SMARTBRIDGE
Let's Bridge the Gap

Introduction to Neural Networks

Neural Network

- This series of lectures will cover key theory aspects:
  - Neurons and Activation Functions
  - Cost Functions
  - Gradient Descent
  - Backpropagation

Neural Network

- Once we build a general high level understanding we will code out all these topics manually with Python, without the use of a deep learning library.
- Then we can move on to using TensorFlow!

Neural Network

- Understanding a high level overview of these key elements will make it much easier to understand what is happening when we begin to use TensorFlow!
- Tensorflow has direct connections to these concepts in its syntax!

Let's get started!

Introduction to the Perceptron

Perceptron

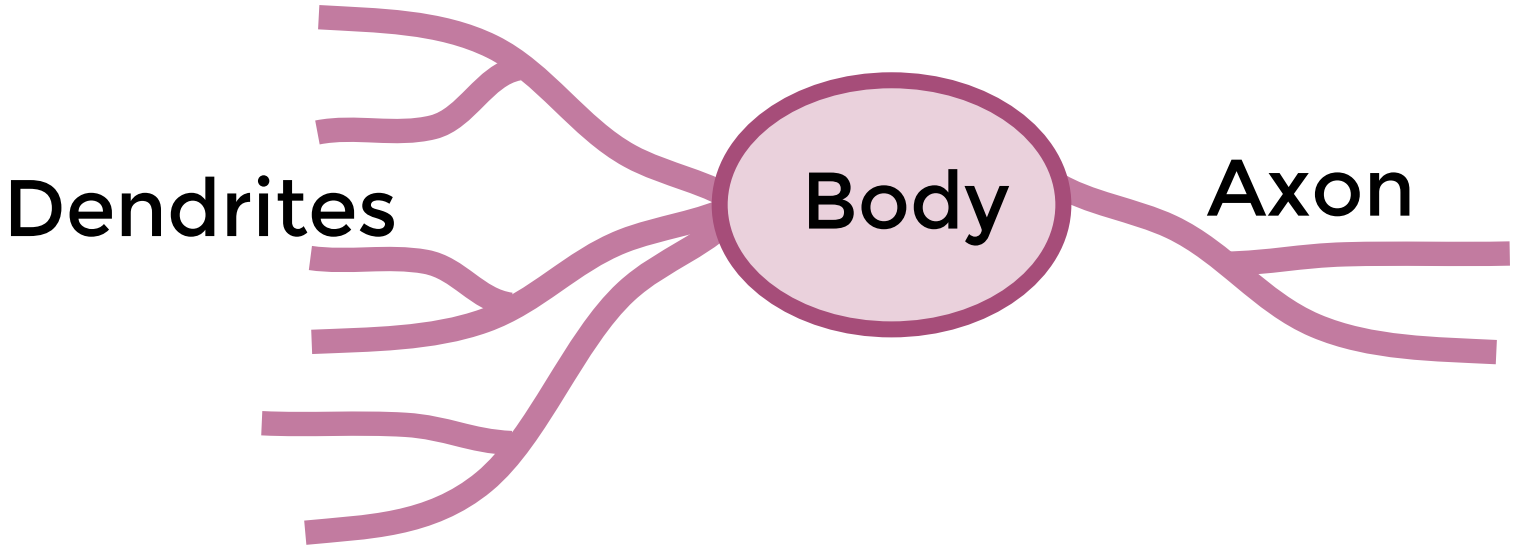
- Before we launch straight into neural networks, we need to understand the individual components first, such as a single “neuron”.

Perceptron

- Artificial Neural Networks (ANN) actually have a basis in biology!
- Let’s see how we can attempt to mimic biological neurons with an artificial neuron, known as a perceptron!

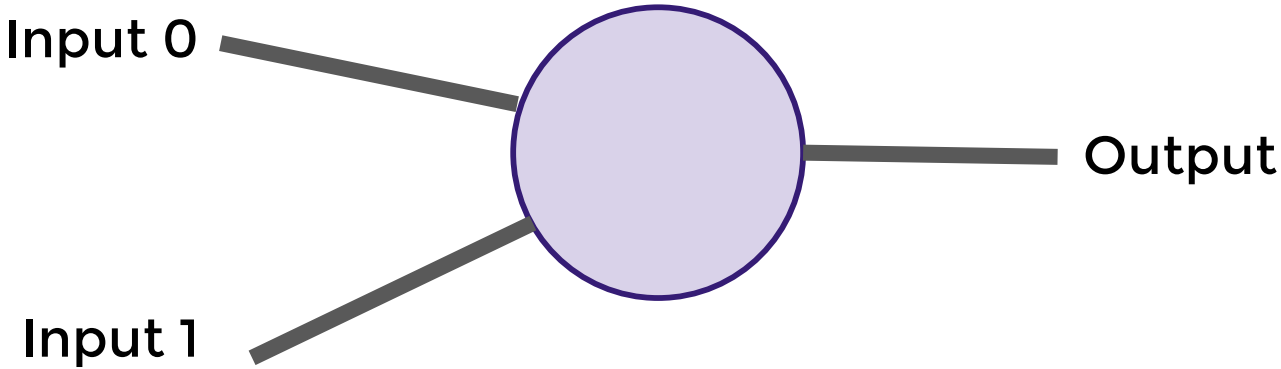
Perceptron

- The biological neuron:



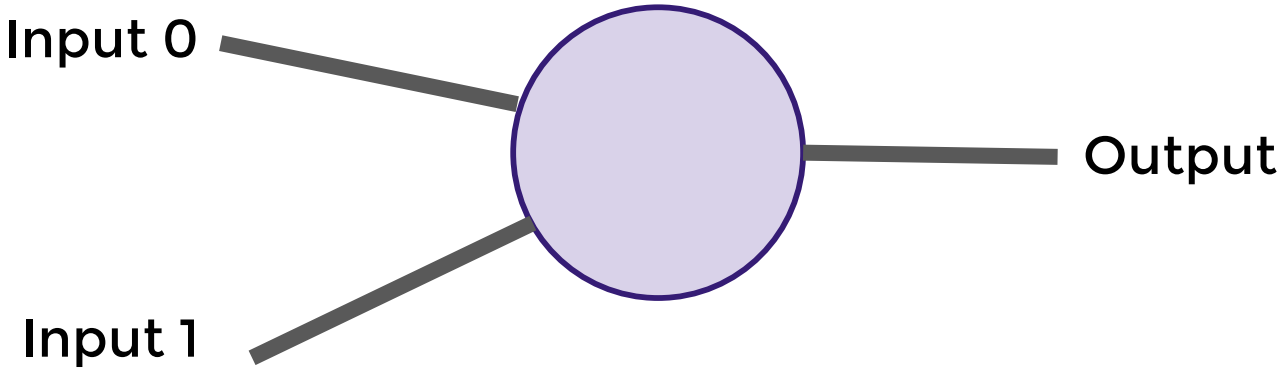
Perceptron

- The artificial neuron also has inputs and outputs!



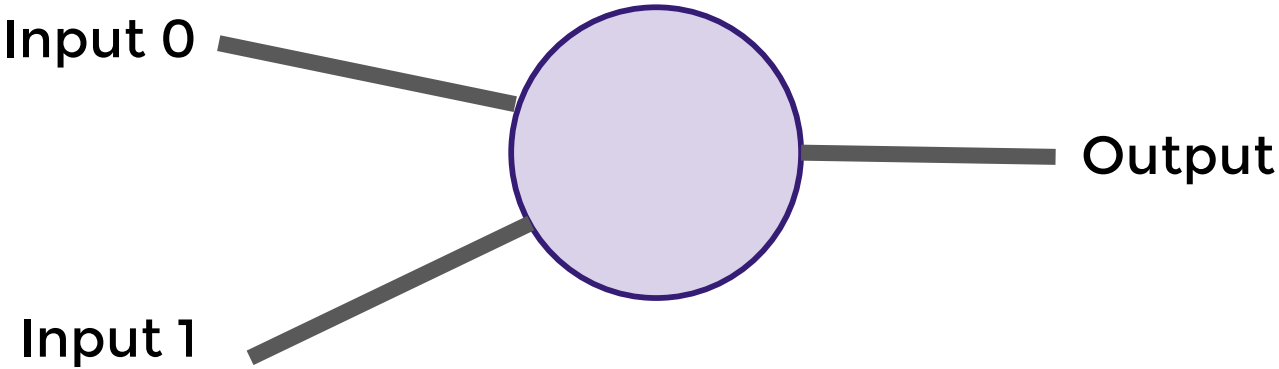
Perceptron

- This simple model is known as a perceptron.



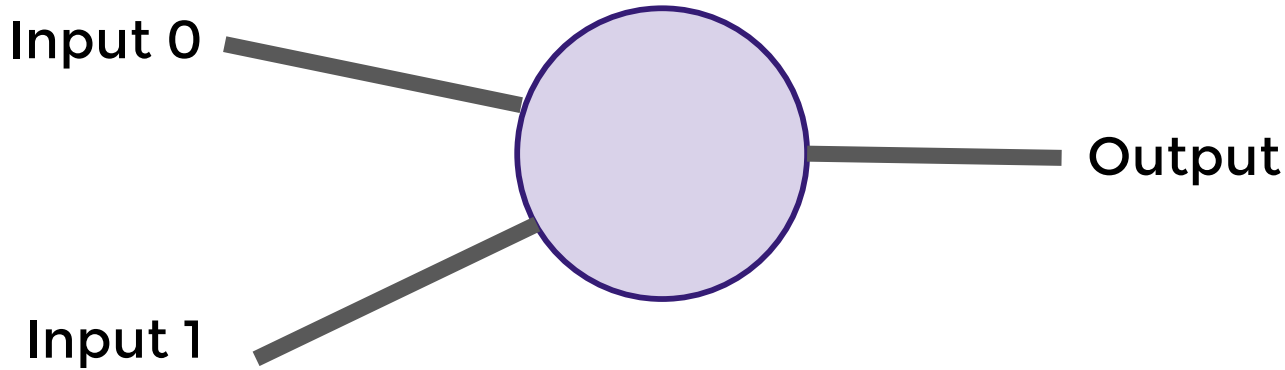
Perceptron

- Simple example of how it can work.



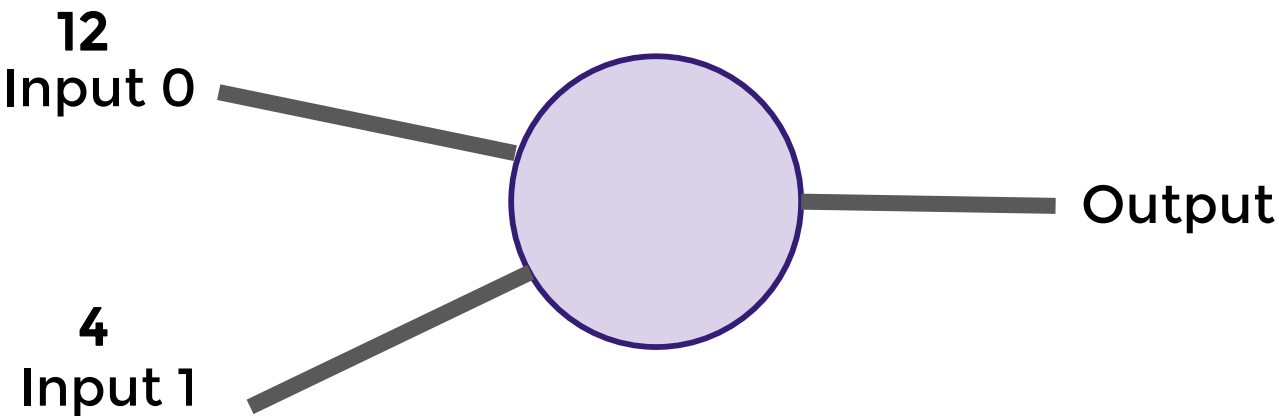
Perceptron

- We have two inputs and an output



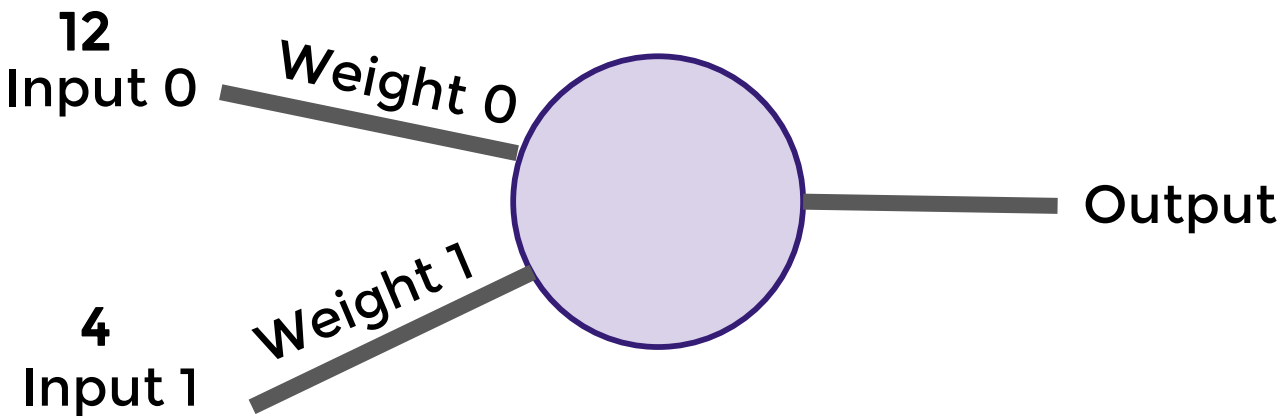
Perceptron

- Inputs will be values of features



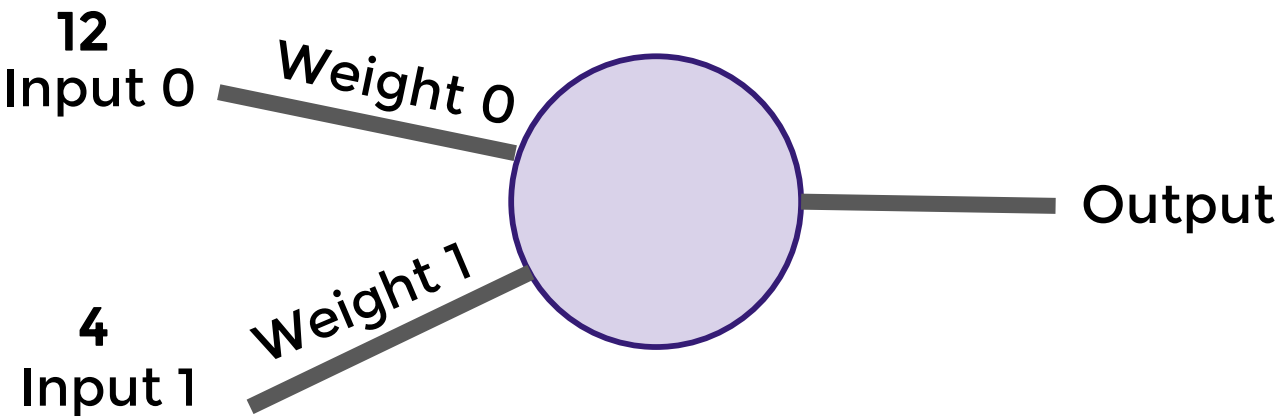
Perceptron

- Inputs are multiplied by a weight



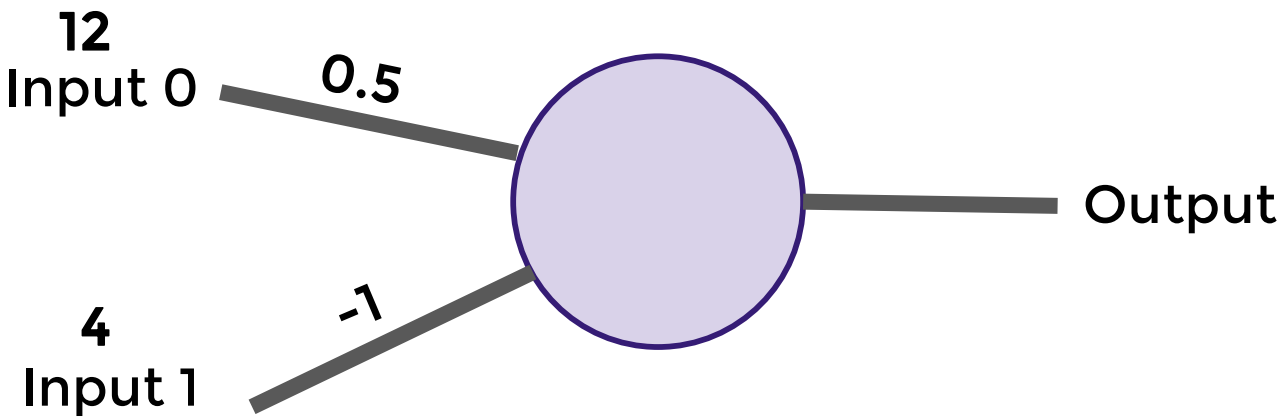
Perceptron

- Weights initially start off as random



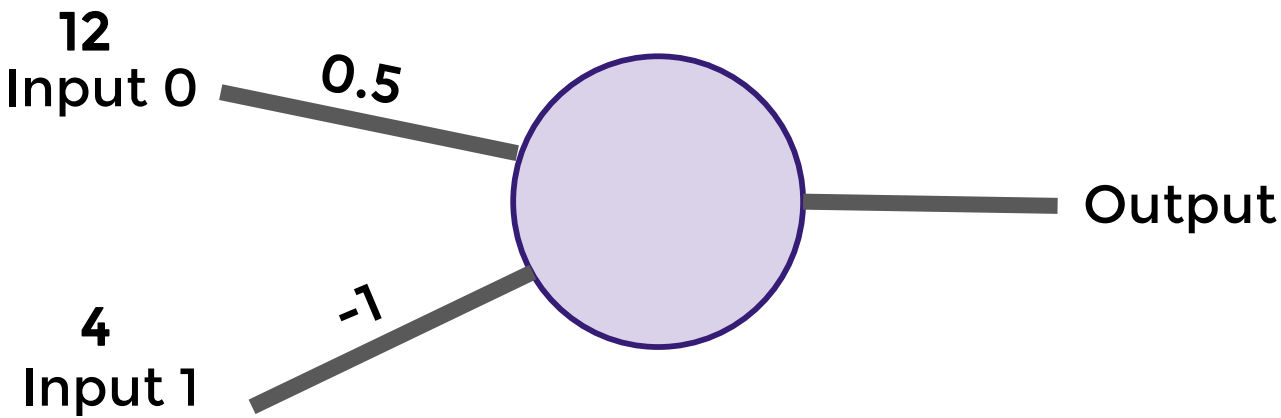
Perceptron

- Weights initially start off as random



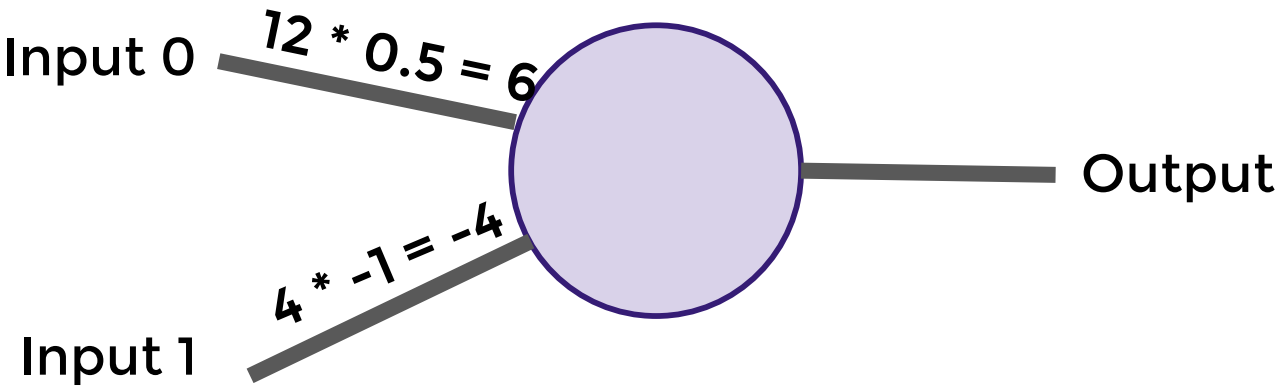
Perceptron

- Inputs are now multiplied by weights



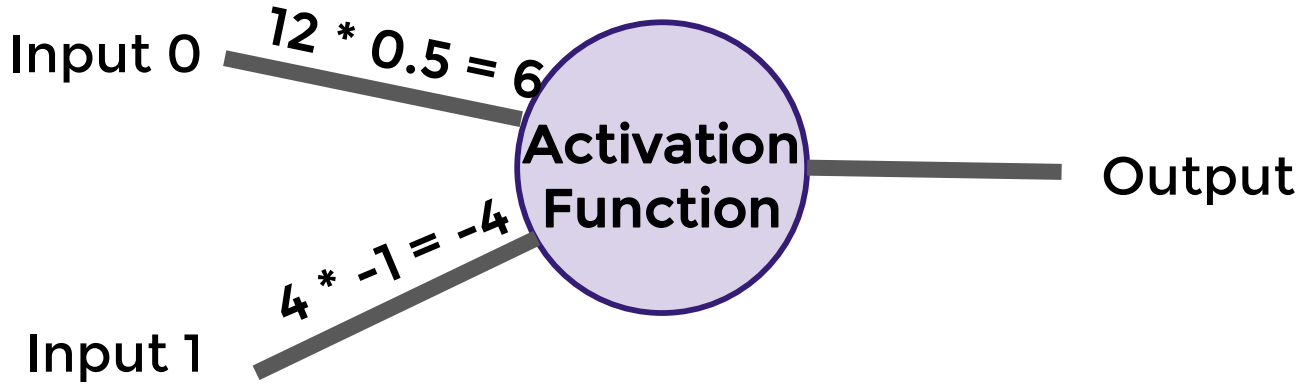
Perceptron

- Inputs are now multiplied by weights



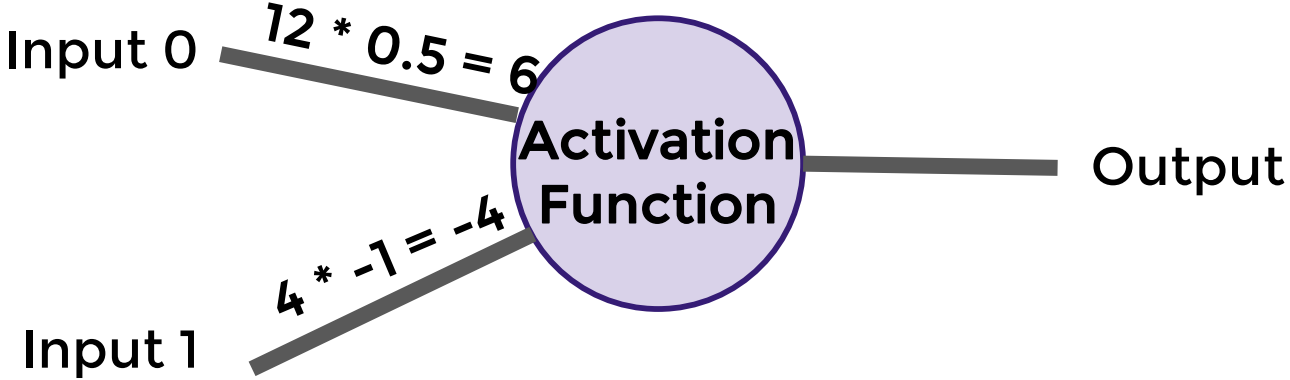
Perceptron

- Then these results are passed to an activation function.



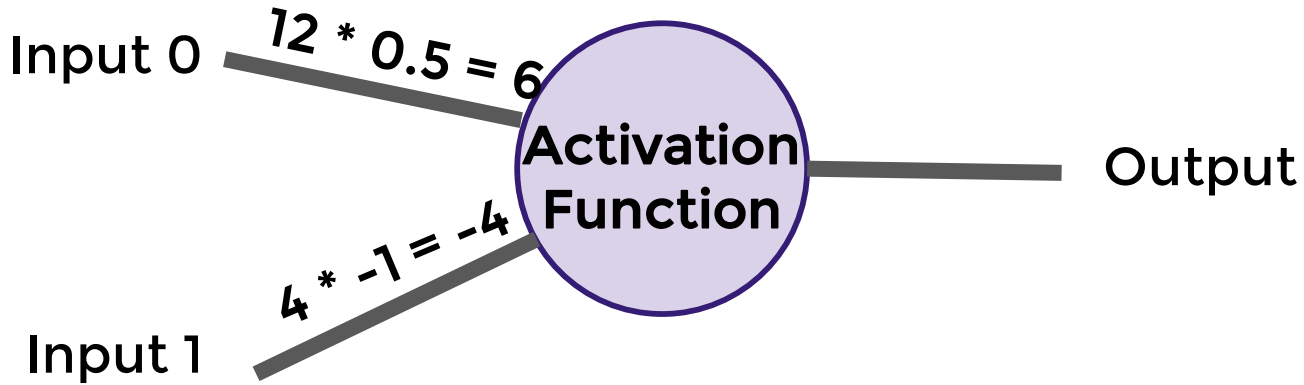
Perceptron

- Many activation functions to choose from, we'll cover this in more detail later!



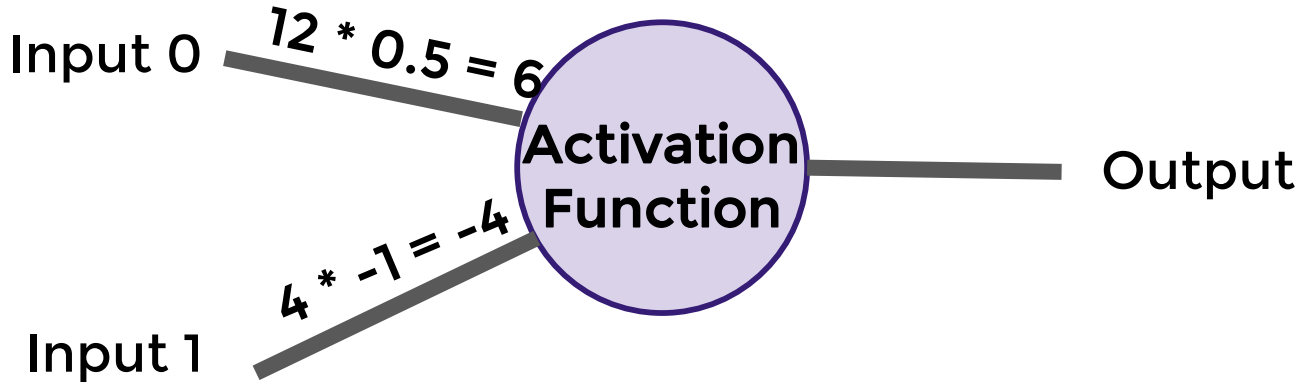
Perceptron

- For now our activation function will be very simple...



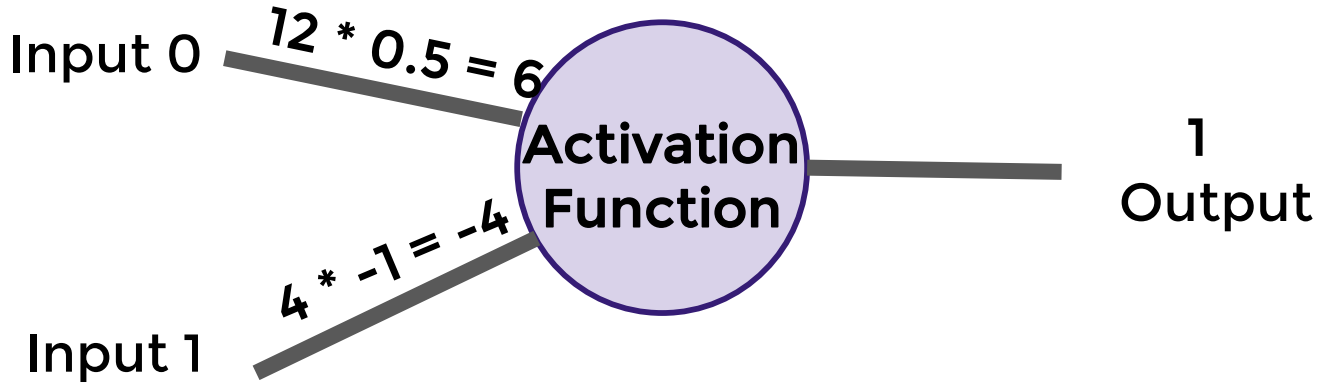
Perceptron

- If sum of inputs is positive return 1, if sum is negative output 0.



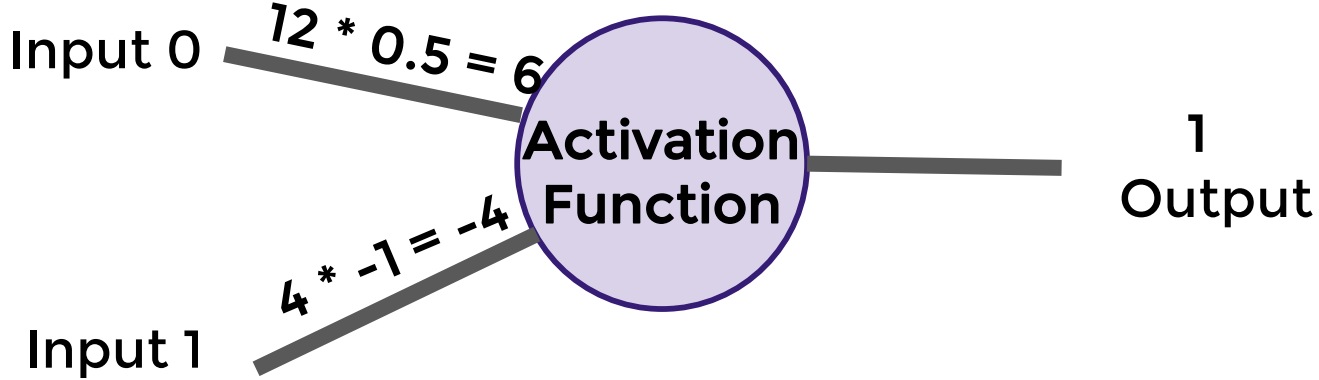
Perceptron

- In this case 6-4=2 so the activation function returns 1.



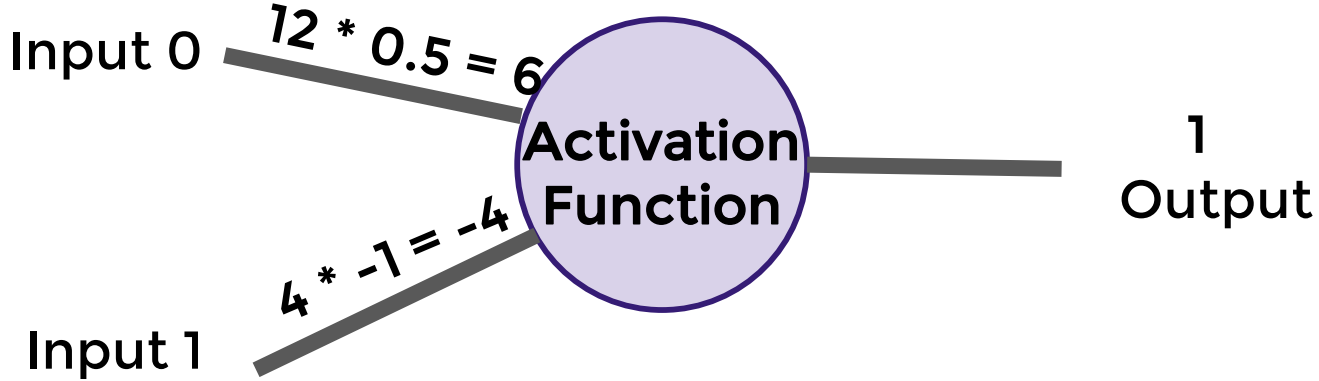
Perceptron

- There is a possible issue. What if the original inputs started off as zero?



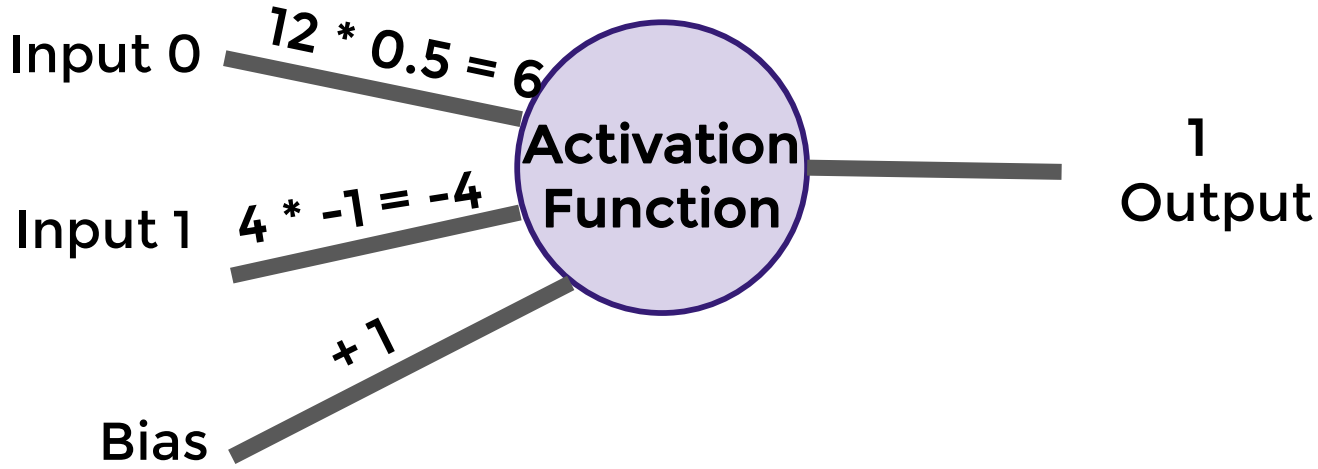
Perceptron

- Then any weight multiplied by the input would still result in zero!



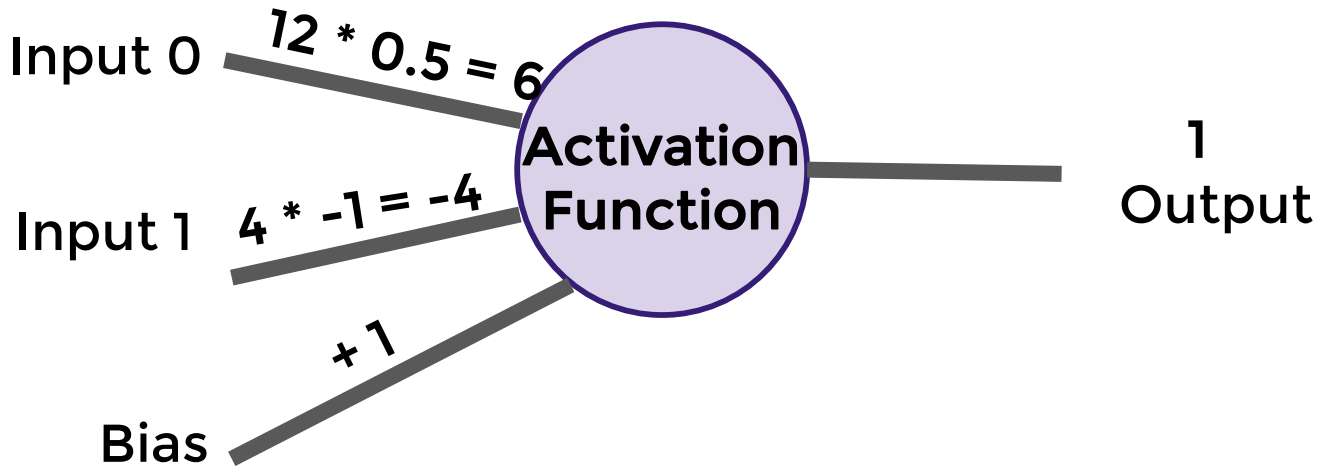
Perceptron

- We fix this by adding in a bias term, in this case we choose 1.



Perceptron

- So what does this look like mathematically?



Perceptron

- Let’s quickly think about how we can represent this perceptron model mathematically:

$$\sum_{i=0}^{n} w_i x_i + b$$

Perceptron

- Once we have many perceptrons in a network we'll see how we can easily extend this to a matrix form!

$$\sum_{i=0}^{n} w_i x_i + b$$

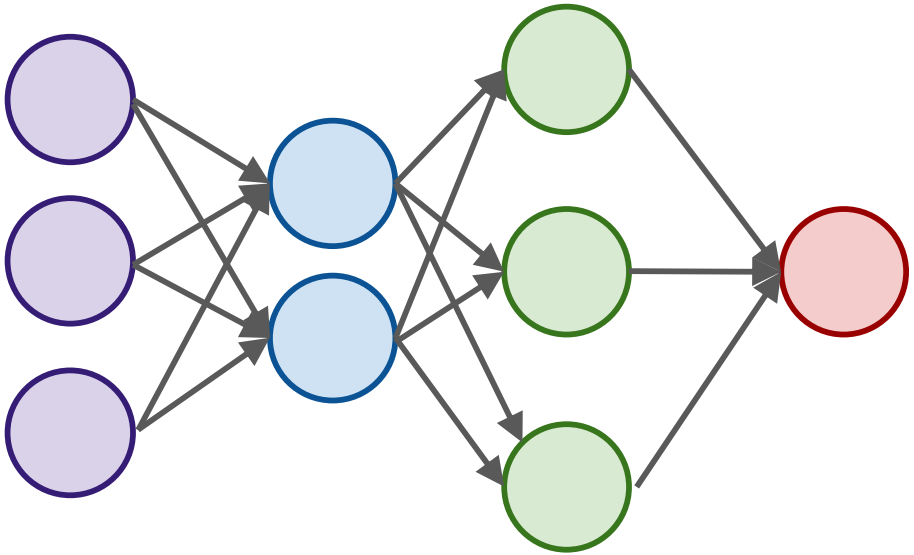
Perceptron

- Review
  - Biological Neuron
  - Perceptron Model
  - Mathematical Representation

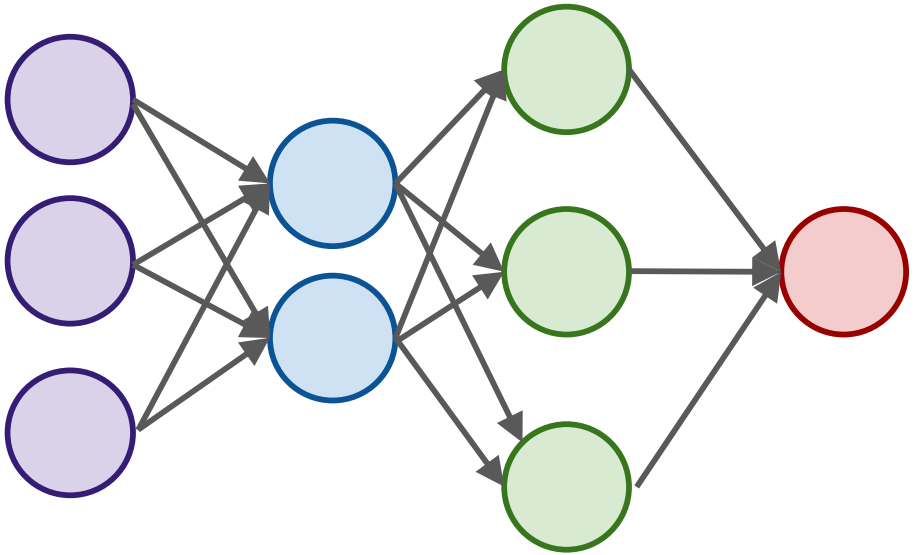
Introduction to Neural Networks

- We've seen how a single perceptron behaves, now let's expand this concept to the idea of a neural network!
- Let's see how to connect many perceptrons together and then how to represent this mathematically!

- Multiple Perceptrons Network



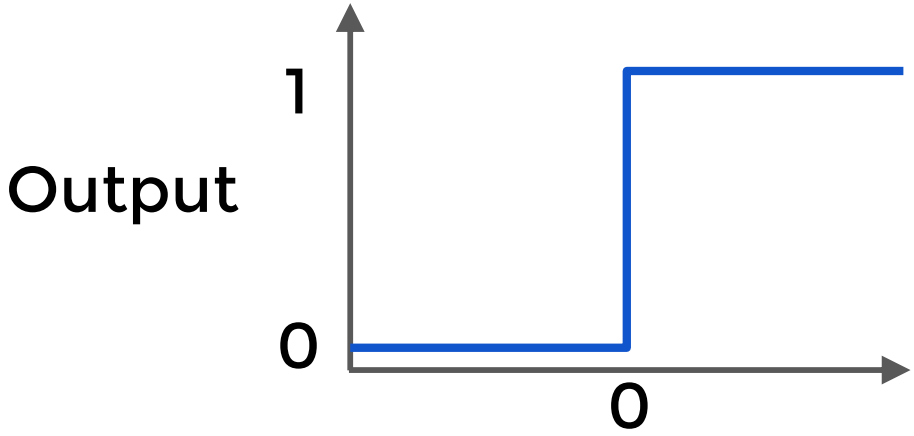
- Input Layer. 2 hidden layers. Output Layer



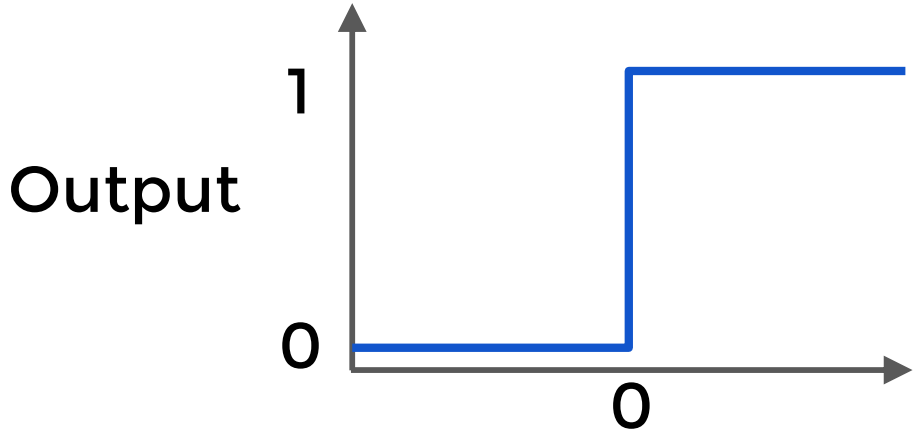
- Input Layers
    - Real values from the data
- Hidden Layers
    - Layers in between input and output
    - 3 or more layers is “deep network”
- Output Layer
    - Final estimate of the output

- As you go forwards through more layers, the level of abstraction increases.
- Let’s now discuss the activation function in a little more detail!

- Previously our activation function was just a simple function that output 0 or 1.

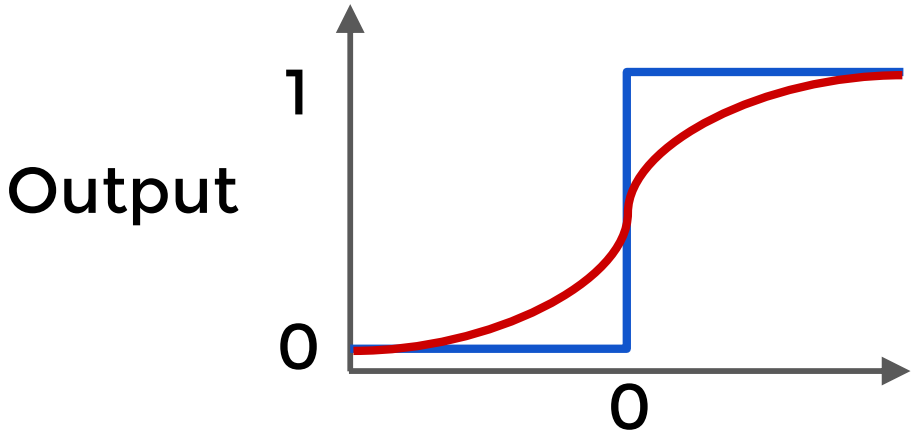


- This is a pretty dramatic function, since small changes aren't reflected.

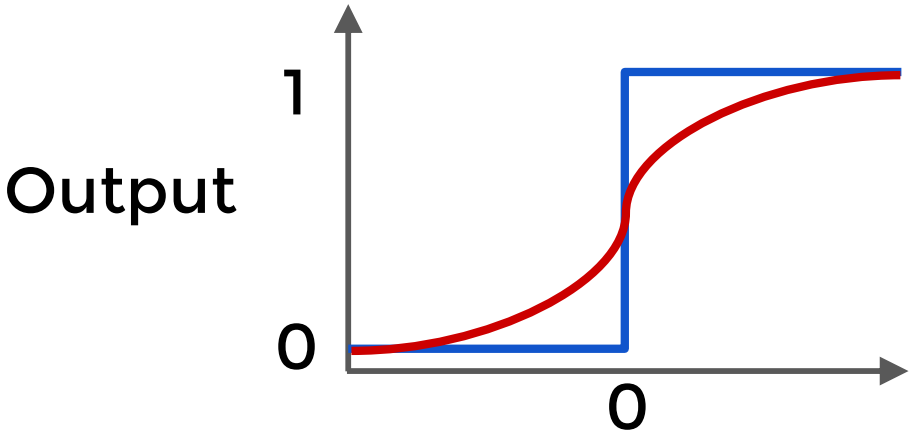


$$z = wx + b$$

- It would be nice if we could have a more dynamic function, for example the red line!



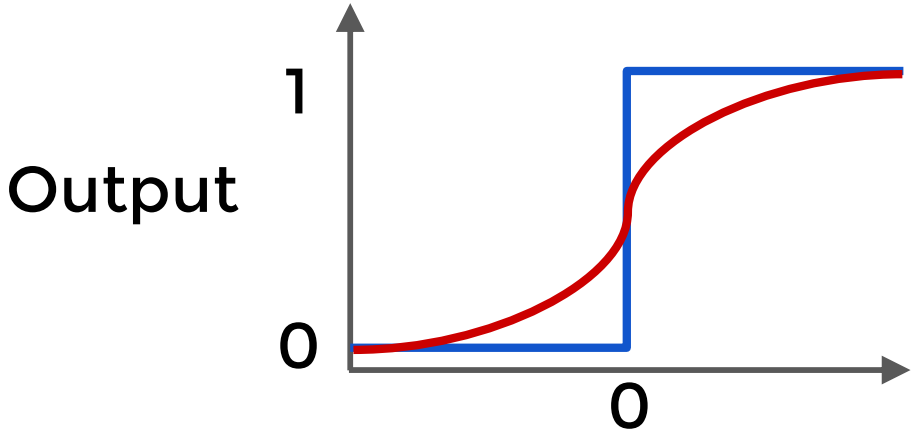
- Lucky for us, this is the sigmoid function!



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

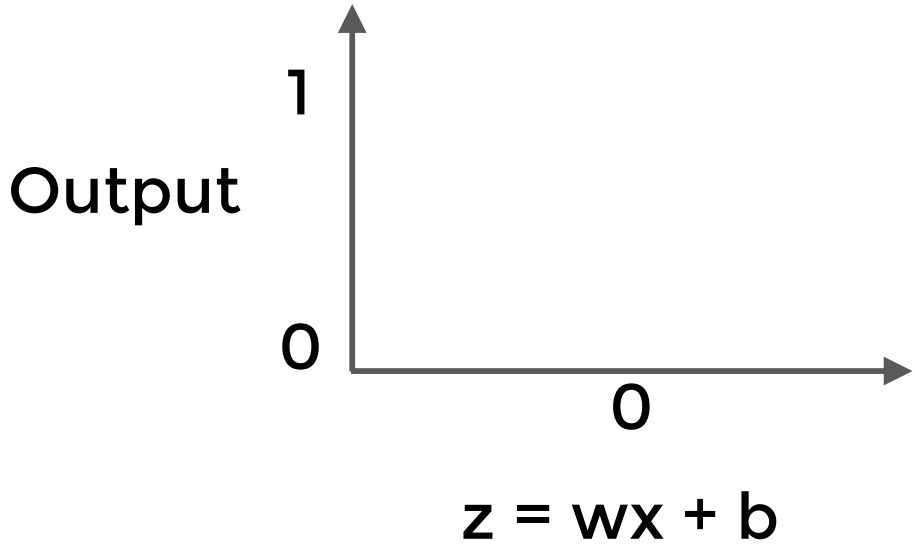
z = wx + b

- Changing the activation function used can be beneficial depending on the task!

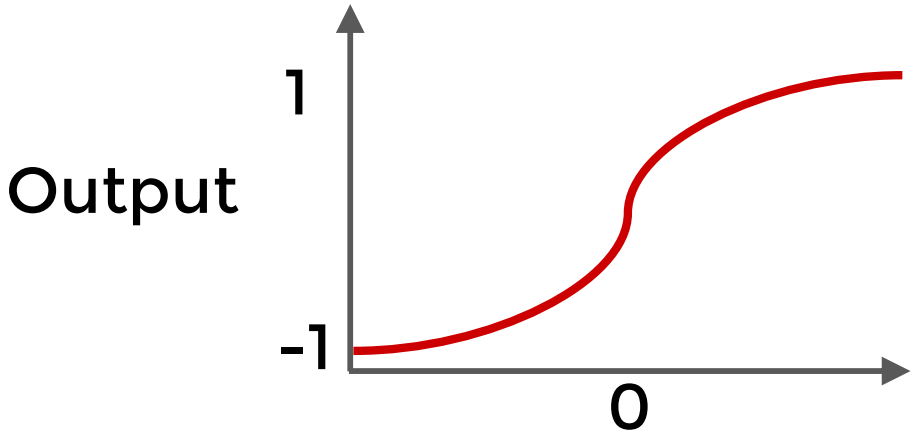


$$f(x) = \frac{1}{1 + e^{-(x)}}$$

- Let’s discuss a few more activation functions that we’ll encounter!



- Hyperbolic Tangent: tanh(z)

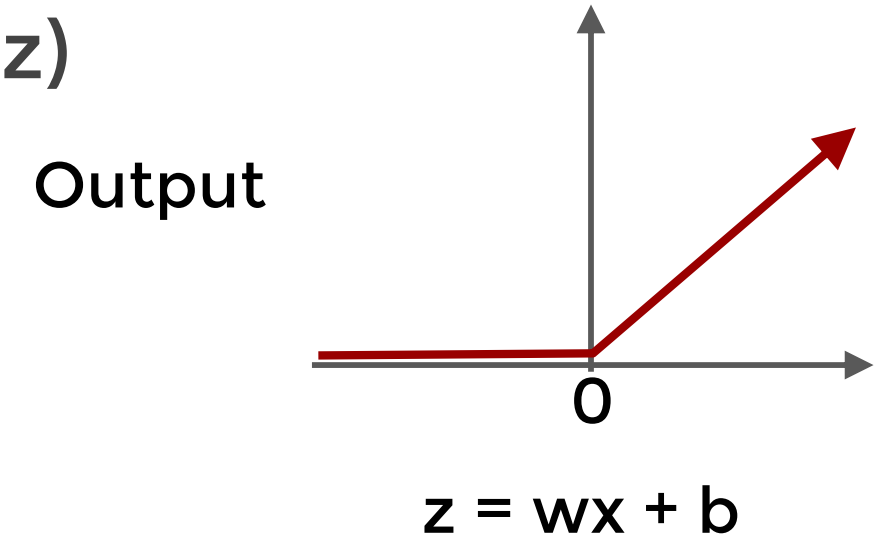


$$\cosh x = \frac{\mathbf{e}^{x} + \mathbf{e}^{-x}}{2}$$

$$\sinh x = \frac{\mathbf{e}^{x} - \mathbf{e}^{-x}}{2}$$

$$\tanh x = \frac{\sinh x}{\cosh x}$$

- Rectified Linear Unit (ReLU): This is actually a relatively simple function: max(0,z)



Perceptron

- ReLu and tanh tend to have the best performance, so we will focus on these two.
- Deep Learning libraries have these built in for us, so we don’t need to worry about having to implement them manually!

- As we continue on, we'll also talk about some more state of the art activation functions.
- Up next, we'll discuss cost functions, which will allow us to measure how well these neurons are performing!

Cost Functions

- Let’s now explore how we can evaluate performance of a neuron!
- We can use a cost function to measure how far off we are from the expected value.

- We'll use the following variables:
  - y to represent the true value
  - a to represent neuron's prediction
- In terms of weights and bias:
  - $w*x + b = z$
  - Pass z into activation function
  - $\sigma(z) = a$

- Quadratic Cost
    - $C = \Sigma(y-a)^2 / n$
- We can see that larger errors are more prominent due to the squaring.
- Unfortunately this calculation can cause a slowdown in our learning speed.

- Cross Entropy
    - $C = (-1/n) \Sigma (y \cdot \ln(a) + (1-y) \cdot \ln(1-a)$
- This cost function allows for faster learning.
- The larger the difference, the faster the neuron can learn.

- We now have 2 key aspects of learning with neural networks, the neurons with their activation function and the cost function.
- We’re still missing a key step, actually “learning”!

- We need to figure out how we can use our neurons and the measurement of error (our cost function) and then attempt to correct our prediction, in other words, “learn”!

- In the next lecture we'll briefly cover how we can do this with Gradient Descent!

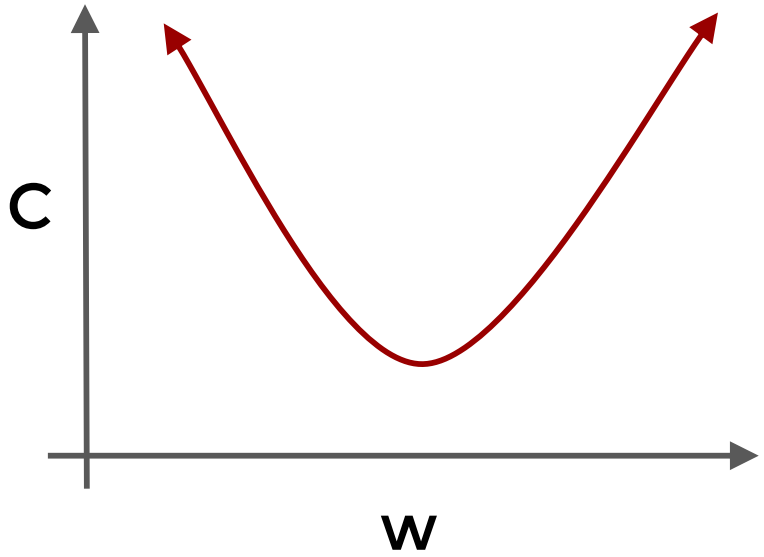
SMARTBRIDGE
Let's Bridge the Gap

Gradient Descent and Backpropagation

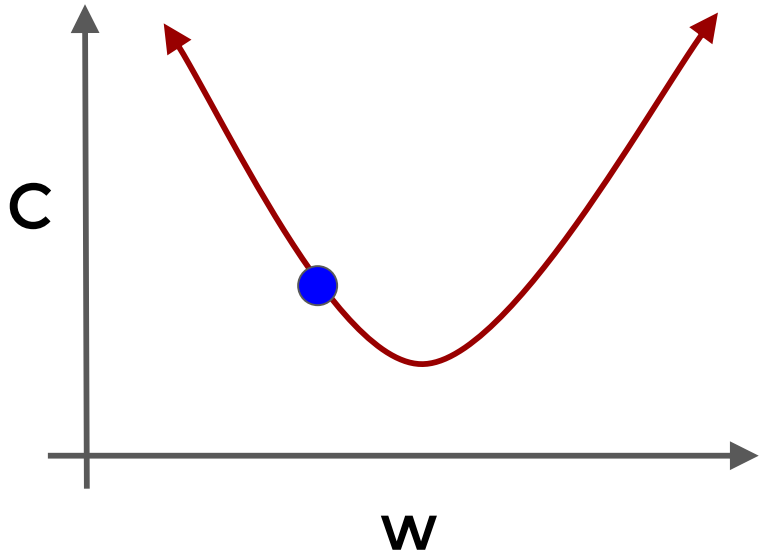
- If you’ve dabbled in machine learning before, you may have already heard of Gradient Descent!
- Let’s quickly go over it with a high level overview!

- Gradient descent is an optimization algorithm for finding the minimum of a function.
- To find a local minimum, we take steps proportional to the negative of the gradient.

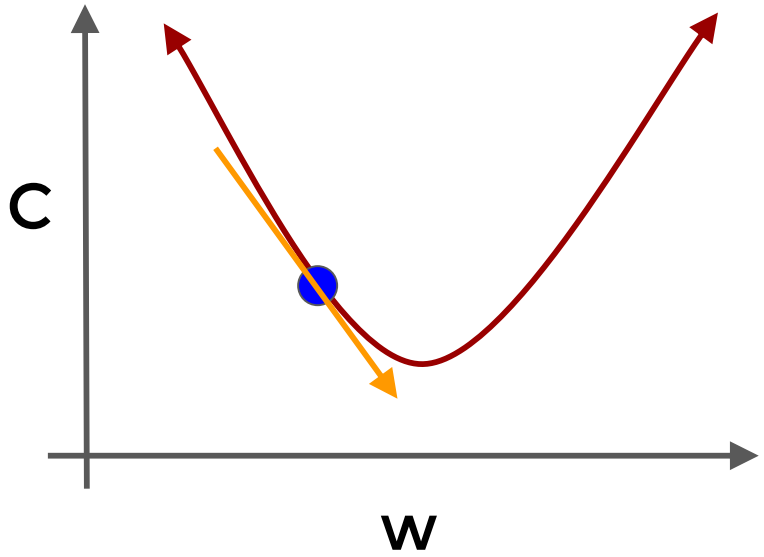
- Gradient Descent (in 1 dimension)



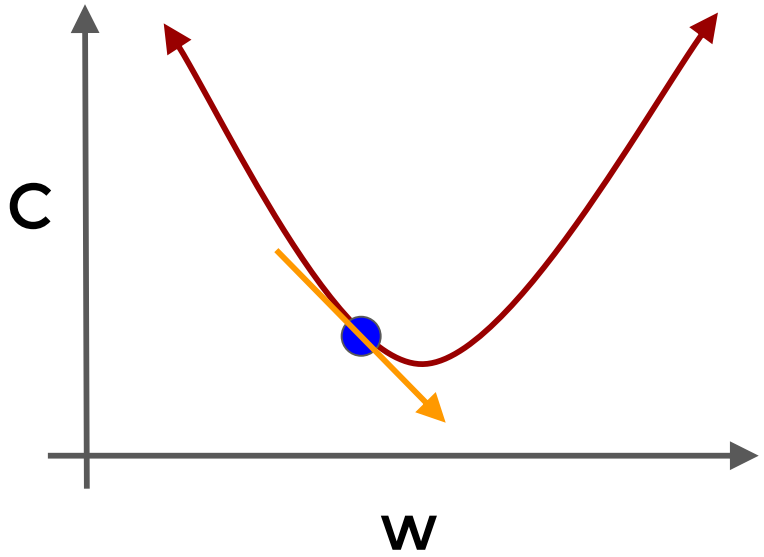
- Gradient Descent (in 1 dimension)



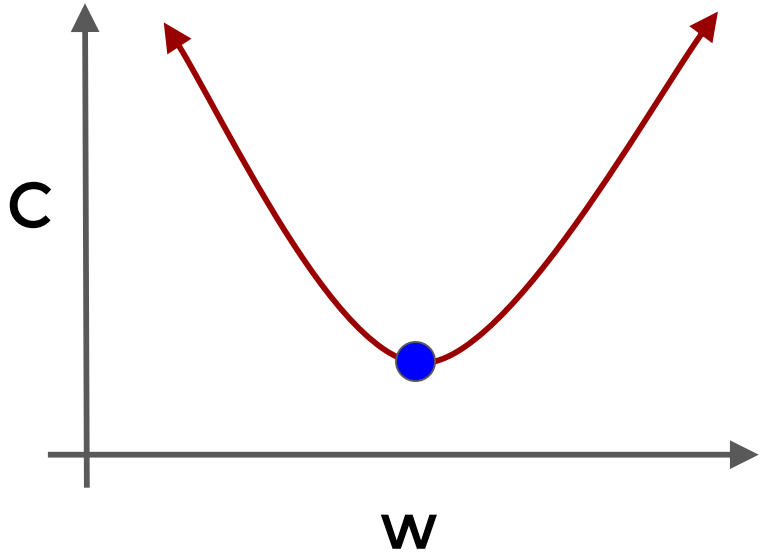
- Gradient Descent (in 1 dimension)



- Gradient Descent (in 1 dimension)



- Visually we can see what parameter value to choose to minimize our Cost!



- Finding this minimum is simple for 1 dimension, but our cases will have many more parameters, meaning we'll need to use the built-in linear algebra that our Deep Learning library will provide!

- Using gradient descent we can figure out the best parameters for minimizing our cost, for example, finding the best values for the weights of the neuron inputs.

- We now just have one issue to solve, how can we quickly adjust the optimal parameters or weights across our entire network?
- This is where backpropagation comes in!

- Backpropagation is used to calculate the error contribution of each neuron after a batch of data is processed.
- It relies heavily on the chain rule to go back through the network and calculate these errors.

- Backpropagation works by calculating the error at the output and then distributes back through the network layers.
- It requires a known desired output for each input value (supervised learning).

- The implementation of backpropagation will be further clarified when we dive into the math example!
- For now let’s finish off our high level discussion with TensorFlow’s playground!

SMARTBRIDGE
Let's Bridge the Gap

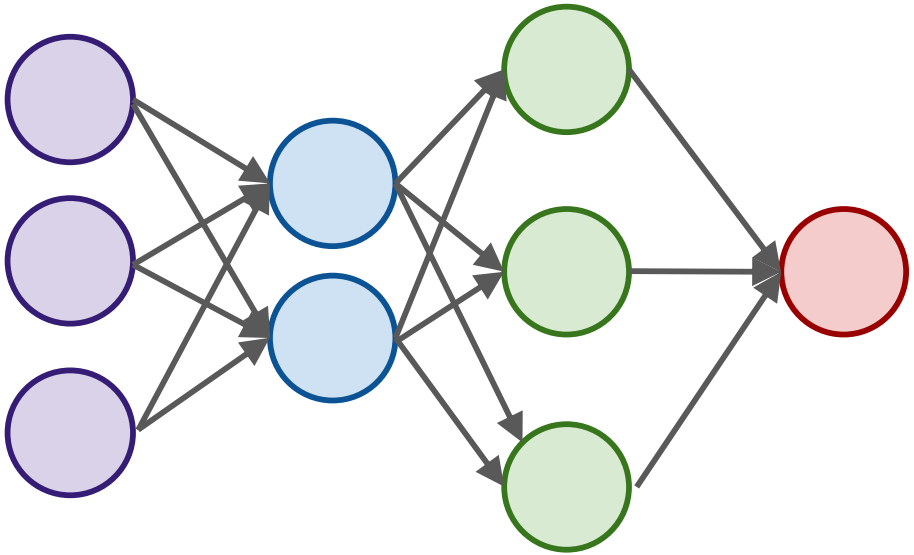
TensorFlow Playground

- Go to:
  - playground.tensorflow.org

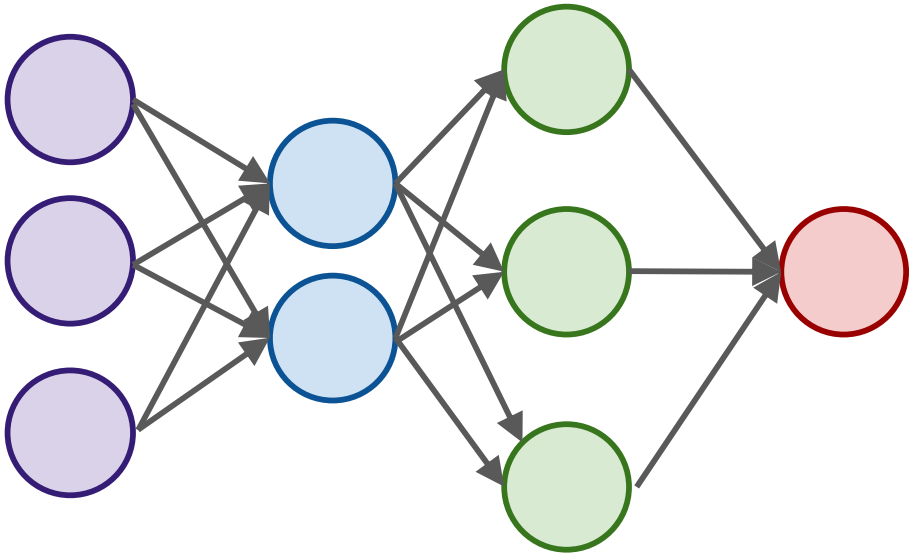
Types of Networks

- Let’s discuss high level overviews of the various types of neural networks
    - Dense Networks
    - Convolutional Networks-CNN
    - Recurrent Neural Networks-RNN
    - Generative Adversarial Networks-GAN

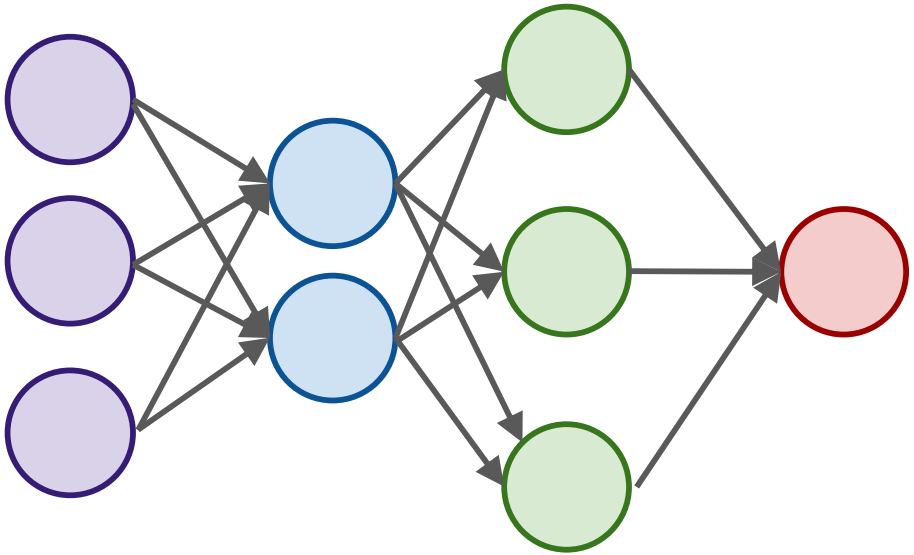
- Dense Networks



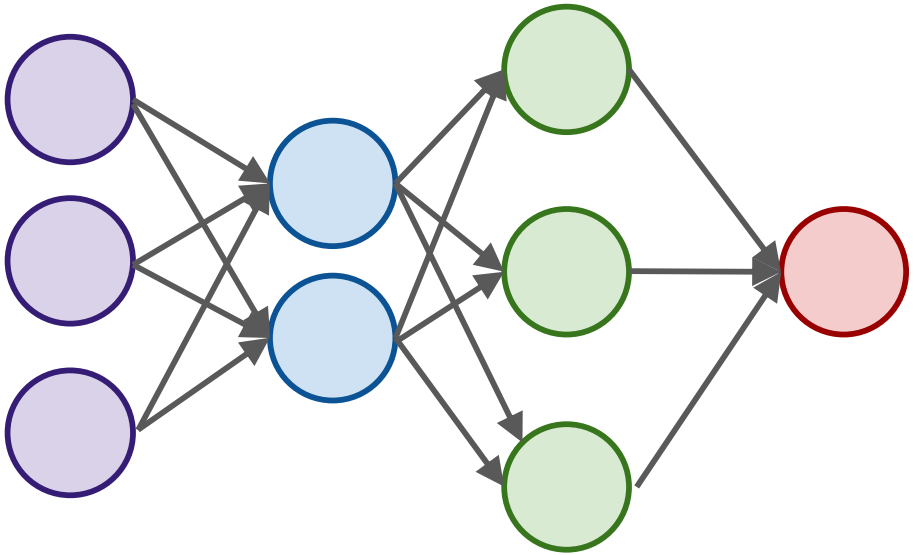
- Convolutional Neural Networks



- Recurrent Neural Networks



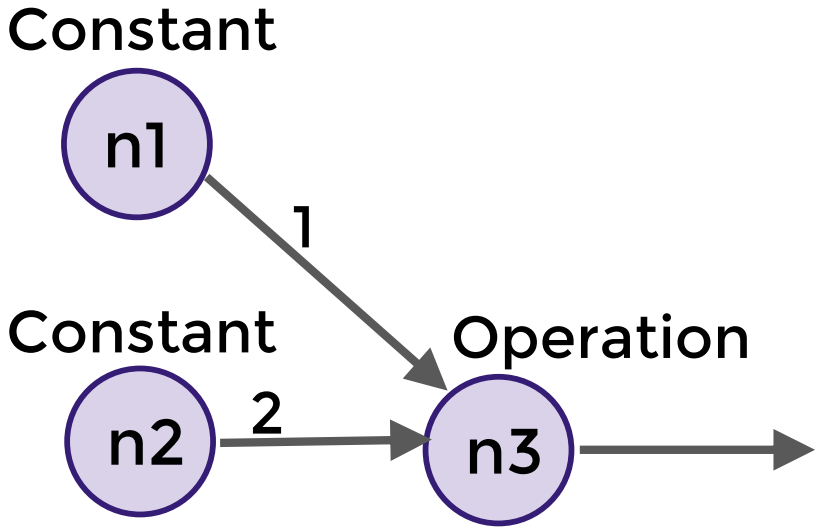
- Generative Adverserial Neural Networks



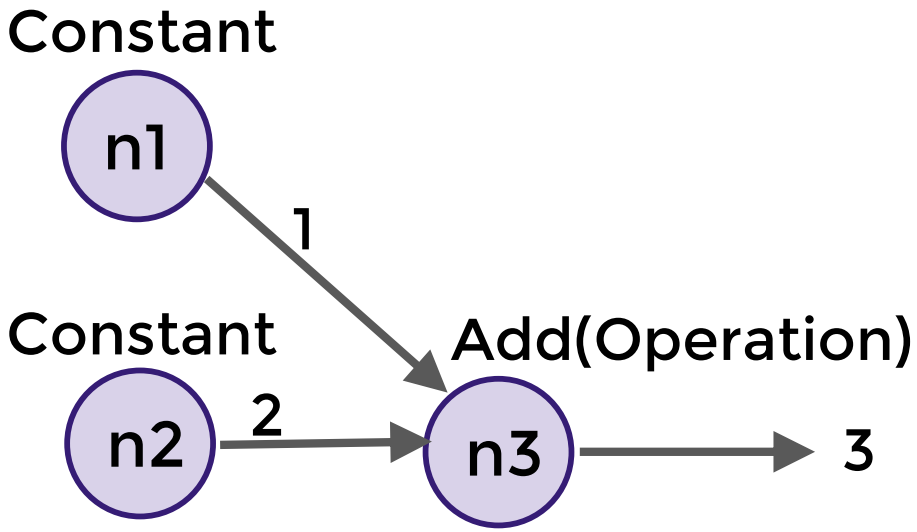
Manual Neural Network Part 2 - Operation

- Operation Class
  - Input Nodes
  - Output Nodes
  - Global Default Graph Variable
  - Compute
    - Overwritten by extended classes

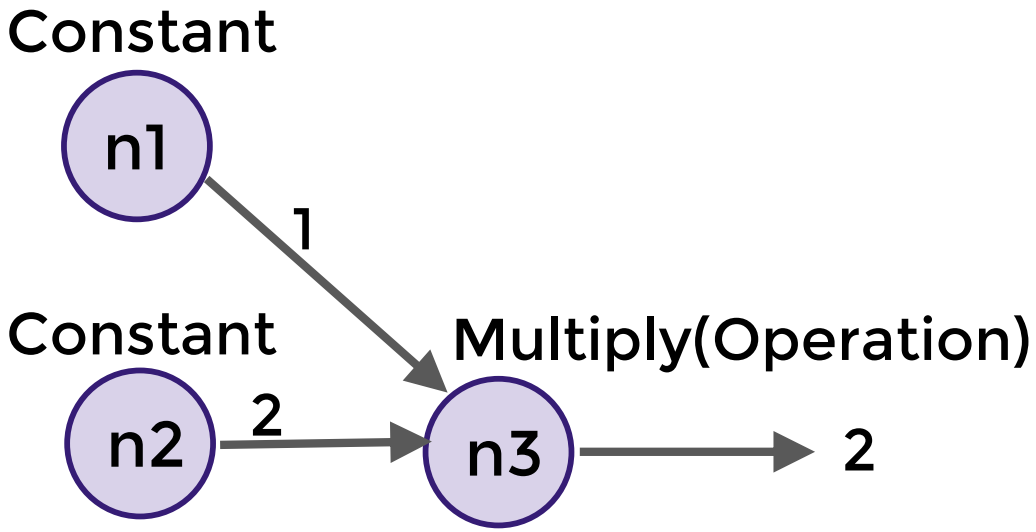
- Graph - A global variable



- Graph



- Graph



Manual Neural Network Variables,Placeholders, and Graphs

- Placeholder - An “empty” node that needs a value to be provided to compute output.
- Variables - Changeable parameter of Graph
- Graph - Global Variable connecting variables and placeholders to operations.

Let's get started!

Manual Neural Network Session

- Now that the Graph has all the nodes, we need to execute all the operations within a Session.
- We'll use a PostOrder Tree Traversal to make sure we execute the nodes in the correct order.

Manual Neural Network Classification

- $y = mx + b$
- $y = -1x + 5$
- Remember that both y and x are features!
- Feat2 = -1*Feat1 + 5
- Feat2 + Feat1 - 5 = 0
- FeatMatrix[1, 1] - 5 = 0