LET THE LEARNING BEGIN …

Single Layer Perceptron and Activation Function

A lengthy yet brief introduction to perceptrons and different type of activation functions

Ansh David · Follow
Published in CodeX
9 min read · Apr 23, 2021

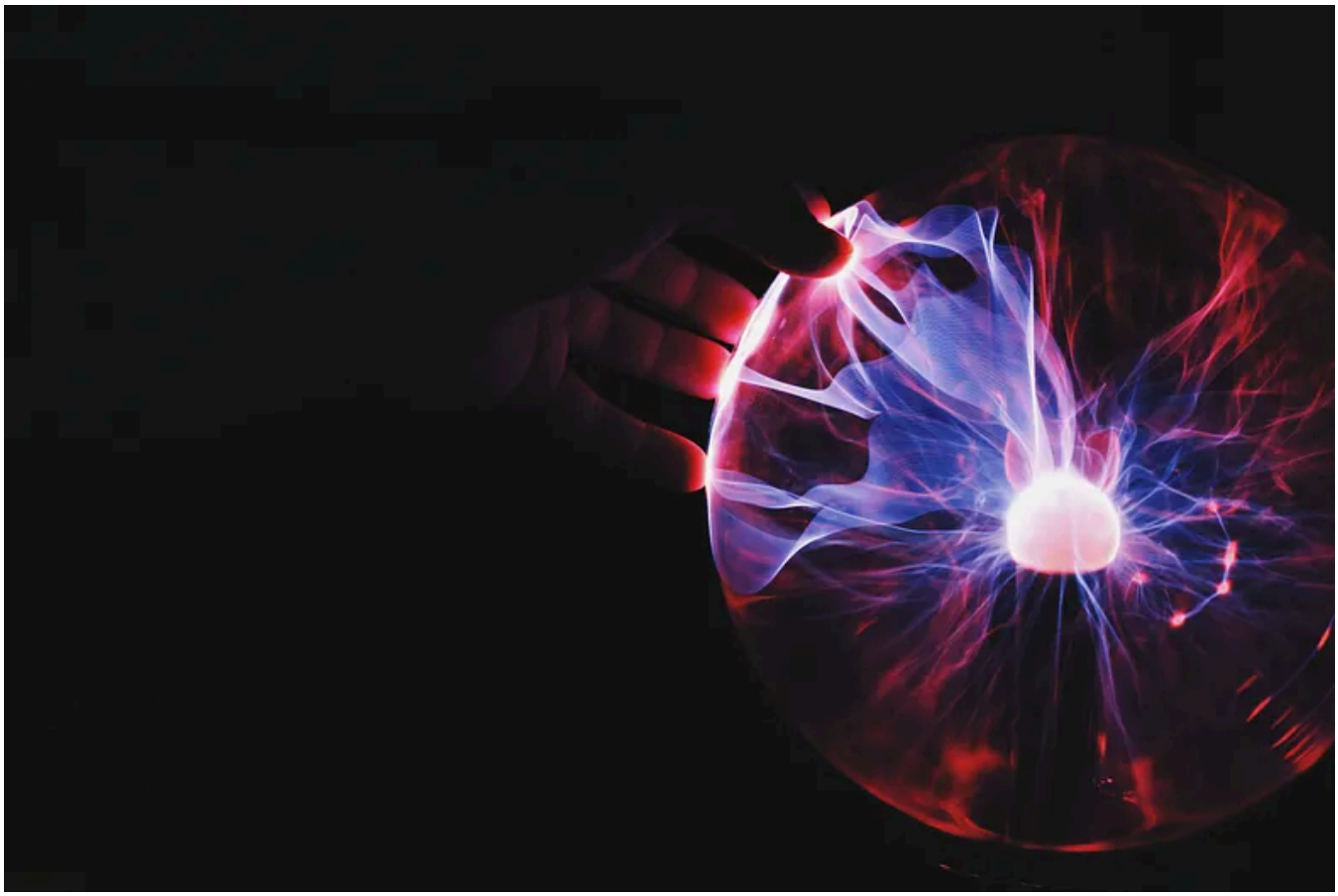
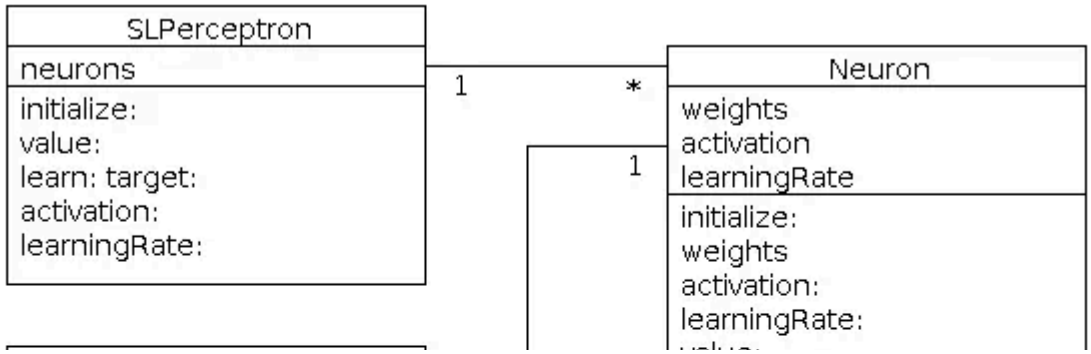
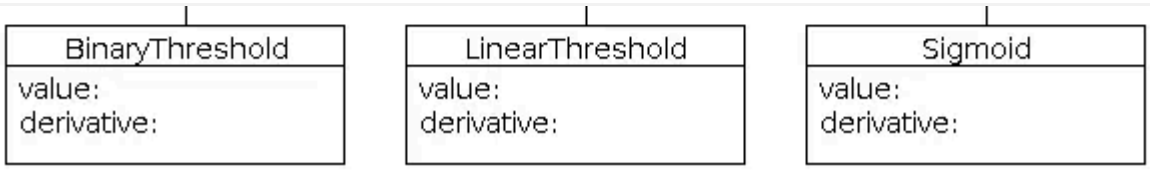


Photo by Ramón Salinero on Unsplash

A single layer perceptron (SLP) is a feed-forward network based on a threshold transfer function. SLP is the simplest type of artificial neural networks and can only classify linearly separable cases with a binary target. Activation functions are mathematical equations that determine the output of a neural network. The function is attached to each neuron in the network, and determines whether it should be activated or not, based on whether each neuron's input is relevant for the model's prediction.

How It Works



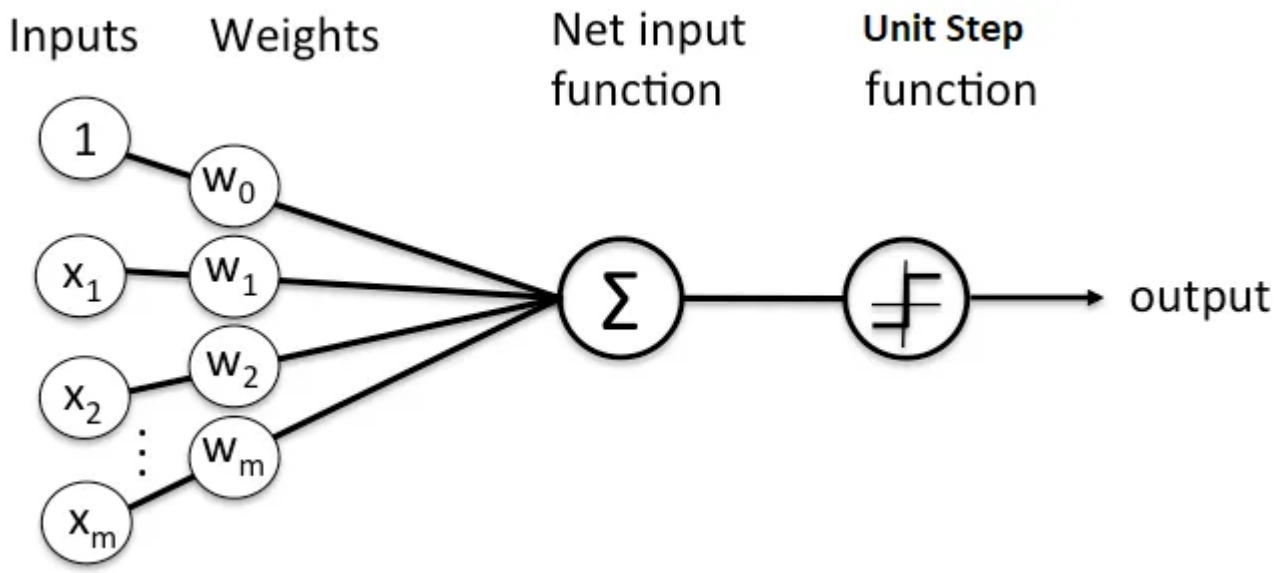


A perceptron is a linear classifier; that is, it is an algorithm that classifies input by separating two categories with a straight line. Input is typically a feature vector x multiplied by weights w and added to a bias b :

$$y = wx + b$$

A single-layer perceptron does not include hidden layers, which allow neural networks to model a feature hierarchy. It is, therefore, a shallow neural network, which prevents it from performing non-linear classification.

Training using Unit Step Function



- label the positive and negative class in our binary classification setting as `1` and `-1`
- linear combination of the input values `x` and weights `w` as $(z=w_1x_1+\cdots+w_mx_m)$
- define an activation function $g(z)$, where if $g(z)$ is greater than a defined threshold θ we predict 1 and -1 otherwise; in this case, this activation function g is an alternative form of a simple **Unit step function**, which is sometimes also called **Heaviside step function**

$$g(z) = \begin{cases} 1, & \text{if } z \geq \theta \\ -1, & \text{otherwise} \end{cases}$$

Summarizing , we end up with :

$$z = \sum_{j=1}^{m} x_j w_j = w^T x$$

where,

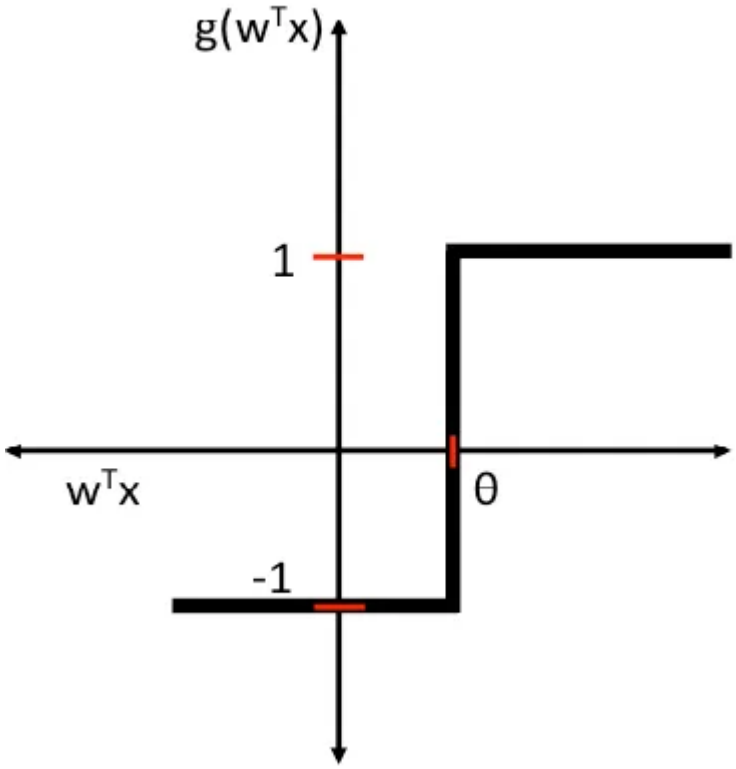
- w is the feature (weight) vector,

- x is an m-dimensional sample from the training dataset:

$$[w = \begin{bmatrix} w_1 \\ \vdots \\ w_m \end{bmatrix} \qquad x = \begin{bmatrix} x_1 \\ \vdots \\ x_m \end{bmatrix}$$

In order to simplify the notation, we bring θ to the left side of the equation and define $w_0 = -\theta$ and $x_0 = 1$ (also known as **bias**)

$$g(z) = \begin{cases} 1, & \text{if } z \geq 0 \\ -1, & \text{otherwise} \end{cases}$$



so that,

$$z = \sum_{j=0}^{m} x_j w_j = w^T x$$

Learning rule

Initial perceptron rule is fairly simple and can be summarized by the following steps:

1. Initialize the weights to `0` or small random numbers.
2. For each training sample x^i: calculate the output value and update the weights.
3. The output value is the class label predicted by the unit step function that we defined earlier and the weight update can be written more formally as $w_j = w_j + \Delta w_j$.
4. The value for updating the weights at each increment is calculated by the learning rule: $\Delta w_j = \eta(target^i - output^i)\, x_j^i$
   where η is the learning rate (a constant between 0.0 and 1.0), **target** is the true class label, and the **output** is the predicted class label.
5. All weights in the weight vector are being updated simultaneously

The convergence of the perceptron is only guaranteed if the two classes are linearly separable. If the two classes can't be separated by a linear decision boundary, we can set a maximum number of passes over the training dataset **epochs** and /or a **threshold** for the number of tolerated misclassifications.

Unit Step Function vs Activation Function

Activation functions are decision making units of neural networks. They calculates net output of a neural node. Herein, **Heaviside step function** is one of the most common activation function in neural networks. The function produces binary output. That is the reason why it also called as binary step function. The function produces 1 (or true) when input passes threshold limit whereas it produces 0 (or

false) when input does not pass threshold. That's why, they are very useful for binary classification studies.

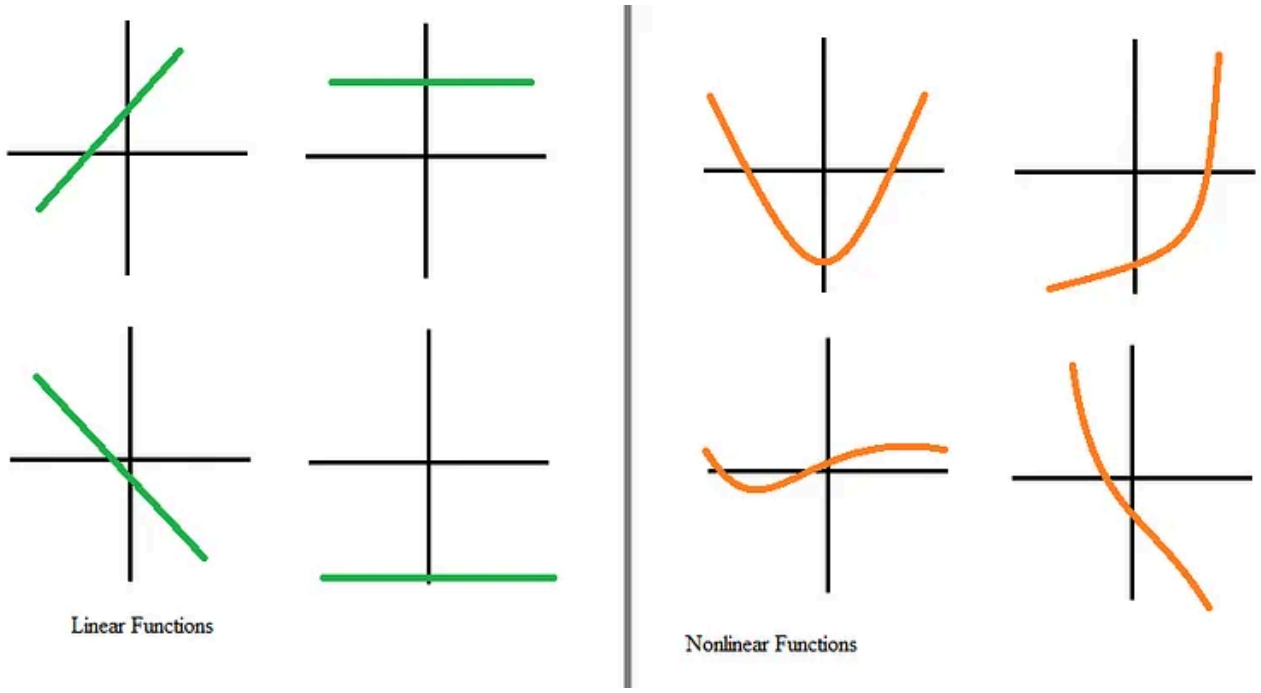
The **heaviside step function** is typically only useful within **single-layer perceptrons**, an early type of neural networks that can be used for classification in cases where the input data is **linearly separable**. However, multi-layer neural networks or multi-layer perceptrons are of more interest because they are general function approximators and they are able to distinguish data that is not linearly separable. Multi-layer perceptrons are trained using **backpropagation**. A requirement for backpropagation is a differentiable activation function. That's because backpropagation uses **gradient descent** on this function to update the network weights.

> *The Heaviside step function is non-differentiable at* `x = 0` *and its derivative is* `0` *elsewhere* `f(x) = x; -∞ → ∞`. *This means gradient descent won't be able to make progress in updating the weights and backpropagation will fail.*

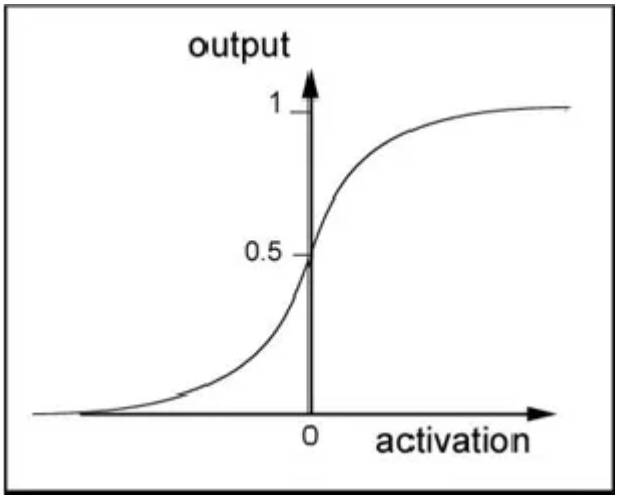
Neural networks are said to be universal function approximators. The main underlying goal of a neural network is to learn complex non-linear functions. If we do not apply any non-linearity in our multi-layer neural network, we are simply trying to separate the classes using a linear hyperplane.

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.
- Divided mainly on the basis of their **range or curves**



Sigmoid or Logistic Activation Function



The main reason why we use sigmoid function is because it exists between `0` and `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. It is often termed as a squashing function as well. It aims to introduce non-linearity in the input space. The non-linearity is where we get the wiggle and the network learns to capture complicated relationships. When a large negative number passed through the sigmoid function becomes `0` and a large positive number becomes `1`.

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

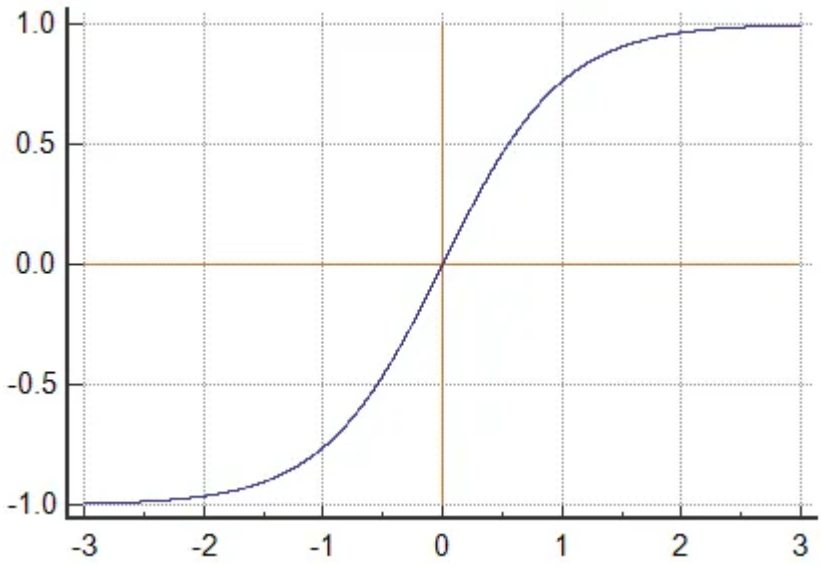
- **Killing gradients:** Sigmoid neurons get saturated on the boundaries and hence the local gradients at these regions is almost zero. The large positive values are squashed near 1 and large negative values are squashed near 0. Hence, effectively making the local gradient to near 0. As a result, during backpropagation, this gradient gets multiplied to the gradient of this neurons' output for the final objective function, hence it will effectively "kill" the gradient and no signal will flow through the neuron to its weights. Also, we have to pay attention to initializing the weights of sigmoid neurons to avoid saturation, because, if the initial weights are too large, then most neurons will get saturated and hence the network will hardly learn.

- **Non zero-centered outputs:** The output is always between 0 and 1, that means that the output after applying sigmoid is always positive hence, during gradient-descent, the gradient on the weights during backpropagation will always be either positive or negative depending on the output of the neuron. As a result, the gradient updates go too far in different directions which makes optimization harder.

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

- The function is **monotonic** but function's derivative is not.

Tanh or hyperbolic tangent Activation Function



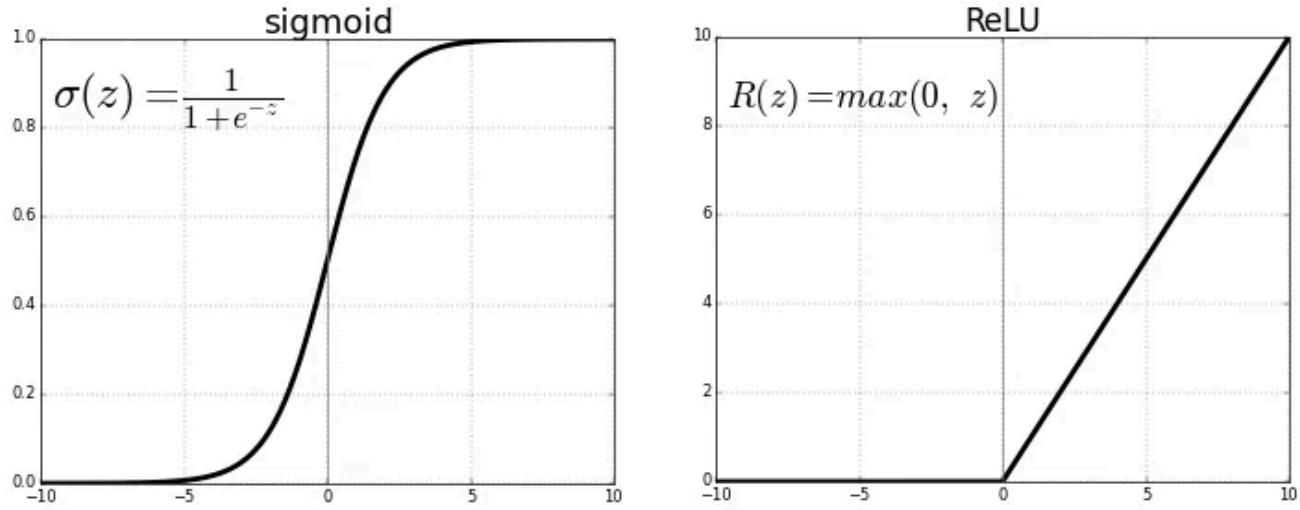
It is basically a shifted sigmoid neuron. It basically takes a real valued number and squashes it between -1 and $+1$. Similar to sigmoid neuron, it saturates at large

positive and negative values. However, its output is always zero-centered which helps since the neurons in the later layers of the network would be receiving inputs that are zero-centered. Hence, in practice, tanh activation functions are preferred in hidden layers over sigmoid.

$$\tanh(x) = \frac{(e^x - e^{-x})}{(e^x + e^{-x})} = \frac{2}{1 + e^{-2x}} - 1 = 2\text{sigmoid}(2x) - 1$$

- The function is **differentiable.**
- The function is **monotonic** while its derivative is not monotonic.
- The tanh function is mainly used classification between two classes.
- **Both tanh and logistic sigmoid activation functions are used in feed-forward nets.**

ReLU or Rectified Linear Unit



Fairly recently, it has become popular as it was found that it greatly accelerates the convergence of stochastic gradient descent as compared to Sigmoid or Tanh activation functions. It basically thresholds the inputs at zero, i.e. all negative values in the input to the ReLU neuron are set to zero. This decreases the ability of the model to fit or train from the data properly. Any negative input given to the ReLU activation function turns the value into zero immediately in the graph, which in turns affects the resulting graph by not mapping the negative values appropriately. The gradient is either `0` or `1` depending on the sign of the input.

$$\mathrm{ReLU}(z) = \max(0, z)$$

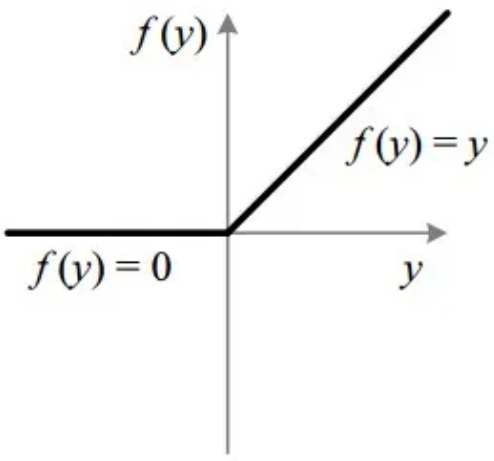
- **Sparsity of Activations:** ReLU and Tanh activation functions would almost always get fired in the neural network, resulting in the almost all the activations getting processed in calculating the final output of the network. Ideally want only a section of neurons to fire and contribute to the final output of the network and hence, we want a section of the neurons in the network to be passive. ReLU gives us this benefit. Hence, due to the characteristics of ReLU, there is a possibility that *50%* of neurons to give `0` activations and thus leading to fewer neurons to fire as a result of which the network becomes lighter and we can compute the output faster.

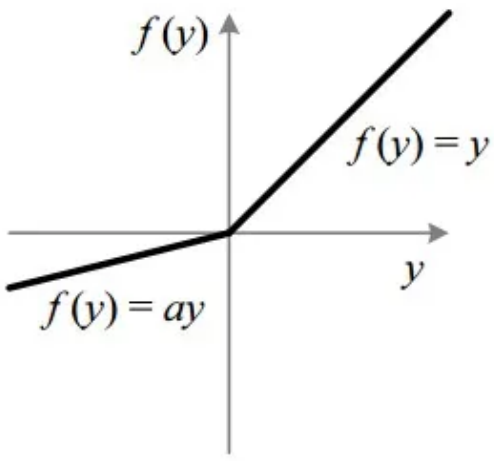
- **Dead Neurons:** Because of the horizontal line in ReLu(for negative `x`), the gradient can go towards `0`. For activations in that region of ReLu, gradient will be `0` because of which the weights will not get adjusted during descent. That means, those neurons which go into that state will stop responding to variations in error/ input (simply because gradient is `0`, nothing changes). This is called dying ReLu problem. This problem can cause several neurons to just die and not respond making a substantial part of the network passive. Hence, neurons that "die" will stop responding to the variations in the output error because of which the parameters will never be updated/updated during backpropagation.

- **Range:** `0` → `∞`

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

Leaky ReLU





Is just an extension of the traditional ReLU function. As we saw that for values less than `0`, the gradient is `0` which results in "Dead Neurons" in those regions. To address this problem, Leaky ReLU comes in handy. That is, instead of defining values less than `0` as `0`, we instead define negative values as a small linear combination of the input. The small value commonly used is `0.01`.

$$\text{LeakyReLU}(z) = \max(0.01 * z, z)$$

- The idea of Leaky ReLU can be extended even further by making a small change. Instead of multiplying `z` with a constant number, we can learn the multiplier and treat it as an additional hyperparameter in our process. This is known as Parametric ReLU.
- **Range:** `-∞ → ∞`
- Function and its **derivative** are **monotonic** in nature.

Reference:

1. https://sebastianraschka.com/Articles/2015_singlelayer_neurons.html
2. https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6
3. https://towardsdatascience.com/activation-functions-b63185778794

Any feedback or constructive criticism is welcomed.

Deep Learning | Perceptron | Activation Functions | Artificial Intelligence | Learning