**Backpropagation**

Back-propagation is a supervised learning algorithm, for training Multi-layer perceptrons (Artificial Neural Networks)

The Backpropagation algorithm looks for the minimum value of the error function in weight space using a technique called the delta rule or **gradient descent**. The weights that minimize the error function are then considered to be a solution to the learning problem.

**Why We Need Backpropagation?**

While designing a Neural Network, in the beginning, we initialize weights with some random values or any variable for that fact.

Now, we have selected some weight values in the beginning, but our model output is way different than our actual output i.e. the error value is huge.

Now, how will you reduce the error?

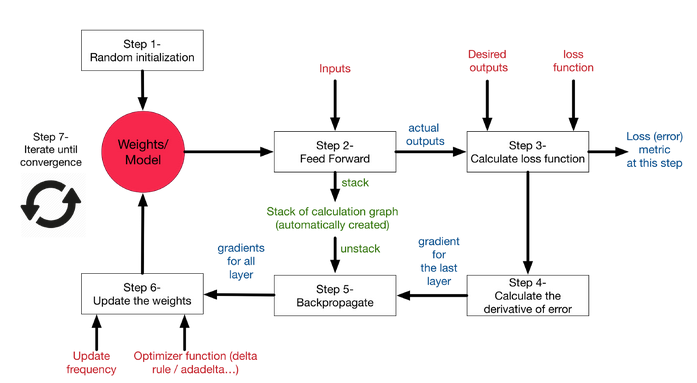
Basically, what we need to do, we need to somehow explain the model to change the parameters (weights), such that error becomes minimum.

Let’s put it in an another way, we need to train our model.

One way to train our model is called as Backpropagation.

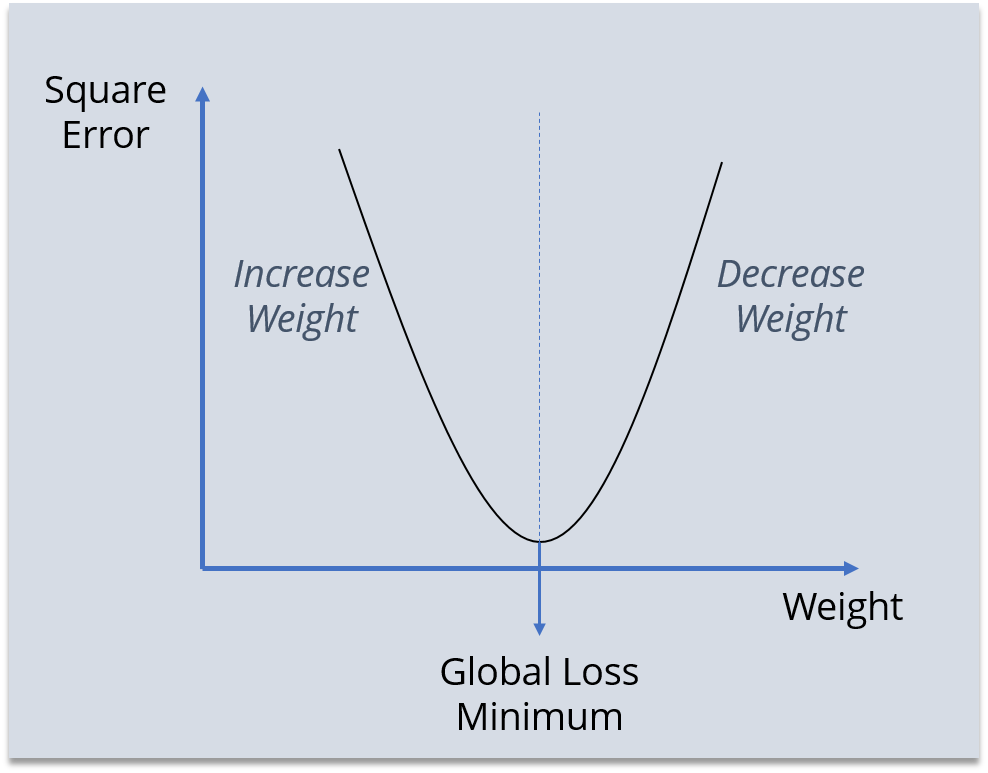
**Steps involved:**

1. **Calculate the error** – How far is your model output from the actual output.
2. **Error minimum?** – Check whether the error is minimized or not.
3. **Update the parameters** – If the error is huge then, update the parameters (weights and biases). After that check the error again. Repeat the process until the error becomes minimum
4. **Model is ready to make a prediction** – Once the error becomes minimum, you can feed some inputs to your model and it will produce the output.



Basically, we need to figure out whether we need to increase or decrease the weight value. Once we know that, we keep on updating the weight value in that direction until error becomes minimum. You might reach a point, where if you further update the weight, the error will increase. At that time you need to stop, and that is your final weight value.

Consider the graph below:



We need to reach the ‘Global Loss Minimum’.

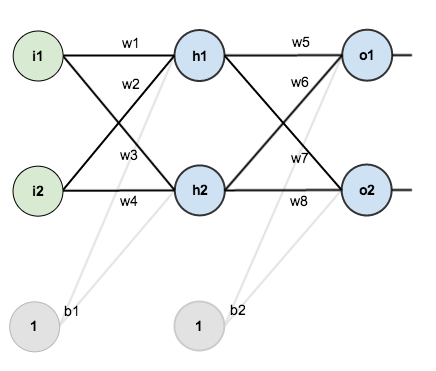
This is nothing but Backpropagation.

Let’s now understand the math behind Backpropagation.

## ****How Backpropagation Works?****

To make things simpler, we’re going to use a neural network with two inputs, two hidden neurons, two output neurons. Additionally, the hidden and output neurons will include a bias.

Here’s the basic structure:



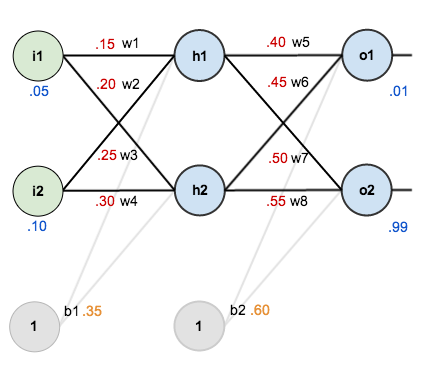
The above network contains the following:

* two inputs
* two hidden neurons
* two output neurons
* two biases

Below are the steps involved in Backpropagation:

* Step – 1: Forward Propagation
* Step – 2: Backward Propagation
* Step – 3: Putting all the values together and calculating the updated weight value

In order to have some numbers to work with, here are the initial weights, the biases, and training inputs/outputs:



The goal of backpropagation is to optimize the weights so that the neural network can learn how to correctly map arbitrary inputs to outputs.

For the rest of this tutorial we’re going to work with a single training set: given inputs 0.05 and 0.10, we want the neural network to output 0.01 and 0.99.

**The Forward Pass**

To begin, lets see what the neural network currently predicts given the weights and biases above and inputs of 0.05 and 0.10. To do this we’ll feed those inputs forward though the network.

We figure out the *total net input* to each hidden layer neuron, *squash* the total net input using an *activation function* (here we use the *logistic function*), then repeat the process with the output layer neurons.

Total net input is also referred to as just *net input*

Here’s how we calculate the total net input for h_1 :

net_{h1} = w_1 * i_1 + w_2 * i_2 + b_1 * 1

net_{h1} = 0.15 * 0.05 + 0.2 * 0.1 + 0.35 * 1 = 0.3775

We then squash it using the logistic function to get the output of h_1:

out_{h1} = \frac{1}{1+e^{-net_{h1}}} = \frac{1}{1+e^{-0.3775}} = 0.593269992

Carrying out the same process for h_2we get:

out_{h2} = 0.596884378

We repeat this process for the output layer neurons, using the output from the hidden layer neurons as inputs.

Here’s the output for o_1:

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

net_{o1} = 0.4 * 0.593269992 + 0.45 * 0.596884378 + 0.6 * 1 = 1.105905967

out_{o1} = \frac{1}{1+e^{-net_{o1}}} = \frac{1}{1+e^{-1.105905967}} = 0.75136507

And carrying out the same process for o_2we get:

out_{o2} = 0.772928465

**Calculating the Total Error**

We can now calculate the error for each output neuron using the [squared error function](http://en.wikipedia.org/wiki/Backpropagation#Derivation) and sum them to get the total error:

E_{total} = \sum \frac{1}{2}(target - output)^{2}

The \frac{1}{2} is included so that exponent is cancelled when we differentiate later on. The result is eventually multiplied by a learning rate anyway so it doesn’t matter that we introduce a constant here.

For example, the target output for o_1is 0.01 but the neural network output 0.75136507, therefore its error is:

E_{o1} = \frac{1}{2}(target_{o1} - out_{o1})^{2} = \frac{1}{2}(0.01 - 0.75136507)^{2} = 0.274811083

Repeating this process for o_2(remembering that the target is 0.99) we get:

E_{o2} = 0.023560026

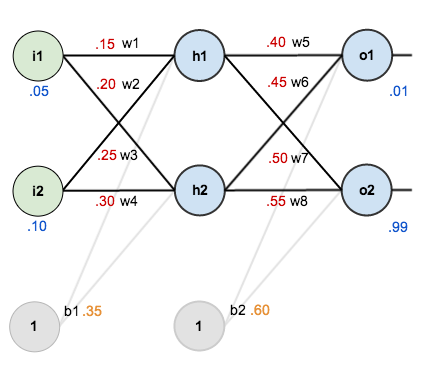
The total error for the neural network is the sum of these errors:

E_{total} = E_{o1} + E_{o2} = 0.274811083 + 0.023560026 = 0.298371109

**The Backwards Pass**

Our goal with backpropagation is to update each of the weights in the network so that they cause the actual output to be closer the target output, thereby minimizing the error for each output neuron and the network as a whole.

**Output Layer**

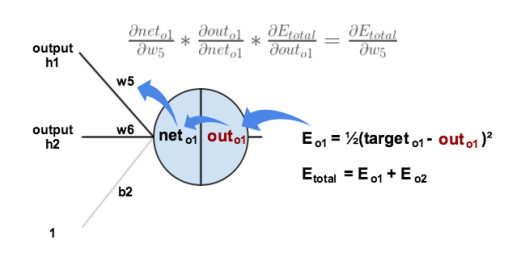
****

Consider w_5. We want to know how much a change in w_5affects the total error, aka \frac{\partial E_{total}}{\partial w_{5}}.

\frac{\partial E_{total}}{\partial w_{5}}is read as “the partial derivative of E_{total}with respect to w_{5}“. You can also say “the gradient with respect to w_{5}“.

By applying the [chain rule](http://en.wikipedia.org/wiki/Chain_rule) we know that:

Visually, here’s what we’re doing:



We need to figure out each piece in this equation.

First, how much does the total error change with respect to the output?

E_{total} = \frac{1}{2}(target_{o1} - out_{o1})^{2} + \frac{1}{2}(target_{o2} - out_{o2})^{2}

\frac{\partial E_{total}}{\partial out_{o1}} = 2 * \frac{1}{2}(target_{o1} - out_{o1})^{2 - 1} * -1 + 0

\frac{\partial E_{total}}{\partial out_{o1}} = -(target_{o1} - out_{o1}) = -(0.01 - 0.75136507) = 0.74136507

-(target - out)is sometimes expressed as out - target

When we take the partial derivative of the total error with respect to out_{o1}, the quantity \frac{1}{2}(target_{o2} - out_{o2})^{2}becomes zero because out_{o1}does not affect it which means we’re taking the derivative of a constant which is zero.

Next, how much does the output of o_1change with respect to its total net input?

The partial [derivative of the logistic function](http://en.wikipedia.org/wiki/Logistic_function#Derivative) is the output multiplied by 1 minus the output:

out_{o1} = \frac{1}{1+e^{-net_{o1}}}

\frac{\partial out_{o1}}{\partial net_{o1}} = out_{o1}(1 - out_{o1}) = 0.75136507(1 - 0.75136507) = 0.186815602

Finally, how much does the total net input of o1change with respect to w_5?

net_{o1} = w_5 * out_{h1} + w_6 * out_{h2} + b_2 * 1

\frac{\partial net_{o1}}{\partial w_{5}} = 1 * out_{h1} * w_5^{(1 - 1)} + 0 + 0 = out_{h1} = 0.593269992

Putting it all together:

\frac{\partial E_{total}}{\partial w_{5}} = 0.74136507 * 0.186815602 * 0.593269992 = 0.082167041

You’ll often see this calculation combined in the form of the [delta rule](http://en.wikipedia.org/wiki/Delta_rule):

\frac{\partial E_{total}}{\partial w_{5}} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1}) * out_{h1}

Alternatively, we have \frac{\partial E_{total}}{\partial out_{o1}}and \frac{\partial out_{o1}}{\partial net_{o1}}which can be written as \frac{\partial E_{total}}{\partial net_{o1}}, aka \delta_{o1}(the Greek letter delta) aka the *node delta*. We can use this to rewrite the calculation above:

\delta_{o1} = -(target_{o1} - out_{o1}) * out_{o1}(1 - out_{o1})

Therefore:

\frac{\partial E_{total}}{\partial w_{5}} = \delta_{o1} out_{h1}

Some sources extract the negative sign from \deltaso it would be written as:

\frac{\partial E_{total}}{\partial w_{5}} = -\delta_{o1} out_{h1}

To decrease the error, we then subtract this value from the current weight (optionally multiplied by some learning rate, eta, which we’ll set to 0.5):

w_5^{+} = w_5 - \eta * \frac{\partial E_{total}}{\partial w_{5}} = 0.4 - 0.5 * 0.082167041 = 0.35891648

[Some](http://en.wikipedia.org/wiki/Delta_rule) [sources](http://aima.cs.berkeley.edu/) use \alpha(alpha) to represent the learning rate, [others use](https://www4.rgu.ac.uk/files/chapter3%20-%20bp.pdf) \eta(eta), and [others](http://web.cs.swarthmore.edu/~meeden/cs81/s10/BackPropDeriv.pdf) even use \epsilon(epsilon).

We can repeat this process to get the new weights w_6, w_7, and w_8:

w_6^{+} = 0.408666186

w_7^{+} = 0.511301270

w_8^{+} = 0.561370121

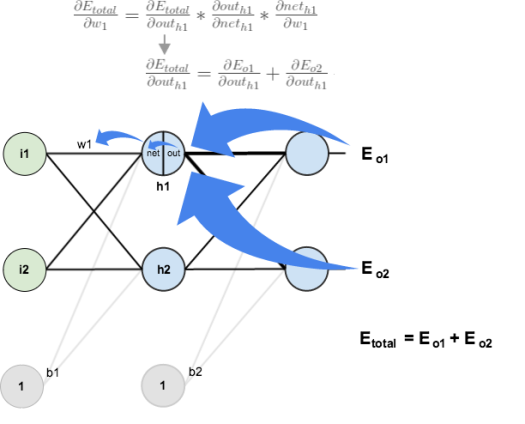
We perform the actual updates in the neural network *after* we have the new weights leading into the hidden layer neurons (ie, we use the original weights, not the updated weights, when we continue the backpropagation algorithm below).

**Hidden Layer**

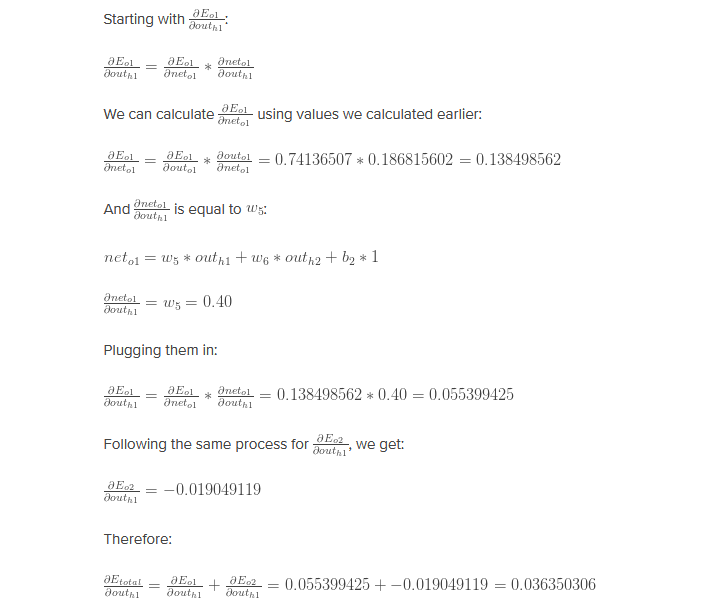
Next, we’ll continue the backwards pass by calculating new values for w_1, w_2, w_3, and w_4.

Big picture, here’s what we need to figure out:

Visually:

[](https://matthewmazur.files.wordpress.com/2015/03/nn-calculation.png)

We’re going to use a similar process as we did for the output layer, but slightly different to account for the fact that the output of each hidden layer neuron contributes to the output (and therefore error) of multiple output neurons. We know that out_{h1}affects both out_{o1}and out_{o2}therefore the \frac{\partial E_{total}}{\partial out_{h1}}needs to take into consideration its effect on the both output neurons:



Now that we have \frac{\partial E_{total}}{\partial out_{h1}}, we need to figure out \frac{\partial out_{h1}}{\partial net_{h1}}and then \frac{\partial net_{h1}}{\partial w}for each weight:

out_{h1} = \frac{1}{1+e^{-net_{h1}}}

\frac{\partial out_{h1}}{\partial net_{h1}} = out_{h1}(1 - out_{h1}) = 0.59326999(1 - 0.59326999 ) = 0.241300709

We calculate the partial derivative of the total net input to h_1with respect to w_1the same as we did for the output neuron:

net_{h1} = w_1 * i_1 + w_3 * i_2 + b_1 * 1

\frac{\partial net_{h1}}{\partial w_1} = i_1 = 0.05

Putting it all together:

\frac{\partial E_{total}}{\partial w_{1}} = 0.036350306 * 0.241300709 * 0.05 = 0.000438568

You might also see this written as:

\frac{\partial E_{total}}{\partial w_{1}} = (\sum\limits_{o}{\delta_{o} * w_{ho}}) * out_{h1}(1 - out_{h1}) * i_{1}

\frac{\partial E_{total}}{\partial w_{1}} = \delta_{h1}i_{1}

We can now update w_1:

w_1^{+} = w_1 - \eta * \frac{\partial E_{total}}{\partial w_{1}} = 0.15 - 0.5 * 0.000438568 = 0.149780716

Repeating this for w_2, w_3, and w_4

w_2^{+} = 0.19956143

w_3^{+} = 0.24975114

w_4^{+} = 0.29950229

Finally, we’ve updated all of our weights! When we fed forward the 0.05 and 0.1 inputs originally, the error on the network was 0.298371109. After this first round of back-propagation, the total error is now down to 0.291027924. It might not seem like much, but after repeating this process 10,000 times, for example, the error plummets to 0.0000351085. At this point, when we feed forward 0.05 and 0.1, the two outputs neurons generate 0.015912196 (vs 0.01 target) and 0.984065734 (vs 0.99 target).