Neural network walk through

The following illustrates the process of training a simple neural network. It documents how every step changes network, both through a the visual representation of the network, and in a dataset.

The walkthrough is based on one round of training for a simple XOR gate (Exclusive OR). In this illustration the inputs will be 0 and 1, which should equal 1 for a XOR gate.

The network has 2 input nodes (1 for each input value), 4 nodes in 1 hidden layer, and 1 output node (for the 1 output value required).

When building neural networks there are a couple of things to keep in mind.

- Node values are typically restricted to the range of 0 to 1, so problem sets are designed in such a manner to be compatible with this constraint.

- Typically you want integer 0's or 1's as inputs, with a goal output of 0 or 1 as well. ie: You shouldn't try to shortcut by creating complex meanings such as an output of 0.0->0.3 representing one meaning, where as 0.3 to 0.7 represents something else. Split these into distinct output points.

This walk through has been constructed as I couldn't find one like it that showed every step of the process and its impact on both the network, and what that would look like in code. The hope is that through tracing every step, you can see how the network tweaks its values as a result of training data and the amount of error in its predictions.

This walkthrough assumes basic mathematical understanding of matrices (mostly transposition and dot products), and the programming concept of arrays.

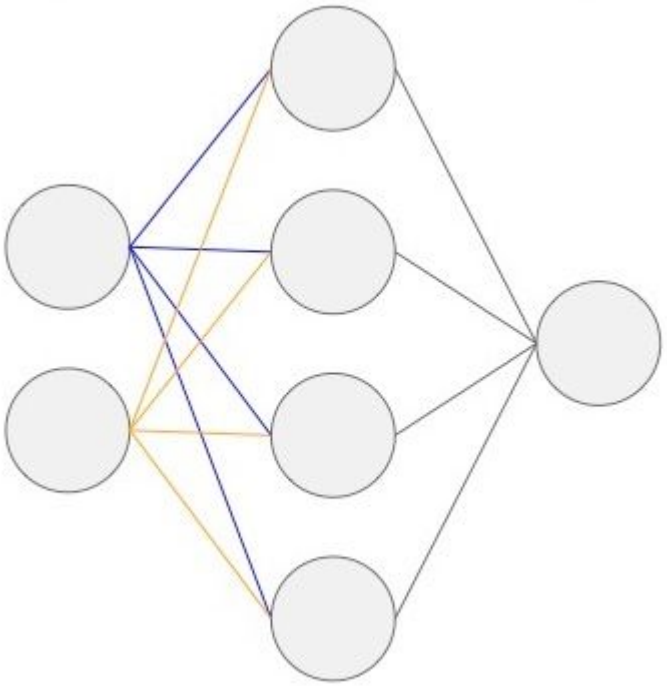
Paul Baumgarten
April 2019
---
web: pbaumgarten.com/ai
twitter: @pbaumgarten

Correction required!
Weights should be randomised to range of 0.0 and 1.0
NOT -1.0 and 1.0 as was previously written

Introducing our neural network

Our network looks like this. It has 2 input nodes, 4 nodes in a "hidden layer" and 1 output node.

At each layer, each node is connected to every node in the subsequent layer. This connections are analogous to the synapses connecting the neurons of our brains. In our NN, each synapse as a "weight" which is a number that will be used to indicate how "important" that connection is.



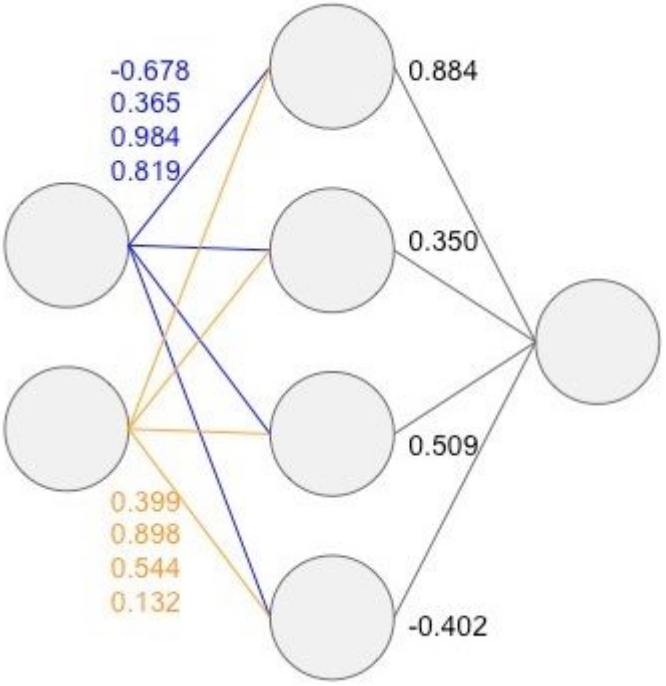
1. Initialise the neural network

Before we begin using our NN, we initialise the weights to random values between `0` and `+1`.

In our example, our random number generator has populated our weights as follows:

```
weights_i2h = [
   [ 0.6784, 0.3990 ],
   [ 0.3645, 0.8981 ],
   [ 0.9840, 0.5440 ],
   [ 0.8190, 0.1316 ]
]

weights_h2o = [
   [ 0.8839, 0.3500, 0.5093, 0.4023 ]
]
```



Programmatically, we are using arrays to store the weights for each set of synapses.

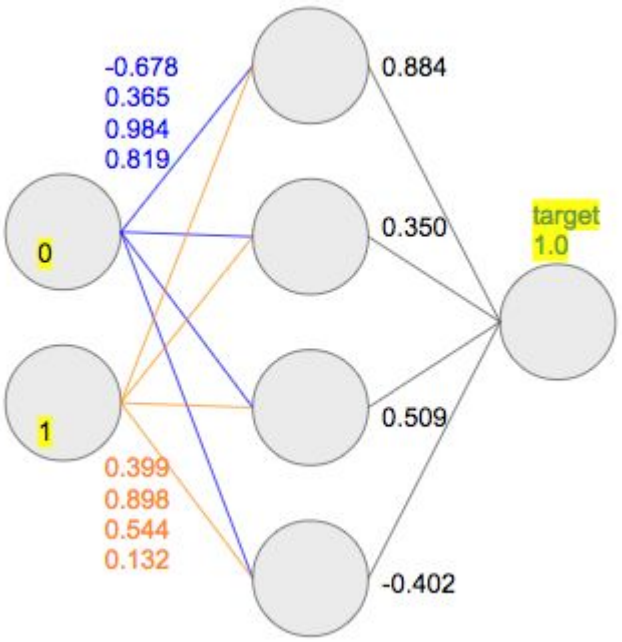
- **`weights_i2h[0]`** contains an array of weight values that contribute toward the first hidden layer node, whereas **`weights_i2h[1]`** contain the weight values that contribute to the second hidden layer node, etc.
- To access the weight value for an individual pathway, such as the 1st input node to the 3rd hidden node, it would be weights_i2h[0][2].
- The same principle applies with **`weights_h2o`**.

2. Enter training input & target data

As we are using a XOR gate, the values of 0 and 1 would ideally provide an output of 1. We will save these values to arrays, and update our model illustration as such...

```
inputs = [
   [ 0.0 ],
   [ 1.0 ]
]

targets = [
   [ 1.0 ]
]
```



3. Calculate raw values for hidden nodes

Use the dot product of the weights_i2h array with the input array to calculate the hidden node values.

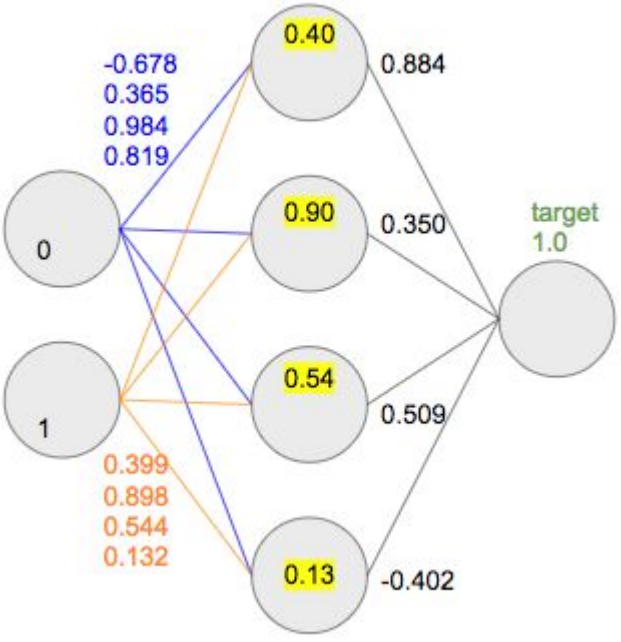
```
inputs = [
   [ 0 ],
   [ 1 ]
]

weights_i2h = [
   [ -0.6784, 0.3990 ],
   [ 0.3645, 0.8981 ],
   [ 0.9840, 0.5440 ],
   [ 0.8190, 0.1316 ]
]
```

		0.0	
		1.0	
-0.6784	0.3990	(0)(-0.6784) + (1)(0.3990)	0.3990
0.3645	0.8981	(0)(0.3645) + (1)(0.8981)	0.8981
0.9840	0.5440	(0)(0.9840) + (1)(0.5440)	0.5440
0.8190	0.1316	(0)(0.8190) + (1)(0.1316	0.1316

The final result is stored in the hidden array as follows:

```
hidden = [
   [ 0.3990 ],
   [ 0.8981 ],
   [ 0.5440 ],
   [ 0.1316 ]
]
```



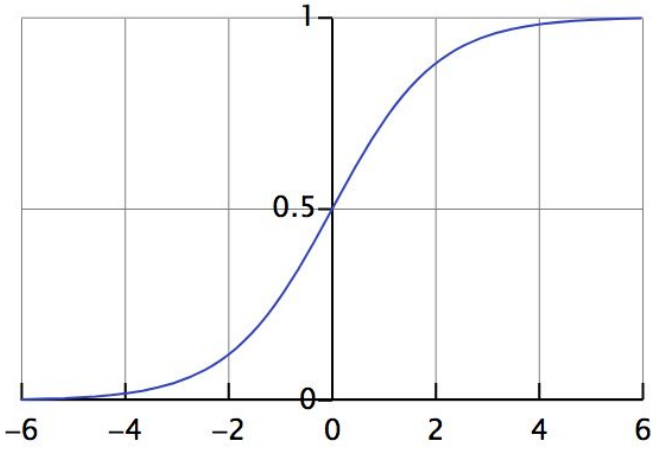
4. Apply an activation function on hidden nodes

An activation function is used to define the outputs of neurons to fit within a particular range. In our case (and typically used for beginner NN's) we'll use the sigmoid function.

The sigmoid function will take any incoming number and force it into a range between 0.0 and 1.0. It's equation and graph is as follows:

$$S(x) = \frac{1}{1 + e^{-x}}$$

(graphics: wikipedia)



So, programmatically it would be a case of writing a function to perform Sigmoid, and then looping over the hidden nodes array to calculate their updated values. For instance, in Python this might look like:

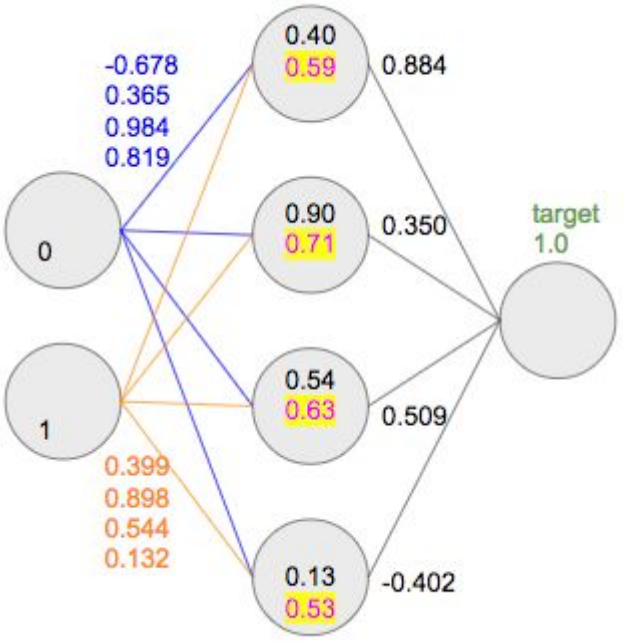
```
def sigmoid(n):
    e = 2.7182818284
    return 1.0/(1.0 + e**(-n))
```

So, using these previous values for our hidden nodes,

```
hidden = [
   [ 0.3990 ],
   [ 0.8981 ],
   [ 0.5440 ],
   [ 0.1316 ]
]
```

We arrive at these new values:

```
hidden = [
   [ 0.5984 ],
   [ 0.7105 ],
   [ 0.6327 ],
   [ 0.5328 ]
]
```



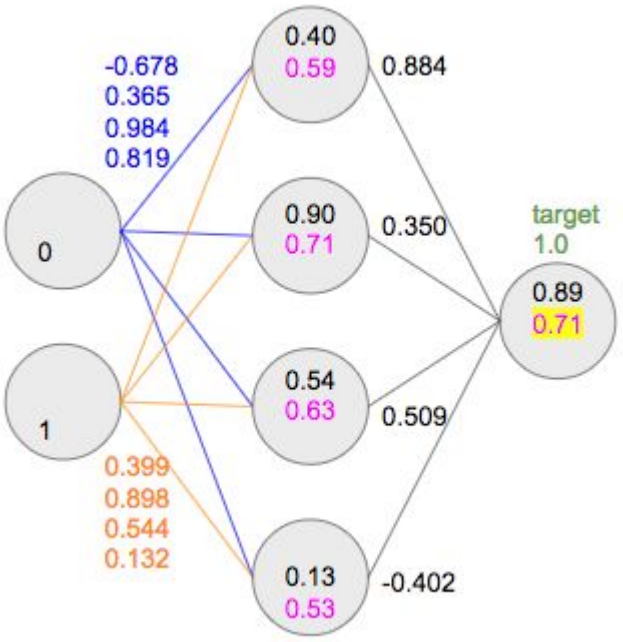
5. Calculate output node values

Repeat the previous steps to find the values for your next layer, in our case, the output nodes.

- Recall our values of the previous layer and the weights of those synapses...

```
hidden = [
   [ 0.5984 ],
   [ 0.7105 ],
   [ 0.6327 ],
   [ 0.5328 ]
]

weights_h2o = [
   [ 0.8839, 0.3500, 0.5093, -0.4023 ]
]
```



- Calculate the dot product of the weights of the previous layer, with that previous layer's final values, to obtain the raw value for our output nodes.

```
outputs = [
   [ 0.8856 ]
]
```

- Apply the activation function to determine the "final value" of the output nodes.

```
outputs = [
   [ 0.7079 ]
]
```

Our neural network has now successfully completed the prediction phase. In this case, the given inputs of 0 and 1, the network has "predicted" the answer is 0.7079. As we know, 0 XOR 1 should give us an answer of 1, so the network is about 71% correct!

6. Calculate error (output layer)

We now move into the next phase: We will calculate the amount of error in the prediction and back-propagate (apply) corrections throughout the network so that it will hopefully become more accurate over time.

A simple process of subtracting our output values from our target values to create a new matrix of error values.

```
outputs = [
   [ 0.7079 ]
]
targets = [
   [ 1 ]
]

output_errors = [
   [ 0.2920 ]
]
```

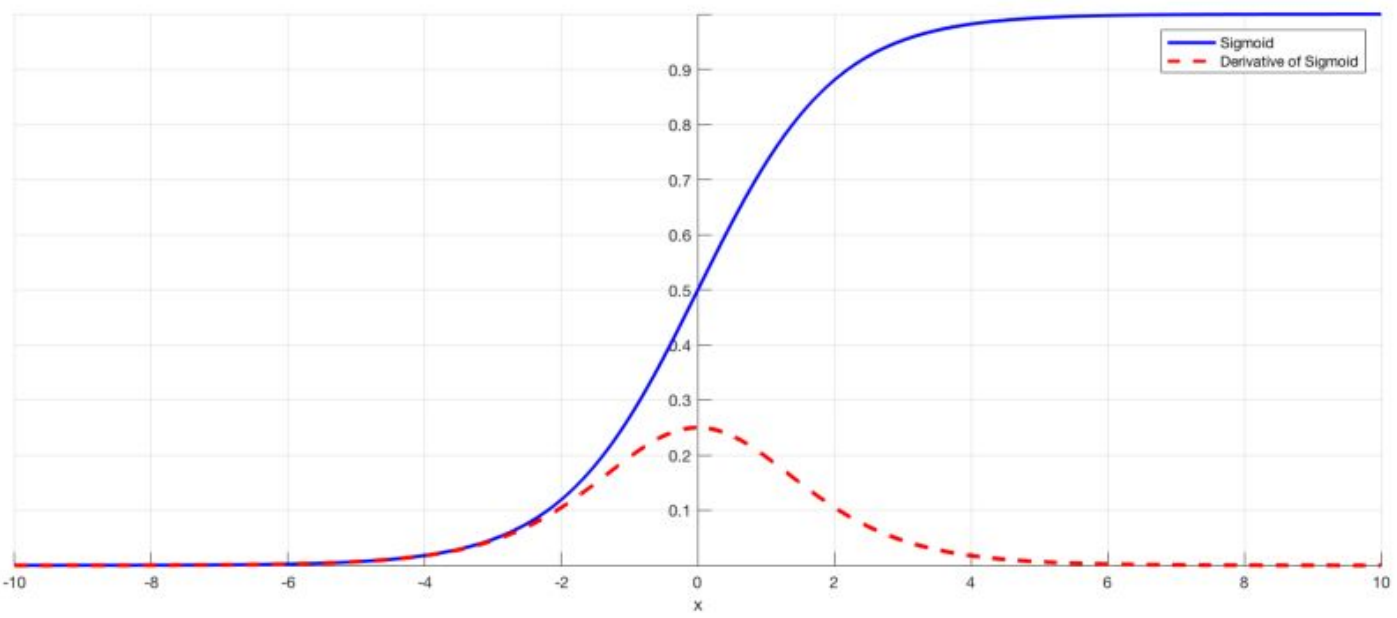
Note: We are keeping everything in matrices here because any real network is going to be outputting more than one node, so we would need to perform this calculation on each individual node.

7. Calculate degree of correction required (output layer)

To increase the accuracy of the network, we adjust the values of the various weights. You might initially think, we just adjust the weights by the raw amount of error. However, we don't know which nodes and synapses caused the error involved, and we want to adjust the network so that it is reliable for different input values as well - not just this one scenario with inputs of 0 and 1. As a result, we moderate or reduce the amount of change over a couple of different steps.

The first step we will apply is to find the derivative of the final output values we obtained. If you haven't done Calculus in mathematics yet, the derivative can be thought of as the gradient or slope of a curve. The graph below shows the Sigmoid function and its derivative on the same axis'.

The derivative of our activation function is useful because it is highest when the "output" value hovers around the midpoint of 0.5. For our NN to be useful, we don't want nodes to be "sitting on the fence", so if they are in the middle, the derivative can be used to increase the size of the shift we will apply. If a node is already heavily one sided, we don't need want to change it as much as otherwise we'll end up with wild variations from one iteration to the next, so we'll just make a little adjustment.



We use Calculus to determine the derivative function. In our case, for the Sigmoid function its derivative could be written thus:

```
def sigmoid_derivative(n):
    return sigmoid(n)*(1.0-sigmoid(n))
```

Using our outputs matrix as the input to this function,

```
outputs = [
   [ 0.7079 ]
]
```

We would obtain the following results

```
derivatives = [
   [ 0.2067 ]
]
```

(sigmoid derivative chart: https://towardsdatascience.com/derivative-of-the-sigmoid-function-536880cf918e)

Once we know the derivatives (remember, a unit of measure of the steepness of the curve), we will multiply each derivative value by each output error value respectively. *Note: this is straight multiplication of cells in matching positions, not the dot product.*

```
output_errors = [
   [ 0.2920 ]
]
derivatives = [
   [ 0.2067 ]
]
```

So the preliminary amount of adjustment for the hidden->output weights is 0.2920 multiplied by 0.2067 which gives us...

```
adjustments_h2o_weights = [
   [ 0.0603 ]
]
```

We will further moderate the amount of adjustment to apply by using a "learning rate". This is just a mathematical constant we apply to the entire network to ensure that no individual datum over-influences the network which could result in wild and unpredictable variations.

In our case, we're using a `learning_rate = 0.1` which we just multiply across every value giving us,

```
adjustments_h2o_weights = [
   [ 0.00603 ]
]
```

8. Adjust hidden->output weights

Prior to performing this step we transpose the hidden layer weights matrix so its alignment matches the matrix we will be applying the results to

this...

```
hidden = [
   [ 0.5984 ],
   [ 0.7105 ],
   [ 0.6327 ],
   [ 0.5328 ]
]
```

to this...

```
hidden_transposed = [
   [ 0.5984, 0.7105, 0.6327, 0.5328 ]
]
```

We will calculate the amount of adjustment (delta) we want to apply to each weight for the hidden-to-output synapses by using the dot product:

* the post-sigmoid predicted values in the hidden layer with
* the adjustment number we produced that was based on the error in the overall result.

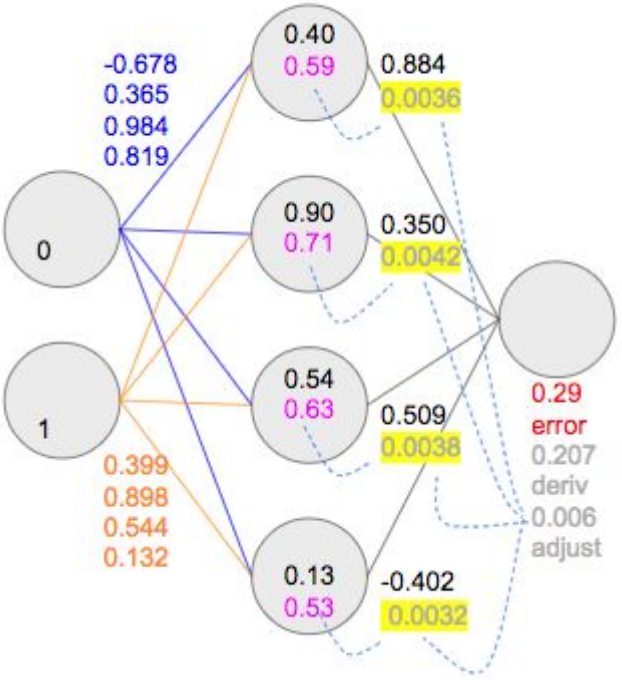
```
hidden_transposed = [
   [ 0.5984, 0.7105, 0.6327, 0.5328 ]
]
```

●

```
adjustments_h2o_weights = [
   [ 0.0060 ]
]
```

=

```
weight_h2o_deltas = [
   [ 0.0036, 0.0042, 0.0038, 0.0032 ]
]
```



Once we have the deltas, we can apply them to each individual synapses through simple matrix addition.

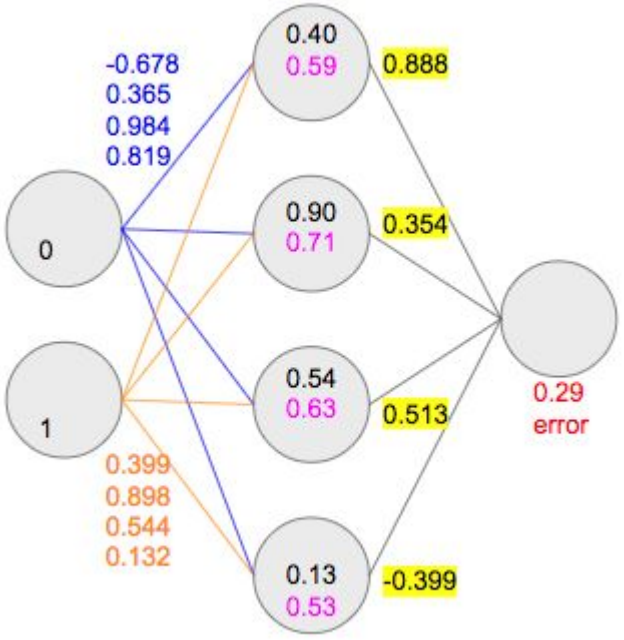
```
weights_h2o = [
   [ 0.8839, 0.3500, 0.5093, -0.4023 ]
]
```

+

```
weight_h2o_deltas = [
   [ 0.0036, 0.0042, 0.0038, 0.0032 ]
]
```

=

```
weights_h2o = [
   [ 0.8875, 0.3543, 0.5131, -0.3991 ]
]
```



9. Calculate error (hidden layer)

Transpose the new hidden->output weights so it is correctly aligned for our next calculations

```
weights_h2o = [
   [ 0.8875, 0.3543, 0.5131, -0.3991 ]
]

weights_h2o_transposed = [
   [ 0.8875 ],
   [ 0.3543 ],
   [ 0.5131 ],
   [ -0.3991 ]
]
```

Calculate the hidden layer error for each node by finding the dot product of each node's individual output value by the error values of the whole network.

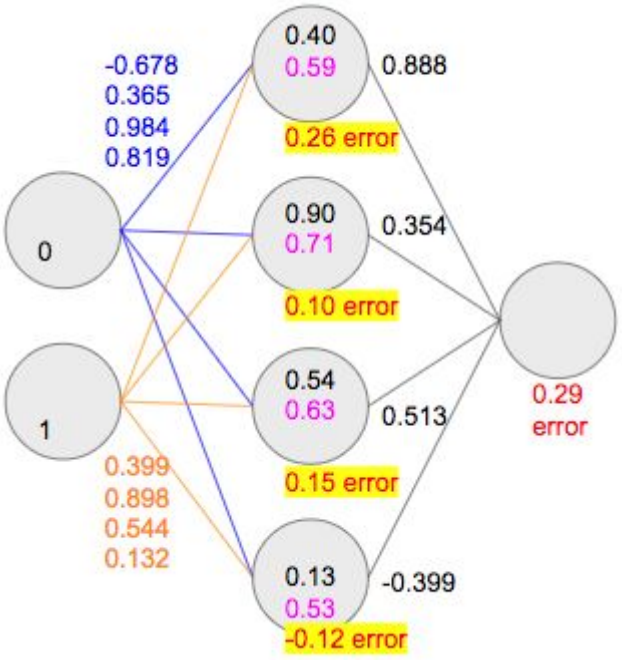
```
weights_h2o_transposed = [
   [ 0.8875 ],
   [ 0.3543 ],
   [ 0.5131 ],
   [ -0.3991 ]
]
```

●

```
output_errors = [
   [ 0.2920 ]
]
```

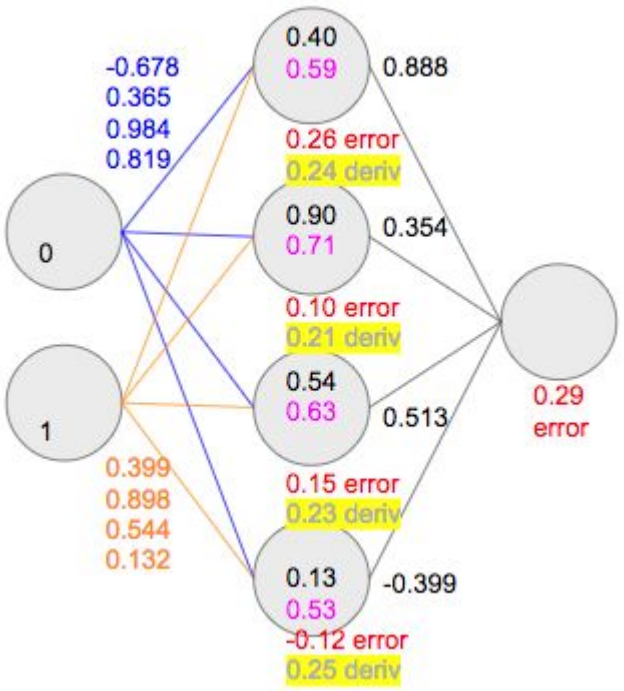
=

```
hidden_errors = [
   [ 0.2591 ],
   [ 0.1034 ],
   [ 0.1498 ],
   [ -0.1165 ]
]
```



10. Calculate degree of correction required (hidden layer)

Repeating the process we used with the output layer, we will find the derivative for each hidden node value, multiply it to the various input->hidden weight errors and apply the learning rate.



1. Find the derivatives

```
hidden = [
   [ 0.5984 ],
   [ 0.7105 ],
   [ 0.6327 ],
   [ 0.5328 ]
]
```

Use the same `sigmoid_derivative()` function discussed earlier on each value to obtain...

```
hidden_derivatives = [
   [ 0.2403 ],
   [ 0.2056 ],
   [ 0.2323 ],
   [ 0.2489 ]
]
```

2. Multiply the individual *derivatives* by the individual *input->hidden errors*
   Note: this is straight multiplication of cells in matching positions, not the dot product.

```
hidden_derivatives = [
   [ 0.2403 ],
   [ 0.2056 ],
   [ 0.2323 ],
   [ 0.2489 ]
]
```

x

```
hidden_errors = [
   [ 0.2591 ],
   [ 0.1034 ],
   [ 0.1498 ],
   [ -0.1165 ]
]
```

=

```
adjustment_i2h_weights = [
   [ 0.0622 ],
   [ 0.0212 ],
   [ 0.0348 ],
   [ -0.0290 ]
]
```

3. Apply the learning_rate of 0.1 to the adjustment_i2h_weights and we get...

```
adjustment_i2h_weights = [
   [ 0.00622 ],
   [ 0.00212 ],
   [ 0.00348 ],
   [ -0.00290 ]
]
```

11. Adjust input->hidden weights

As with the hidden layer process, we will transpose the matrix of the incoming values, the input layer.

so from this…

```
inputs = [
   [ 0 ],
   [ 1 ]
]
```

to this…

```
inputs_transposed = [
   [ 0, 1 ]
]
```

Calculate amount of change that should apply to each input->hidden weight by calculating the dot product

```
inputs_transposed = [
   [ 0, 1 ]
]
```

●

```
adjustment_i2h_weights = [
   [ 0.00622 ],
   [ 0.00212 ],
   [ 0.00348 ],
   [ -0.00290 ]
]
```

=

```
weight_i2h_deltas = [
   [ 0, 0.00622 ],
   [ 0, 0.00212 ],
   [ 0, 0.00348 ],
   [ 0, -0.00290 ]
]
```

Apply the changes to the input->hidden weights.

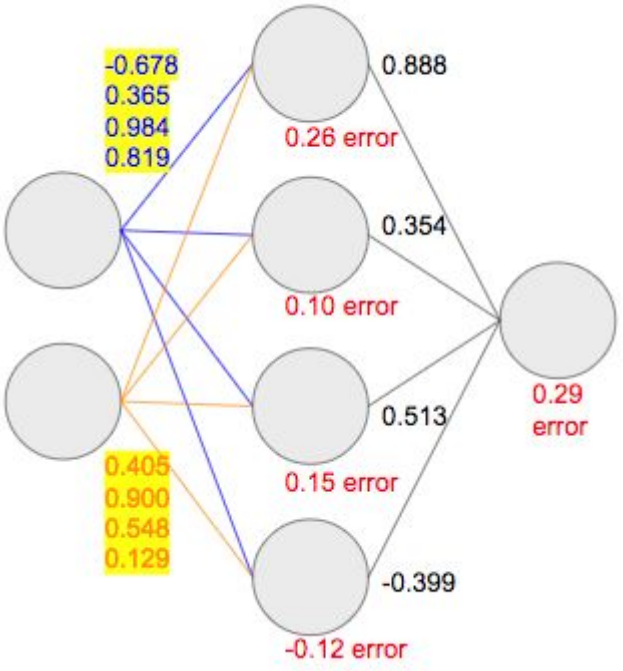
```
weights_i2h = [
   [ -0.6784, 0.3990 ],
   [ 0.3645, 0.8981 ],
   [ 0.9840, 0.5440 ],
   [ 0.8190, 0.1316 ]
]
```

+

```
weight_i2h_deltas = [
   [ 0, 0.0062 ],
   [ 0, 0.0021 ],
   [ 0, 0.0034 ],
   [ 0, -0.0029 ]
]
```

=

```
weights_i2h = [
   [ -0.6784, 0.4052 ],
   [ 0.3645, 0.9002 ],
   [ 0.9840, 0.5475 ],
   [ 0.8190, 0.1287 ]
]
```



That's it!!

The corrections have been fully propagated through the network and it is ready to begin the next round!

For more, and to see a Python implementation of this network, please check my AI pages at https://pbaumgarten.com/ai