**Feedforward neural networks** are [artificial neural networks](https://brilliant.org/wiki/artificial-neural-network/) where the connections between units do not form a [cycle](https://brilliant.org/wiki/graphs/##graphs-basic). Feedforward neural networks were the first type of artificial neural network invented and are simpler than their counterpart, [recurrent neural networks](https://brilliant.org/wiki/recurrent-neural-network/). They are called feedforward because information only travels forward in the network (no loops), first through the input nodes, then through the [hidden nodes](https://brilliant.org/wiki/artificial-neural-network/#putting-it-all-together) (if present), and finally through the output nodes.

Feedfoward neural networks are primarily used for [supervised learning](https://brilliant.org/wiki/supervised-learning/) in cases where the data to be learned is neither sequential nor time-dependent.

## Singe-layer Perceptron

The simplest type of feedforward neural network is the [**perceptron**](https://brilliant.org/wiki/perceptron/), a feedforward neural network with no [hidden units](https://brilliant.org/wiki/artificial-neural-network/#putting-it-all-together). Thus, a perceptron has only an input layer and an output layer. The output units are computed directly from the sum of the product of their [weights](https://brilliant.org/wiki/artificial-neural-network/#a-computational-model-of-the-neuron) with the corresponding input units, plus some [bias](https://brilliant.org/wiki/artificial-neural-network/#a-computational-model-of-the-neuron).

Historically, the perceptron's output has been binary, meaning it outputs a value of **0 or 1**. This is achieved by passing the aforementioned product sum into the [step function](https://brilliant.org/wiki/artificial-neural-network/#a-computational-model-of-the-neuron) *H*(*x*). This is defined as

**H**(x) = {

1 if  *x* ≥ 0

0 if  *x* < 0

}​

For a binary perceptron with *n*-dimensional input *x*, *n*-dimensional weight vector *w*, and bias *b*, the 1-dimensional **output** is

**output** ={

1 ​​if  *w*⋅*x*+*b* ≥ 0

0 if  *w*⋅*x*+*b* < 0

}​

Since the perceptron divides the input space into two classes, 0 and 1, depending on the values of *w* and *b*, it is known as a [linear classifier](https://brilliant.org/wiki/linear-classifier/). The line separating the two classes is known as the **classification boundary** or decision boundary. In the case of a two-dimensional input (as in the diagram) it is a line, while in higher dimensions this boundary is a [hyperplane](https://brilliant.org/wiki/hyperplane/" \o "hyperplane" \t "_blank). The weight vector *w* defines the [slope](https://brilliant.org/wiki/slope/) of the classification boundary while the bias *b* defines the [intercept](https://brilliant.org/wiki/intercept/?wiki_title=intercept).

More general single-layer perceptrons can use activation functions other than the step function *H*(*x*).

Typical choices are:

**ReLu function** { x, x > 0; 0 otherwise } = max(x, 0)

**sigmoid function** *σ*(*x*)=1/(1+*e*−*x*)

**hyperbolic tangent**

Use of any of these functions ensures the output is a continuous number (as opposed to binary), and thus not every activation function yields a linear classifier.

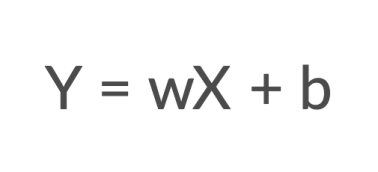
Generally speaking, a perceptron with activation function *g*(*x*) has output

*o*=*g*(*w*⋅*x*+*b*)

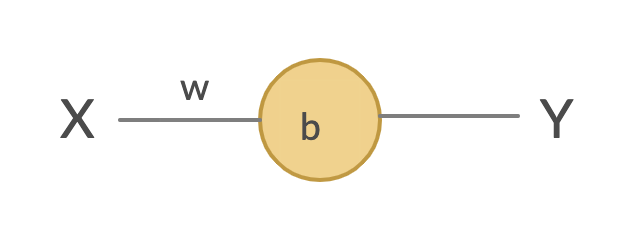
In order for a perceptron to learn to correctly classify a set of input-output pairs **(x, y)** it has to adjust the weights ***w*** and bias ***b*** in order to learn a good classification boundary.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Let’s start with something we all are familiar with. The most famous linear equation.



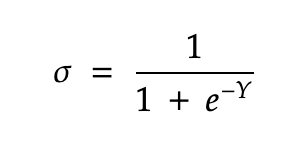
And now, let’s represent it in a Neural Network format.



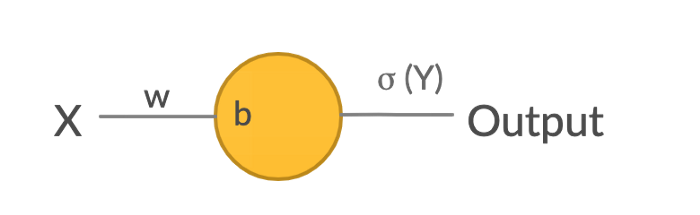
This is one of the most important part of understanding neural network — that it is just a different representation of our linear equations. When we pass an input (X) to the system, it multiplies it by a weight (***w***) and adds a bias (***b***) to it and gives us a prediction (Y). That’s all!

**Activation function**

The only issue with the above equation is that Y can have any range of values dependent on X, w and b. So, to control our output value between 0–1, let’s add a function at the Neuron. This is called activation function. One of the famous one is sigmoid function.

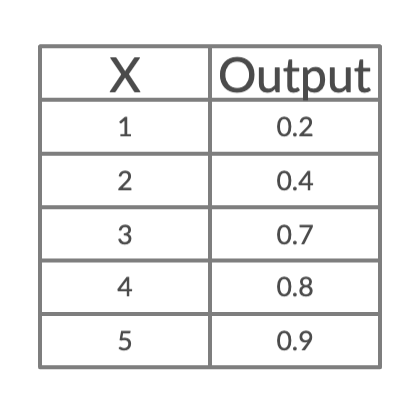


So, this is how our new neural network will look with the activation function.



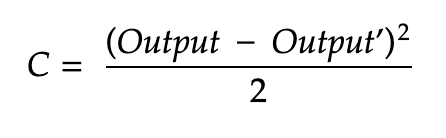
**Training and Test sets**

These are the actual values based on which we want to train our model. This is usually in this format — (X,Y). So, when you feed X to the system, you should receive Y as an output.



**Cost Function**

In simple words, it’s the difference between your predicted value Output’ and actual value of Output. It’s usually calculated as mean squared error (MSE).



*Whats our aim while working with neural network?*

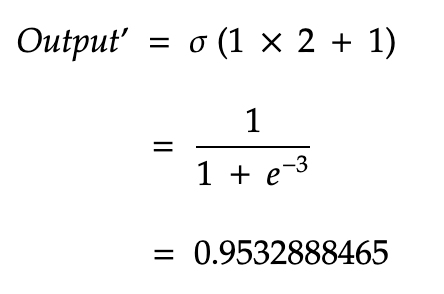
So, we are usually given training set and test set. The aim is to figure out **w** and **b**such that cost function is minimum. This is achieved via — **Backpropogation**.

So, let’s start with the basic mathematical calculations.

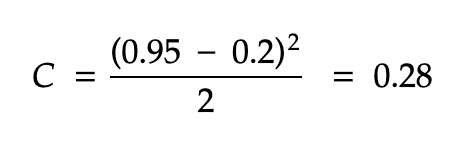
**Iteration 1**

*Taking* ***X*** *= 1 and* ***Output*** *= 0.2*

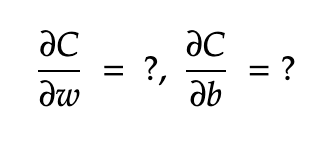
First let’s start with assumption,*say* ***w*** *= 2 and* ***b*** *= 1.*



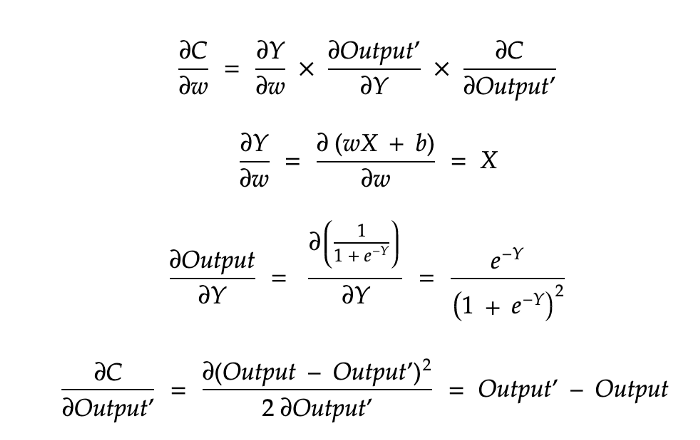
So, we got 0.95 but we expected output to be 0.2. So, the Cost function would be:



Now, we want to see how to change ***w*** and ***b***to decrease **C**. (our goal, remember!). Let me repeat again, so, basically how should we change ***w/b***, so that **C** will change. Yes, that’s exactly what differential calculus is!



Let’s take the first term for ***w***:



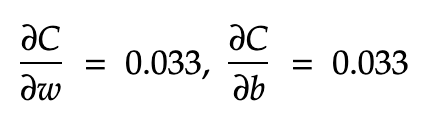
**Line 1**: Via chain rule, added few terms on the right side. It’s still the same expression if you think about it.

**Line 2-4**: Now, let’s try to simplify each terms on the right side individually. These are just differential calculations rules applied. I would highly advise you to revisit the rules to get a good understanding of the above. It’s pretty straightforward in case u even remotely remembers it.

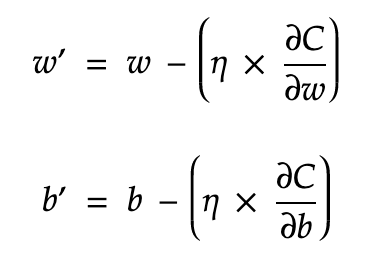
Similarly, repeat the process for ***b***:



So, the final values of change would be:



So, this is the change we got. So, let’s change w and b, by subtracting the above value. Let’s take learning **rate = η** = 0.5. Basically, we add learning rate to make sure we don’t update w and b too much.



So tada, the new values of ***w*** = 1.98, ***b*** = 0.98

Now, keep repeating the process for other values of the training set and keep updating ***w*** and ***b***. That’s what we call Stochastic Gradient Descent using back propagation.

Here, have tried to explain briefly how back propagation actually works using the simplest NN. In real world though, we never have such simple neural network. There, we have many neurons and weights and biases but the overall idea remains the same.