = [1, 0.8, 0.5, 0.7, 1];

x6 = [0, 0.2, 0.3, 0.6, 1];

% each of the augmented vectors are placed into a single vector to churn

% through

inputs = [x1; x2; x3; x4; x5; x6]';

% associated outputs

outputs = [1, -1, 1, -1, 1, -1];

[cycleError, patternError] = validation(allWeights, inputs, outputs);

function [output, patternError] = validation(weights, inputs, outputs)

% for each weight, calculate error from inputs

output = zeros(1, 10);

weightedIndex = 1;

patternError = zeros(1, 60);

patternIndex = 1;

for weightIndex = 1:10

sumErrors = 0;

if weightIndex == 11

weightedIndex = 1;

end

for inputIndex = 1:6

currentError = calculateVariation(outputs(:,inputIndex),...

weights(:,weightedIndex),...

inputs(:,inputIndex));

sumErrors = sumErrors + currentError;

patternError(:, patternIndex) = currentError;

patternIndex = patternIndex + 1;

disp(["inputIndex",inputIndex, currentError]);

disp(inputs(:,inputIndex)');

end

weightedIndex = weightedIndex + 1;

output(:,weightIndex) = sumErrors;

end

end

function error = calculateVariation(expectedOutput, weight, input)

actualOutput = sign(weight' \* input);

error = 0.5\*(expectedOutput - actualOutput)^2;

end

## Question 1.2.a 1.2.b continuous perceptron weight generation code

% see tut1 for assistance

clc;

clear;

% augmented input vectors

x1 = [0.8, 0.5, 0, 0.1, 1];

x2 = [0.2, 0.1, 1.3, 0.9, 1];

x3 = [0.9, 0.7, 0.3, 0.3, 1];

x4 = [0.2, 0.7, 0.8, 0.2, 1];

x5 = [1, 0.8, 0.5, 0.7, 1];

x6 = [0, 0.2, 0.3, 0.6, 1];

% each of the augmented vectors are placed into a single vector to churn

% through

y = [x1; x2; x3; x4; x5; x6]';

% associated outputs

d = [1, -1, 1, -1, 1, -1];

% given learning constant

learningConstant = 0.25;

% given cycles

cycles = 50;

% starting weight

w = [0.2309, 0.5839, 0.8436, 0.4764, -0.6475]';

% a counter for the cycles to be measure against

inputCounter = 1;

% setting up the output matrix

[dRows, dCols] = size(x1');

output = zeros(dRows, cycles);

% note that there are six patterns

cycleErrors = zeros(1, cycles/6);

cycleIndex = 1;

for index = 1:cycles

output(:, index) = w;

[w, cycleError] = continuousCorrection(w, learningConstant, y(:, inputCounter), d(:, inputCounter));

cycleErrors(:, cycleIndex) = cycleErrors(:, cycleIndex) + cycleError^2;

inputCounter = inputCounter + 1;

if inputCounter > size(d)

%disp([inputCounter, index, inputCounter > size(d), cycleIndex]);

inputCounter = 1;

cycleErrors(:, cycleIndex) = 0.5 \* cycleErrors(:, cycleIndex);

cycleIndex = cycleIndex + 1;

end

end

disp(output);

disp(cycleErrors);

% weight correction formula given, need to look at activation functions

function [outputWeight, error] = continuousCorrection(weight, learningConstant, input, expectedValue)

v = weight' \* input;

z = (2 / (1 + exp(-v))) - 1;

error = expectedValue - z;

rate = 0.5\*(1 - z^2);

r = error \* rate;

outputWeight = weight + learningConstant \* r \* input;

end

## Question 1.2.a and 1.2.b weight generation

This is every weight that has been cycled through via the code referenced on the previous page.

| Cycle | Weight | | | | |
| --- | --- | --- | --- | --- | --- |
| 1 | 0.2309 | 0.5839 | 0.8436 | 0.4764 | -0.6475 |
| 2 | 0.3367 | 0.6500 | 0.8436 | 0.4896 | -0.5153 |
| 3 | 0.3089 | 0.6361 | 0.6635 | 0.3649 | -0.6539 |
| 4 | 0.3972 | 0.7048 | 0.6929 | 0.3943 | -0.5558 |
| 5 | 0.3676 | 0.6012 | 0.5745 | 0.3647 | -0.7038 |
| 6 | 0.4421 | 0.6608 | 0.6117 | 0.4169 | -0.6293 |
| 7 | 0.4421 | 0.6366 | 0.5754 | 0.3443 | -0.7502 |
| 8 | 0.5443 | 0.7004 | 0.5754 | 0.3571 | -0.6225 |
| 9 | 0.5147 | 0.6856 | 0.3831 | 0.2240 | -0.7705 |
| 10 | 0.6046 | 0.7556 | 0.4131 | 0.2539 | -0.6706 |
| 11 | 0.5761 | 0.6557 | 0.2990 | 0.2254 | -0.8132 |
| 12 | 0.6575 | 0.7209 | 0.3398 | 0.2825 | -0.7317 |
| 13 | 0.6575 | 0.7003 | 0.3089 | 0.2208 | -0.8345 |
| 14 | 0.7542 | 0.7608 | 0.3089 | 0.2329 | -0.7137 |
| 15 | 0.7278 | 0.7476 | 0.1370 | 0.1138 | -0.8459 |
| 16 | 0.8140 | 0.8146 | 0.1657 | 0.1426 | -0.7501 |
| 17 | 0.7874 | 0.7213 | 0.0591 | 0.1159 | -0.8834 |
| 18 | 0.8691 | 0.7867 | 0.0999 | 0.1731 | -0.8017 |
| 19 | 0.8691 | 0.7691 | 0.0736 | 0.1204 | -0.8896 |
| 20 | 0.9581 | 0.8247 | 0.0736 | 0.1315 | -0.7783 |
| 21 | 0.9371 | 0.8143 | -0.0626 | 0.0372 | -0.8831 |
| 22 | 1.0154 | 0.8752 | -0.0365 | 0.0633 | -0.7961 |
| 23 | 0.9904 | 0.7875 | -0.1367 | 0.0382 | -0.9213 |
| 24 | 1.0671 | 0.8489 | -0.0983 | 0.0920 | -0.8446 |
| 25 | 1.0671 | 0.8334 | -0.1215 | 0.0455 | -0.9219 |
| 26 | 1.1471 | 0.8834 | -0.1215 | 0.0555 | -0.8219 |
| 27 | 1.1311 | 0.8754 | -0.2257 | -0.0166 | -0.9021 |
| 28 | 1.2002 | 0.9292 | -0.2027 | 0.0065 | -0.8252 |
| 29 | 1.1765 | 0.8461 | -0.2977 | -0.0173 | -0.9440 |
| 30 | 1.2465 | 0.9021 | -0.2627 | 0.0317 | -0.8740 |
| 31 | 1.2465 | 0.8882 | -0.2836 | -0.0101 | -0.9436 |
| 32 | 1.3177 | 0.9327 | -0.2836 | -0.0012 | -0.8546 |
| 33 | 1.3054 | 0.9265 | -0.3637 | -0.0566 | -0.9163 |
| 34 | 1.3663 | 0.9739 | -0.3433 | -0.0363 | -0.8486 |
| 35 | 1.3436 | 0.8946 | -0.4339 | -0.0590 | -0.9618 |
| 36 | 1.4072 | 0.9455 | -0.4021 | -0.0145 | -0.8982 |
| 37 | 1.4072 | 0.9328 | -0.4212 | -0.0526 | -0.9618 |
| 38 | 1.4705 | 0.9723 | -0.4212 | -0.0447 | -0.8827 |
| 39 | 1.4609 | 0.9675 | -0.4836 | -0.0879 | -0.9307 |
| 40 | 1.5150 | 1.0096 | -0.4656 | -0.0699 | -0.8706 |
| 41 | 1.4934 | 0.9340 | -0.5520 | -0.0915 | -0.9786 |
| 42 | 1.5513 | 0.9804 | -0.5230 | -0.0510 | -0.9206 |
| 43 | 1.5513 | 0.9686 | -0.5406 | -0.0861 | -0.9792 |
| 44 | 1.6080 | 1.0041 | -0.5406 | -0.0790 | -0.9084 |
| 45 | 1.6004 | 1.0002 | -0.5900 | -0.1133 | -0.9464 |
| 46 | 1.6489 | 1.0380 | -0.5738 | -0.0971 | -0.8925 |
| 47 | 1.6283 | 0.9660 | -0.6562 | -0.1177 | -0.9954 |
| 48 | 1.6815 | 1.0085 | -0.6296 | -0.0804 | -0.9422 |
| 49 | 1.6815 | 0.9976 | -0.6459 | -0.1130 | -0.9965 |
| 50 | 1.7325 | 1.0295 | -0.6459 | -0.1066 | -0.9328 |

## Question 1.2.c cycle error production

This is the error generated at each cycle of the evolution of the neural network.

| Cycle | Cumulative Patterns | Error |
| --- | --- | --- |
| 1 | 6 | 3.6064 |
| 2 | 12 | 3.0135 |
| 3 | 18 | 2.4616 |
| 4 | 24 | 2.0177 |
| 5 | 30 | 1.6858 |
| 6 | 36 | 1.4391 |
| 7 | 42 | 1.2520 |
| 8 | 48 | 1.1065 |
| 9 | 50 | 0.5175 (0.9904) |

Note that there the patterns are 6, so every 6 patterns, the cycle error would be captured, hence the full 9th cycle (requiring the 54th pattern to cycle through) is 0.5175 since there isn’t enough data to capture together yet. Hence it is considered an incomplete cycle as noted by the number of patterns it has cycled through.

Though, the second number in the brackets is what would have happened if you cycled through 54 cycles instead of the listed 50.

## Question 1.2.d validating the training set code

clc;

clear;

seventhWeight = [0.442131840727600;0.636597872834342;0.575441294301345;0.344346399188018;-0.750222787022720];

finalWeight = [1.73253526080533;1.02953015227645;-0.645882485236181;-0.106634592428199;-0.932766308404683];

% augmented input vectors

x1 = [0.8, 0.5, 0, 0.1, 1];

x2 = [0.2, 0.1, 1.3, 0.9, 1];

x3 = [0.9, 0.7, 0.3, 0.3, 1];

x4 = [0.2, 0.7, 0.8, 0.2, 1];

x5 = [1, 0.8, 0.5, 0.7, 1];

x6 = [0, 0.2, 0.3, 0.6, 1];

% each of the augmented vectors are placed into a single vector to churn

% through

y = [x1; x2; x3; x4; x5; x6]';

% associated outputs

d = [1, -1, 1, -1, 1, -1];

[yRows, yCols] = size(y);

seventhWeightErrors = zeros(1, yCols);

finalWeightErrors = zeros(1, yCols);

for index = 1:yCols

finalWeightErrors(:, index) = validation(finalWeight, y(:,index), d(:, index));

seventhWeightErrors(:, index) = validation(seventhWeight, y(:,index), d(:, index));

end

disp(finalWeightErrors);

disp(seventhWeightErrors);

function error = validation(weight, input, expectedValue)

v = weight' \* input;

z = (2 / (1 + exp(-v))) - 1;

disp([expectedValue, z]);

error = expectedValue - z;

end