

## 2.1.0 – Download Python Files – Complete

### 2.1.1 – Implement truth table from HW1.4 using hard-activation perceptron learning rule.

For this, I explore changing the value for the learning constant alpha as can be seen in the following table:

Alpha	# of Iterations	Total Error	Final Weights
0.03	12	0.0	0.43 0.13 0.4 -0.59
0.3	3	0.0	0.4 -0.2 0.7 -0.8
3	9	0.0	10.0 1.0 7.0 -12.5
30	3	0.0	31 1 31 -59

We can see that in each case, Total Error arrives to 0.0 in a relative few number of errors. This can be attributed to the hard activation function used here, which may or may not result in large margins when Total Error is 0.0.

### 2.1.2 – Implement using soft-activation function

When we change the activation function to a hard activation function however, we notice that many of the alpha configurations result in high number of iterations (noting 100 iterations is the maximum allowed by our scripts). Surprisingly however, with Alpha = 30, only 3 iterations are required to reach Total Error of 0.0. We can assume with initialization of weights to 1 and alpha at 30, the algorithm made a lucky jump to a “perfect” solution.

Alpha	# of Iterations	Total Error	Final Weights
0.03	100	0.36396	1.13 -0.21 1.12 -1.71
0.3	100	0.012412	3.1 -0.2 3.08 -4.63
3	25	0.000955	4.5 -0.2 4.45 -6.73
30	3	0.0	31.0 1.0 31.0 -55.42

### 2.1.3 – Implement with Delta training rule

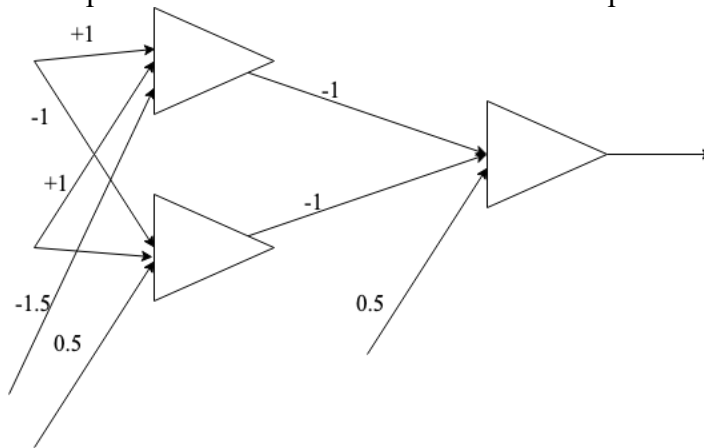
Finally, we see with the delta training rule a similar method to the soft activation function. This time however, we have two parameters which we can interact with, alpha and gain. We can see as gain decreases in this case, we appear to have worse learning results or higher Total Error given a number of iteration.

Alpha	Gain (k)	# of Iterations	Total Error	Final Weights
0.022	1.0	100	1.403353	1.15 -0.12 1.14 -1.73
0.22	1.0	100	0.113424	2.62 -0.11 2.61 -3.92
<b>2.2</b>	<b>1.0</b>	<b>39</b>	<b>0.000936</b>	<b>6.7 -1.19 7.53 -10.95</b>
22	1.0	100	08.0	1.0 1.0 1.0 -15.28
0.022	0.5	100	3.387994	1.01 -0.22 1.01 -1.68
0.22	0.5	100	0.561794	3.46 -0.26 3.43 -5.16
2.2	0.5	100	0.04014	6.34 -0.2 6.32 -9.49
22	0.5	100	8.0	-4.48 -6.72 4.85 -17.16

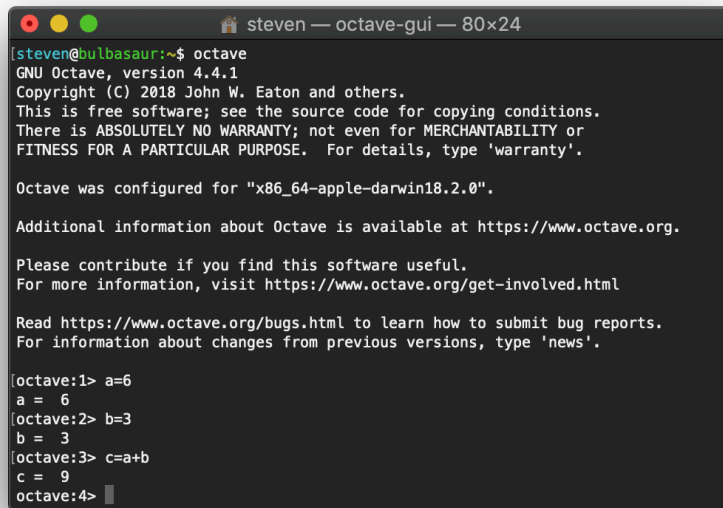
0.022	0.1	100	8.144669	0.81 0.4 0.81 -0.15
0.22	0.1	100	5.033809	2.01 -1.43 2.0 -4.48
2.2	0.1	100	1.450447	11.35 -1.63 11.24 -16.88
22	0.1	100	0.114108	26.19 -1.07 26.07 -39.19

## 2.2 – Design neural network to perform XOR function

The thinking behind this problem goes as follows. We are given up to three neurons in our multilayer network. One neuron must be used for output, while the other two would be used in a previous layer (at the input). With two layers at the input, we can give each of these neurons a specific goal to detect the AND operation of the inputs. For the first neuron, we take the AND of the values as-is. This is used to detect first if both inputs are  $1$ . If so, then this first neuron fires. The second neuron attempts to detect if both inputs are  $0$  and if so, outputs a  $1$  otherwise  $0$ . If we can detect both AND cases, we can invert this to produce the XOR case from the final output neuron.



### 2.3.a – Install Octave – COMPLETE



```
steven — octave-gui — 80x24
[steven@bulbasaur:~]$ octave
GNU Octave, version 4.4.1
Copyright (C) 2018 John W. Eaton and others.
This is free software; see the source code for copying conditions.
There is ABSOLUTELY NO WARRANTY; not even for MERCHANTABILITY or
FITNESS FOR A PARTICULAR PURPOSE. For details, type 'warranty'.

Octave was configured for "x86_64-apple-darwin18.2.0".

Additional information about Octave is available at https://www.octave.org.

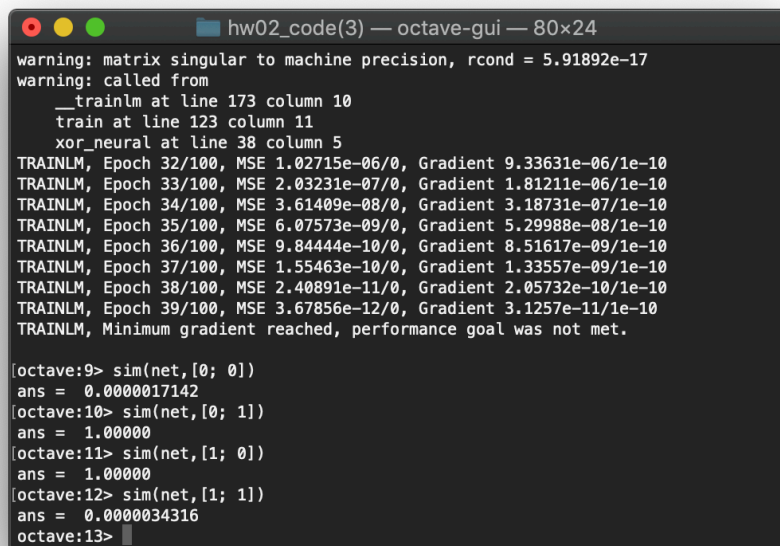
Please contribute if you find this software useful.
For more information, visit https://www.octave.org/get-involved.html

Read https://www.octave.org/bugs.html to learn how to submit bug reports.
For information about changes from previous versions, type 'news'.

octave:1> a=6
a = 6
octave:2> b=3
b = 3
octave:3> c=a+b
c = 9
octave:4>
```

### 2.3.b – Octave XOR Problem

Admittedly, there were some warning that appeared from old versions of `nnet` as well as cases where a matrix internally could not be inverted (matrix was singular), but the training appears to have successfully classified the XOR problem. We can notice in the screenshot below for inputs  $\{0,0\}$  and  $\{1,1\}$ , the output from the network does not given an exact value of 0 as expected, but the results are small enough that they can be understood to be equal to zero.



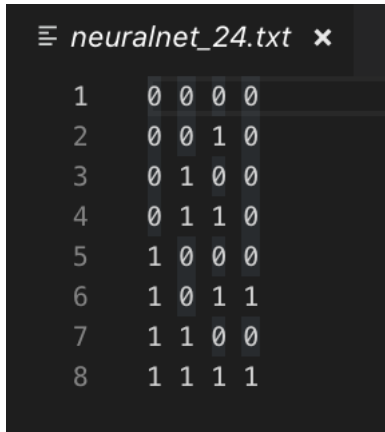
```
hw02_code(3) — octave-gui — 80x24
warning: matrix singular to machine precision, rcond = 5.91892e-17
warning: called from
  __trainlm at line 173 column 10
  train at line 123 column 11
  xor_neural at line 38 column 5
TRAINLM, Epoch 32/100, MSE 1.02715e-06/0, Gradient 9.33631e-06/1e-10
TRAINLM, Epoch 33/100, MSE 2.03231e-07/0, Gradient 1.81211e-06/1e-10
TRAINLM, Epoch 34/100, MSE 3.61409e-08/0, Gradient 3.18731e-07/1e-10
TRAINLM, Epoch 35/100, MSE 6.07573e-09/0, Gradient 5.29988e-08/1e-10
TRAINLM, Epoch 36/100, MSE 9.84444e-10/0, Gradient 8.51617e-09/1e-10
TRAINLM, Epoch 37/100, MSE 1.55463e-10/0, Gradient 1.33557e-09/1e-10
TRAINLM, Epoch 38/100, MSE 2.40891e-11/0, Gradient 2.05732e-10/1e-10
TRAINLM, Epoch 39/100, MSE 3.67856e-12/0, Gradient 3.1257e-11/1e-10
TRAINLM, Minimum gradient reached, performance goal was not met.

octave:9> sim(net,[0; 0])
ans = 0.0000017142
octave:10> sim(net,[0; 1])
ans = 1.00000
octave:11> sim(net,[1; 0])
ans = 1.00000
octave:12> sim(net,[1; 1])
ans = 0.0000034316
octave:13>
```

## 2.4 – Feed-forward Network in Octave

Using the `xor_neural.m` as a base, we are able to edit the input file to contain the truth table. One important note was to ensure bias of 1.0 was not stored in the input file. For whatever reason, bias confused the learning algorithm. I set each layer (hidden and input) to have a single neuron and kept the same activation functions from the `xor_neural.m` example file.

Screenshots of `training.txt` file and code below as requested in the homework. These files are also attached.



1	0	0	0	0
2	0	0	1	0
3	0	1	0	0
4	0	1	1	0
5	1	0	0	0
6	1	0	1	1
7	1	1	0	0
8	1	1	1	1

```
C neuralnet_24.m x
1  #####
2  # Contains: Neural Network for truthtable in HW1.4
3  # Name: 2_4_neuralnet.m
4  # Course Instructor: Milos Manic
5  # Provided by: Course Instructor
6  #
7  # Derivative of: xor_neural.m
8  #####
9
10
11  clear all
12
13  #load Truth Table from .txt file#
14  xorTT=load("neuralnet_24.txt");
15  dim=size(xorTT);
16  num_entries=dim(1);
17
18  #Seperate input matrix and output vector from Truth Table#
19  in=xorTT(:,1:end-1);
20  out=xorTT(:,end);
21
22  #Transpose them for Neural Toolbox function usage#
23  in=in';
24  out=out';
25
26  #minmax is required for creating a Feed-Forward Network in Octave
27  minmax_in=minmax(in);
28
29  #Create Feed-Forward Network using newff function#
30  #<For additional help, type "help newff"(without quotes) on the Octave prompt#
31  MLPnet=newff(minmax_in,[1 1],{"tansig","logsig"},"trainlm","learngdm","mse");
32
33  #Save Neural Network Structure in a text file
34  saveMLPStruct(MLPnet,"neuralnet_struct_24.txt");
35
36  #Show training performance every 1 step#
37  MLPnet.trainParam.show = 1;
38
39  #Train the neural network#
40  [net]=train(MLPnet,in,out);
41
42
43  for a = [0,1]
44      for b = [0,1]
45          for c = [0,1]
46              |      sprintf("sim(net,[%d;%d;%d]) = %f", a,b,c, sim(net,[a;b;c]))
47              end
48          end
49  end
50  |
```

## 2.5 – Extra Credit, design XOR network with only two neurons

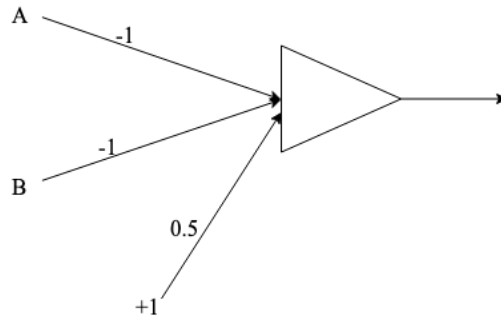
To accomplish this, we must consider XOR into logical parts. First, we consider XOR to be the same as  $\sim \text{EQUALS}$  such that

$\text{XOR}(A,B) == \sim \text{EQUALS}(A,B)$  which can be broken down into two parts which we can design our neurons to accomplish. Given inputs A,B

$C = (\sim A \text{ and } \sim B)$

$D = \sim(C \text{ or } (A \text{ and } B)) = \text{XOR problem}$

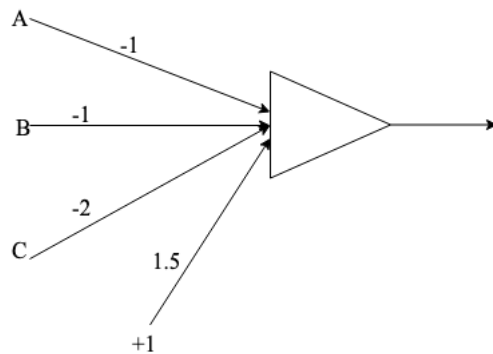
Thus, we first attempt to create a neuron which can solve for C:



Once we solve this, we can use it as an input for a new neuron for D. Note, the truth table for D would be as follows, though certain rows (such as A=0,B=0,C=0) are impossible because C would never evaluate to the given neuron for given values of A and B, however the rows are presented for completeness. The orange highlighted rows are those which would not happen, however we do evaluate the neuron for these values.

A	B	C	$\sim(C \text{ or } (A \text{ and } B))$
1	1	1	0
1	1	0	0
1	0	1	0
1	0	0	1
0	1	1	0
0	1	0	1
0	0	1	0
0	0	0	1

From this, we create a neuron:



Which we can use as the final output layer. If we stick these two neurons together, we get the following network where inputs A and B are inputs into both neurons, but the first neuron is an additional input into the second neuron keeping the network as a directed acyclic graph.

