On Machine Intelligence

WHY ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING ARE CHANGING THE WORLD

EXAMPLES / MACHINE LEARNING

A Simple Neural Network In Octave – Part 1

DECEMBER 19, 2015 STEPHEN OMAN 1 COMMENT

Getting started with neural networks can seem to be a daunting prospect, even if you have some programming experience. The many examples on the Internet dive straight into the mathematics of what the neural network is doing or are full of jargon that can make it a little difficult to understand what's going on, not to mention how to implement it in actual code.

So I decided to write a post to help myself understand the mechanics and it turns out that it will require a few parts to get through it!

Anyway, to start with, there is a great free numerical computation package called Octave that you can use to play around with Machine Learning concepts. Octave itself does not know about neural networks, but it does know how to do fast matrix multiplication. This important feature of Octave will be made clear later.

You can find out more and download the Octave software here:

https://www.gnu.org/software/octave/

A nice toy problem to start with is the XOR problem. XOR means “exclusive OR’ and it is best explained in a table:

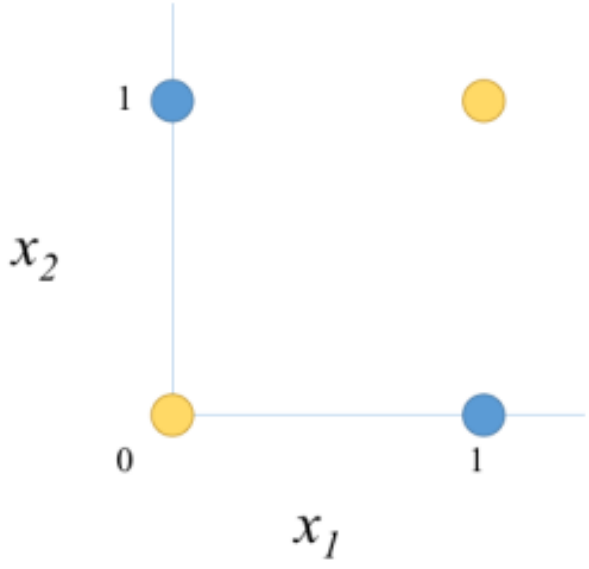
Given this input		Produce this output
x_1	x_2	y
0	0	0
0	1	1
1	0	1

1	1	0

What the table shows is that there are two inputs (labelled x_1 and x_2) and one output (labelled y). When x_1 and x_2 are both set to 0, the output we expect is also 0. Similarly, when x_1 and x_2 are both set to 1, the output is also 0. However, when x_1 and x_2 are set to different inputs, then the output will be 1.

The challenge is to build a neural network that can successfully learn to produce the correct output given the four different inputs in the table.

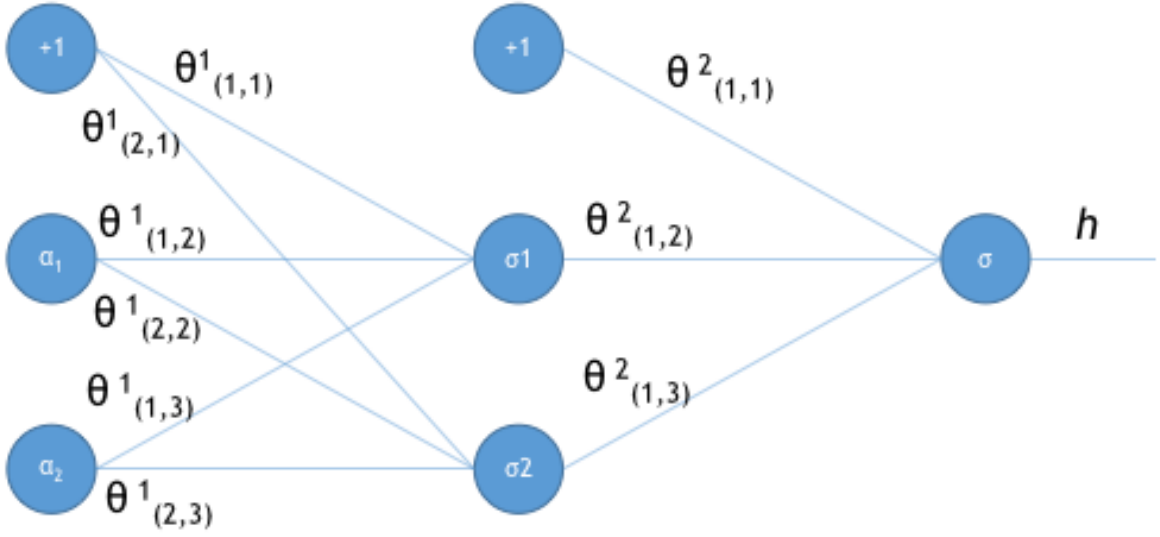
Let's have a quick look at a graphical representation of the problem:



The graph shows the two inputs x_1 and x_2 on their respective axes. Where x_1 and x_2 have the same value, the graph shows yellow circles and similarly where x_1 and x_2 are different, the graph shows blue circles.

There's an important constraint that the graph shows clearly. It isn't possible to draw a single straight line across the graph so that the yellow circles are on one side and the blue circles are on the other side. This is called "linear-separability". So the XOR problem is not linearly separable, which means that we are going to need a multi-layer neural network to solve it.

The diagram below shows a typical configuration for a neural network that can be trained to solve the XOR problem.



There are a number of things to note about about this particular network. Firstly, the inputs in

the table above (x_1 and x_2), are mapped directly onto the nodes represented by a_1 and a_2. Secondly, this first layer of nodes also contains a bias node, that has its output always set to +1. Thirdly, the nodes in the middle of the diagram are collectively called the hidden nodes or hidden layer. It also contains a bias node set to +1. The outputs of the other nodes are labelled with a small greek letter sigma, which will become clearer below. Lastly, the output of the network is labelled h.

It's useful to represent the inputs as a vector (a one-dimensional matrix) that looks like this:

$$A^1 = \begin{bmatrix} 1 \\ a_1 \\ a_2 \end{bmatrix}$$

This can be translated directly into Octave as:

```
A1 = [1; 0; 0];
```

In the above example, 1 represents the bias node, and the two zeros represent the first row of the variables from our table above replacing the a_1 and a_2 in the vector. You can replace the two zeros with values from other rows on the table to see what happens to the output after we've built up the network.

The links from the nodes in the first layer to the nodes in the second layer have weights associated with them, denoted by the letter theta (along with a superscript 1) in the diagram of our network. Similarly, the weights in layer 2 are also shown with a superscript 2.

The subscript numbers identify the nodes at either end of the link, in the form (i,j), where i is the node receiving the signal and j is the node sending the signal. This is slightly counter-intuitive, where the normal expectation is that the signal moves from i to j. The reason for this is that it is possible to represent all the weights at a given layer in a single matrix that looks like this:

$$\Theta^1 = \begin{bmatrix} \theta_{1,1} & \theta_{1,2} & \theta_{1,3} \\ \theta_{2,1} & \theta_{2,2} & \theta_{2,3} \end{bmatrix}$$

Typically, the initial values of the weights in a network are set to random values between -1 and +1, since we have no idea what they actually should be. We can do this in Octave as follows:

```
THETA1 = 2*rand(2,3) - 1;
```

And for Θ^2, which has 3 nodes linked to the one final node:

```
THETA2 = 2*rand(1,3) - 1;
```

So now we can simply multiply those matrices together to work out what the input to the second layer is:

$$Z^2=\Theta^1 A^1$$

where Z^2 represents the input to the second layer of nodes. This multiplication will result in another vector (1 dimensional matrix). Our Octave code for this is simply:

```
Z2 = [1; THETA1 * A1];
```

Note that we add the extra 1 as the first element in the vector to represent the bias that will be needed as an input into layer 3.

This is a lot better (and faster) than having to calculate each of these inputs separately. In fact, most machine learning libraries will provide fast matrix multiplication (and other matrix operations), precisely because it is an efficient way to model machine learning strategies.

To calculate the output of layer 2, we must apply a function to the input. The typical function used is called a sigmoid function (represented by the sigma in the network diagram) and it looks like this in Octave:

```
function [result] = sigmoid(x)
    result = 1.0 ./ (1.0 + exp(-x));
end
```

So the output of layer 2 is the sigmoid of the input, or

$$A^2=\sigma(Z^2)$$

which in Octave is:

```
A2 = sigmoid(Z2);
```

We repeat the process for layer 3, multiplying the output of layer 2 by the matrix of weights for

layer 2 to get the input for layer 3 and then getting the sigmoid of the result:

$$Z^3 = \Theta^2 A^2$$

$$h = \sigma(Z^3)$$

The output from the network is then a single value, called our hypothesis (*h*). This is the network's guess at the output given it's input. The Octave code for this is:

```
Z3 = THETA2 * A2;
h = sigmoid(Z3);
```

That's the network fully constructed. We can put any values from the table in the front (putting them in the A1 vector) and see what the output from the network is (*Hypothesis*).

Here is an example of the above code running:

```
GNU Octave, version 3.8.0
Copyright (C) 2013 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-darwin13.0.0".

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

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

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

>> function [result] = sigmoid(x)
  result = 1.0 ./ (1.0 + exp(-x));
end
>> A1 = [1; 0; 0];
>> THETA1 = 2*rand(2,3) - 1;
>> THETA2 = 2*rand(1,3) - 1;
>> Z2 = [1; THETA1 * A1];
>> A2 = sigmoid(Z2);
>> Z3 = THETA2 * A2;
>> h = sigmoid(Z3);
>> h
h =  0.59183
>>
```

It is almost certain that the network will get the wrong answer (outputting a 1 when it should be outputting a 0 and vice versa). The example above shows *h* to be 0.59183 (you may get a completely different value), which is clearly wrong, as for a (0,0) input, we should get an output of 0.

In the next post in this series, we'll look at how to get the network to learn from its mistakes and how it can get much better at outputting correct values.

Here is the next part of this series

Please share this post! Thanks :)

A Simple Neural Network in Octave - Part 2
In "Examples"

Solving XOR with a Neural Network in Python
In "Examples"

Solving XOR with a Neural Network in TensorFlow
In "Artificial Intelligence"

CODE, MACHINE LEARNING ALGORITHMS, OCTAVE

PREVIOUS POST
Poor use for IBM's AI Watson: Predicting Popular Sale Items

NEXT POST
A Simple Neural Network in Octave – Part 2

One thought on "A Simple Neural Network In Octave – Part 1"

2 – Show HN: A simple neural network in Octave to solve the XOR problem says:

REPLY

JANUARY 14, 2016 AT 11:27 PM

[□] Read more here: https://aimatters.wordpress.com/2015/12/19/a-simple-neural-network-in-octave-part-1/ [□]

★ Like

Leave a Reply

Enter your comment here...

ABOUT ME

Stephen Oman

Lifelong coder, interested in artificial intelligence since finding a listing for Eliza in a computer magazine in 1985.B.Sc. Computer Science and M.Sc. (Research) in AI from Trinity College Dublin.

View Full Profile

Follow On Machine Intelligence 18

SOCIAL

ARCHIVE

March 2015 (3)

Follow On Machine Intelligence

FOLLOW BLOG VIA EMAIL

Enter your email address to follow this blog and receive notifications of new posts by email.

Join 1,086 other followers

Enter your email address

FOLLOW

On Machine Intelligence

WHY ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING ARE CHANGING THE WORLD

EXAMPLES / MACHINE LEARNING

A Simple Neural Network in Octave – Part 2

JANUARY 3, 2016 STEPHEN OMAN LEAVE A COMMENT

In the last post in this short series, we looked at how to build a small neural network to solve the XOR problem. The network we built can generate an output for any of the inputs that we gave it from the table. It's most likely that it sometimes gets the wrong answer, because the weights on the links were randomly generated.

Now we'll look at getting the network to learn from it's mistakes so that it gets better each time. In the context of neural networks, learning means adjusting those weights so it gets

better and better at giving the right answers.

The first step is to find out how wrong the network is. This is called the cost or loss of the network. There are several ways to do this from the simple to the complex. A common approach is called “logistic regression”, which avoids problems of the neural network getting stuck in it’s learning.

Recall from our XOR table that there are two possible outputs, either 1 or 0. There will be two different versions of the cost depending on the output:

$$Cost(h_\theta(x),y) = \begin{cases} -log(h_\theta(x)) & if \quad y=1 \\ -log(1-h_\theta(x)) & if \quad y=0 \end{cases}$$

Remember that $h_\theta(x)$ is the hypothesis that the network produces, given the input $x=(x_1,x_2)$ and the weights θ. “log” is the standard mathematical function that calculates the natural logarithm of the value given (hence the name “logistic regression”).

We can combine these two instances together in a single formula that makes it easier and faster to translate into code:

$$Cost(h_\theta(x),y) = -ylog(h_\theta(x))-(1-y)log(1-h_\theta(x))$$

So our code to represent this:

```
y = 0;
J = ((y * log(h)) + ((1 - y) * log(1 - h))) * -1 ;
```

This gives us a good idea how the network is performing on this particular input, with y set to the expected output from the first row of the XOR table.

A better network will have a lower cost, so our goal in training the network is to minimize the cost. The algorithm used to do that is called the back propagation algorithm (or backprop for short).

The first step is to evaluate the error at the output node (layer 3):

$$\delta^3 = h - y$$

Note that the superscript 3 refers to the layer and doesn't mean cubed! The code in Octave is:

```
delta3 = h - y;
```

So far, so straightforward. Backprop takes the error at a particular layer within a network and propagates it backward through the network. In general, the formula for that is:

$$\delta^{l-1} = (\Theta^{(l-1)})^T \delta^l .* g'(z^{(l-1)})$$

This looks a little complicated so let's dissect it to make it easier to understand and translate into Octave code.

Firstly, the *l* superscript on some of the terms in the formula refers to the layer in the network. The next layer we need to work out is layer 2, so *l* = 2. We already know what δ^3 is from the previous code above.

Next, $\Theta^{(l-1)}$ refers to the matrix of weights at the layer *l-1*. The T superscript means to transpose the matrix (basically swap the rows and columns). This makes it easier for us to multiple the two matrices. Octave being what it is makes this very easy for us, so the first part of the formula is easy to translate:

```
THETA2' * delta3
```

Note the apostrophe after THETA2. This is Octave's way of transposing a matrix.

The second part of the formula represents the derivative of the activation function. Fortunately, there is an easy way to calculate that without getting into the complexity of calculus:

$$g'(Z) = Z .* (1 - Z)$$

The ".*" means element-wise multiplication of a matrix. So each term in the first matrix is multiplied by the corresponding term in the second matrix. In Octave, it's almost exactly the same (for layer 2):

```
Z2 .* (1 - Z2)
```

Putting those two parts together then:

```
delta2 = ((THETA2' * delta3) .* (Z2 .* (1 - Z2)))(2:end);
```

Remember that these are matrix operations, so delta2 will be a matrix too. The last part of the

line of code (2:end) means to take the second element of the matrix to the end. The reason this is done is because we are not propagating an error back from the bias node.

At this point, we have the error in the output at layer 3 and layer 2. Since layer 1 is the input layer, there isn't any error there, so we don't need to calculate anything.

The last step in the backprop is to adjust the weights now that we know those errors. This is also quite simple:

$$\delta(\Theta^{l}) = \alpha(\delta^{l+1}A^{l})$$

The formula says that the change to the weights $\Theta^{(l)}$ is the activation values multiplied by the errors, adjusted by the value α, which represents the learning rate. The learning rate governs how large the adjustments to the weights are. It's a bit of a misnomer in that it doesn't relate to how "fast" the network learns, although that can be the effect of having a relatively larger value. For solving the XOR problem, a learning rate of 0.01 is sufficient.

So in Octave:

```
THETA2 = THETA2 - (0.01 * (delta3 * A2'));
THETA1 = THETA1 - (0.01 * (delta2 * A1'));
```

Once we've adjusted the weights in the matrices, the network will give us a slightly different hypothesis for each of our inputs. We can recalculate the cost function above to see how much improvement there is.

If we repeat the calculations, the cost should start reducing and the network will get better and better at producing the desired output.

```
>>
>> y = 0;
>> J = 0;
>> J = J + ((y * log(h)) + ((1 - y) * log(1 - h))) * -1 ;
>> delta3 = h - y;
>> delta2 = ((THETA2' * delta3) .* (Z2 .* (1 - Z2)))(2:end);
>> THETA2 = THETA2 - (0.01 * (delta3 * A2'));
>> THETA1 = THETA1 - (0.01 * (delta2 * A1'));
>> Z2 = [1; THETA1 * A1];
>> A2 = sigmoid(Z2);
>> Z3 = THETA2 * A2;
>> h = sigmoid(Z3);
>> h
h =  0.59054
>>
```

In the next post, we'll pull all the code together and see what happens when we start training the network across all the examples.

The last part of this series is here.

Please share this post! Thanks :)

A Simple Neural Network In Octave - Part 1
In "Examples"

A Simple Neural Network in Octave - Part 3
In "Examples"

Solving XOR with a Neural Network in TensorFlow
In "Artificial Intelligence"

PREVIOUS POST
A Simple Neural Network In Octave – Part 1

NEXT POST
A Simple Neural Network in Octave – Part 3

Leave a Reply

Enter your comment here...

ABOUT ME

Stephen Oman

Lifelong coder, interested in artificial intelligence since finding a listing for Eliza in a computer magazine in 1985.B.Sc. Computer Science and M.Sc. (Research) in AI from Trinity College Dublin.

View Full Profile

SOCIAL

ARCHIVE

March 2015 (3)

FOLLOW BLOG VIA EMAIL

Enter your email address to follow this blog and receive notifications of new posts by email.

Join 1,086 other followers

Enter your email address

Sign me up

On Machine Intelligence

WHY ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING ARE CHANGING THE WORLD

EXAMPLES / MACHINE LEARNING

A Simple Neural Network in Octave – Part 3

JANUARY 6, 2016 STEPHEN OMAN LEAVE A COMMENT

This is the final post in a short series looking at implementing a small neural network to solve the XOR problem in Octave.

In the first post, we looked at how to set up the parameters of the network (the weights between the nodes), feed in an example and get the network to predict some output. The second post looked at implementing the back propagation algorithm, which adjusts those parameters in the network to improve the accuracy of it's outputs.

Now it's time to pull those pieces together so that we can let the network adjust the parameters to get as close to the right output as we desire.

First, we'll set up a function to run all the training examples through the network. Presenting all the training examples once (one after the other) is called an epoch. The goal of this function will be to return the updated parameters after an epoch:

```
function [THETA1_new, THETA2_new] = xor_nn(XOR, THETA1, THETA2, in
```

Just a quick note on the notation and parameters in this function header. The first part after the word "function" specifies what this function returns. In our case, we want new versions of the parameters in the network (represented by THETA1 and THETA2).

The name of the function is "xor_nn" and the parameters to the function are contained within the brackets:

- XOR This is the representation of the training set.
- THETA1 & THETA2 Current values for the parameters in the network.
- init_w=0 This tells the function to initialise the weights in the network
- learn=0 This tells the network to learn from the examples (see below)

- alpha=0.01 This is the learning rate (default value is 0.01)

Note that any parameter that has an “=” sign is optional and will get the default value shown if it is not provided explicitly.

The first step in our function is to check if we need to initialize the weights.

```
if (init_w == 1)
    THETA1 = 2*rand(2,3) - 1;
    THETA2 = 2*rand(1,3) - 1;
endif
```

This is simply the same code as before, inside an “if” block.

Now we initialize the cost variable. This is done for every epoch:

```
J = 0.0;
```

In the previous post, we looked at updating the weights in the network after calculating the cost after the network has processed a single training example. This is a specific type of learning called “online learning”. There is another type of learning called “batch learning”, and this works by updating the weights once after *all* the training examples have been processed. Let’s implement that instead in our function.

We need to record the number of training examples and the total delta across all the training examples (rather than just across one as before). So here are the extra variables required:

```
T1_DELTA = zeros(size(THETA1));
T2_DELTA = zeros(size(THETA2));
m = 0;
```

Remember that THETA1 and THETA2 are matrices, so we need to initialize every element of the matrix, and make our delta matrices the same size. We’ll also use “m” to record the number of training examples that we present to the network in the epoch.

Now lets set up a loop to present those training examples to the network one by one:

```
for i = 1:rows(XOR)
```

This simply says repeat the following block for the same number of rows that exist in our XOR data set.

Now put in the code from Part 1 that processes an input:

```
A1 = [1; XOR(i,1:2)'];
Z2 = THETA1 * A1;
A2 = [1; sigmoid(Z2)];
Z3 = THETA2 * A2;
h = sigmoid(Z3);
```

```
J = J + ( XOR(i,3) * log(h) ) + ( (1 - XOR(i,3)) * log(1 - h) );
m = m + 1;
```

Note the slight change in moving the bias node from the layer input calculation (Z3) to the output from the previous layer (A2). This just makes the code slightly simpler.

Then we add the code from Part 2, inside a test to see if we are in learning mode. The code has been slightly modified as we are implementing batch learning rather than online learning. In batch mode, we want to calculate the errors across all the examples:

```
if (learn == 1)
    delta3 = h - XOR(i,3);
    delta2 = ((THETA2' * delta3) .* (A2 .* (1 - A2)))(2:end);
    T2_DELTA = T2_DELTA + (delta3 * A2');
    T1_DELTA = T1_DELTA + (delta2 * A1');
else
    disp('Hypothesis for '), disp(XOR(i,1:2)), disp('is '), disp(h
endif
```

If we're not learning from this example, then we simply display the cost of this particular example.

That's the end of the loop, so we close the block in Octave with:

```
endfor
```

Now we calculate the average cost across all the examples:

```
J = J / -m;
```

This gives us a useful guide to see if the cost is reducing each time we run the function (which it should!).

Now we'll update the weights in the network, remembering to divide by the number of training examples as we are in batch mode:

```
if (learn==1)
    THETA1 = THETA1 - (alpha * (T1_DELTA / m));
    THETA2 = THETA2 - (alpha * (T2_DELTA / m));
else
    disp('J: '), disp(J);
endif
```

Lastly, we'll set the return values as our new updated weights:

```
THETA1_new = THETA1;
THETA2_new = THETA2;
```

And ending the function:

```
endfunction
```

So that's all that's required to run all the training examples through the network. If you run the function a few times, you should see the cost of the network reducing as we expect it to:

```
>> XOR = [0,0,0; 0,1,1; 1,0,1; 1,1,0];
>> THETA1 = 0;
>> THETA2 = 0;
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 1, 1, 0.01);
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2);
Hypothesis for
   0   0
is
 0.70597
Hypothesis for
   0   1
is
 0.69698
Hypothesis for
   1   0
is
 0.70113
Hypothesis for
   1   1
is
 0.69185
J:
 0.77933
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 0, 1, 0.01);
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 0, 1, 0.01);
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 0, 1, 0.01);
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 0, 1, 0.01);
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 0, 1, 0.01);
>> [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2);
Hypothesis for
   0   0
is
 0.70263
Hypothesis for
   0   1
is
 0.69376
Hypothesis for
   1   0
is
 0.69776
Hypothesis for
   1   1
is
 0.68861
J:
 0.77625
>>
```

If you run the function as shown, you'll see that the cost (J) reduces each time, but not by a lot. It would be a bit boring to have to run the function each time, so let's set up a short script to run it many times:

```
XOR = [0,0,0; 0,1,1; 1,0,1; 1,1,0];
THETA1 = 0;
THETA2 = 0;

[THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 1, 1, 0.01);
```

```
for i = 1:100000
    [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2, 0, 1, 0.01);
    if (mod(i,1000) == 0)
        disp('Iteration : '), disp(i)
        [THETA1, THETA2] = xor_nn(XOR, THETA1, THETA2);
    endif
endfor
```

There's not a lot here that's new. The first call to the xor_nn function initialises the weights. The loop calls the function 100,000 times, printing out the cost every 1000 iterations. That may seem like a lot of function calls, but remember that the network weights are being adjusted by a small amount each time.

If you run that script, you should see something like this as the output:

```
Iteration :
 1000
Hypothesis for
   0   0
is
 0.47689
Hypothesis for
   0   1
is
 0.48140
Hypothesis for
   1   0
is
 0.52354
Hypothesis for
   1   1
is
 0.52798
J:
 0.69423
Iteration :
 2000
Hypothesis for
   0   0
is
 0.47606
Hypothesis for
   0   1
is
 0.48130
Hypothesis for
   1   0
is
 0.52079
Hypothesis for
   1   1
is
 0.52589
J:
 0.69409
```

When the loop gets to the end, the output will be close to this (the exact numbers may be different):

```
Iteration :
 99000
Hypothesis for
   0   0
is
 0.019508
Hypothesis for
   0   1
is
 0.97161
Hypothesis for
   1   0
is
 0.97153
Hypothesis for
   1   1
is
 0.028313
J:
 0.026528
Iteration :
 100000
Hypothesis for
   0   0
is
 0.019090
Hypothesis for
   0   1
is
 0.97231
Hypothesis for
   1   0
is
 0.97223
Hypothesis for
   1   1
is
 0.027530
J:
 0.025859
>>
```

As you can see, the network guesses small numbers (close to 0) for the first and last XOR examples and high (close to 1) for the two middle examples. This is close to what we want the network to do, so we've successfully trained this particular network to recognise the XOR function.

If you wish to download the code directly, it's available here on Github:

https://github.com/StephenOman/Octave/tree/master/xor%20neural%20network

There are a lot of things that we should do to make sure that the algorithm is optimised, including checking that the default learning rate of 0.01 is actually the right rate. But that is a job for another day.

Now that you've seen a simple neural network recognizing patterns and learning from examples, you can implement your own in Octave for other more interesting problems.

Be the first to like this.

Solving XOR with a Neural Network in TensorFlow
In "Artificial Intelligence"

A Simple Neural Network in Octave - Part 2
In "Examples"

Solving XOR with a Neural Network in Python
In "Examples"

ALGORITHMS, MACHINE LEARNING ALGORITHMS

PREVIOUS POST
A Simple Neural Network in Octave – Part 2

NEXT POST
Solving XOR with a Neural Network in Python

Leave a Reply

Enter your comment here...

ABOUT ME

Stephen Oman

Lifelong coder, interested in artificial intelligence since finding a listing for Eliza in a computer magazine in 1985.B.Sc. Computer Science and M.Sc. (Research) in AI from Trinity College Dublin.

View Full Profile

Follow On Machine Intelligence 18

SOCIAL

ARCHIVE

March 2015 (3)

Follow On Machine Intelligence

FOLLOW BLOG VIA EMAIL

Enter your email address to follow this blog and receive notifications of new posts by email.

Join 1,086 other followers

Enter your email address

FOLLOW