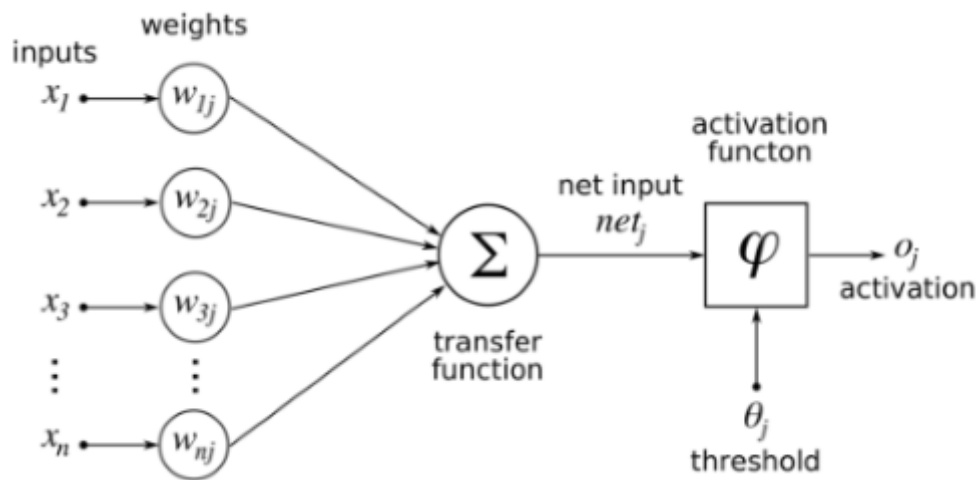


# Activation Functions

In artificial neural networks, the *activation* function (also referred to as a unit or neuron) defines the output of that node given an input or set of inputs. Every output from an activation function is the input to every activation function in the following layer. A standard computer chip circuit can be seen as a digital network of activation functions that can be "ON" or "OFF", depending on input.

Activation functions serve as *threshold*, *classification*, or sometime even called a *partition*. Bengio et al. refers to this as [Space Folding](#). What activation functions do is basically create a partition thereby dividing the original space into two partitions (typically).



There have been over 640 different variations of activation function proposals and there is no definitive guide for which activation function works best on specific problems. However, there are only a handful of activation functions that are recommended for use, and of those, the most popular are described below. It's a trial and error process where one should try different set of functions and see which one works best on the problem at hand.

Activation functions generally have similar characteristics:

1. **Non-Linear** - Most data that neural networks process is non-linear in nature. Activation functions provide this non-linearity in order to model non-linear relationships.
2. **Differentiable** - Plotting many activation functions typically result in a graph with a smooth, constantly changing slope. This makes it easier to identify the direction of the tangent slope to know which direction to update the weights in order to reduce error (find the global minimum with respect to a cost function).
3. **Fixed Range** - Activation functions typically transform the data into a fixed range such as Sigmoid which is between 0 and 1. These ranges are small typically between 0 and 1 or -1 and 1. Having small ranges makes learning faster and more efficient.

Activation functions are typically denoted by  $a$  where:

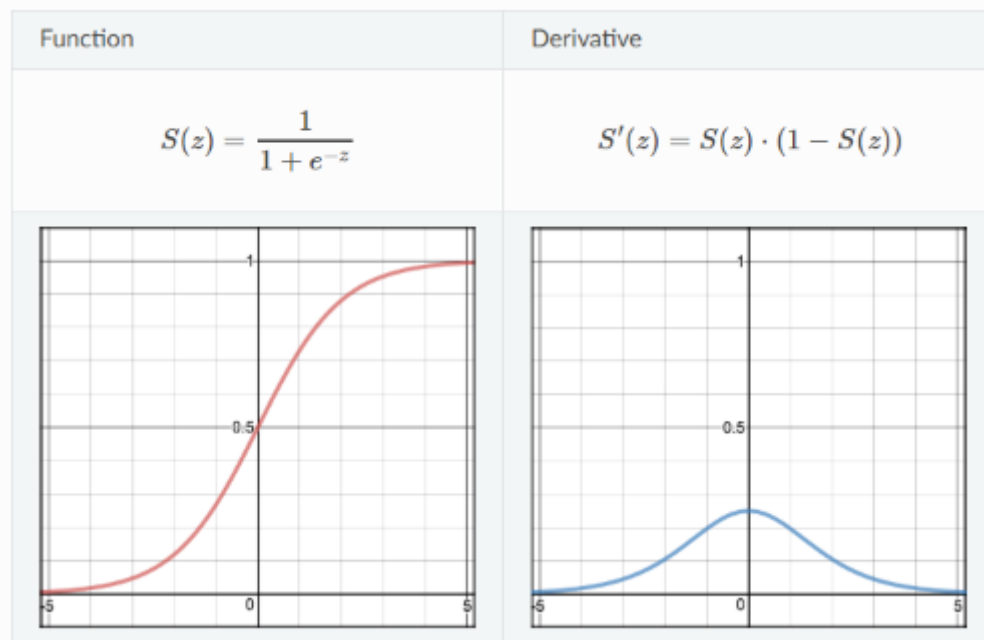
$a_i^{(j)}$  is an activation function of the  $i$ th activation function (neuron) in the  $j$ th hidden layer.

$a_1^{(2)}$  is an activation function of the 1<sup>st</sup> activation function (neuron) in the 2<sup>nd</sup> hidden layer.

## Sigmoid ( $\sigma$ )

$$g(z) = \frac{1}{1+e^{-z}}$$

The Sigmoid function returns a value that is between 0 and 1 and is a good candidate for classification problems. Therefore, it works nicely as an output function since it can return a probability as the output where a high positive probability would be closer to 1. The Sigmoid function, as shown below, only has a  $y$  value range between 0 and 1. This means for every input  $z$ , no matter how large, will always result in a small change in  $y$  value. As a result, when plotting the loss for Sigmoid, it's gradient (derivative) is very close to zero which gives rise to the problem of *vanishing gradients*. Because of this small gradient, using it for an activation function makes learning slow and more difficult. The network will stop learning or learning is drastically slow depending on use case and until gradient / computation gets hit by floating point value limits.



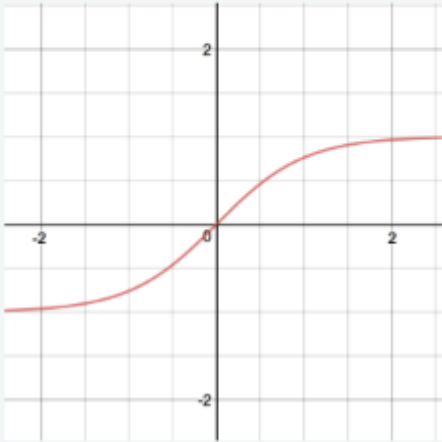
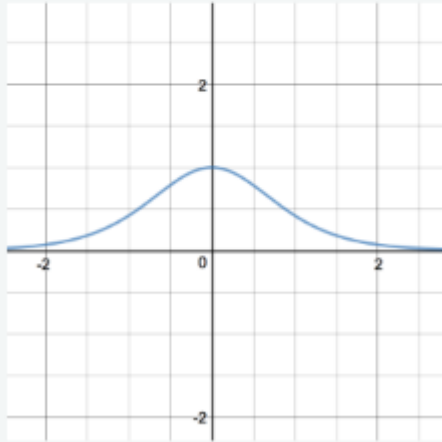
```
import numpy as np

def sigmoid(z):
    a = 1 / (1 + np.exp(-z))
    return a
```

## Hyperbolic Tangent (Tanh)

$$g(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$$

Hyperbolic Tangent is better than sigmoid in that the range of values don't center around 0.5 but 0 so it is generally preferred over Sigmoid for learning. The gradient is more prominent than Sigmoid but, like the Sigmoid, the gradient is still small making learning slow.

Function	Derivative
$\tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$	$\tanh'(z) = 1 - \tanh(z)^2$
	

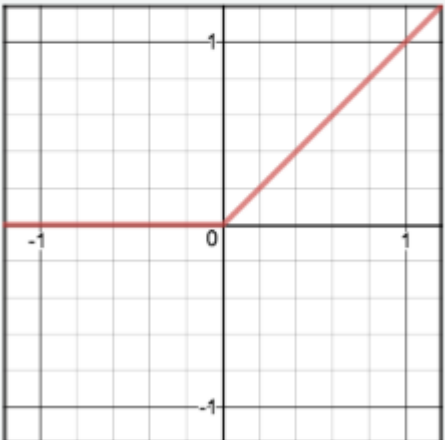
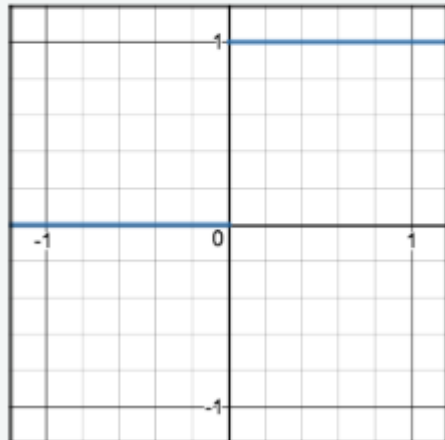
```
import numpy as np

def tanh(z):
    a = np.tanh(z)
    return a
```

## Rectified Linear Unit (ReLU)

$$g(z) = \max\{0, z\}$$

ReLU /rel-you/ range is 0 to *inf*, avoids the vanishing gradient problem and is less intensive computationally since the math is simpler. It's still non-linear in nature and is recommended to only be used in neural network hidden layers. If used in deep networks, the probability of resulting in dead neurons increases because some gradients can become fragile during training and cause a weight update that cause the activation unit to never activate. More specifically, when training during back propagation if an  $x$  value is less than 0, the resulting gradient is 0 and will stop responding to input variations (e.g. always outputting 0). This scenario is often times called the *dying ReLU problem*. In an effort to overcome the dying ReLU problem, changing the way parameters are initialized using random asymmetry can help (See also: [Dying ReLU and Initialization: Theory and Numerical Examples white paper](#) by Lu Lu and Yeonjong Shin, et. al.)

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ 0 & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ 0 & z < 0 \end{cases}$
	

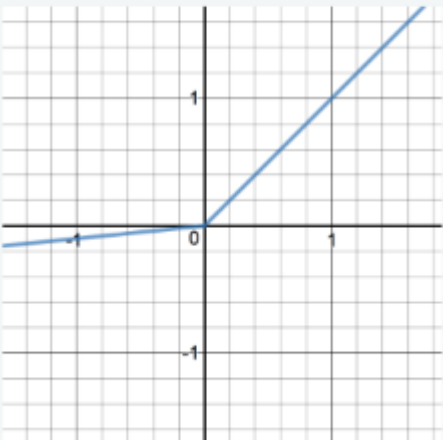
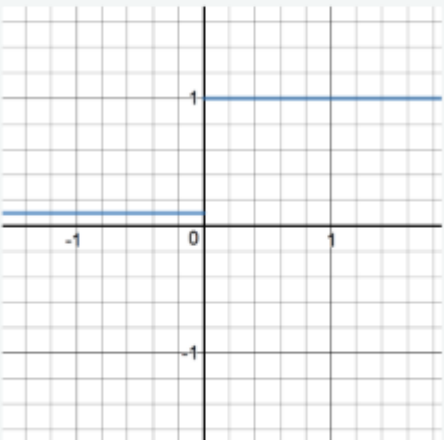
```
import numpy as np

def relu(a):
    a = np.maximum(0, z)
    return a
```

## Leaky Rectified Linear Unit (Leaky ReLU)

$$g(z) = \max\{\alpha \cdot z, z\}$$

Leaky ReLU attempts to overcome the zero gradient issue of ReLU. It overcomes this by supplying a small, non-zero gradient,  $\alpha$  value (usually 0.01) for  $z$  values  $\leq 0$ . However, since it possesses linearity using a constant gradient for values  $\leq 0$  it really can't be used for complex classification problems like Sigmoid or Tanh.

Function	Derivative
$R(z) = \begin{cases} z & z > 0 \\ \alpha z & z \leq 0 \end{cases}$	$R'(z) = \begin{cases} 1 & z > 0 \\ \alpha & z < 0 \end{cases}$
	

```
import numpy as np

def leaky_relu(z):
    a = np.maximum(0.1 * z, z)
    return a
```

## Randomized ReLU

[https://isaacchanghau.github.io/post/activation\\_functions/](https://isaacchanghau.github.io/post/activation_functions/)

## Maxout

## Softmax

## ELU

