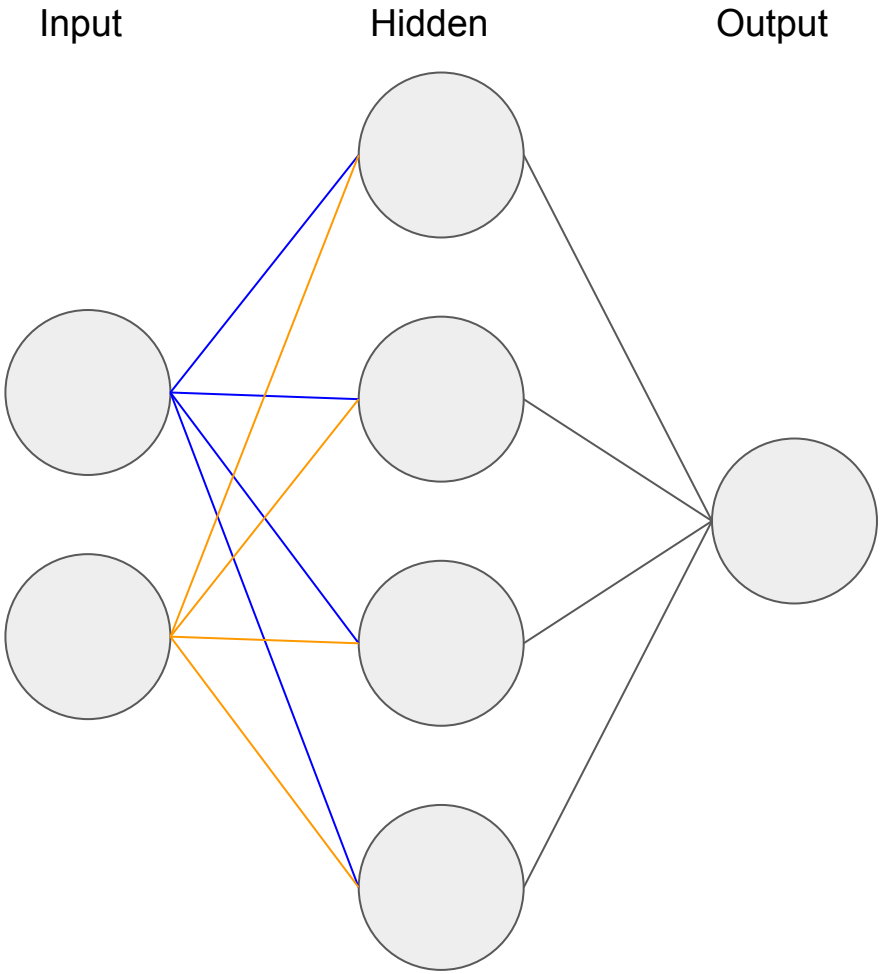


Note

The following is a walk through to train a simple neural network on a XOR gate (Exclusive-OR) for one round of training (out of, typically, 1000s). In this illustration the inputs will be 0 and 1, which should equal 1 for a XOR gate.

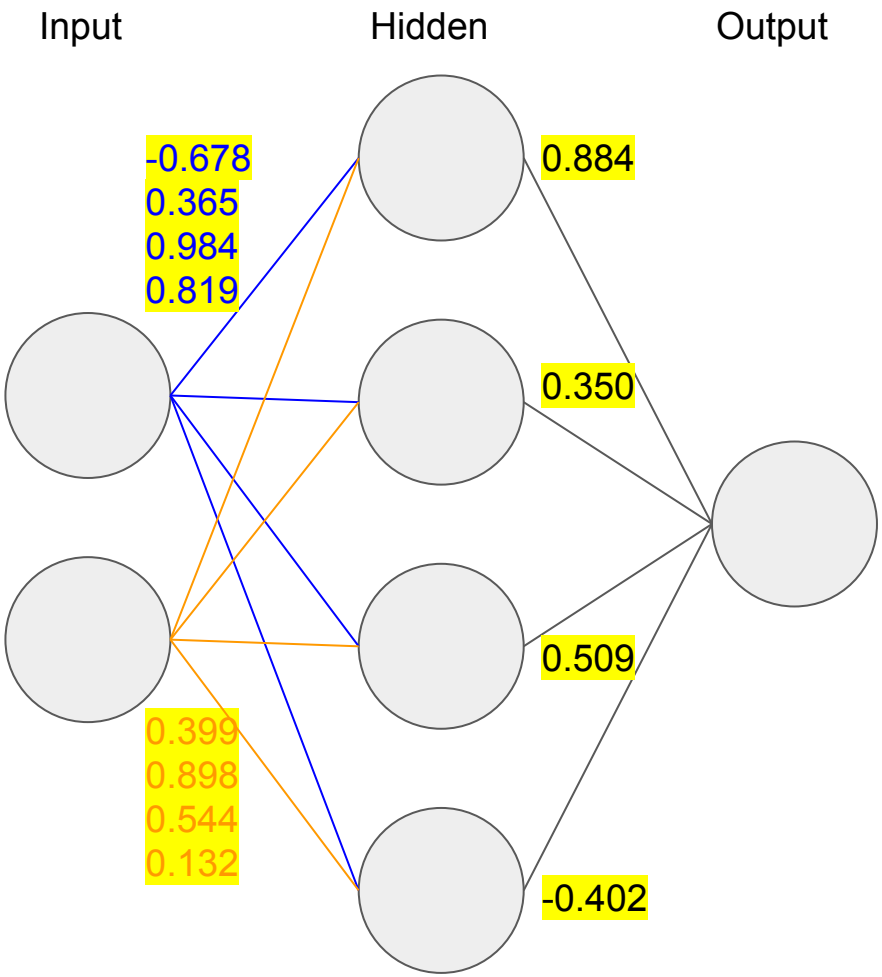
The network has 2 input nodes (1 for each input value), 4 nodes in 1 hidden layer, and 1 output node (for the 1 output value required).

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 the goal of an output of 0 or 1 as well (ie: don't try to have a network aim for 0.0->0.3 have one meaning, 0.3 to 0.7 have another etc, instead add additional output nodes for each desired meaning).

Every calculation is provided so that it (hopefully) doesn't matter if you don't understand sample code applying abstract mathematical methods (such occurs with Tensorflow, PyTorch or even Numpy). Tracing every step should allow you to see how the network tweaks its values as a result of training data and the amount of error in its predictions.

Paul Baumgarten
February 2019



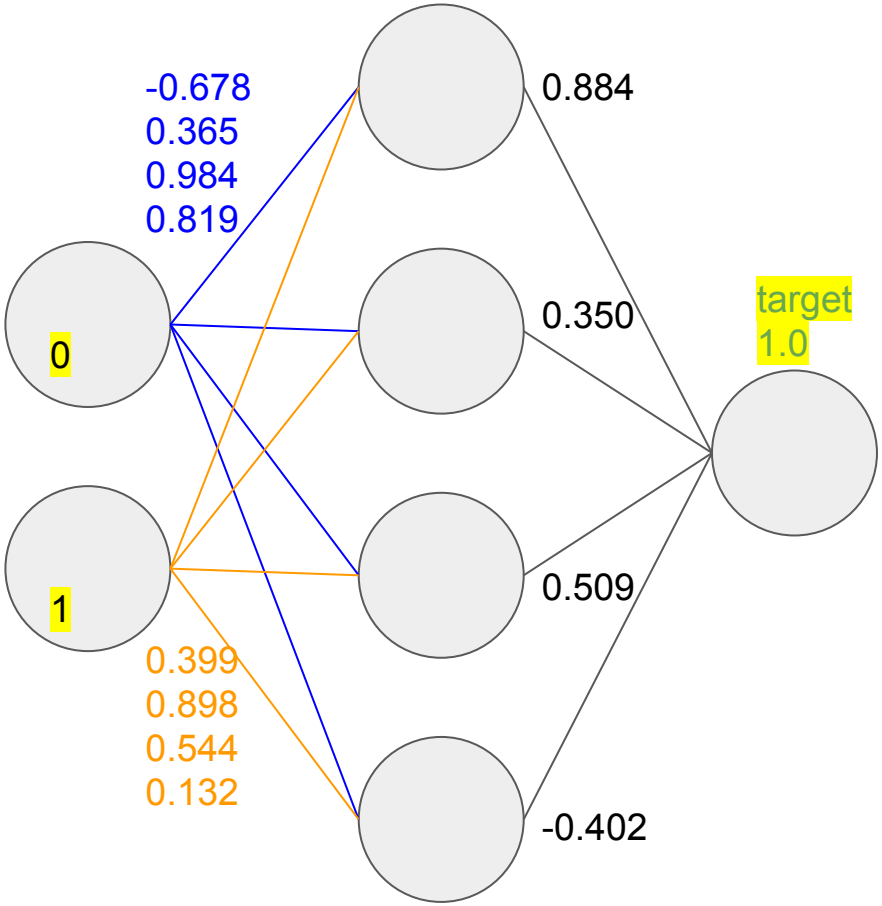
Note

Before beginning any training rounds, initialise the network with randomly assigned weights for each pathway.

Use random numbers between -1.0 and 1.0

After

```
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 ]
]
```

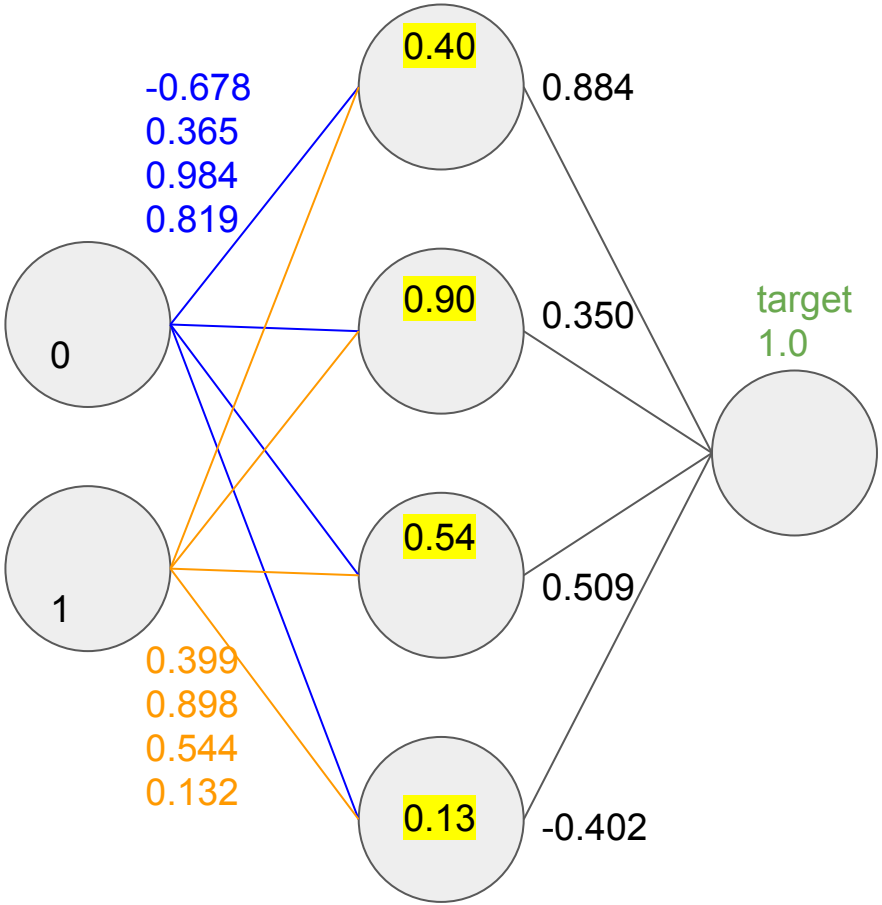


Note

Obtain training inputs and target data

After

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

targets = [
    [ 1 ]
]
```



Note

Perform matrix dot product to calculate raw values of hidden layer nodes

Before

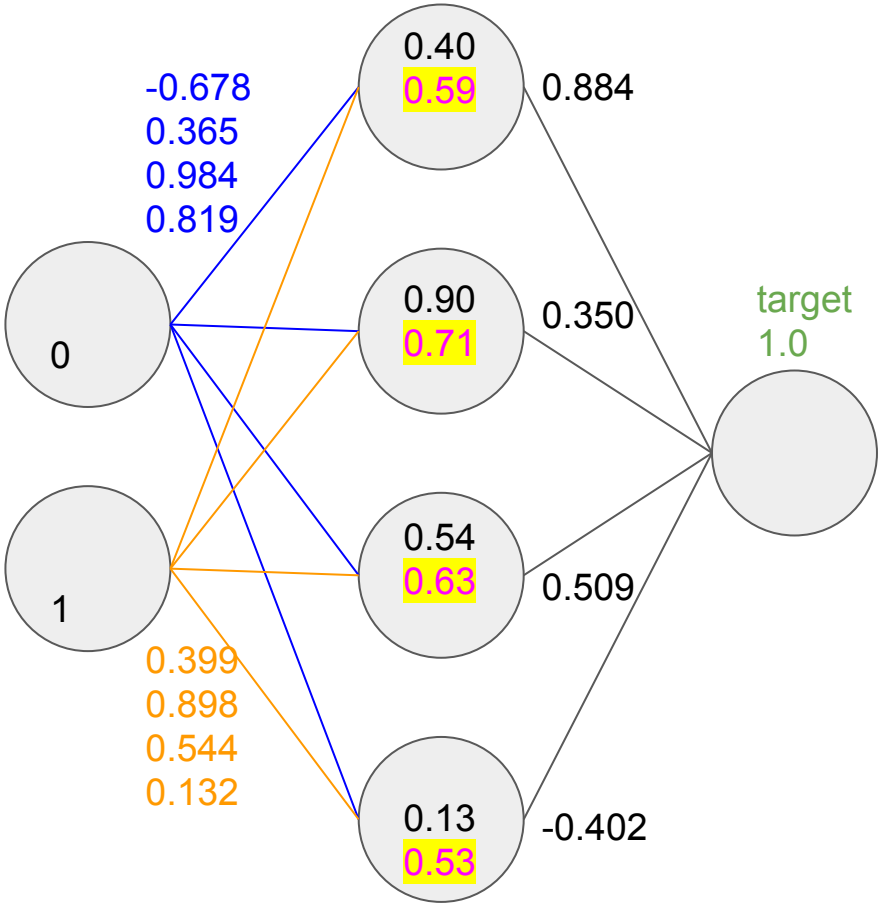
```
inputs = [
   [ 0 ],
   [ 1 ]
]
weights_i2h = [
   [ -0.6784, 0.3990 ],
   [ 0.3645, 0.8981 ],
   [ 0.9840, 0.5440 ],
   [ 0.8190, 0.1316 ]
]
```

Execute

```
hidden = Matrix.multiply(weights_i2h, inputs);
```

After

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



Note

Apply activation function to force values into range 0 to 1

Before

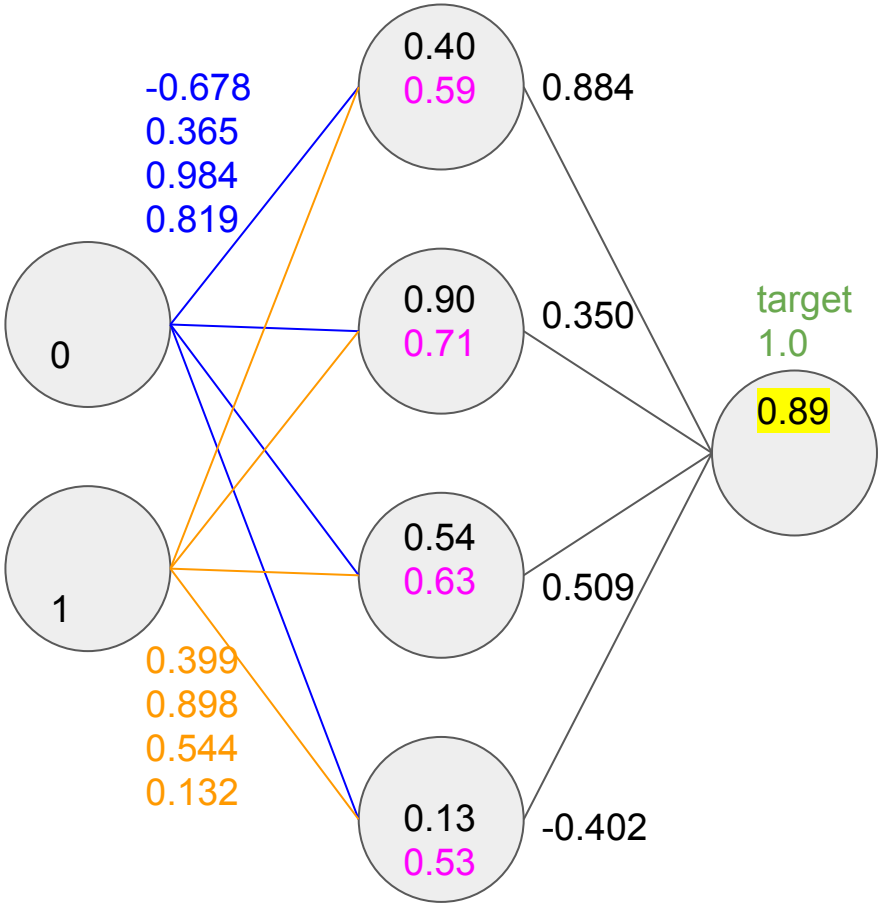
```
hidden = [
   [ 0.3990 ],
   [ 0.8981 ],
   [ 0.5440 ],
   [ 0.1316 ]
]
```

Execute

```
hidden.map(sigmoid);
```

After

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



Note

Use matrix dot product to calculate raw value of output node

Before

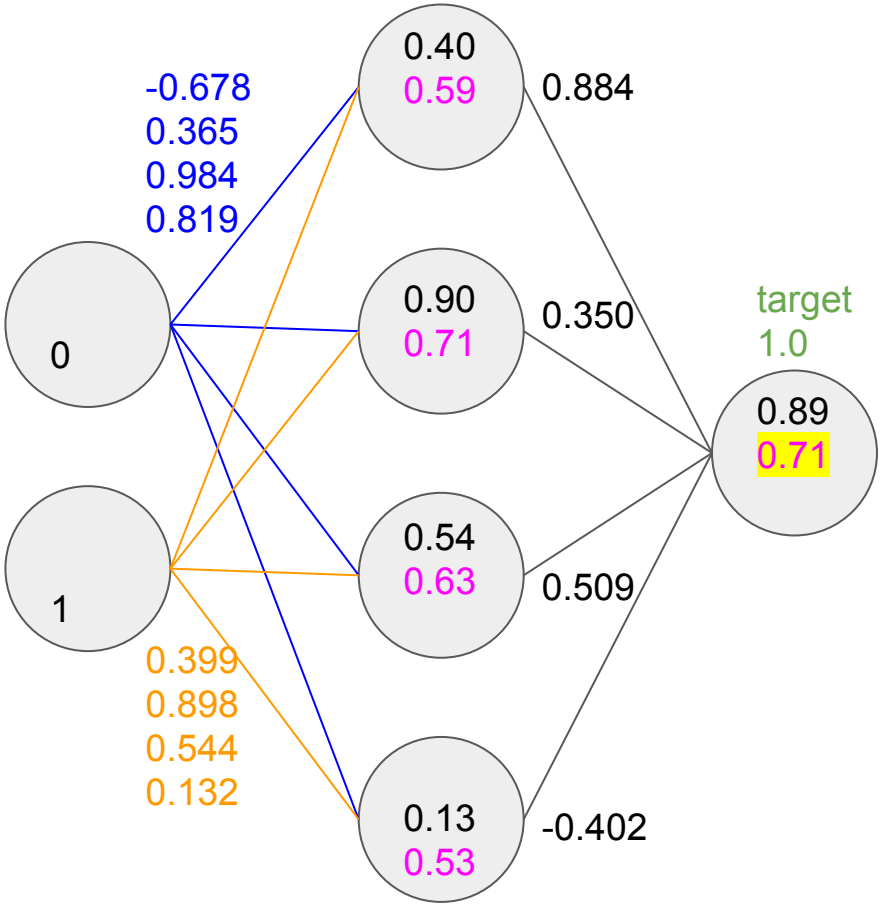
```
hidden = [
   [ 0.5984 ],
   [ 0.7105 ],
   [ 0.6327 ],
   [ 0.5328 ]
]

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

Execute

```
outputs = Matrix.multiply(weights_h2o, hidden);
```

After

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



Note

Apply activation function to force values to range 0 to 1

Before

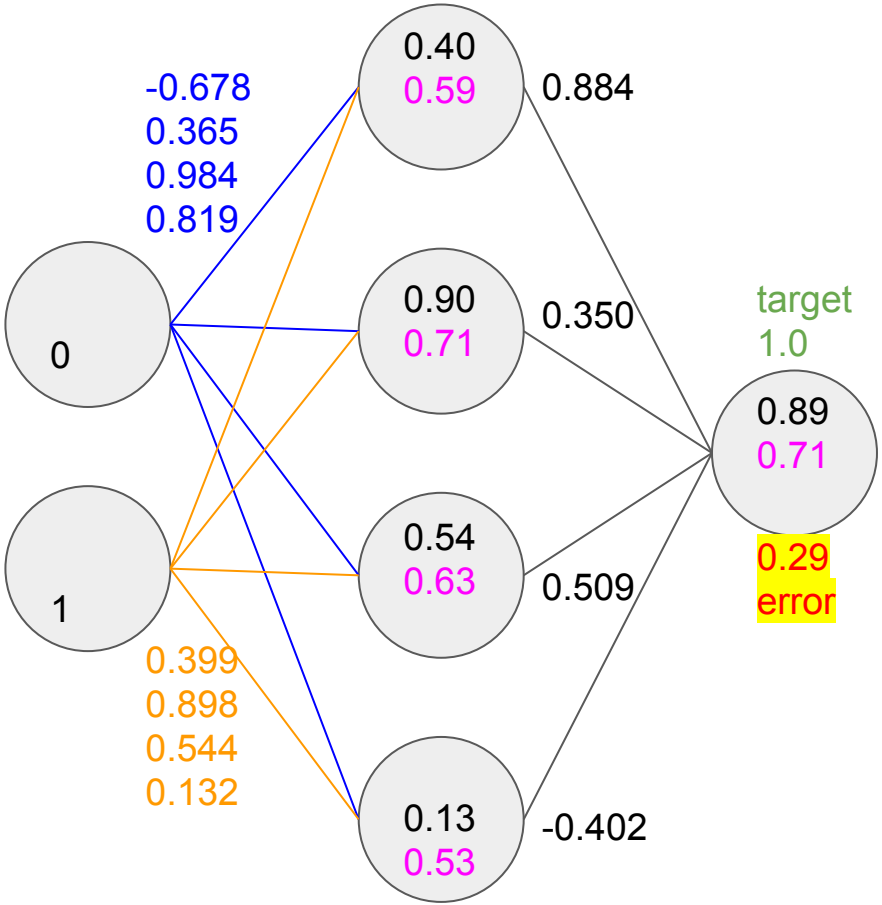
```
outputs = [
   [ 0.8856 ]
]
```

Execute

```
outputs = outputs.map(sigmoid);
```

After

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



Note

Calculate raw error of prediction

Before

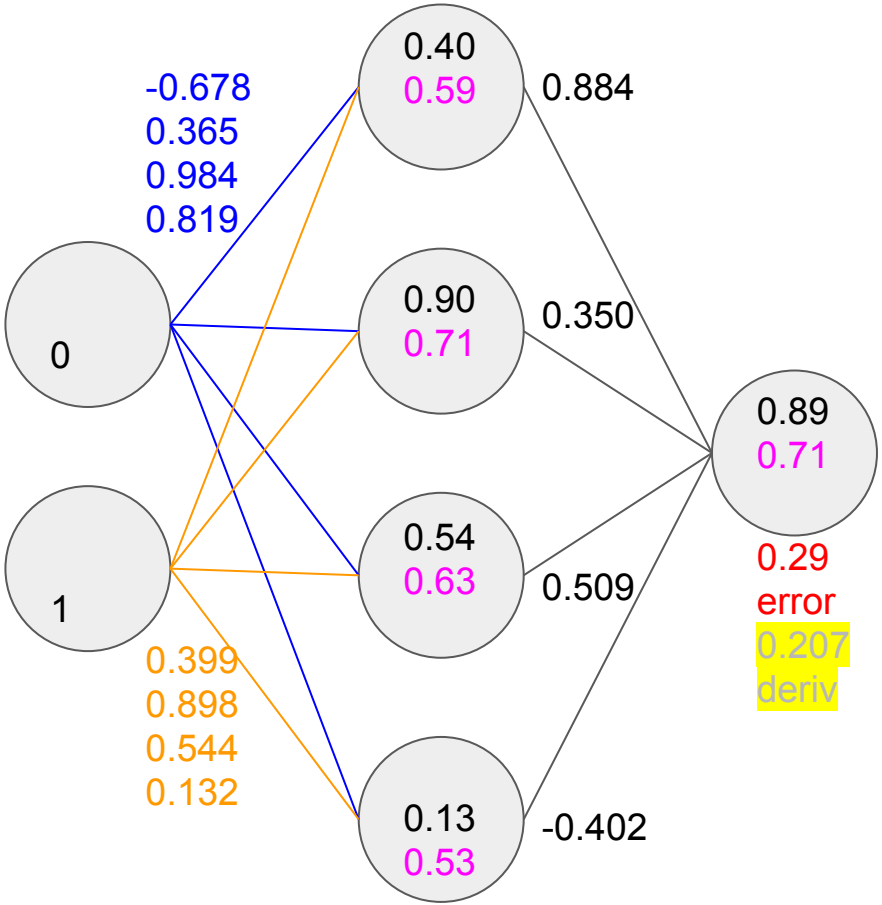
```
outputs = [
   [ 0.7079 ]
]
targets = [
   [ 1 ]
]
```

Execute

```
output_errors = Matrix.subtract(targets, outputs);
```

After

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



Note

To help determine the necessary change in the hidden->output weights, we look at the derivative (gradient) of the activated predicted output.

Why? The gradient, or slope, is highest at the midpoint of the function. For our NN to be useful, we don't want nodes to be "sitting on the fence", so if they are in the middle, we use this to increase the size of the shift we will apply. If a node is pretty much already one sided, we don't need to bother changing it much.

Before

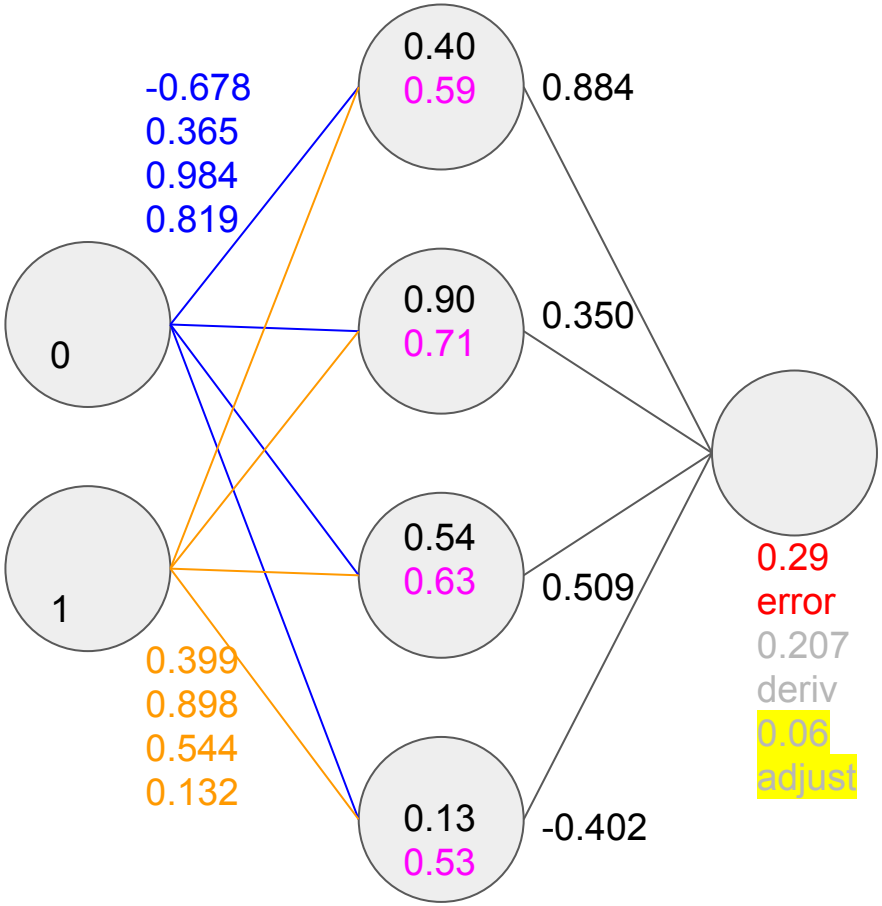
```
outputs = [
   [ 0.7079 ]
]
```

Execute

```
gradients = Matrix.map(outputs, sigmoid_derivative);
```

After

```
gradients = [
   [ 0.2067 ]
]
```



Note

Apply the gradient to influence how much error correction we wish to apply for this round

Before

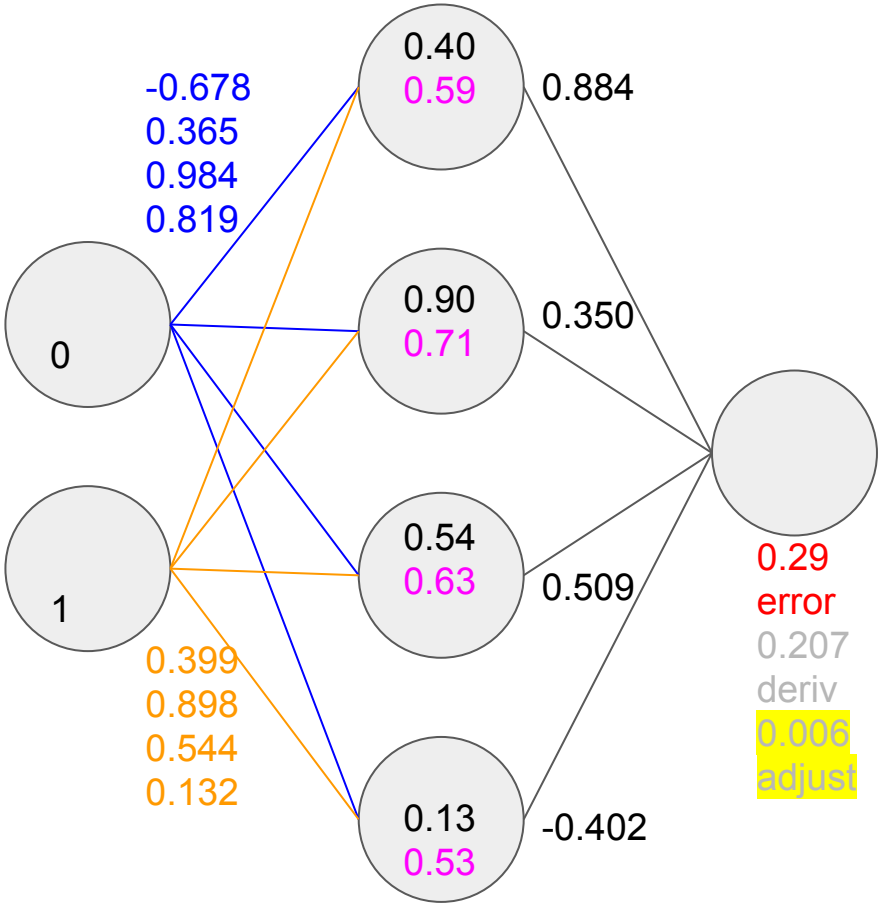
```
output_errors = [
   [ 0.2920 ]
]
gradients = [
   [ 0.2067 ]
]
```

Execute

```
adjustments_h2o_weights = gradients.multiply(output_errors);
```

After

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



Note

Apply the learning rate to influence how much error correction we wish to apply for this round. We don't want any individual training datum to over-influence the NN and cause wild variations.

Before

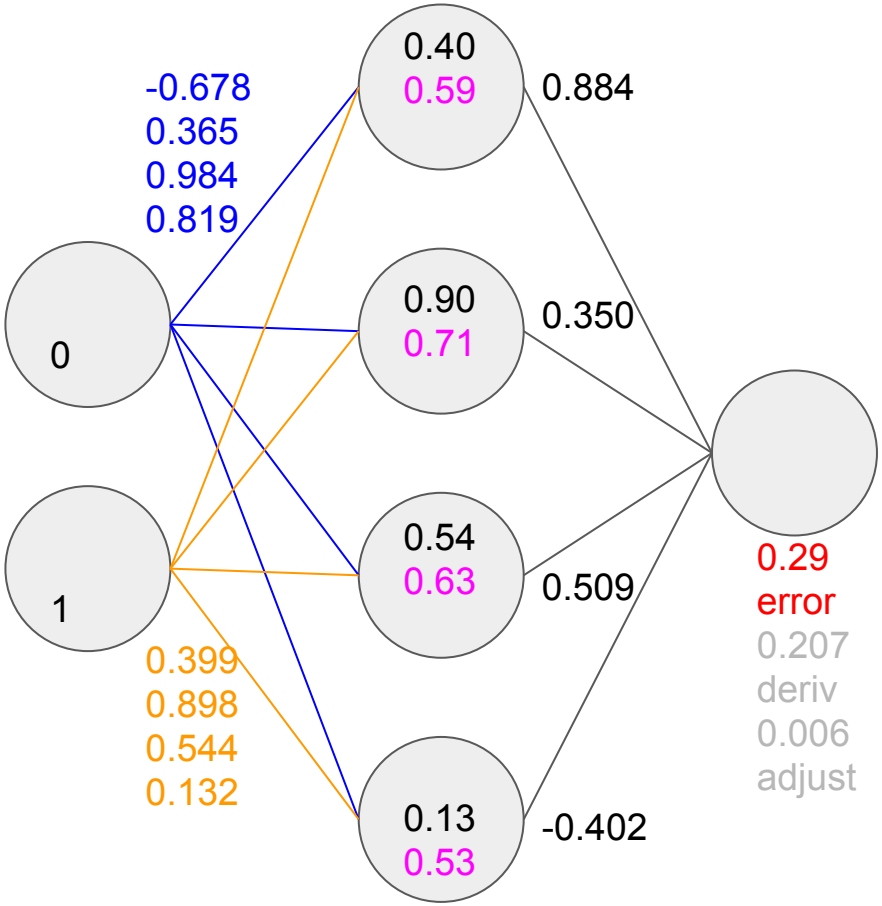
```
adjustments_h2o_weights = [
   [ 0.0603 ]
]
```

Execute

```
adjustments_h2o_weights =
adjustments_h2o_weights.multiply(learning_rate);;
```

After

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



Note

Transpose the hidden layer so it is correctly orientated for later calculations

Before

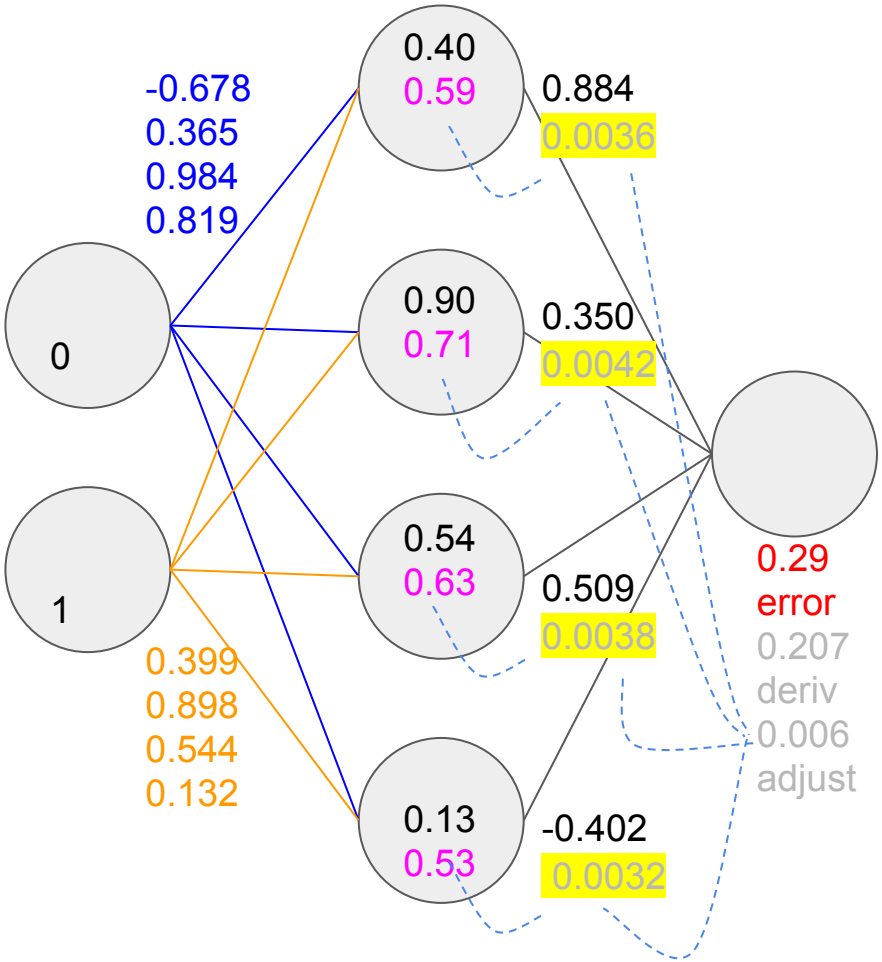
```
hidden =
[
   [ 0.5984 ],
   [ 0.7105 ],
   [ 0.6327 ],
   [ 0.5328 ]
]
```

Execute

```
hidden_transposed = Matrix.transpose(hidden);
```

After

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



Note

```
Calculate changes to hidden->output weights by multiplying:
* the post-sigmoid predicted values in the hidden layer by
* the adjustment number we produced that was based on the
error in the overall result.
```

Before

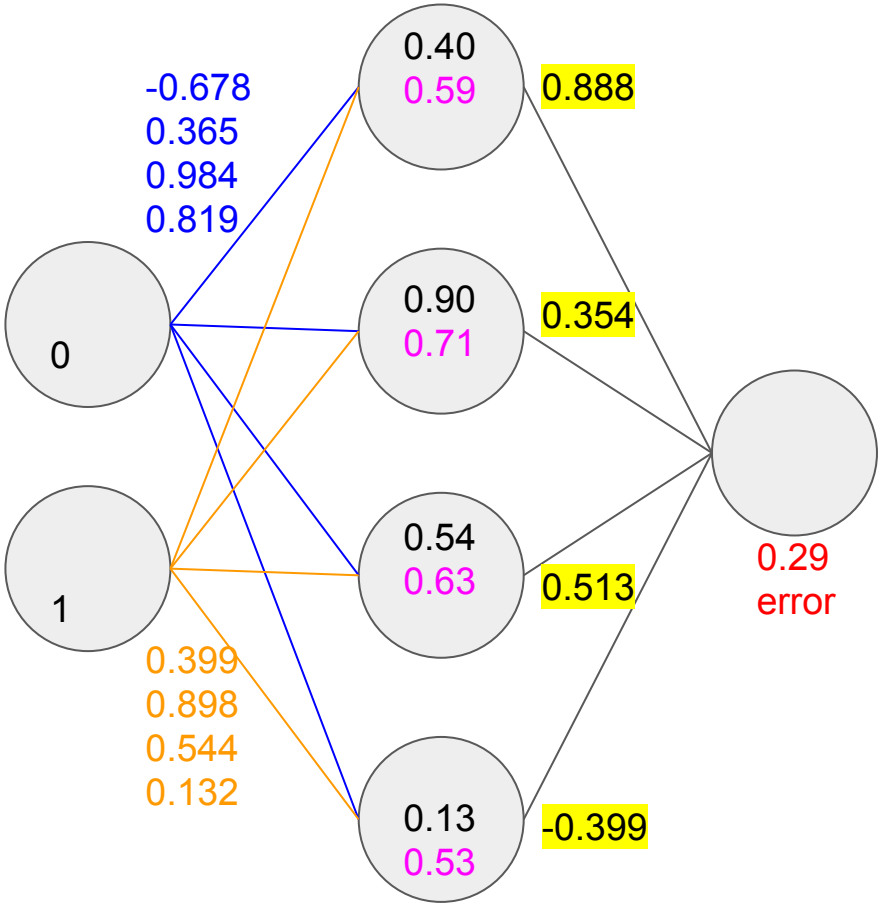
```
adjustments_h2o_weights = [
   [ 0.0060 ]
]
hidden_transposed = [
   [ 0.5984, 0.7105, 0.6327, 0.5328 ]
]
```

Execute

```
weight_h2o_deltas = Matrix.multiply(adjustments_h2o_weights,
hidden_transposed);
```

After

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



Note

Apply changes to the hidden->output weights

Before

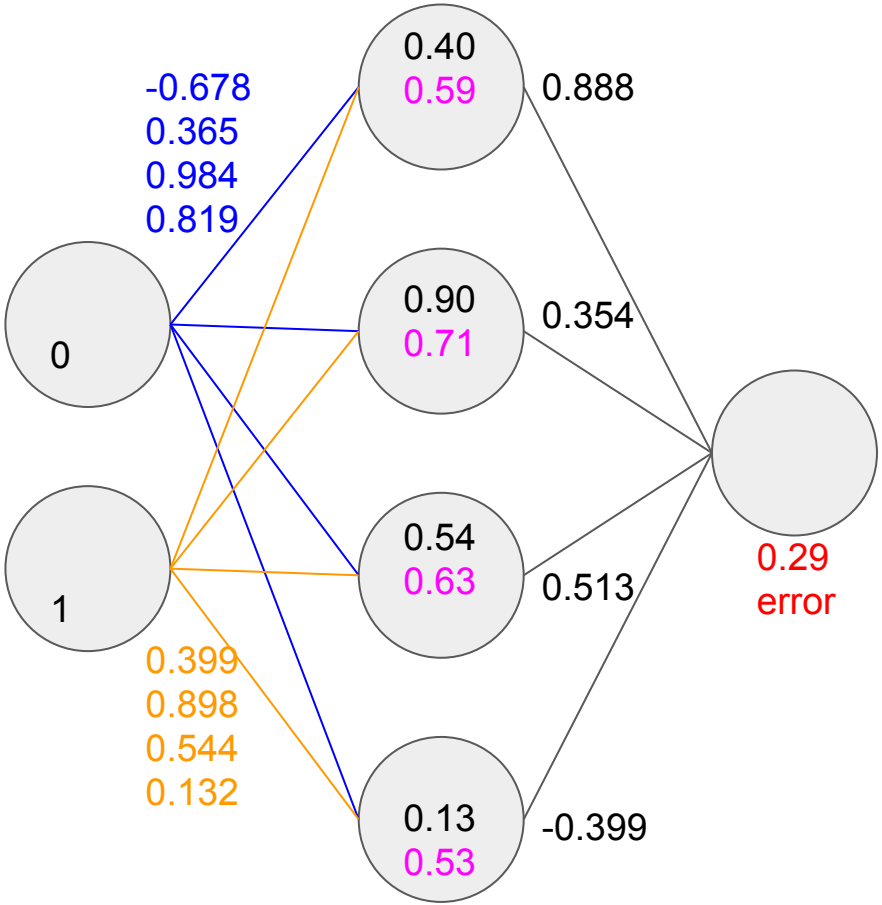
```
weights_h2o = [
   [ 0.8839, 0.3500, 0.5093, -0.4023 ]
]
weight_h2o_deltas = [
   [ 0.0036, 0.0042, 0.0038, 0.0032 ]
]
```

Execute

```
weights_h2o = weights_h2o.add(weight_h2o_deltas);
```

After

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



Note

Transpose the new hidden->output weights for use in later calculations

Before

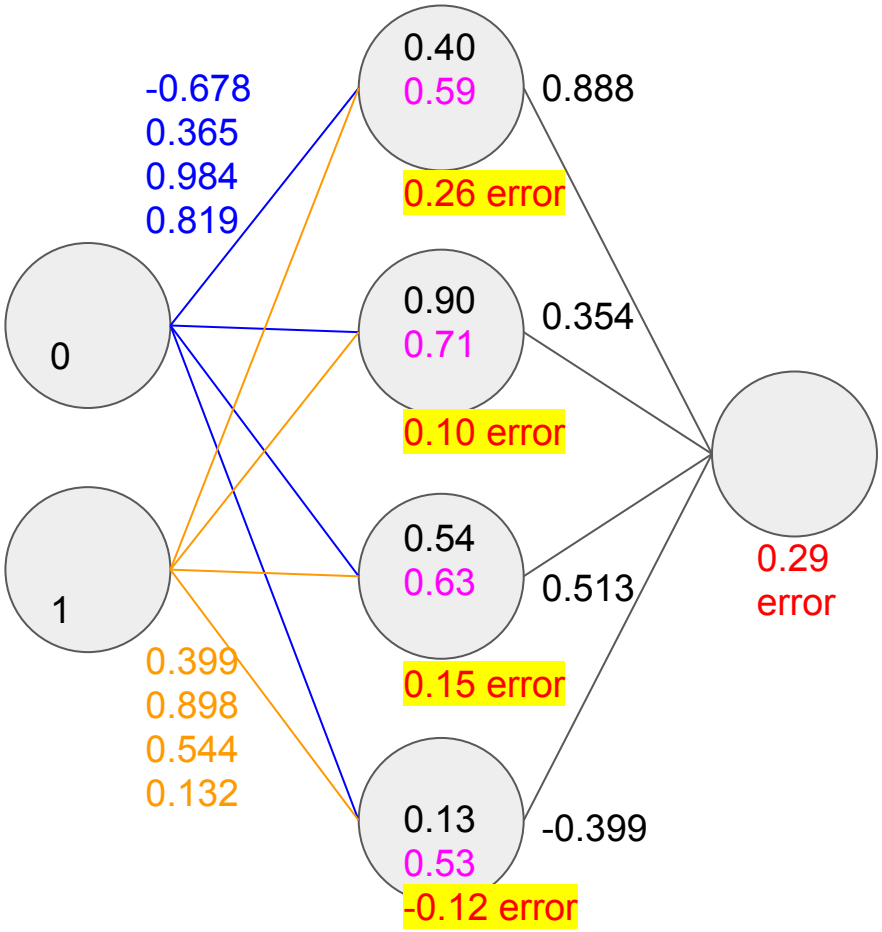
```
weights_h2o = [
   [ 0.8875, 0.3543, 0.5131, -0.3991 ]
]
```

Execute

```
weights_h2o_transposed = Matrix.transpose(weights_h2o);
```

After

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



Note

```
Hidden layer error for each node =
* that node's weighting toward the output multiplied by
* the error of the output
```

Before

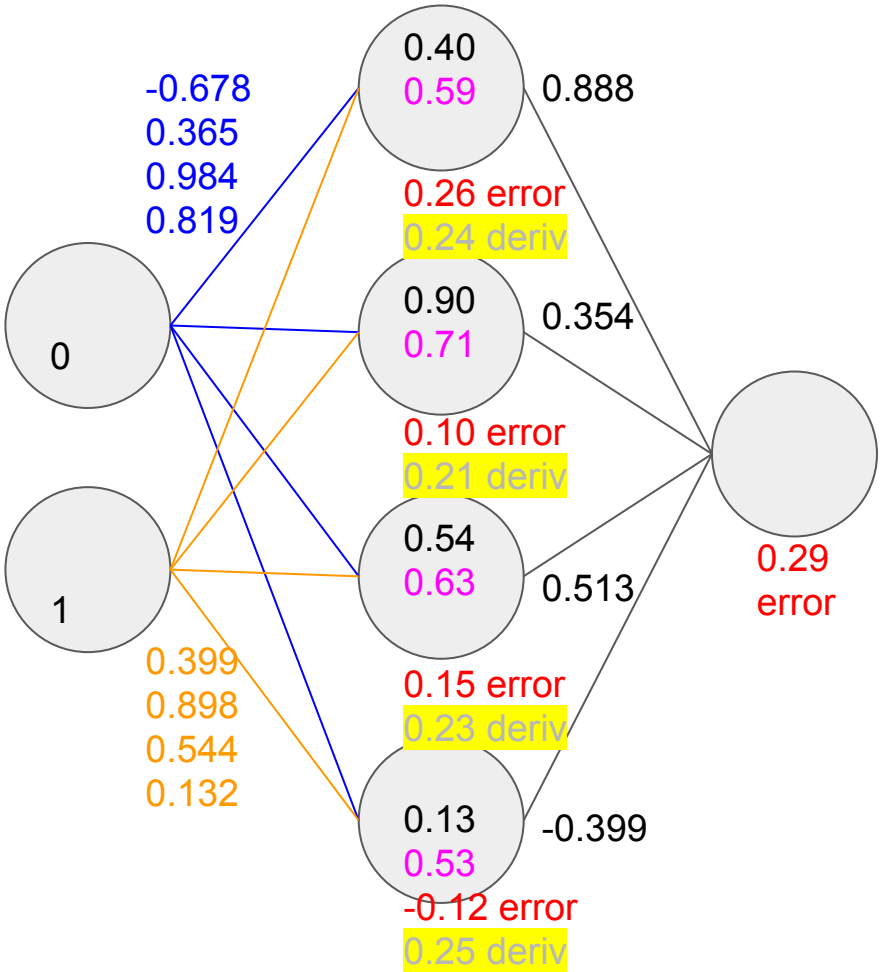
```
weights_h2o_transposed = [
   [ 0.8875 ],
   [ 0.3543 ],
   [ 0.5131 ],
   [ -0.3991 ]
]
output_errors = [
   [ 0.2920 ]
]
```

Execute

```
hidden_errors = Matrix.multiply(weights_h2o_transposed,
output_errors);
```

After

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



Note

Find the gradient for each hidden-node value (again, to determine degree of shifting we should enforce)

Before

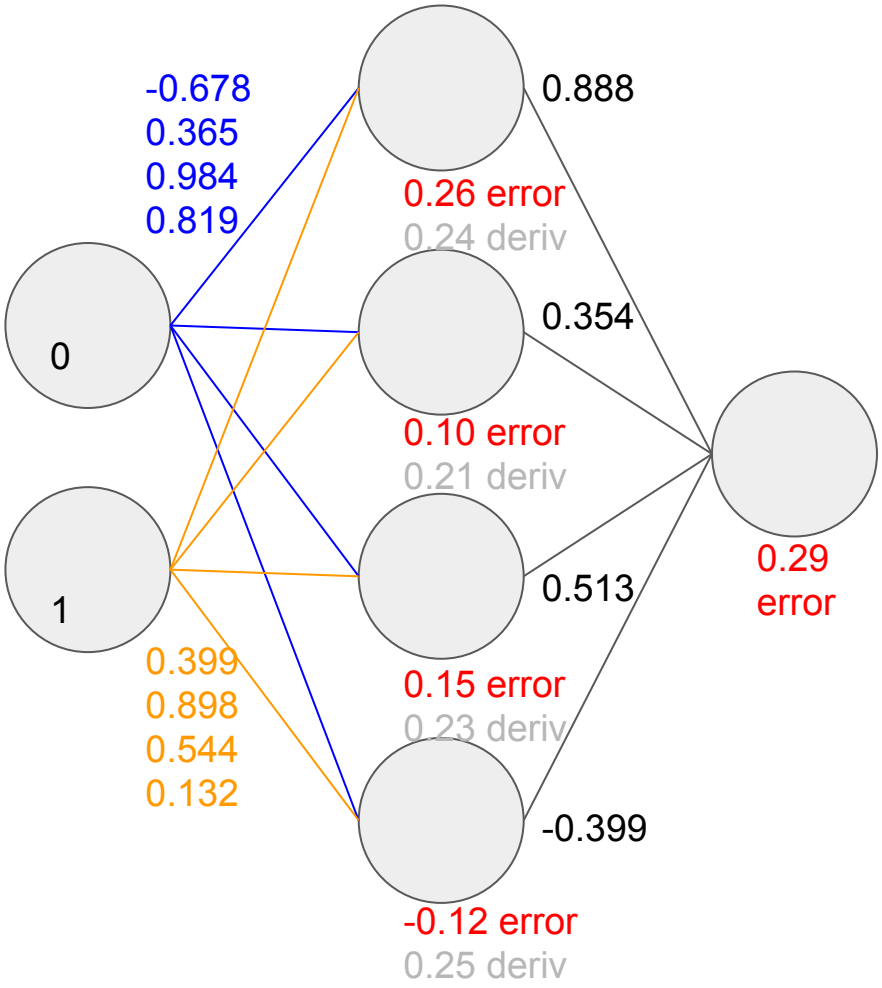
```
hidden = [
    [ 0.5984 ],
    [ 0.7105 ],
    [ 0.6327 ],
    [ 0.5328 ]
]
```

Execute

```
hidden_gradients = Matrix.map(hidden, sigmoid_derivative);
```

After

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



Note

Multiply "hidden layer errors" by the "gradients" to find adjustment factor required for each incoming path weighting.

Before

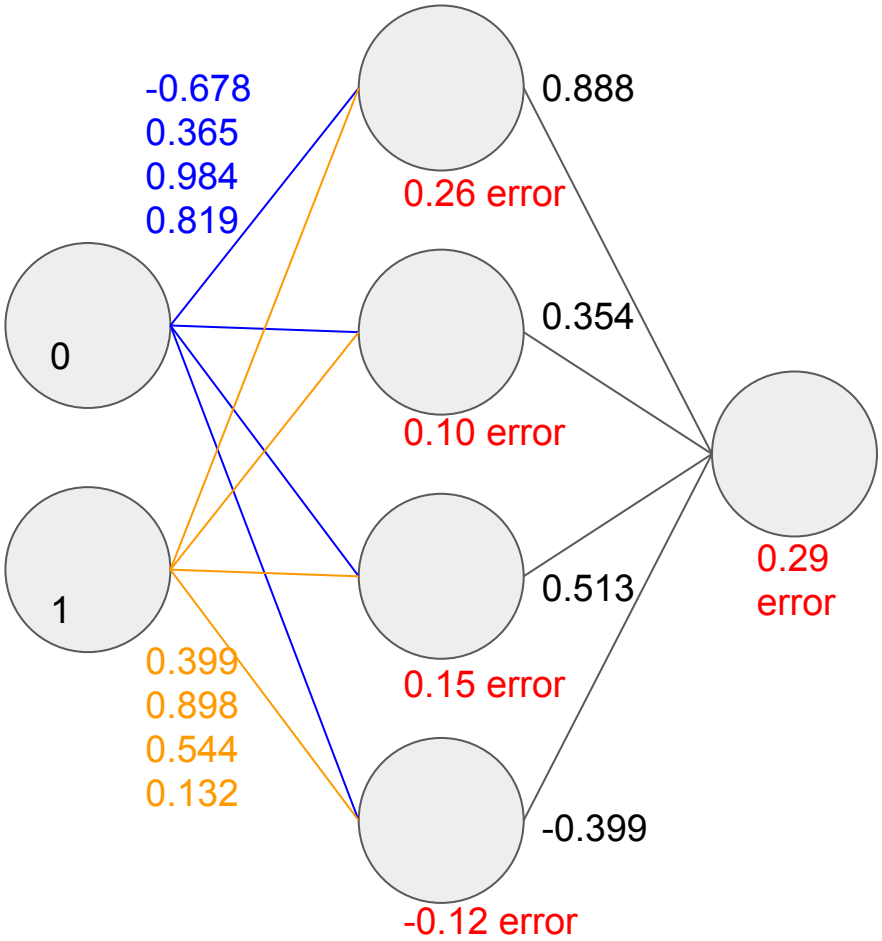
```
hidden_gradient = [
   [ 0.2403 ],
   [ 0.2056 ],
   [ 0.2323 ],
   [ 0.2489 ]
]
hidden_errors = [
   [ 0.2591 ],
   [ 0.1034 ],
   [ 0.1498 ],
   [ -0.1165 ]
]
```

Execute

```
adjustment_i2h_weights= hidden_gradient.multiply(hidden_errors)
```

After

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



Note

Apply learning rate (again, so no one individual training dutum has too much influence on the NN)

Before

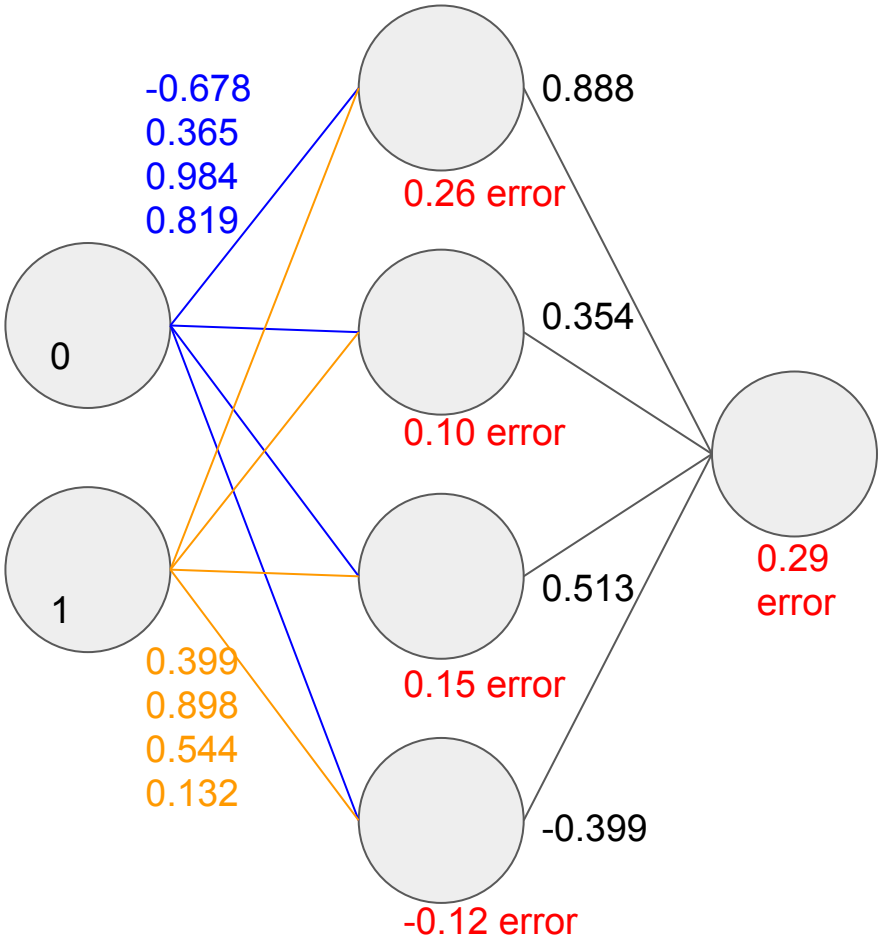
```
adjustment_i2h_weights = [
   [ 0.0622 ],
   [ 0.0212 ],
   [ 0.0348 ],
   [ -0.0290 ]
]
```

Execute

```
adjustment_i2h_weights.multiply(learning_rate);
```

After

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



Note

Transpose the inputs so we can use them on the next step

Before

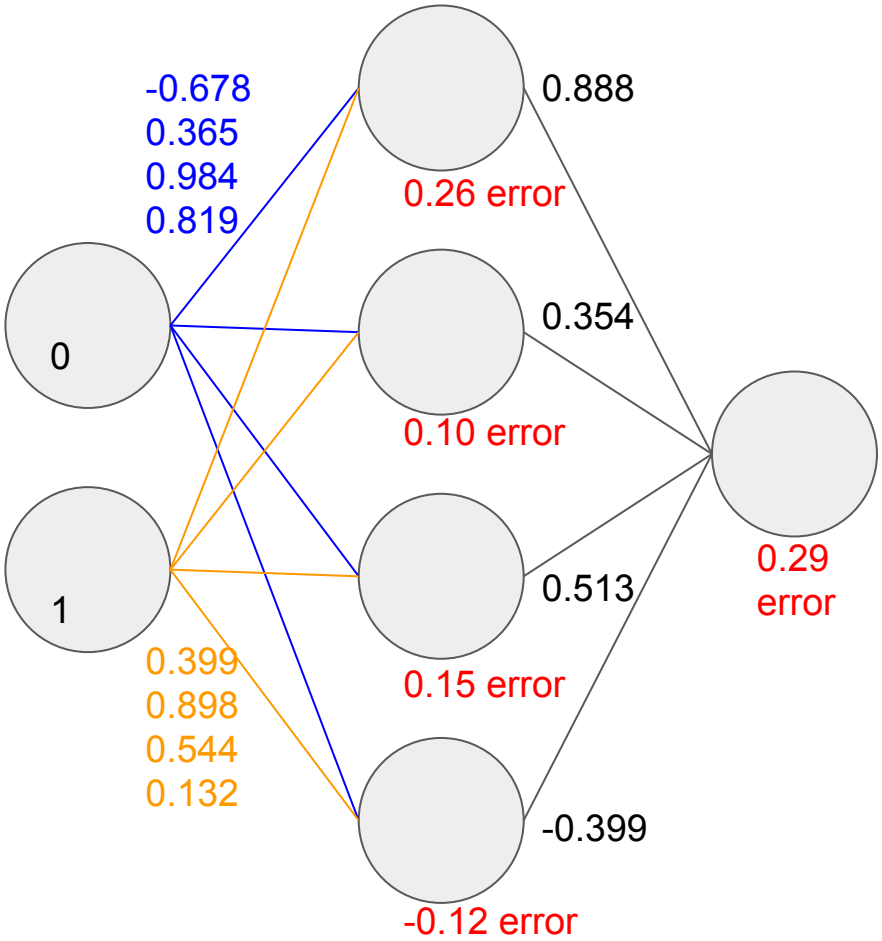
```
inputs = [
   [ 0 ],
   [ 1 ]
]
```

Execute

```
inputs_transposed = Matrix.transpose(inputs);
```

After

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



Note

Calculate amount of change that should apply to each input->hidden weight.

Before

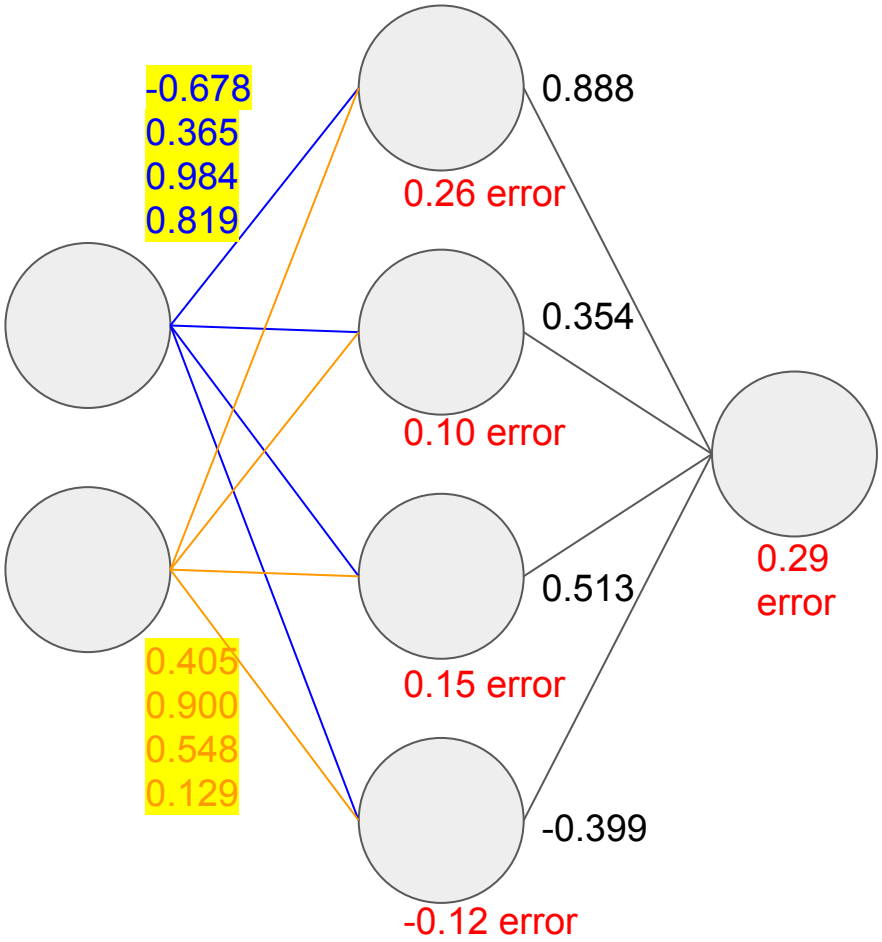
```
adjustment_i2h_weights = [
   [ 0.00622 ],
   [ 0.00212 ],
   [ 0.00348 ],
   [ -0.00290 ]
]
inputs_transposed = [
   [ 0, 1 ]
]
```

Execute

```
weight_i2h_deltas = Matrix.multiply(adjustment,
inputs_transposed);
```

After

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



Note

Apply changes to input->hidden weights

Before

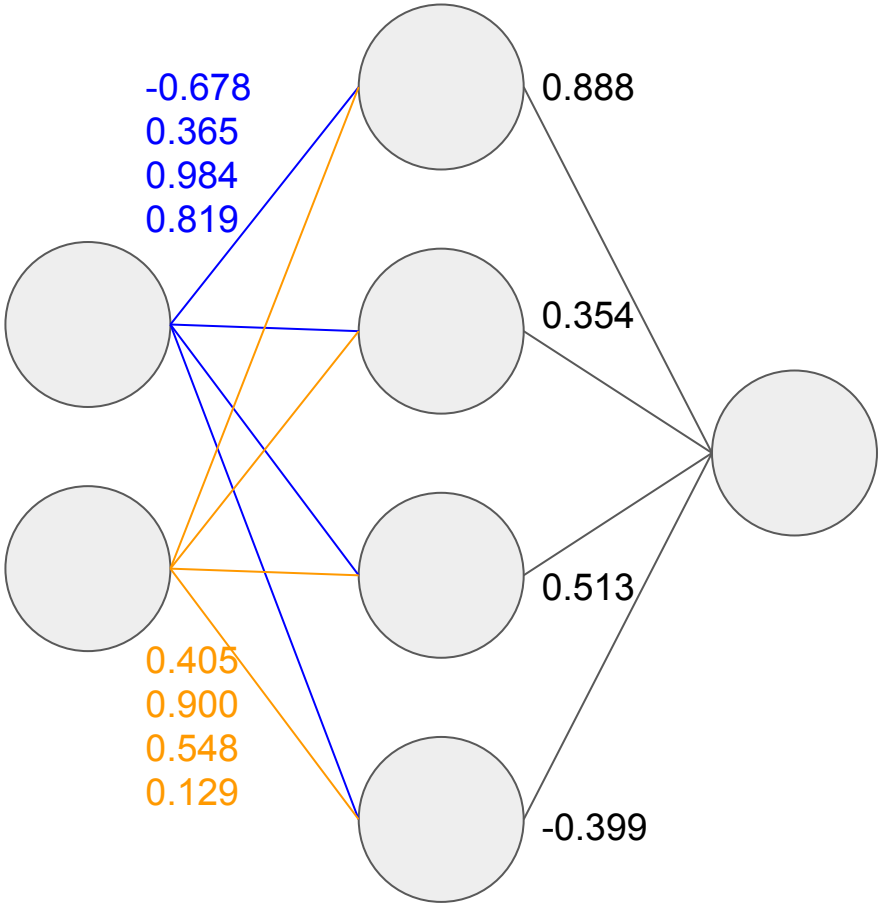
```
weights_i2h = [
   [ -0.6784, 0.3990 ],
   [ 0.3645, 0.8981 ],
   [ 0.9840, 0.5440 ],
   [ 0.8190, 0.1316 ]
]
weight_i2h_deltas = [
   [ 0, 0.0062 ],
   [ 0, 0.0021 ],
   [ 0, 0.0034 ],
   [ 0, -0.0029 ]
]
```

Execute

```
weights_i2h.add(weight_i2h_deltas);
```

After

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



Note

Ready to perform more training!