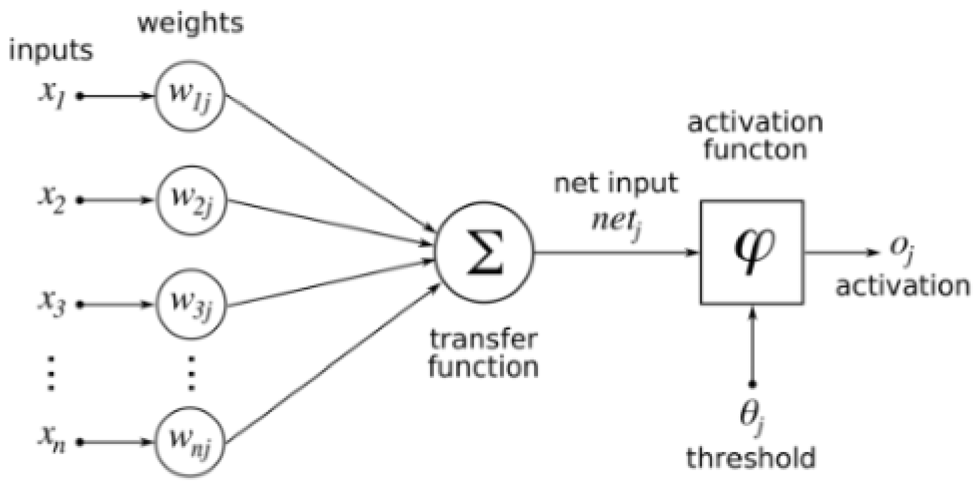
Activation Functions

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 bascially 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 handfull of activation functions that ar 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.

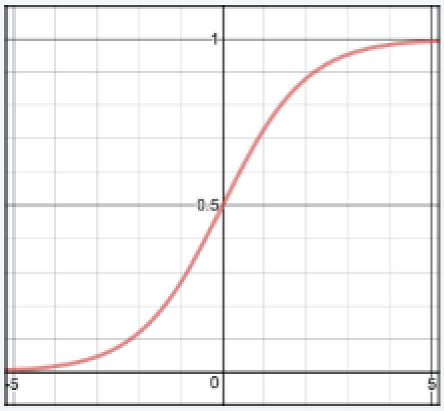
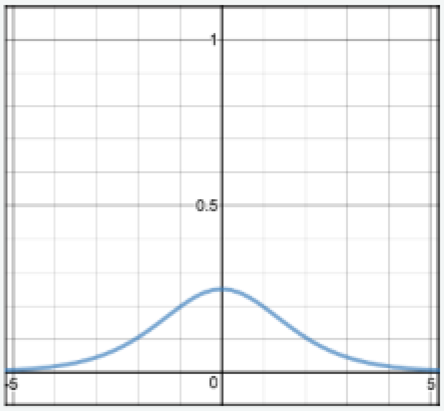
Activation functions are typically denoted by a where:

> $a_i^{(j)}$ is an activation function of the ith activation function (neuron) in the jth hidden layer.
>
> $a_1^{(2)}$ is an activation function of the 1^{st} activation function (neuron) in the 2^{nd} hidden layer.

Sigmoid (σ)

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

The Sigmoid function returns a value that is between 0 and 1 and is a good candidate for classifiation 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.

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

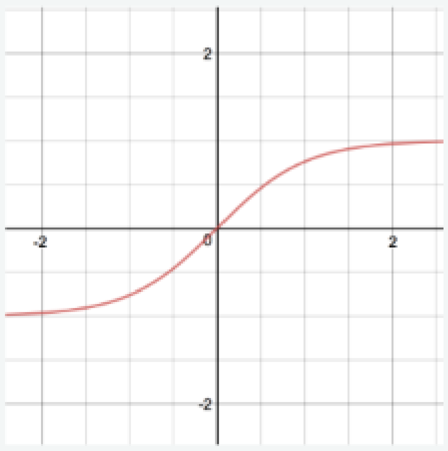
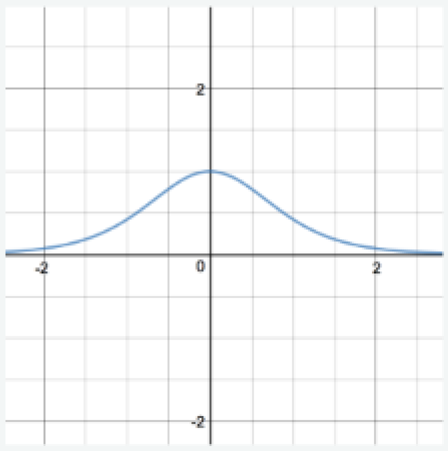
```python
import numpy as np

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

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

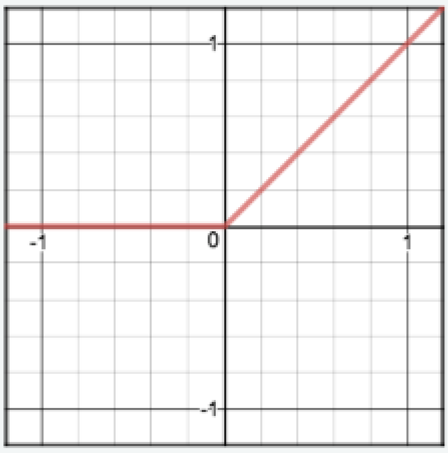
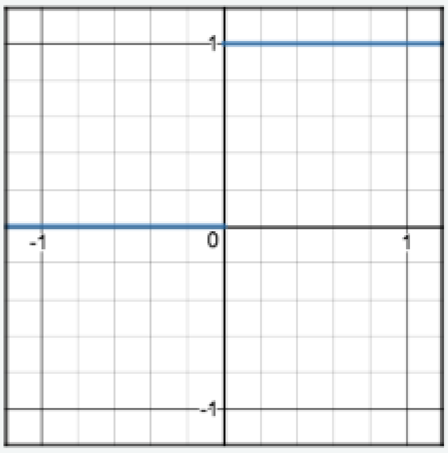
```python
import numpy as np

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

Rectified Linear Unit (ReLU)

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

ReLU 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 proabability of resulting in dead neurons increases because some gradients can become fragile during training and cause a weight update that cause the actiation unit to never activate. More specifically, when training during back propogation if an x value is less than 0, the resulting gradient is 0 and will stop responding to input variations (e.g. always outputing 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 asymetry can help (See also: Dying ReLU and Initialization: Theory and Numerical Examples white paper by Lu Lu and Yeonjong Shin, et. al.)

Function	Derivative
$R(z) = \left\{ \begin{matrix} z & z > 0 \\ 0 & z <= 0 \end{matrix} \right\}$	$R'(z) = \left\{ \begin{matrix} 1 & z > 0 \\ 0 & z < 0 \end{matrix} \right\}$
	

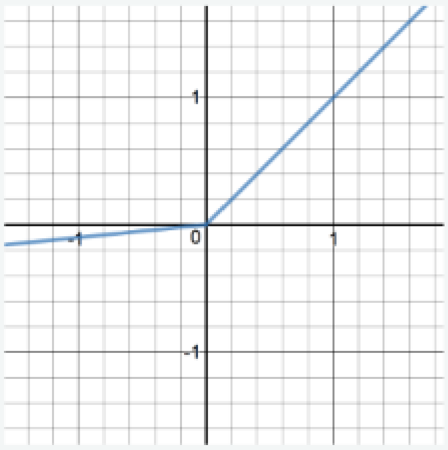
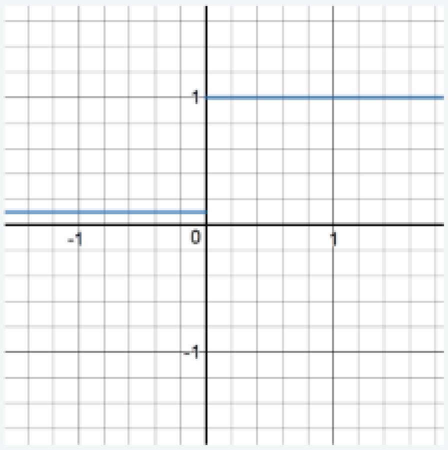
```python
import numpy as np

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, α value (usually 0.01) for z values ≤ 0. However, since it possesses liniarity using a constant gradient for values ≤ 0 it really can't be used for complex classification problems like Sigmoid or Tanh.

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

```python
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/

Maxout

Softmax

ELU

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