**Department of Computer Science and Engineering**

|  |  |  |
| --- | --- | --- |
| **Course Code:CSE422** |  | **Credits: 1.5** |
| **Course Name: Artificial Intelligence** |  | **Prerequisite:** CSE111, CSE221 |

**Lab 11  
Neural Network**

1. **Lab Overview:**

This lab is for gaining experience in implementing, tuning, and experimenting with neural networks. On successful completion of these lab, a student will

Understand the philosophy of artificial neural network, and understand how neural networks (NN) work. How to tune the key parameters of NN algorithms.

1. **Learning Objective:**
2. What is NN?
3. How to solve problems using NN?
4. Implement and experiment with NN algorithms and libraries
5. **Lesson Fit:**

There is pre-requisite to this lab: CSE111, CSE221. You should have intensive Programming Knowledge and capability to understand algorithms.

1. **Acceptance and Evaluation**

Performed lab tasks will be evaluated by the Lab Instructor (LI)

* 1. Short viva will be conducted in each Lab or occasionally to examine your work.
  2. You may work in groups but be aware that you will be evaluated individually; hence active participation during the Lab work demonstration is recommended.
  3. There will be Lab handout after your work you have to handover it to LI

1. **Learning Outcome:**

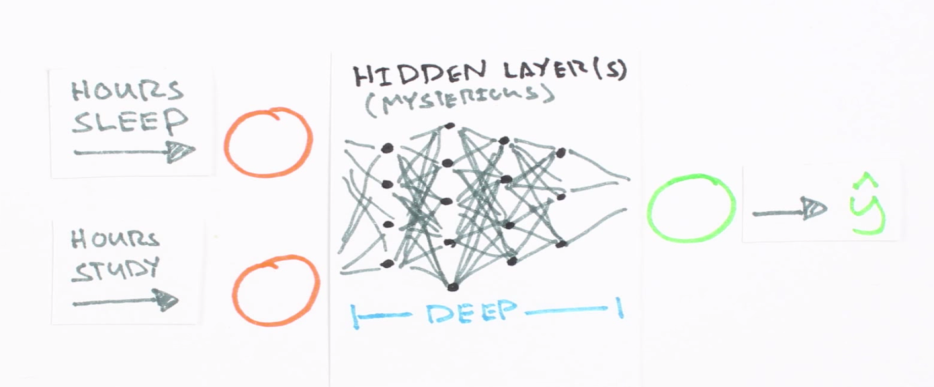
After this Lab, the students will be able to:

* 1. Understand basics Neural Networks
  2. Get an overview how to implement NN algorithms and related library

1. **Activity Detail**
   1. **Hour: 1-2  
      Getting Started:**
      1. Have a look on a the file " Neural Network Basics" and " First Neural Network in Python " in TSR
      2. Check \\TSR to see available datasets, codes, and tutorials
      3. http://stevenmiller888.github.io/mind-how-to-build-a-neural-network/

**Discussion:**

**What is a Neural Network?**

[Artificial neural networks](https://en.wikipedia.org/wiki/Artificial_neural_network) are statistical learning models, inspired by biological neural networks (central nervous systems, such as the brain), that are used in [machine learning](https://en.wikipedia.org/wiki/List_of_machine_learning_concepts). These networks are represented as systems of interconnected “neurons”, which send messages to each other. The connections within the network can be systematically adjusted based on inputs and outputs, making them ideal for supervised learning.

**Underlying Concept**

A neural network is a collection of “neurons” with “synapses” connecting them. The collection is organized into three main parts: the input layer, the hidden layer, and the output layer. Note that you can have n hidden layers, with the term “deep” learning implying multiple hidden layers.

Hidden layers are necessary when the neural network has to make sense of something really complicated, contextual, or non-obvious, like image recognition. The term “deep” learning came from having many hidden layers. These layers are known as “hidden”, since they are not visible as a network output.

The circles represent neurons and lines represent synapses. Synapses take the input and multiply it by a “weight” (the “strength” of the input in determining the output). Neurons add the outputs from all synapses and apply an activation function.

Training a neural network basically means calibrating all of the “weights” by repeating two key steps, forward propagation and back propagation.

Since neural networks are great for regression, the best input data are numbers (as opposed to discrete values, like colors or movie genres, whose data is better for statistical classification models). The output data will be a number within a range like 0 and 1 (this ultimately depends on the activation function—more on this below).

In forward propagation, we apply a set of weights to the input data and calculate an output. For the first forward propagation, the set of weights is selected randomly.

In back propagation, we measure the margin of error of the output and adjust the weights accordingly to decrease the error.

Neural networks repeat both forward and back propagation until the weights are calibrated to accurately predict an output.

**Discussion Continues:**

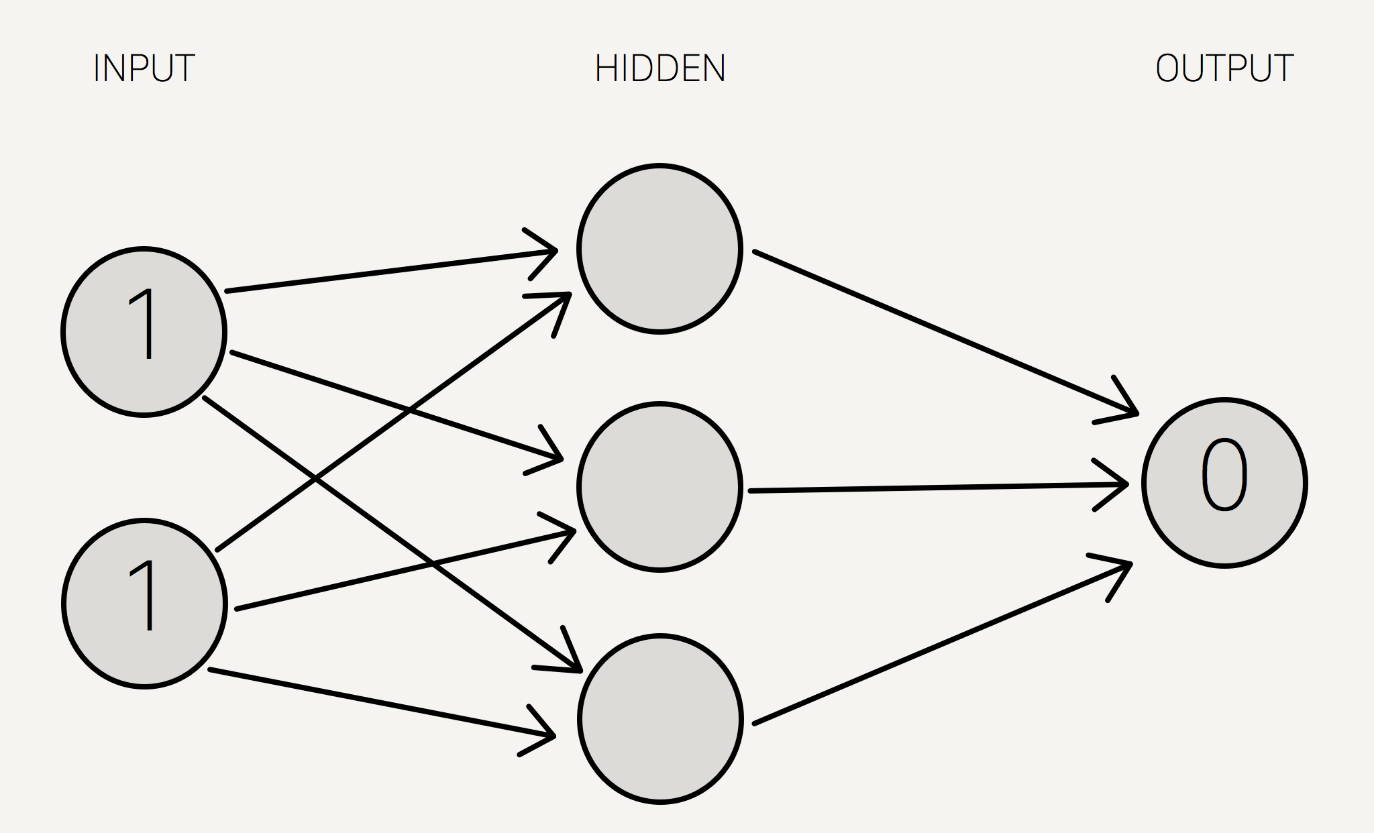
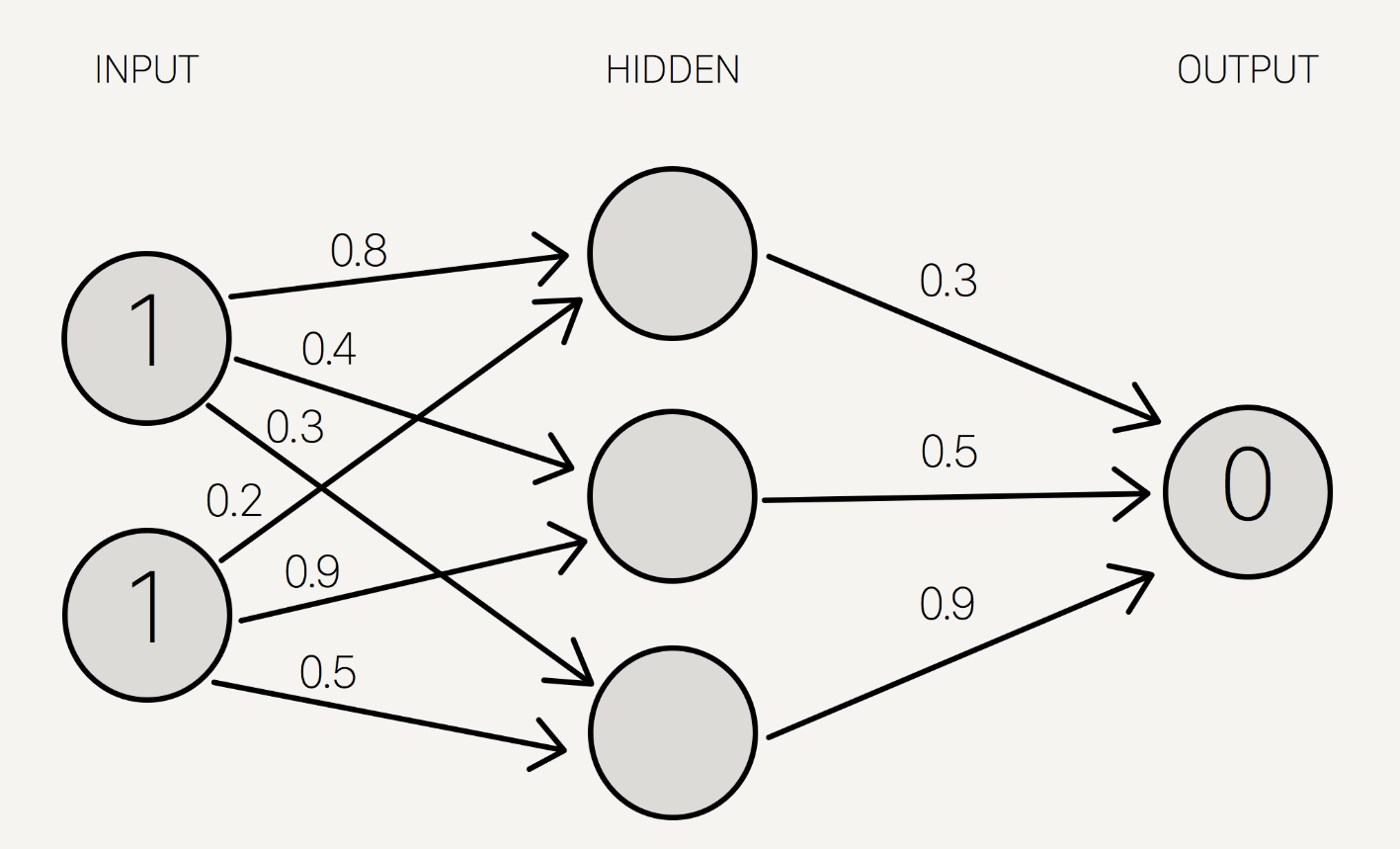
Next, we’ll walk through a simple example of training a neural network to function as an [“Exclusive or” (“XOR”) operation](https://en.wikipedia.org/wiki/Exclusive_or) to illustrate each step in the training process. *Note that all calculations will show figures truncated to the thousandths place.*

**Forward propagation**

|  |  |  |
| --- | --- | --- |
| **X1** | **X2** | **Output** |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

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.

Let’s use the last row from the above table, (1, 1) => 0, to demonstrate forward propagation:



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

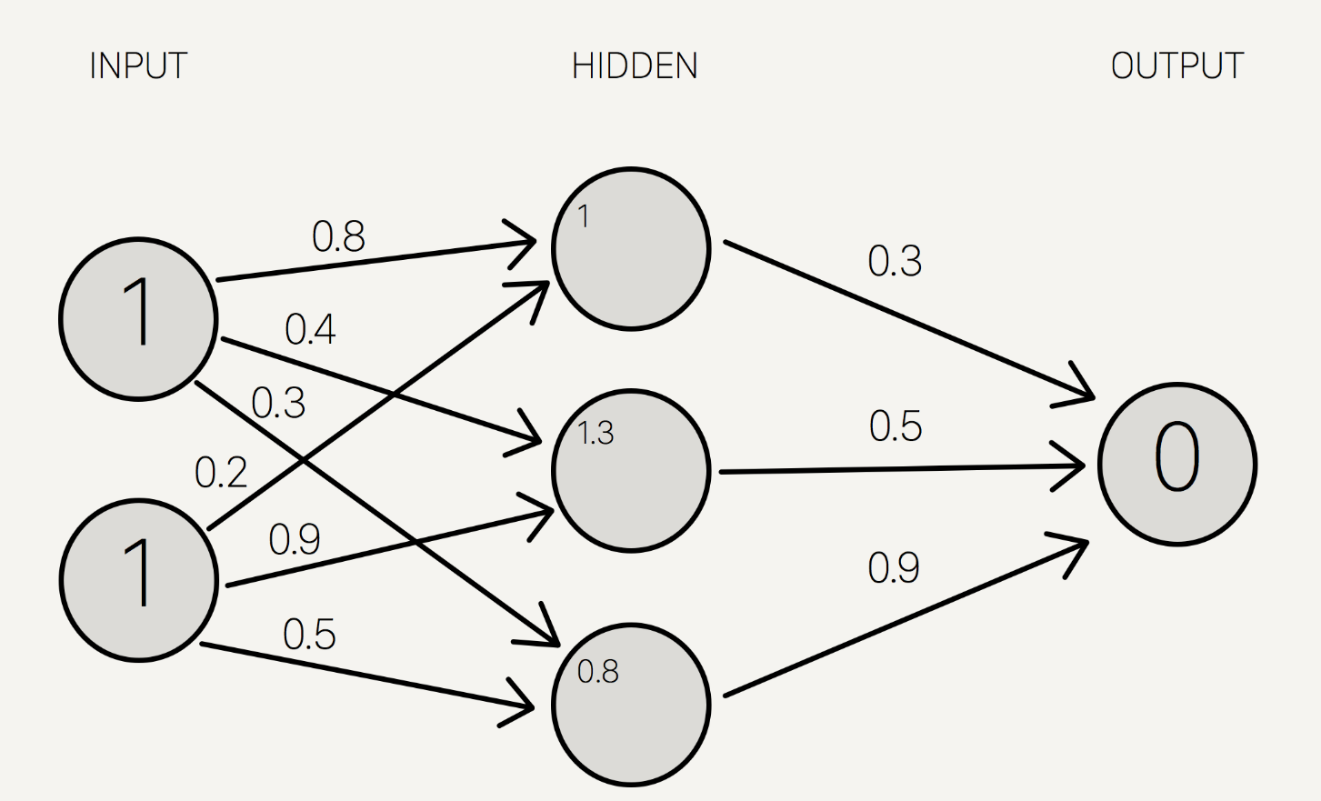
We now assign weights to all of the synapses. Note that these weights are selected randomly (based on Gaussian distribution) since it is the first time we’re forward propagating. The initial weights will be between 0 and 1, but note that the final weights don’t need to be.

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.

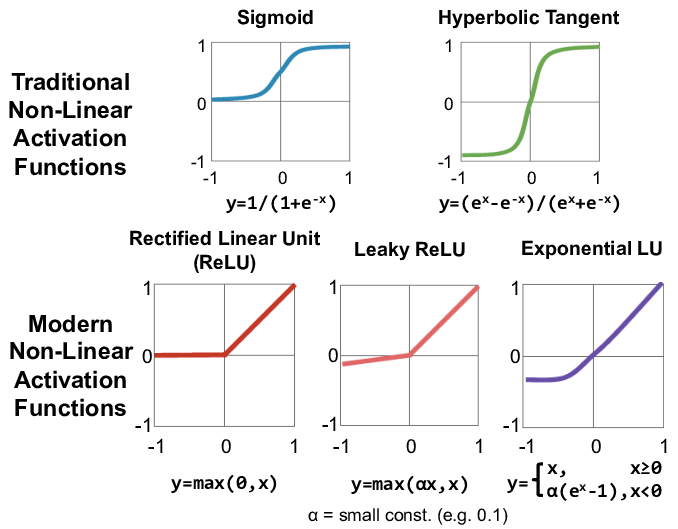
1 \* 0.8 + 1 \* 0.2 = 1

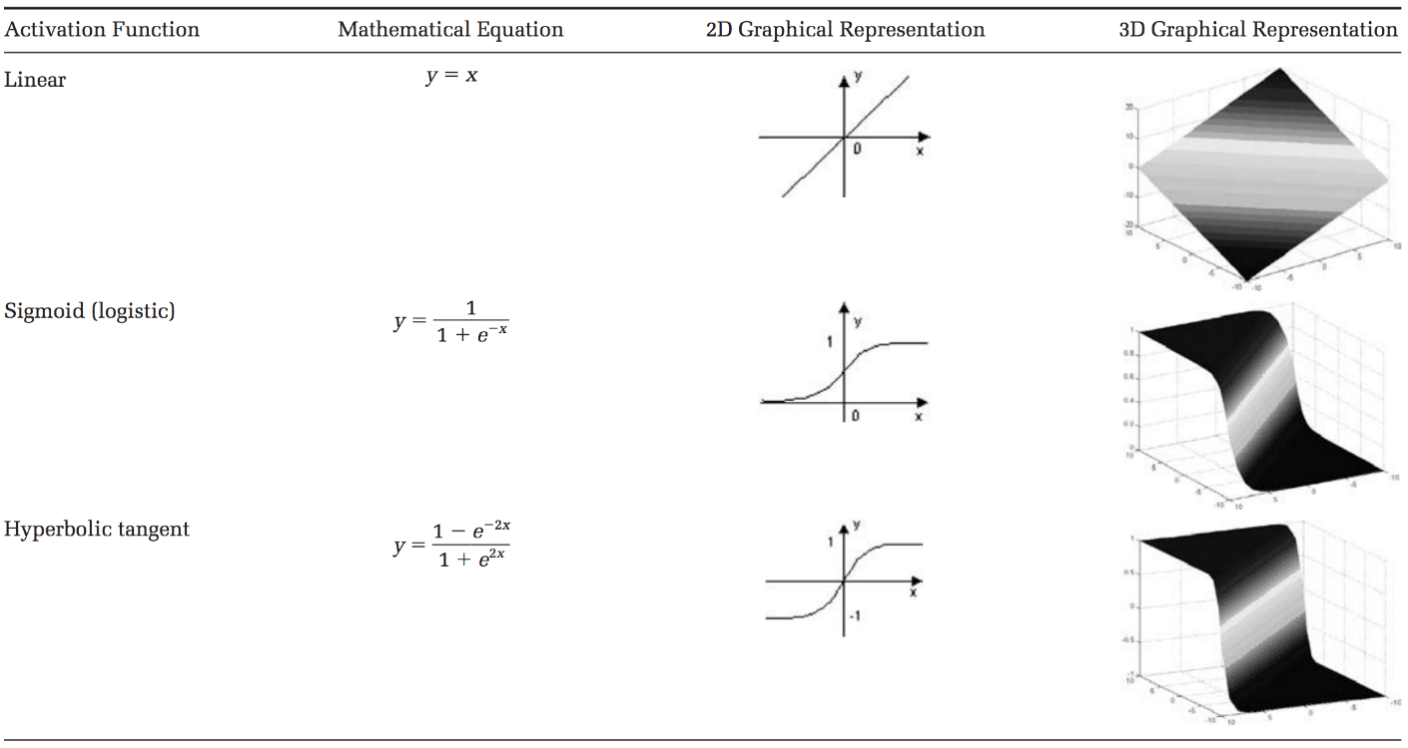
1 \* 0.4 + 1 \* 0.9 = 1.3

1 \* 0.3 + 1 \* 0.5 = 0.8

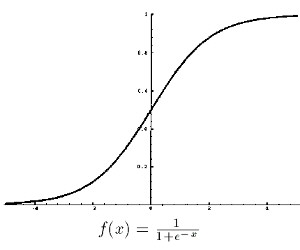
We put these sums smaller in the circle, because they’re not the final value:

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 and are necessary for neural networks to model complex non-linear patterns that simpler models might miss.



There are many types of activation functions—linear, sigmoid, hyperbolic tangent, even step-wise.

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:



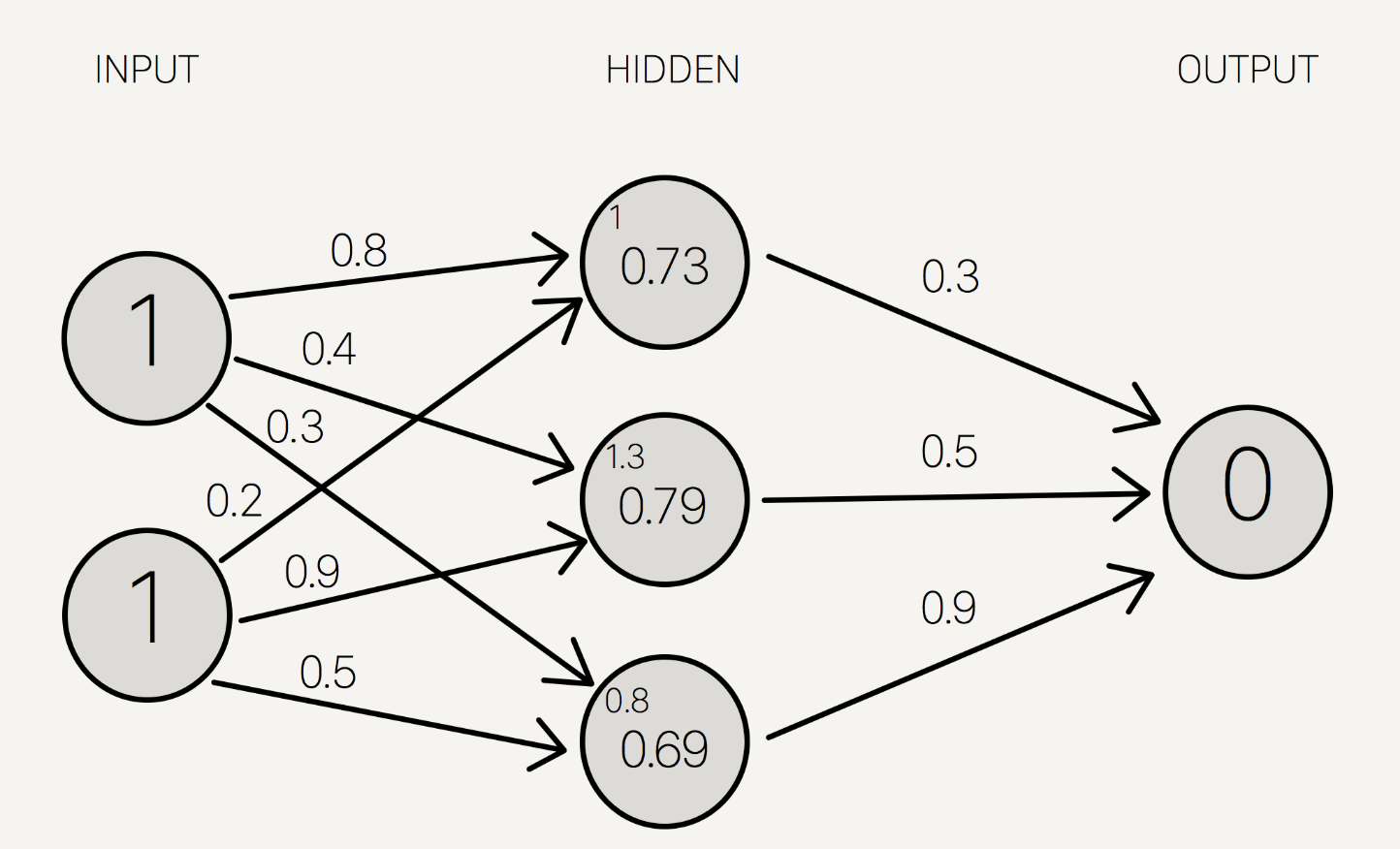
And applying S(x) to the three hidden layer *sums*, we get:

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*:



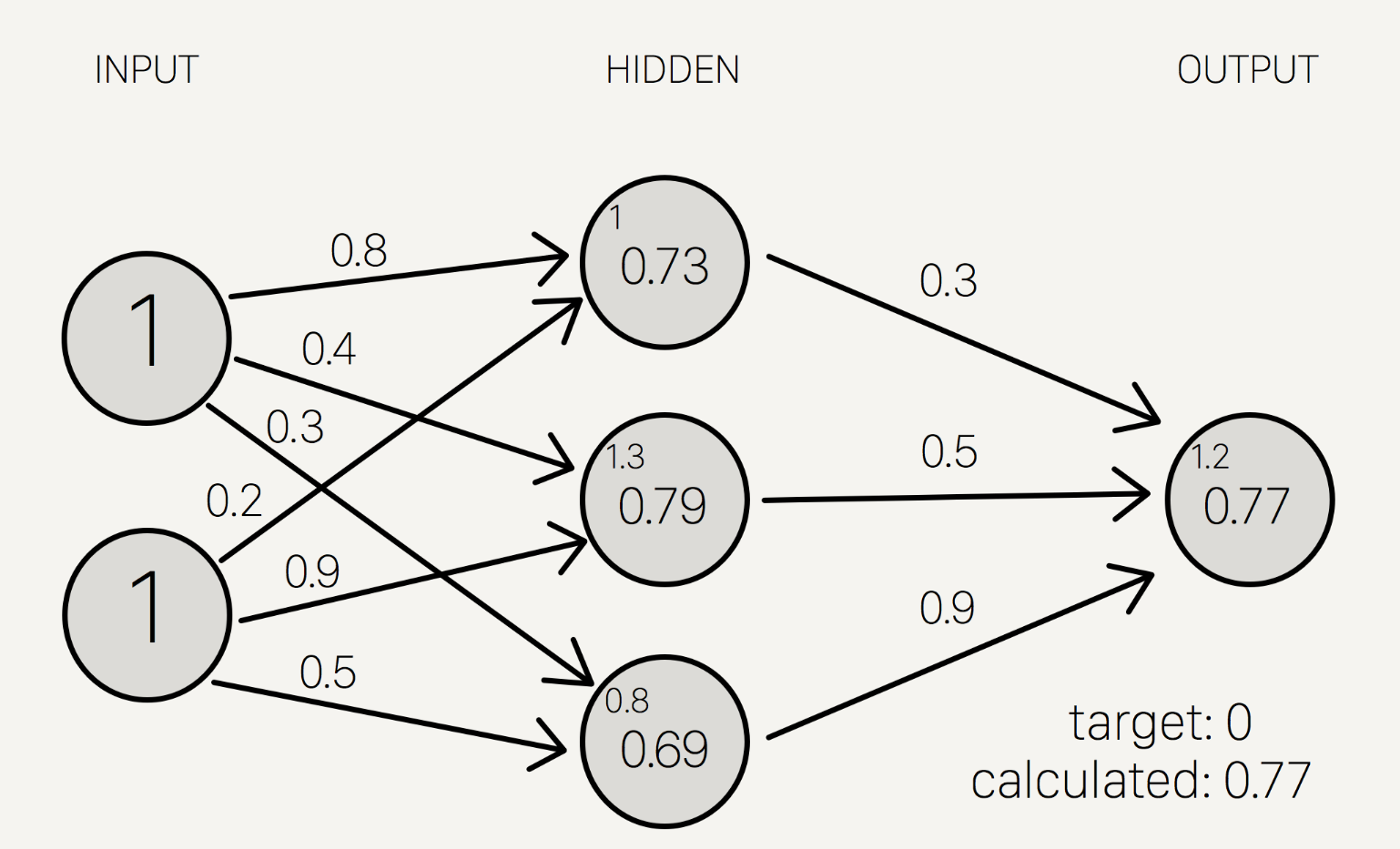
Then, we sum the product of the hidden layer results with the second set of weights (also determined at random the first time around) to determine the output sum.

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

This is our full diagram:



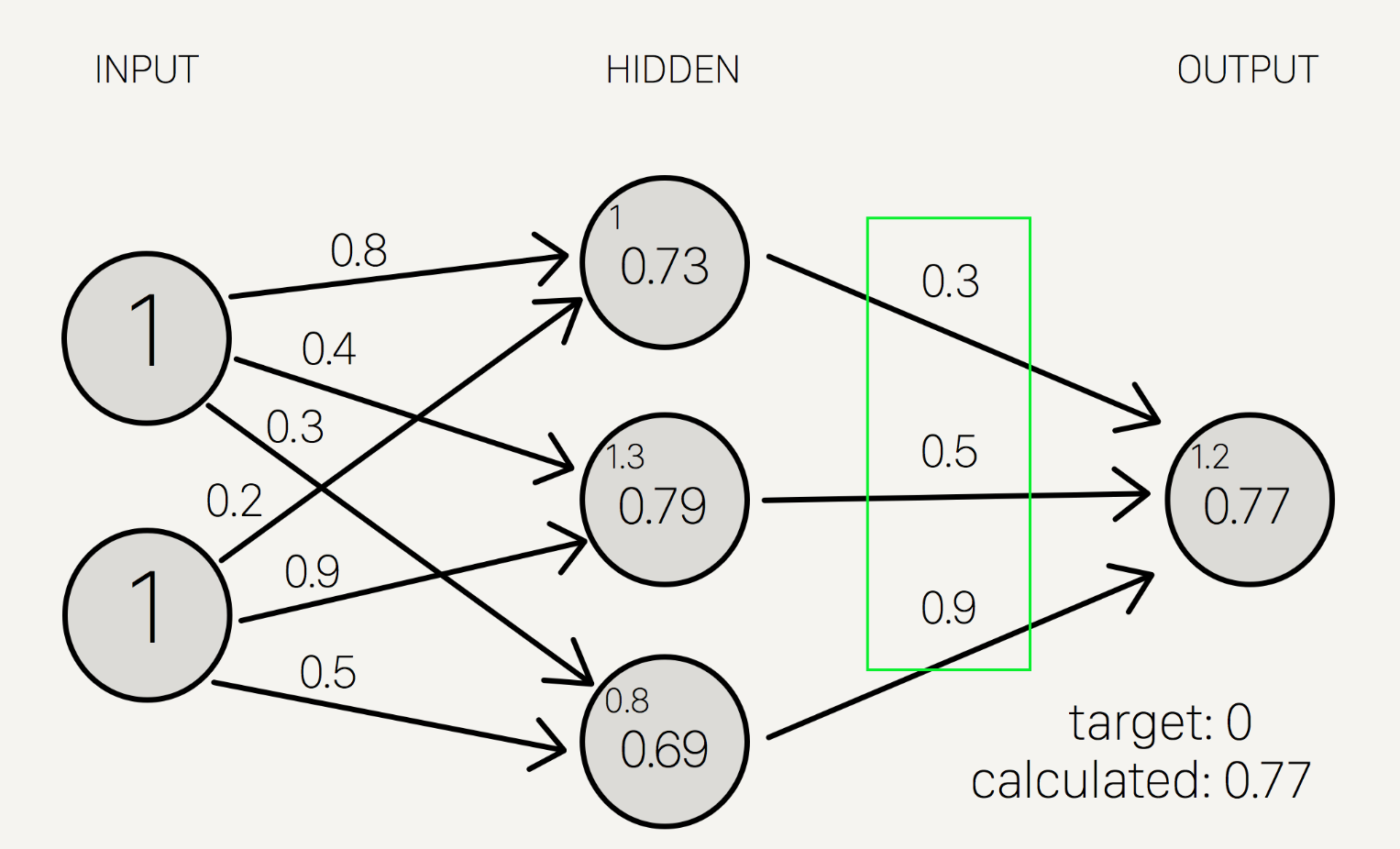
Because we used a random set of initial weights, the value of the output neuron is off the mark; in this case by +0.77 (since the target is 0). If we stopped here, this set of weights would be a great neural network for inaccurately representing the XOR operation.

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

**Back propagation**

To improve our model, we first have to quantify just how wrong our predictions are. Then, we adjust the weights accordingly so that the margin of errors are decreased.

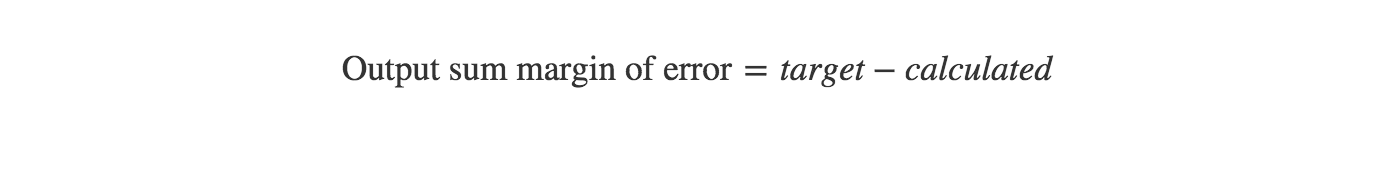
Similar to forward propagation, back propagation calculations occur at each “layer”. We begin by changing the weights between the hidden layer and the output layer.



Calculating the incremental change to these weights happens in two steps:

A) We find the margin of error of the output result (what we get after applying the activation function) to back out the necessary change in the output sum (we call this ***delta output sum***) and

B) We extract the change in weights by multiplying ***delta output sum*** by the hidden layer results.

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

And doing the math:

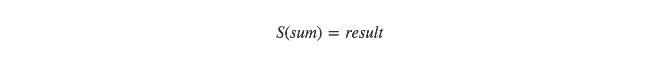
Target = 0

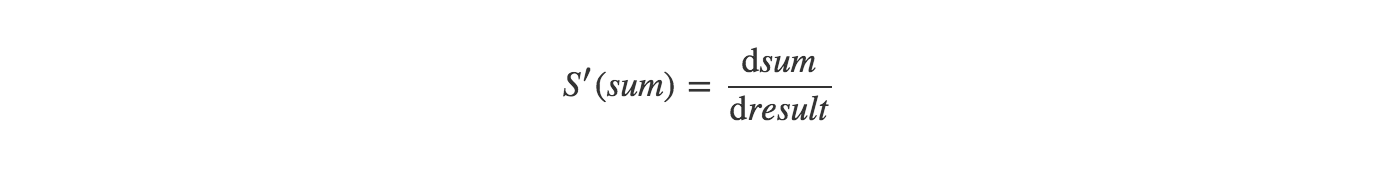
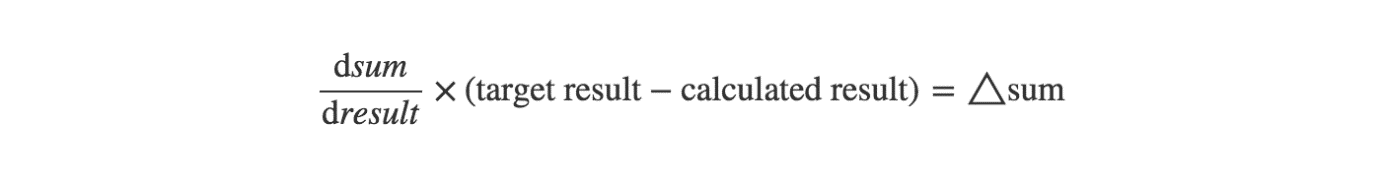
Calculated = 0.77

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.

To refresh your memory, the activation function, sigmoid, takes the sum and returns the result:

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:

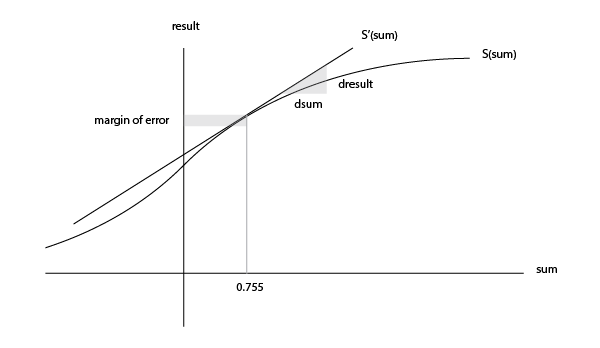
Since the output sum margin of error is the difference in the result, we can simply multiply that with the rate of change to give us the delta output sum:

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)

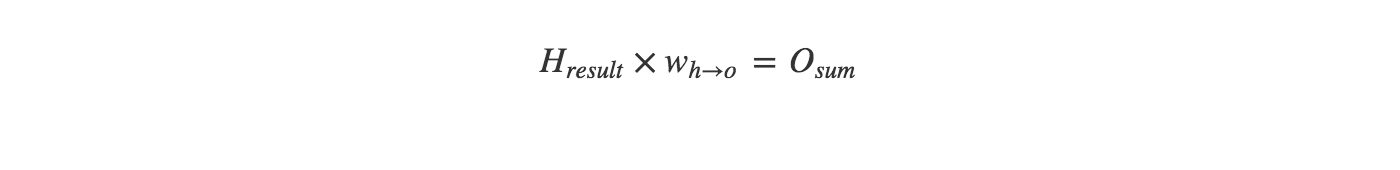
Delta output sum = S'(1.235) \* (-0.77) = S(1.235)\* (1- S(1.235)) \* (-.77)

Delta output sum = -0.13439890643886018

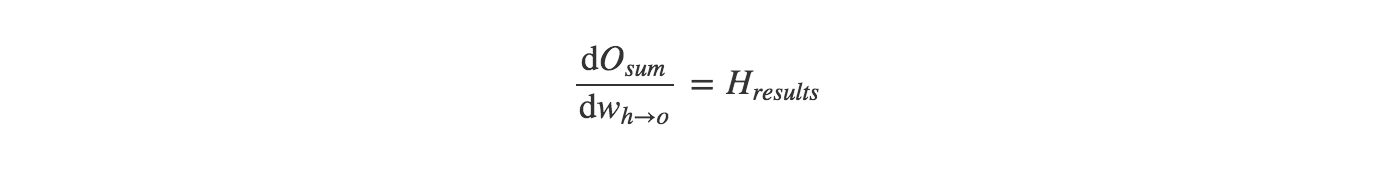
Here is a graph of the sigmoid function to give you an idea of how we are using the derivative to move the input towards the right direction. Note that this graph is not to scale.

Now that we have the proposed change in the output layer sum (-0.13), let’s use this in the derivative of the output sum function to determine the new change in weights.

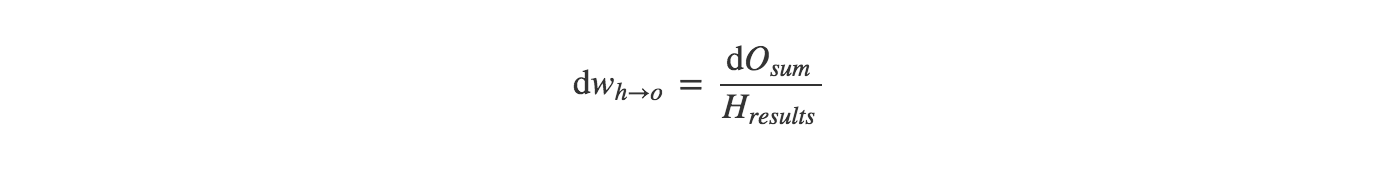
As a reminder, the mathematical definition of the output sum is the product of the hidden layer result and the weights between the hidden and output layer:



The derivative of the output sum is:



..which can also be represented as:



This relationship suggests that a greater change in output sum yields a greater change in the weights; input neurons with the biggest contribution (higher weight to output neuron) should experience more change in the connecting synapse.

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

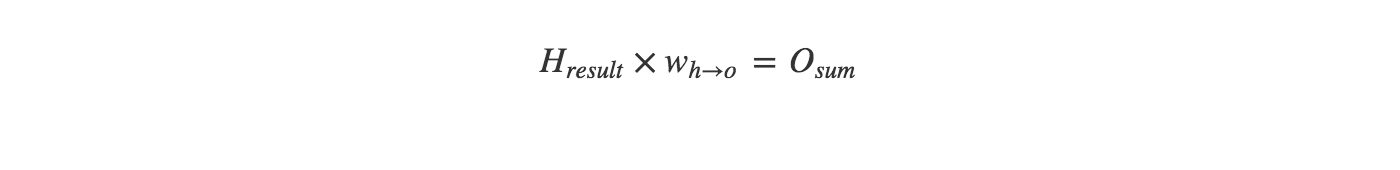
new w7 = 0.1162

new w8 = 0.329

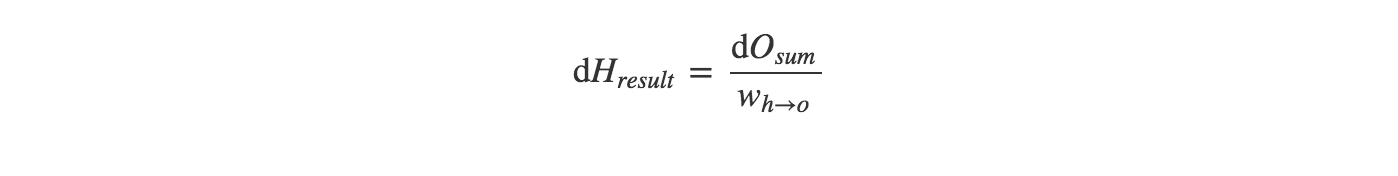
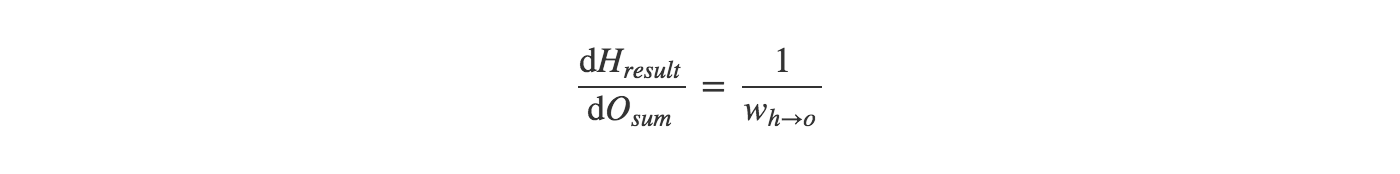
new w9 = 0.708

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.

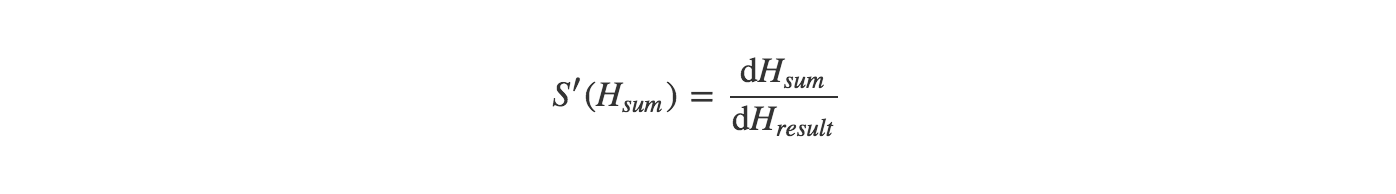
Remember that the relationship between the hidden result, the weights between the hidden and output layer, and the output sum is:



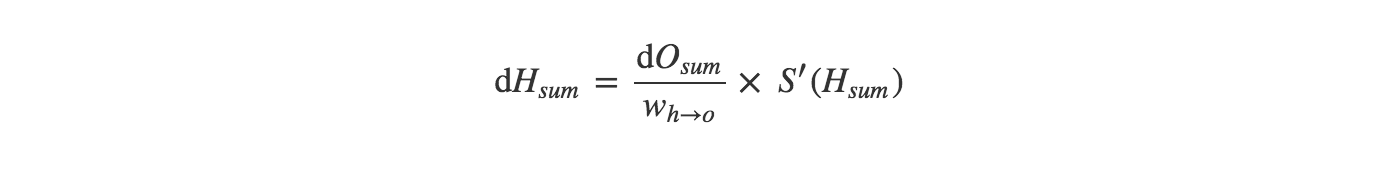
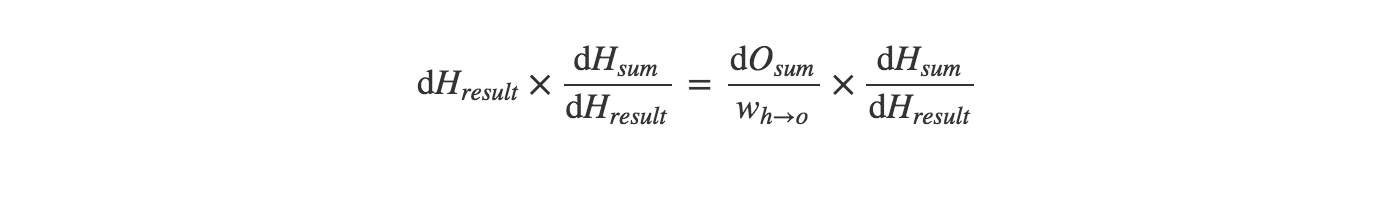
Instead of deriving for output sum, let’s derive for hidden result as a function of output sumto ultimately find out delta hidden sum:



Also, remember that the change in the hidden result can also be defined as:



Let’s multiply both sides by sigmoid prime of the hidden sum:



All of the pieces in the above equation can be calculated, so we can determine the delta hidden sum:

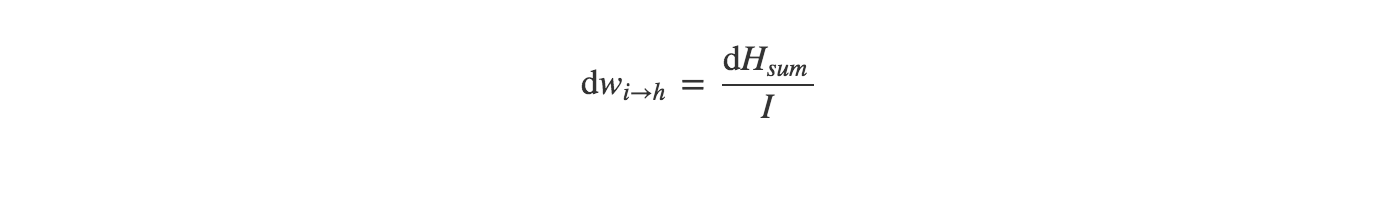
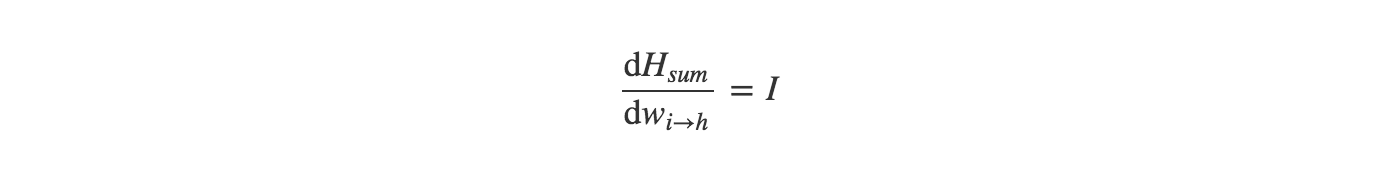
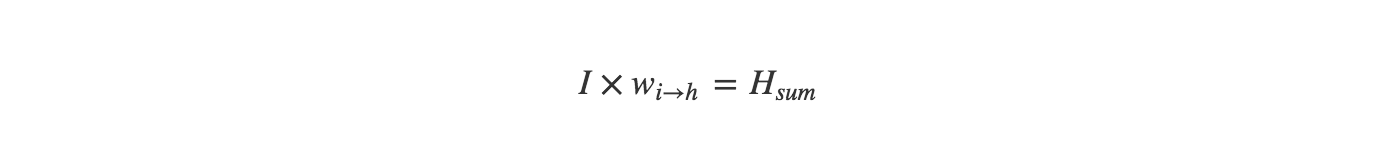
Delta hidden sum = delta output sum / hidden-to-outer weights \* S'(hidden sum)

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). The input data here is equivalent to the hidden results in the earlier back propagation process to determine the change in the hidden-to-output weights. Here is the derivation of that relationship, similar to the one before:



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 = 0.712

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

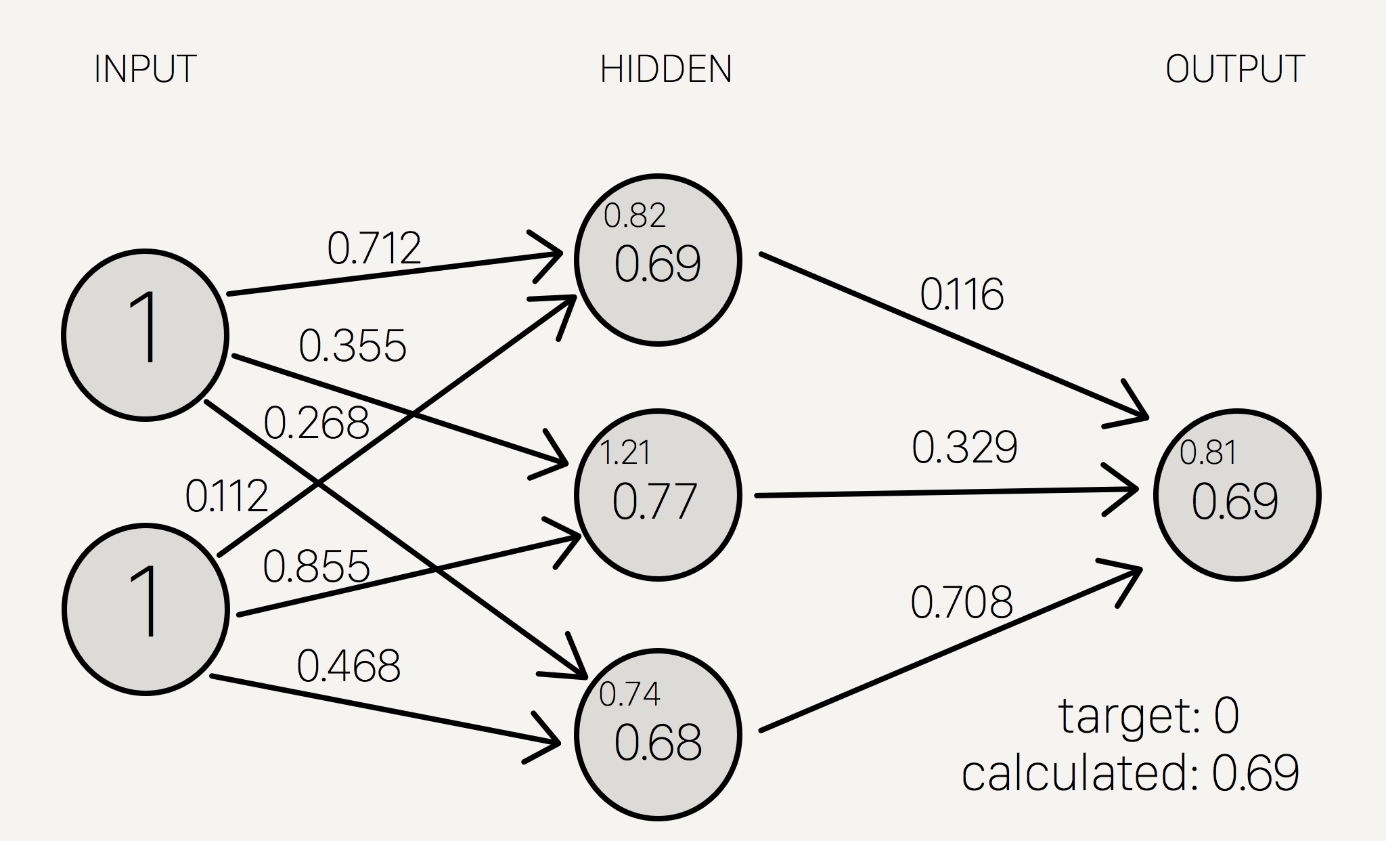
w7: 0.3 w7: 0.1162

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!!

Source: <https://stevenmiller888.github.io/mind-how-to-build-a-neural-network/>

Resource:

* <https://en.wikipedia.org/wiki/Backpropagation>
* <https://www.codingame.com/playgrounds/9487/deep-learning-from-scratch---theory-and-implementation/gradient-descent-and-backpropagation>

Coding:

Play with the codes of "XOR.ipnb" and "neuralNetworksFromScrach.py"

**Assigned Task**

**Consider Iris Data Set Classification Problem:**

we will use Iris Data Set Classification Problem for this demonstration. Iris Data Set is famous dataset in the world of pattern recognition and it is considered to be “Hello World” example for machine learning classification problems. It was first introduced by [Ronald Fisher](https://en.wikipedia.org/wiki/Ronald_Fisher), British statistician and botanist, back in 1936. In his paper *The use of multiple measurements in taxonomic problems,*he used data collected for three different classes of Iris plant: *Iris setosa*, *Iris virginica,*and *Iris versicolor*.

This dataset contains 50 instances for each class. What is interesting about it is that first class is linearly separable from the other two, but the latter two are not linearly separable from each other. Each instance has five attributes:

* Sepal length in cm
* Sepal width in cm
* Petal length in cm
* Petal width in cm
* Class (*Iris setosa*, *Iris virginica, Iris versicolor*)

In next chapter we will build Neural Network using Keras, that will be able to predict the class of the Iris flower based on the provided attributes.

**Code**

Keras programs have similar to the workflow of TensorFlow programs. We are going to follow this procedure:

* Import the dataset
* Prepare data for processing
* Create the model
* Training
* Evaluate accuracy of the model
* Predict results using the model

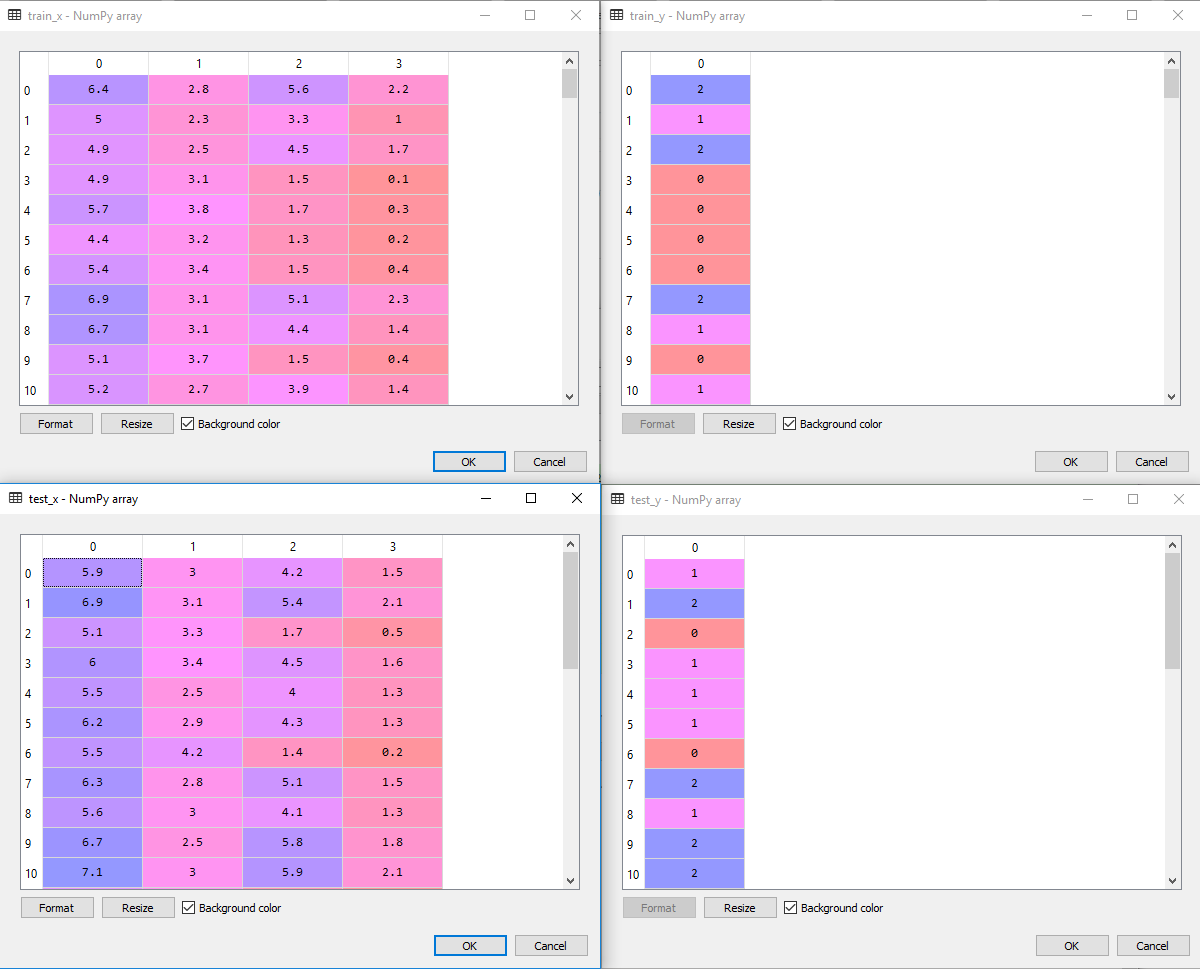
Training and evaluating processes are crucial for any Artificial Neural Network. These processes are usually done using two datasets, one for training and other for testing the accuracy of the trained network. In the real world, we will often get just one dataset and then we will split them into two separate datasets. For the training set, we usually use 80% of the data and another 20% we use to evaluate our model. Here is the list of the libraries that we need to import.

|  |  |
| --- | --- |
|  | # Importing libraries |
|  | fromkeras.modelsimport Sequential |
|  | fromkeras.layersimport Dense |
|  | fromkeras.utilsimportnp\_utils |
|  | importnumpy |
|  | import pandas aspd |

As you can see we are importing Keras dependencies, *NumPy*and P*andas. NumPy* is the fundamental package for scientific computing and *Pandas*provides easy to use data structures and data analysis tools.

After we imported libraries, we can proceed with importing the data and preparing it for the processing. We are going to use *Pandas* for importing data:

|  |  |
| --- | --- |
|  | # Import training dataset |
|  | training\_dataset=pd.read\_csv('iris\_training.csv', names=COLUMN\_NAMES, header=0) |
|  | train\_x=training\_dataset.iloc[:, 0:4].values |
|  | train\_y=training\_dataset.iloc[:, 4].values |
|  |  |
|  | # Import testing dataset |
|  | Write code like above |
|  |  |
|  |  |

Firstly, we used *read\_csv*function to import the dataset into local variables, and then we separated inputs *(train\_x, test\_x)* and expected outputs *(train\_y, test\_y)*creating four separate matrixes. Here is how they look like:

However, our data is not prepared for processing yet. If we take a look at our expected output values, we can notice that we have three values: 0, 1 and 2. Value 0 is used to represent Iris setosa, value 1 to represent Iris versicolor and value 2 to represent virginica. The good news about these values is that we didn’t get string values in the dataset. If you end up in that situation, you would need to use some kind of encoder so you can format data to something similar as we have in our current dataset. For this purpose, one can use ***[LabelEncoder](http://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.LabelEncoder.html" \t "_blank)***of sklearn library. Bad news about these values in the dataset is that they are not applicable to *Sequential* model. What we want to do is reshape the expected output from a vector that contains values for each class value to a matrix with a boolean for each class value. This is called [**one-hot encoding**](https://en.wikipedia.org/wiki/One-hot). In order to achieve this, we will use *np\_utils*from the Keras library:

|  |  |
| --- | --- |
|  | # Encoding training dataset |
|  | encoding\_train\_y=np\_utils.to\_categorical(train\_y) |
|  |  |
|  | # Encoding training dataset |
|  | encoding\_test\_y=np\_utils.to\_categorical(test\_y) |

If you still have doubt what one-hot encoding is doing, observe image below. There are displayed *train\_y* variable and *encoding\_train\_y* variable. Notice that first value in *train\_y* is 2 and see the corresponding value for that row in *encoding\_train\_y.*

Once we imported and prepared the data we can create our model. We already know we need to do this by using *Sequence*and *Dense* class. So, let’s do it following tutorial given. Remember input\_dim this variable value represents the column number or features used in dataset

We need to create:

* one input layer with four nodes, because we are having four attributes in our input values
* two hidden layers with ten neurons each
* one output layer with three neurons, because we are having three output classes

In hidden layers, neurons use [**Rectifier activation function**](http://rubikscode.net/2017/11/20/common-neural-network-activation-functions/), while in output layer neurons use Softmax activation function (ensuring that output values are in the range of 0 and 1). After that, we compile our model, where we define our [**cost function**](http://rubikscode.net/2018/01/15/how-artificial-neural-networks-learn/) and optimizer. In this instance, we will use Adam [**gradient descent optimization algorithm**](http://rubikscode.net/2018/01/15/how-artificial-neural-networks-learn/) with a logarithmic cost function (called *categorical\_crossentropy*in Keras).

|  |
| --- |
|  |
|  |
|  | # Compiling model |
|  | model.compile(loss='categorical\_crossentropy', optimizer='adam', metrics=['accuracy']) |

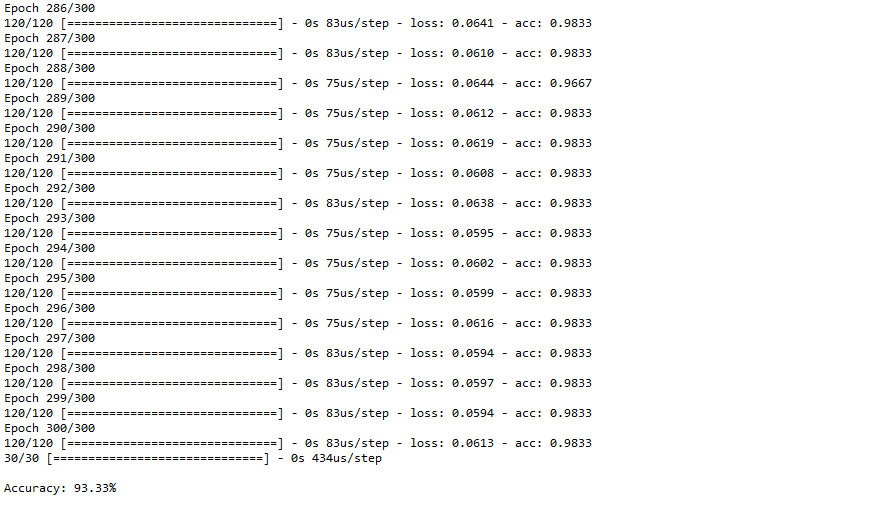
Finally, we can train our network:

|  |  |
| --- | --- |
|  | # Training a model |
|  | model.fit(use required parameters) |

And evaluate it:

|  |  |
| --- | --- |
|  | # Evaluate the model |
|  | scores =model.evaluate(use required parameters) |
|  | print("\nAccuracy: %.2f%%"% (scores[1]\*100)) |

If we run this code, we will get these results:



**Task 01:** Mark 20 **Time: 2** hour

(It is Not a Group Task, Try Individually)

Implement prediction model, you will provide the data for a particular iris flower Neural network has to tell which flower it belong, it will show flower name.

Hints: You have to use encoding decoding technique here, so that output can be decoded into flowers name. Go through the file "First Neural Network in Python" of TSR for helping tips.

**Evaluation Process (VIVA and Written answers):** You have to explain your program and show your work to the Lab Instructor. Instructor may ask you some questions to understand your depth of knowledge and expertise level.