1) What is Activation Function?

It’s just a thing function that you use to get the output of node. It is also known as Transfer Function.

2) Why we use Activation functions with Neural Networks?

It is used to determine the output of neural network like yes or no.

It maps the resulting values in between 0 to 1 or -1 to 1 etc. (depending upon the function).

The Activation Functions can be basically divided into 2 types-

Linear Activation Function

Non-linear Activation Functions

Basically we deal with non leaner activation function as linear function does not help with the complexity or various parameters of usual data

that is fed to Neural Network.

**Non-linear Activation Function**

The Nonlinear Activation Functions are the most used activation functions. Nonlinearity helps to makes the graph look something like this. It makes it easy for the model to generalize or adapt with variety of data and to differentiate between the output.



The main terminologies needed to understand for nonlinear functions are:

***Derivative or Differential:*** *Change in y-axis w.r.t. change in x-axis.It is also known as slope.*

***Monotonic function:*** *A function which is either entirely non-increasing or non-decreasing.*

The Nonlinear Activation Functions are mainly divided on the basis of their **range or curves**-

## ****1. Sigmoid or Logistic Activation Function****

he Sigmoid Function curve looks like a S-shape.



The main reason why we use sigmoid function is because it exists between **(0 to 1).** Therefore, it is especially used for models where we have to **predict the probability** as an output.Since probability of anything exists only between the range of **0 and 1,** sigmoid is the right choice. Its monotonic function.

The function is **differentiable**. That means, we can find the slope of the sigmoid curve at any two points.

The **softmax function** is a more generalized logistic activation function which is used for multiclass classification.

import numpy as np

import math

x=np.arange(-11,10,0.2)

f=1/(1+np.exp(-x))

print(f)

plt.scatter(x, f)

plt.show()

Tanh or hyperbolic tangent Activation Function:

tanh is also like logistic sigmoid but better. The range of the tanh function is from (-1 to 1). tanh is also sigmoidal (s - shaped).



**Fig: tanh v/s Logistic Sigmoid**

*Both tanh and logistic sigmoid activation functions are used in feed-forward nets.*

3. ReLU (Rectified Linear Unit) Activation Function:

The ReLU is the most used activation function in the world right now.Since, it is used in almost all the convolutional neural networks or deep learning.



**Fig: ReLU v/s Logistic Sigmoid**

As you can see, the ReLU is half rectified (from bottom). f(z) is zero when z is less than zero and f(z) is equal to z when z is above or equal to zero.

**Range:** [ 0 to infinity)

The function and its derivative **both are** **monotonic**.

But the issue is that all the negative values become zero immediately which decreases the ability of the model to fit or train from the data properly.

**Why derivative/differentiation is used ?**

When updating the curve, to know in **which direction** and **how much** to change or update the curve depending upon the slope.That is why we use differentiation in almost every part of Machine Learning and Deep Learning.



**Fig: Activation Function Cheetsheet**

How Neural Network Works :

[“Exclusive or” (“XOR”) operation](https://en.wikipedia.org/wiki/Exclusive_or) to illustrate each step in the training process.

### **Forward Propagation**

The XOR function can be represented by the mapping of the below inputs and outputs, which we’ll use as training data. It should provide a correct output given any input acceptable by the XOR function.

input | output

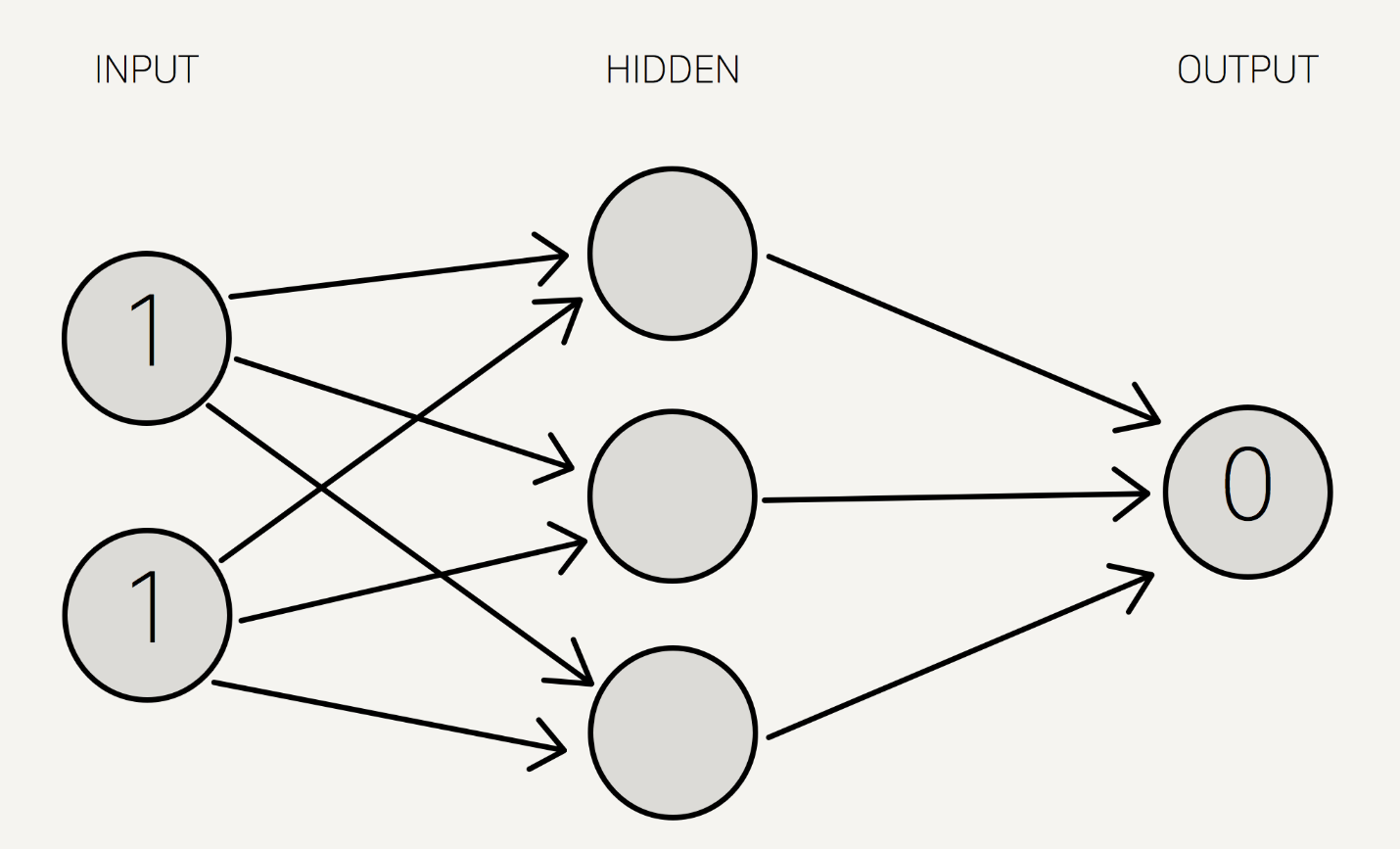
--------------

0, 0 | 0

0, 1 | 1

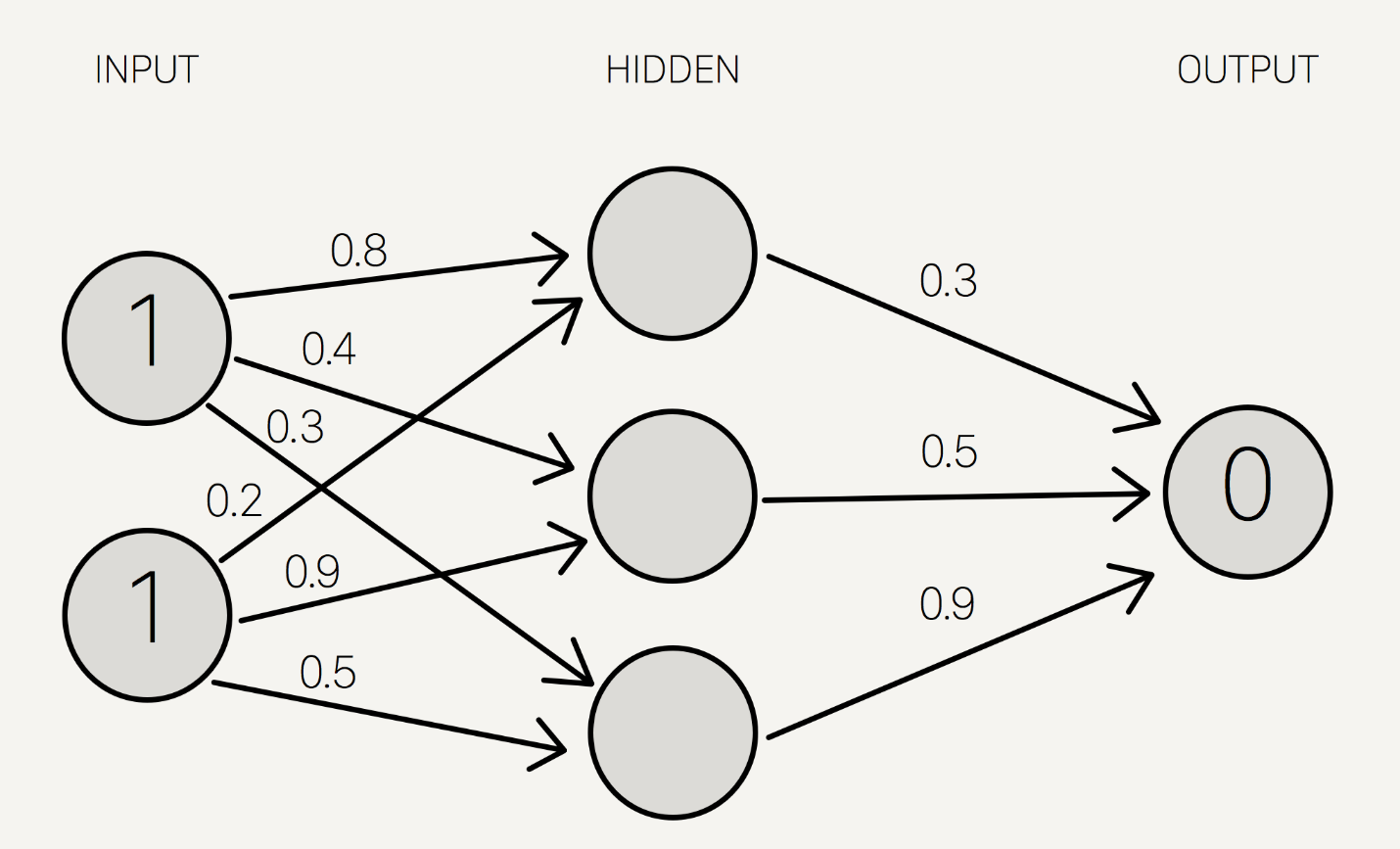
1, 0 | 1

1, 1 | 0



*Note that we use a single hidden layer with only three neurons for this example.*

We now assign weights to all of the synapses. Note that these weights are selected randomly (based on Gaussian distribution) since it is the first time we’re forward propagating. The initial weights will be between 0 and 1, but note that the final weights don’t need to be.



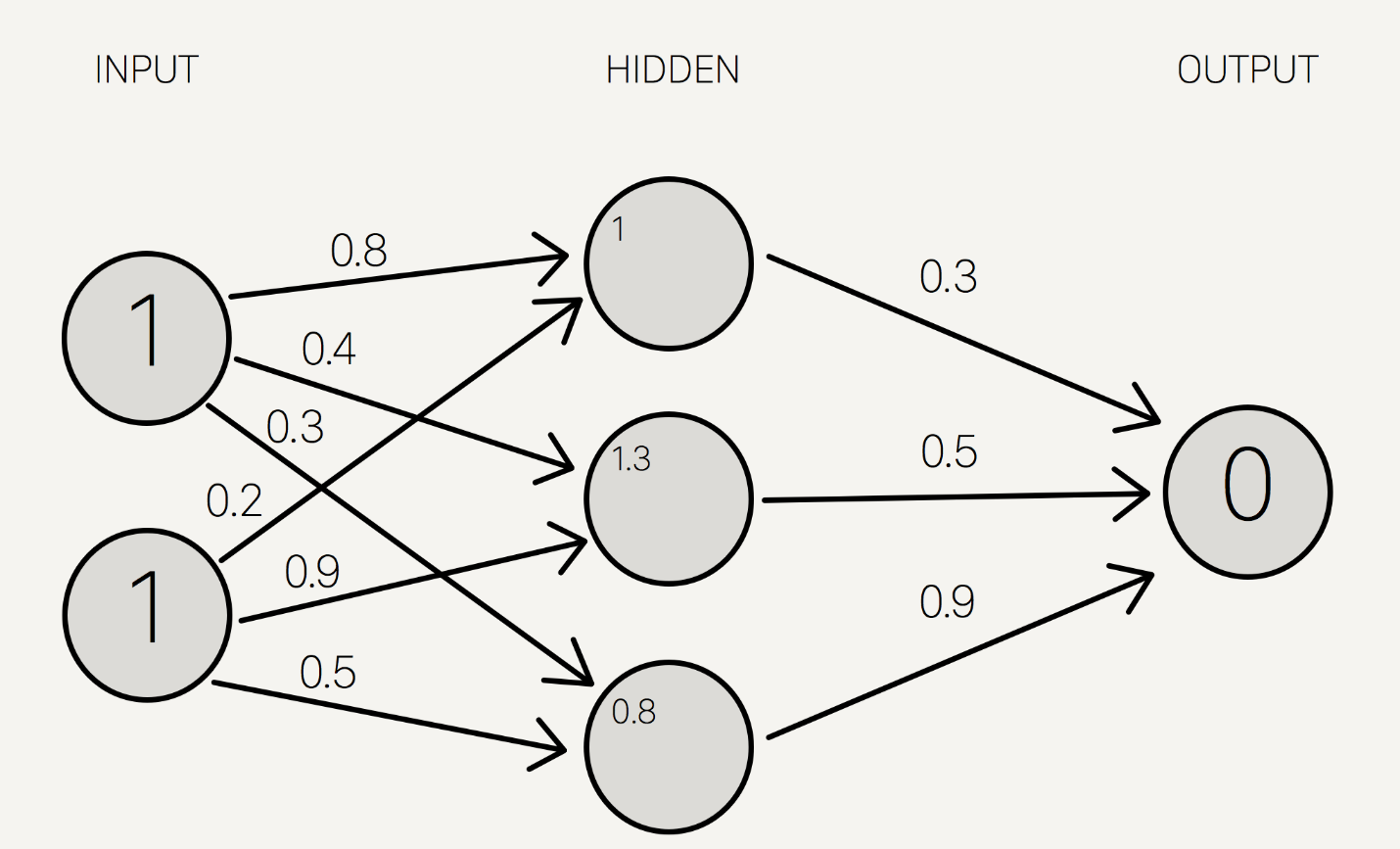
We sum the product of the inputs with their corresponding set of weights to arrive at the first values for the hidden layer. You can think of the weights as measures of influence the input nodes have on the output.

1 \* 0.8 + 1 \* 0.2 = 1

1 \* 0.4 + 1 \* 0.9 = 1.3

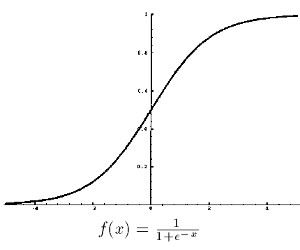
1 \* 0.3 + 1 \* 0.5 = 0.8

We put these sums smaller in the circle, because they’re not the final value:



To get the final value, we apply the [activation function](https://en.wikipedia.org/wiki/Activation_function) to the hidden layer sums. The purpose of the activation function is to transform the input signal into an output signal.

For our example, let’s use the [sigmoid function](https://en.wikipedia.org/wiki/Sigmoid_function) for activation. The sigmoid function looks like this, graphically:



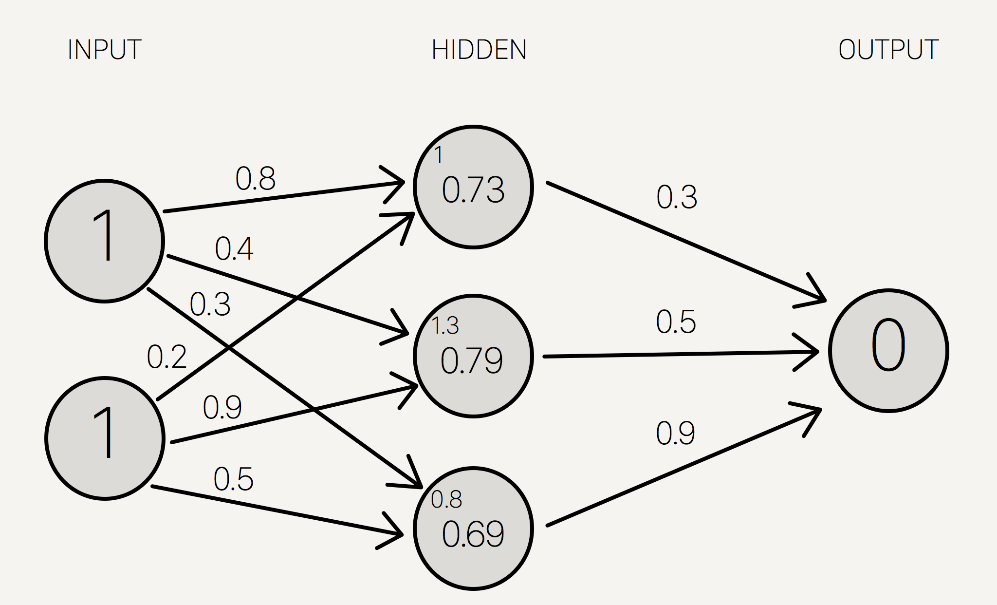
And applying S(x) to the three hidden layer sums, we get:

S(1.0) = 0.73105857863

S(1.3) = 0.78583498304

S(0.8) = 0.68997448112

We add that to our neural network as hidden layer results:

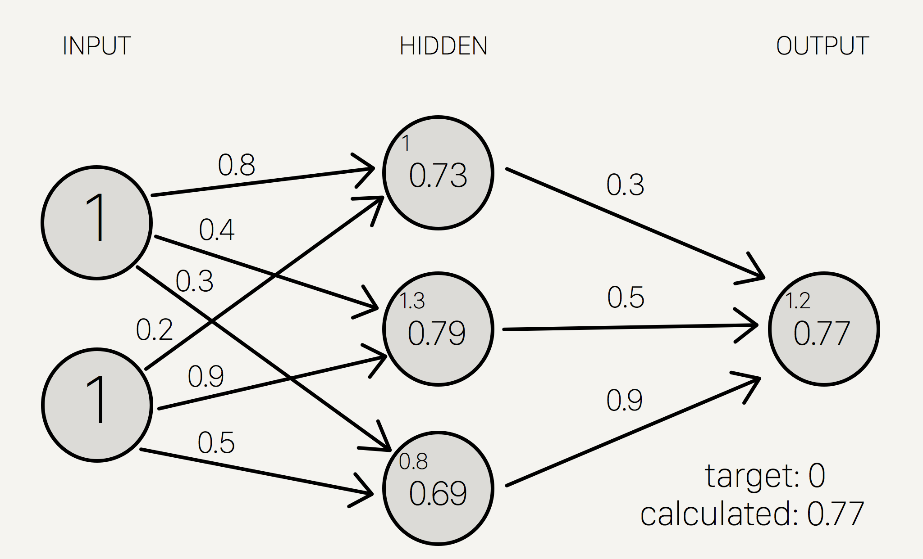


Then, we sum the product of the hidden layer results with the second set of weights (also determined at random the first time around) to determine the output sum.

0.73 \* 0.3 + 0.79 \* 0.5 + 0.69 \* 0.9 = 1.235

..finally we apply the activation function to get the final output result.

S(1.235) = 0.7746924929149283



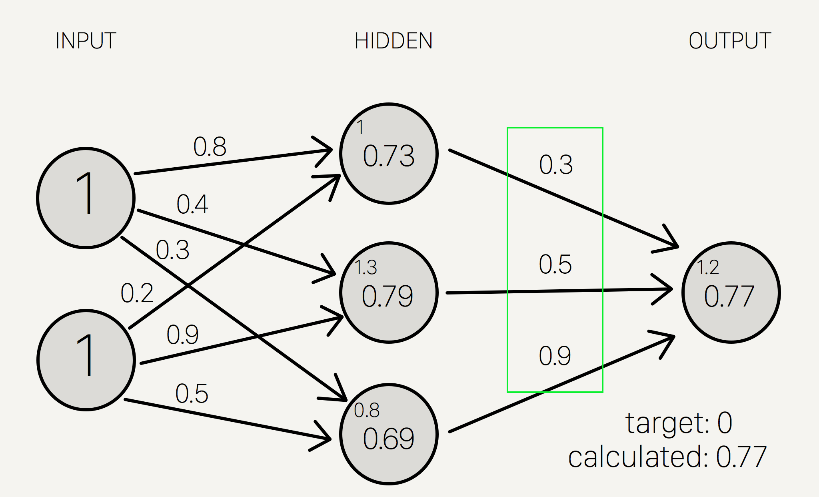
Because we used a random set of initial weights, the value of the output neuron is off the mark; in this case by +0.77 (since the target is 0). If we stopped here, this set of weights would be a great neural network for inaccurately representing the XOR operation.

Let’s fix that by using back propagation to adjust the weights to improve the network!

Back Propagation:

Then, we adjust the weights accordingly so that the margin of errors are decreased.

Similar to forward propagation, back propagation calculations occur at each “layer”. We begin by changing the weights between the hidden layer and the output layer.



The output sum margin of error is the target output result minus the calculated output result:

output sum margin of erro=target -calculated

And doing the math:

Target = 0

Calculated = 0.77

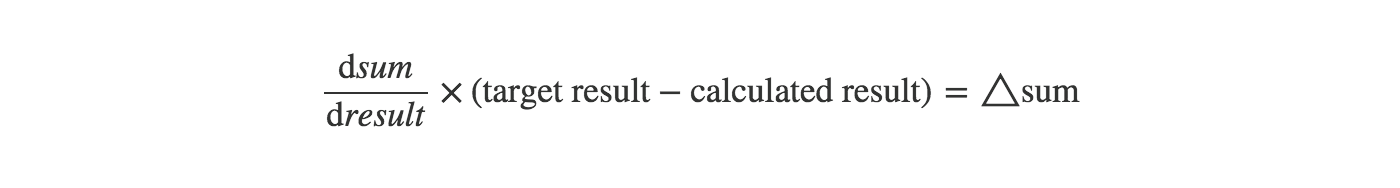
Target - calculated = -0.77

To calculate the necessary change in the output sum, or delta output sum, we take the derivative of the activation function and apply it to the output sum. In our example, the activation function is the sigmoid function.

So the derivative of sigmoid, also known as sigmoid prime, will give us the rate of change (or “slope”) of the activation function at the output sum:

S’(Sum)=dsum/dresult

Since the output sum margin of error is the difference in the result, we can simply multiply that with the rate of change to give us the delta output sum:



Conceptually, this means that the change in the output sum is the same as the sigmoid prime of the output result. Doing the actual math, we get:

Delta output sum = S'(sum) \* (output sum margin of error)

Delta output sum = S'(1.235) \* (-0.77)

Delta output sum = -0.13439890643886018

Let’s do the math:

hidden result 1 = 0.73105857863

hidden result 2 = 0.78583498304

hidden result 3 = 0.68997448112

Delta weights = delta output sum / hidden layer results

Delta weights = -0.1344 / [0.73105, 0.78583, 0.69997]

Delta weights = [-0.1838, -0.1710, -0.1920]

old w7 = 0.3

old w8 = 0.5

old w9 = 0.9

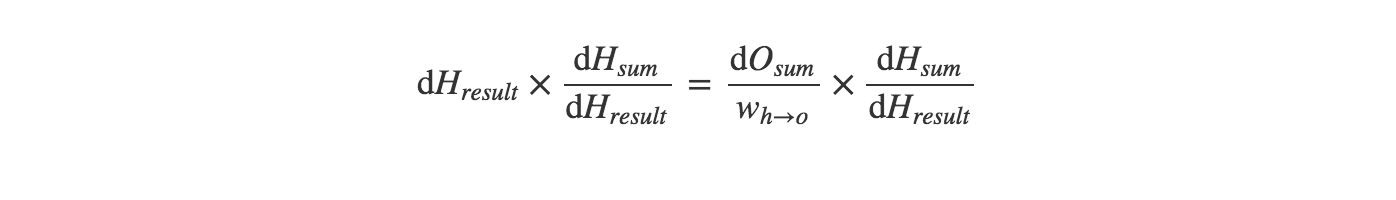
How we get it :

new w7 = (.3-0.1838)=0.1162

new w8 =(.5-.1710)= 0.329

new w9 = (.9-0.1920)=0.708

To determine the change in the weights between the input and hidden layers, we perform the similar, but notably different, set of calculations. Note that in the following calculations, we use the initial weights instead of the recently adjusted weights from the first part of the backward propagation.



Delta hidden sum = delta output sum / hidden-to-outer weights \* S'(hidden sum)

Delta hidden sum = -0.1344 / [0.3, 0.5, 0.9] \* S'([1, 1.3, 0.8])

Delta hidden sum = [-0.448, -0.2688, -0.1493] \* [0.1966, 0.1683, 0.2139]

Delta hidden sum = [-0.088, -0.0452, -0.0319]

Once we get the delta hidden sum, we calculate the change in weights between the input and hidden layer by dividing it with the input data, (1, 1).

Let’s do the math:

input 1 = 1

input 2 = 1

Delta weights = delta hidden sum / input data

Delta weights = [-0.088, -0.0452, -0.0319] / [1, 1]

Delta weights = [-0.088, -0.0452, -0.0319, -0.088, -0.0452, -0.0319]

old w1 = 0.8

old w2 = 0.4

old w3 = 0.3

old w4 = 0.2

old w5 = 0.9

old w6 = 0.5

new w1 = (0.8-.088)=0.712

new w2 = 0.3548

new w3 = 0.2681

new w4 = 0.112

new w5 = 0.8548

new w6 = 0.4681

Here are the new weights, right next to the initial random starting weights as comparison:

old new

-----------------

w1: 0.8 w1: 0.712

w2: 0.4 w2: 0.3548

w3: 0.3 w3: 0.2681

w4: 0.2 w4: 0.112

w5: 0.9 w5: 0.8548

w6: 0.5 w6: 0.4681

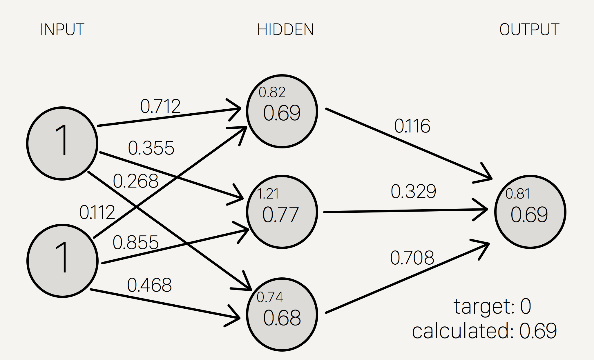
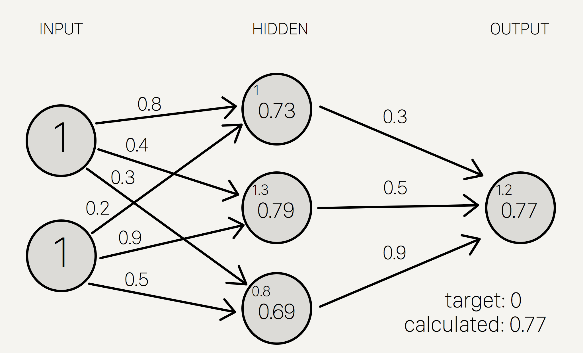
w7: 0.3 w7: 0.1162

w8: 0.5 w8: 0.329

w9: 0.9 w9: 0.708

Once we arrive at the adjusted weights, we start again with forward propagation. When training a neural network, it is common to repeat both these processes thousands of times (by default, Mind iterates 10,000 times).

And doing a quick forward propagation, we can see that the final output here is a little closer to the expected output:

Through just one iteration of forward and back propagation, we’ve already improved the network!!