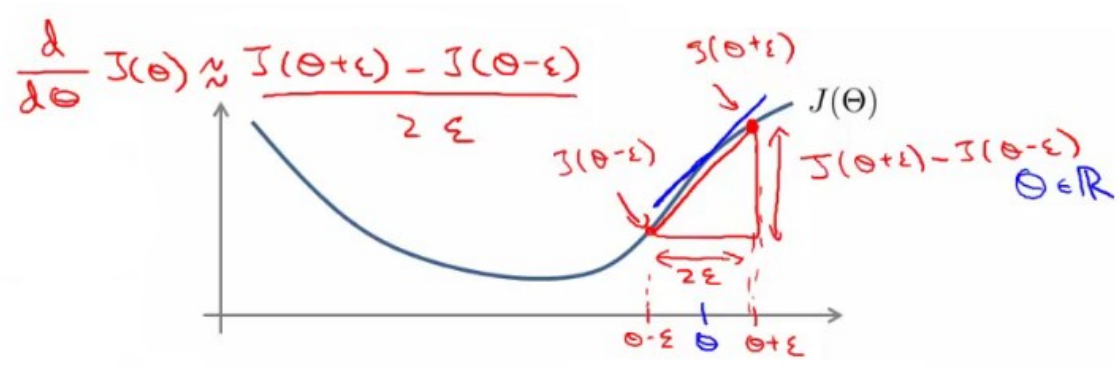
# Gradient Checking

Gradient checking will assure that our backpropagation works as intended. Often times, it is normal for small bugs to creep in the backpropagtion code. There is a very simple way of checking if the written code is bug free. It is based on calculating the slope of cost function manually by taking marginal steps ahead and behind the point at which the gradient is returned by backpropagation.




As visible in the plot above, the gradient approximation can be calculated using the centred difference formula as follows,

$$\frac{\partial}{\partial \Theta} J(\Theta) \approx \frac{J(\Theta + \epsilon) - J(\Theta - \epsilon)}{2\epsilon}$$

With multiple theta matrices, we can approximate the derivative with respect to $\Theta_j$ as follows:

$$\frac{\partial}{\partial \Theta_j} J(\Theta) \approx \frac{J(\Theta_1, \ldots, \Theta_j + \epsilon, \ldots, \Theta_n) - J(\Theta_1, \ldots, \Theta_j - \epsilon, \ldots, \Theta_n)}{2\epsilon}$$

A small value of $\epsilon = 10^{-4}$ guarantees that the math works out properly. If the value for $\epsilon$ is too small, we can end up with the actual slope derivative.

In previous module, we saw how to calculate the Delta vector using back propagation algorithm. So once we compute our Delta vector using the centered difference formula, we can check that if the both Delta vectors are appproximately equal which proves that the back propagation algorithm is working well.

# Random Initialization

Initializing all theta weights to zero does not work with neural networks. When we backpropagate, all nodes will update to the same value repeatedly. Instead we can randomly initialize our weights for our $\Theta$ matrices using the following method:

Hence, we initialize each $\Theta_{ij}^{(l)}$ to a random value between $[-\epsilon, \epsilon]$. Using the above formula guarantees that we get the desired bound. The same procedure applies to all the $\Theta's$

# Activation Functions

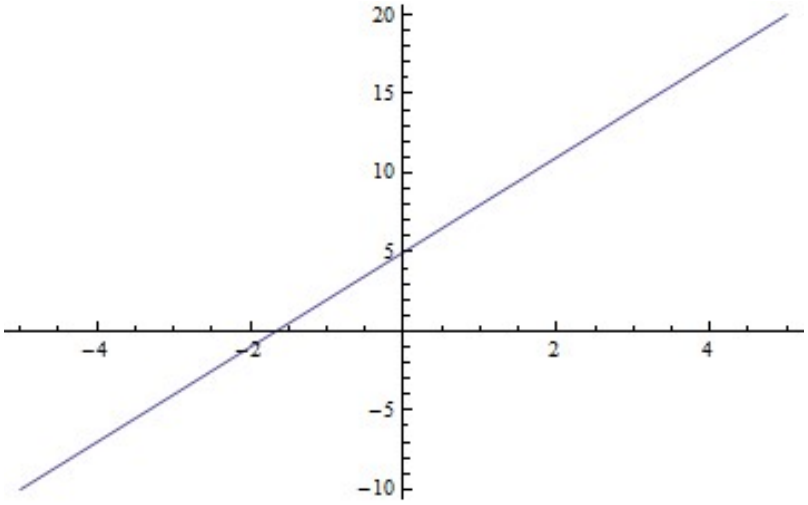
Activation functions are used to determine the firing of neurons in a neural network. Given a linear combination of inputs and weights from the previous layer, the activation function controls how we'll pass that information on to the next layer. An ideal activation function is both nonlinear and differentiable.

- The Non-linear behavior of an activation function allows our neural network to learn nonlinear relationships in the data.
- Differentiability is important because it allows us to backpropagate the model's error when training to optimize the weights.

## Linear Activation Function

$$f(x) = cx$$

A straight line function where activation is proportional to input ( which is the weighted sum from neuron ). We can definitely connect a few neurons together and if more than 1 fires, we could take the max ( or softmax) and decide based on that.




- Think about connected layers. Each layer is activated by a linear function. That activation in turn goes into the next level as input and the second layer calculates weighted sum on that input and it in turn, fires based on another linear activation function. No matter how many layers we have, if all are linear in nature, the final activation function of last layer is nothing but just a linear function of the input of first layer.
- That means these two layers ( or N layers ) can be replaced by a single layer. Ah! We just lost the ability of stacking layers this way. No matter how we stack, the whole network is still equivalent to a single layer with linear activation ( a combination of linear functions in a linear manner is still another linear function ).

## 1. Perceptron

While this is the original activation first developed when neural networks were invented, it is no longer used in neural network architectures because it's incompatible with backpropagation. Backpropagation allows us to find the optimal weights for our model using a version of gradient descent; unfortunately, the derivative of a perceptron activation function cannot be used to update the weights (since it is 0).

$$f(x) = \{0\ for\ x < 0, \quad 1\ for\ x \geq 0\}$$

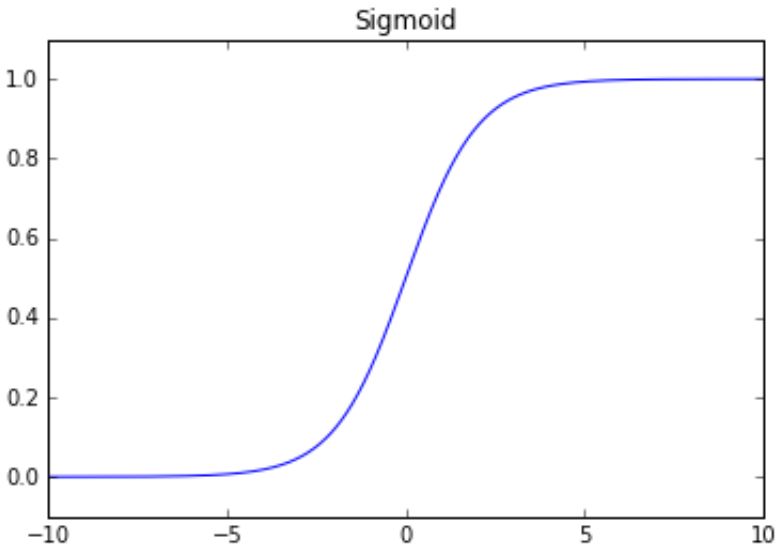
$$f'(x) = \{0\ for\ x \neq 0, \quad ?\ for\ x = 0\}$$

## 3. Sigmoid

The sigmoid functions are one of the most widely used activation functions today. However, it has fallen out of practice to use this activation function in real-world neural networks due to a problem known as the vanishing gradient.

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

$$f'(x) = f(x)(1 - f(x))$$




If you notice, towards either end of the sigmoid function, the Y values tend to respond very less to changes in X. What does that mean? The gradient at that region is going to be small. It gives rise to a problem of “vanishing gradients”. So what happens when the activations reach near the “near-horizontal” part of the curve on either sides? Gradient is small or has vanished ( cannot make significant change because of the extremely small value ).

## 3. Hyperbolic Tangent

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

$$f'(x) = 1 - f(x)^{2}$$

This looks very similar to sigmoid. In fact, it is a scaled sigmoid function!

$$\tanh(x) = 2\,sigmoid(2x) - 1$$

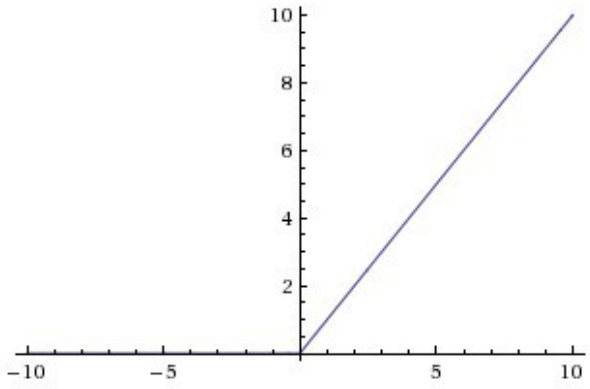
Like sigmoid, tanh also has the vanishing gradient problem.

## 4. Rectified Linear Units (ReLU)

The ReLU is used in almost all the convolutional neural networks or deep learning. ReLU is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. That is a good point to consider when we are designing deep neural nets.

$$f(x) = max(0, x)$$

$$f'(x) = \{0\ for\ x < 0, \quad 1\ for\ x \geq 0\}$$




The ReLU function is as shown above. It gives an output x if x is positive and 0 otherwise. The function and its derivative both are monotonic.

## Dying ReLU Problem

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. There are variations in ReLU to mitigate this issue by simply making the horizontal line into non-

## 5. Softmax Activation

The softmax function is commonly used as the output activation function for multi-class classification because it takes as input a vector of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. That is, prior to applying softmax, some vector components could be negative, or greater than one; and might not sum to 1; but after applying softmax, each component will be in the interval (0,1), and the components will add up to 1, so that they can be interpreted as probabilities.

As a result, we can consider the softmax function as a categorical probability distribution. This allows you to communicate a degree of confidence in your class predictions.

$$f_i\left(\vec{x}\right) = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \text{ for } \mathrm{i} = 1, . . , \mathrm{J}$$
$$\frac{\partial f_i\left(\vec{x}\right)}{\partial x_j} = f_i\left(\vec{x}\right)\left(\delta_{ij} - f_i\left(\vec{x}\right)\right)$$

where we apply the standard exponential function to each element $z_i$ of the input vector $z$ and normalize these values by dividing by the sum of all these exponentials; this normalization ensures that the sum of the components of the output vector $f\left(\vec{x}\right)$ is 1.

## How to choose activation functions?

***Hidden Layers:***

- Rectified linear unit (ReLU) is a preferred choice for all hidden layers because its derivative is 1 as long as z is positive and 0 when z is negative. In some cases, leaky rely can be used just to avoid exact zero derivatives.
- Both sigmoid and tanh functions are not suitable for hidden layers because if z is very large or very small, the slope of the function becomes very small which slows down the gradient descent.

***Output Layer:***

- Sigmoid functions works better for classifiers because approximating a classifier function as combinations of sigmoid is easier than maybe ReLu.

In [ ]: