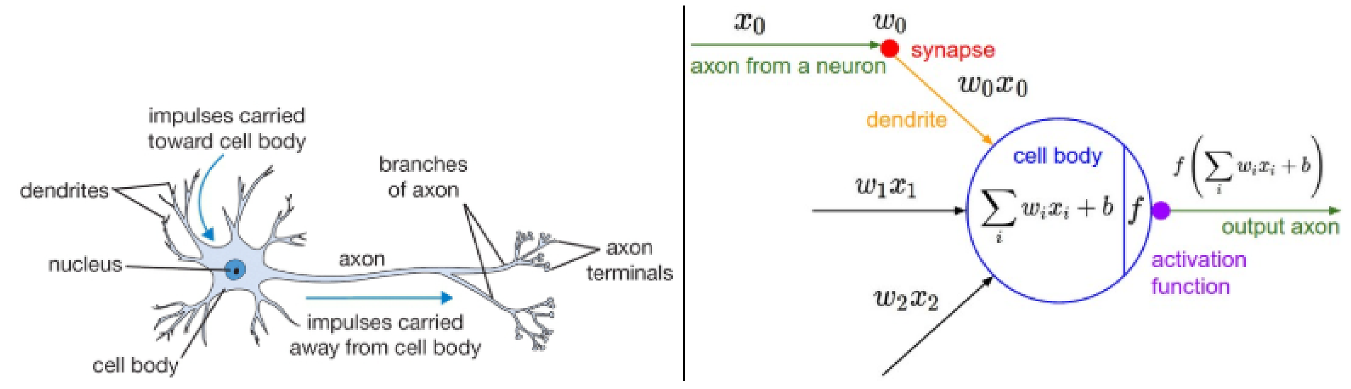
# Everything you need to know about “Activation Functions” in Deep learning models

This article is your **one-stop solution to every possible question related to activation functions that can come into your mind** that are used in deep learning models. **These are basically my notes on activation functions and all the knowledge that I have about this topic summed together in one place**. So, without going into any unnecessary introduction, let’s get straight down to business.

## Contents

## What is an activation function?

Simply put, an activation function is a function that is added into an artificial neural network in order to help the **network learn complex patterns in the data**. When comparing with a neuron-based model that is in our brains, the activation function is at the end deciding **what is to be fired to the next neuron**. That is exactly what an activation function does in an ANN as well. **It takes in the output signal from the previous cell and converts it into some form that can be taken as input to the next cell**. The comparison can be summarized in the figure below.



A cartoon drawing of a biological neuron (left) and its mathematical model (right).

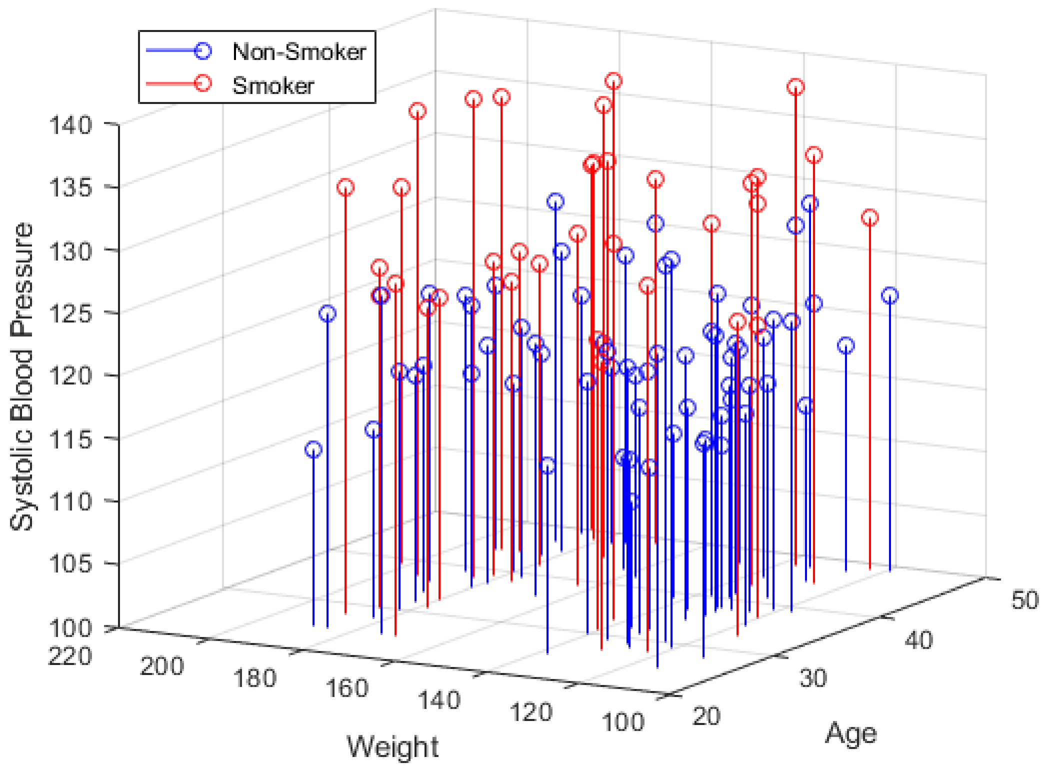
Source: cs231n by Stanford

## Why is there a need for it?

There are multiple reasons for having non-linear activation functions in a network.

1. Apart from the biological similarity that was discussed earlier, they also help in keeping the value of the output from the neuron restricted to a certain limit as per our requirement. This is important because input into the activation function is **W*x + b** where **W** is the weights of the cell and the **x** is the inputs and then there is the bias **b** added to that. This value if not restricted to a certain limit can go very high in magnitude especially in case of very deep neural networks that have millions of parameters. This will lead to computational issues. For example, there are some activation functions (like softmax) that out specific values for different values of input (0 or 1).

2. The most important feature in an activation function is its ability to add non-linearity into a neural network. To understand this, let's consider multidimensional data such as shown in the figure below:



A linear classifier using the three features(weight, Systolic Blood Pressure and Age in this figure) can give us a line through the 3-D space but it will never be able to exactly learn the pattern that makes a person a smoker or a non-smoker(the classification problem in hand) because the pattern that defines this classification is simply **not linear.** In come the artificial neural networks. What if we use an ANN with a single cell but without an activation function. So our output is basically **W*x + b.** But this is no good because **W*x also has a degree of 1**, hence linear and **this is basically identical to a linear classifier.**

What if we stack multiple layers. Let's represent $n^{th}$ layer as a function $f_n(x)$. So we have:

$$\mathbf{o(x) = f_n(f_{n-1}(\ldots.f_1(x))}$$

However, this is also not complex enough especially for problems with very high patterns such as that faced in computer vision or natural language processing.

In order to make the model get the power (aka the higher degree complexity) to learn the non-linear patterns, specific non-linear layers (activation functions) are added in between.

## Desirable features of an activation function

1. **Vanishing Gradient problem:** Neural Networks are trained using the process gradient descent. The gradient descent consists of the backward propagation step which is basically chain rule to get the change in weights in order to reduce the loss after every epoch. Consider a two-layer network and the first layer is represented as $f_1(x)$ and the second layer is represented as $f_2(x)$. The overall network is $o(x) = f_2(f_1(x))$. If we calculate weights during the backward pass, we get $o`(x) = f_2(x)*f_1`(x)$. Here $f_1(x)$ is itself a compound function consisting of *Act*($W_1*x_1 + b_1$) where *Act* is the activation function after layer 1. Applying chain rule again, we clearly see that $f_1`(x) =$ *Act*($W_1*x_1 + b_1$)$*x_1$ which means it also depends directly on the activation value. Now imagine such a chain rule going through multiple layers while backpropagation. If the value of *Act*() is between 0 and 1, then several such values will get multiplied to calculate the gradient of the initial layers. This reduces the value of the gradient for the initial layers and those layers are not able to learn properly. In other words, their gradients tend to vanish because of the depth of the network and the activation shifting the value to zero. This is called the **vanishing gradient problem**. So we want our activation function to not shift the gradient towards zero.

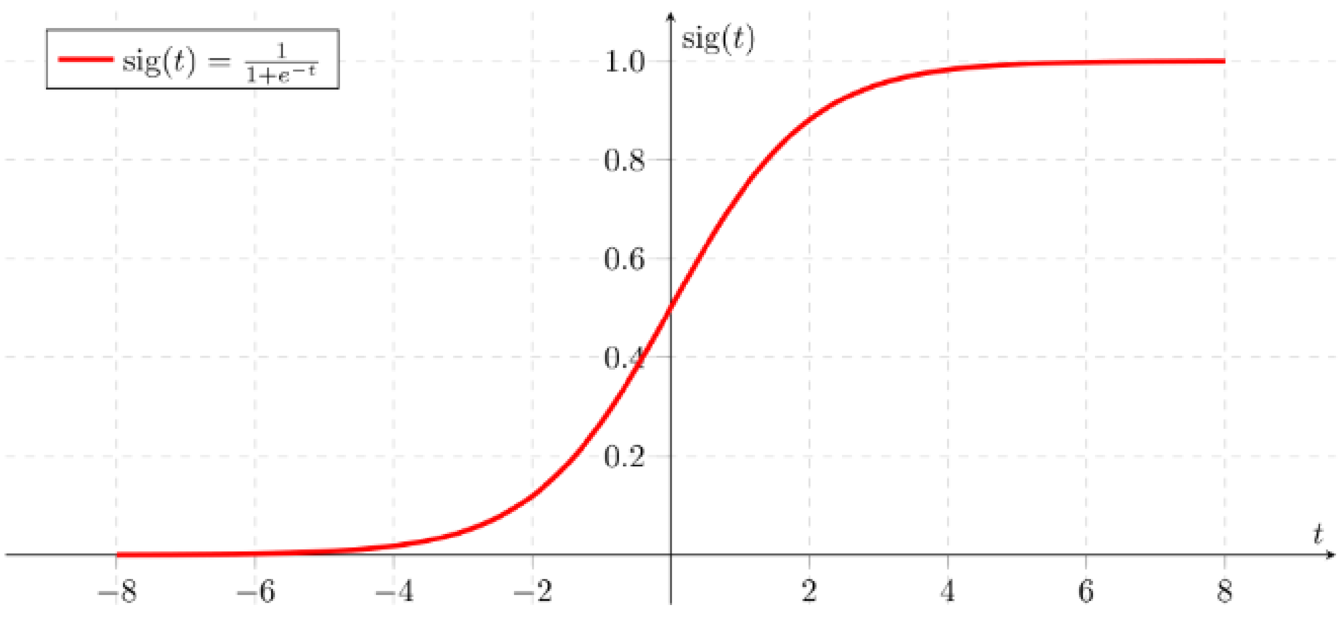
2. **Zero-Centered:** Output of the activation function should be symmetrical at zero so that the gradients do not shift to a particular direction.

3. **Computational Expense**: Activation functions are applied after every layer and need to be calculated millions of times in deep networks. Hence, they should be computationally inexpensive to calculate.

4. **Differentiable:** As mentioned, neural networks are trained using the gradient descent process, hence the layers in the model need to differentiable or at least differentiable in parts. **This is a necessary requirement for a function to work as activation function layer.**

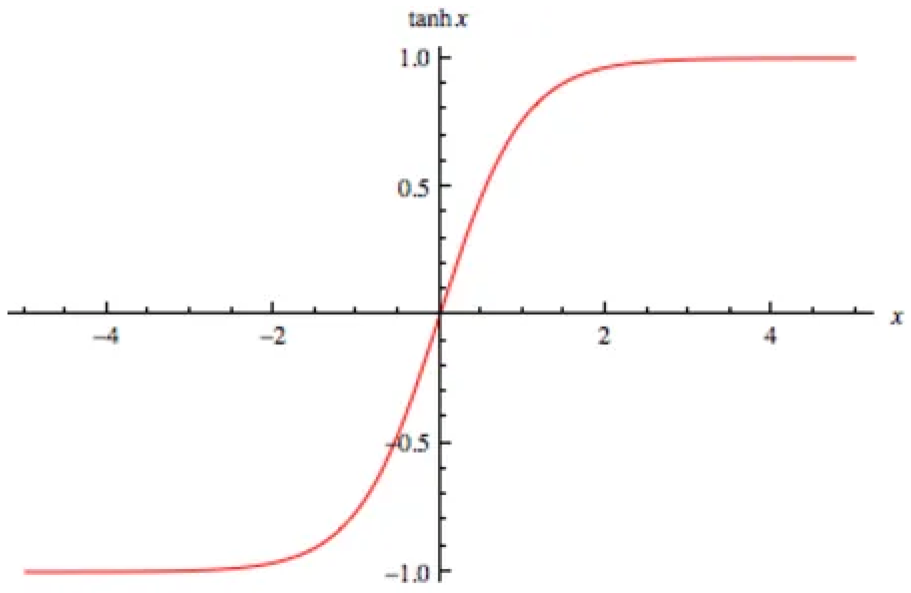
## Various non-linear activations in use

- **Sigmoid:** The sigmoid is defined as:



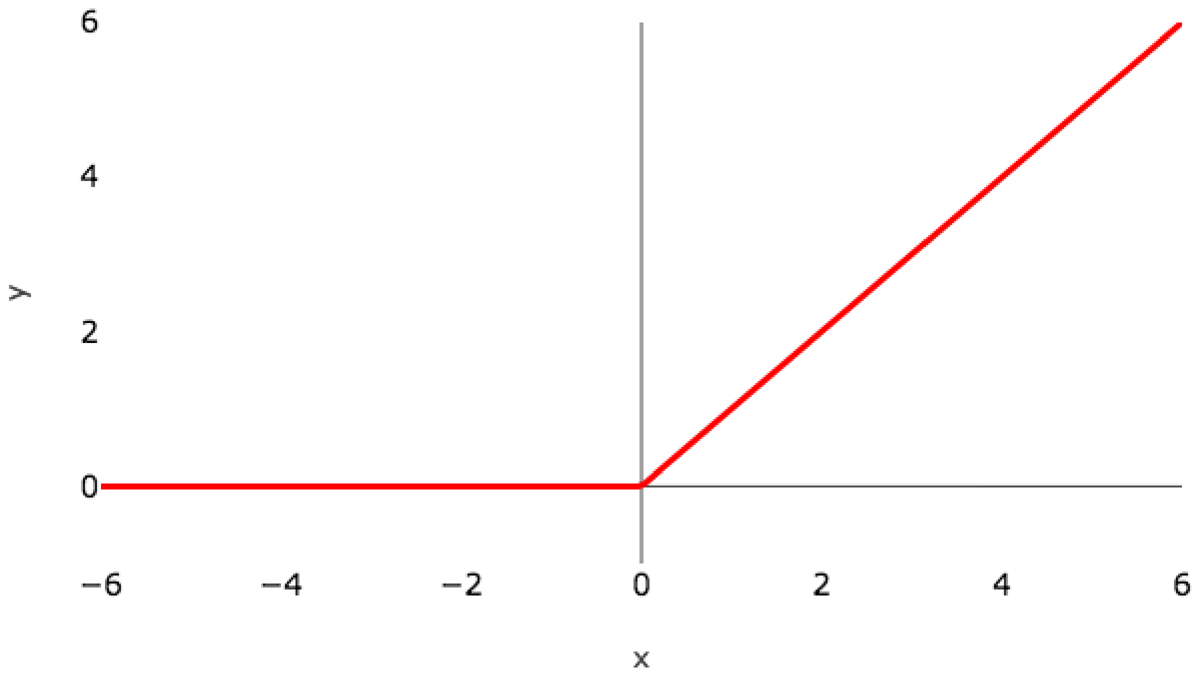
**This activation function is here only for historical reasons and never used in real models.** It is computationally expensive, causes vanishing gradient problem and not zero-centred. This method is generally used for binary classification problems.

- **Softmax**: The softmax is a more generalised form of the sigmoid. It is used in **multi-class classification problems**. Similar to sigmoid, it produces values in the range of 0–1 therefore it is used as the final layer in classification models.
- **Tanh:** The tanh is defined as:



If you compare it to sigmoid, it solves just one problem of being zero-centred.

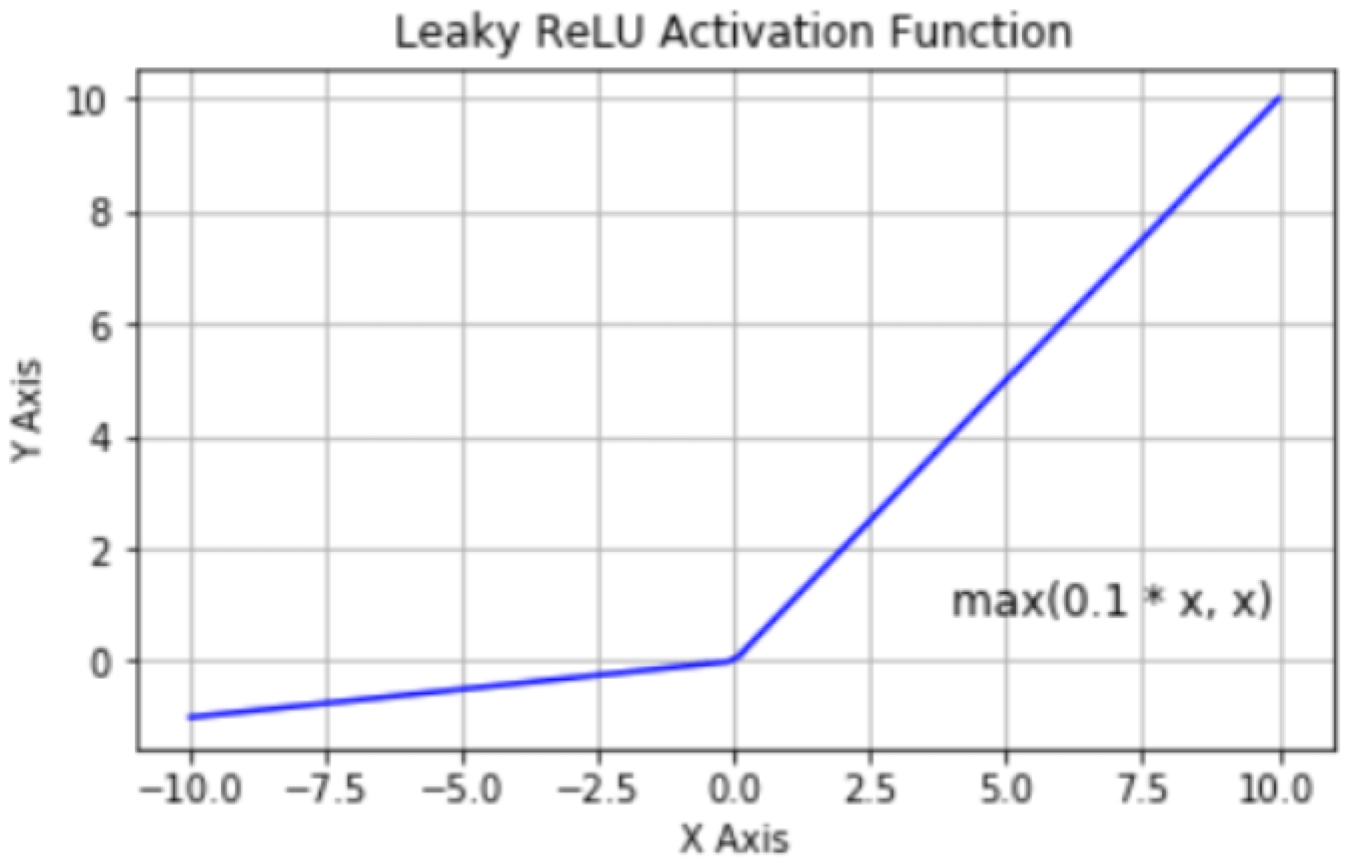
- **ReLU**: ReLU **(Rectified Linear Unit)** is defined as **f(x) = max(0,x):**



This is a widely used activation function, especially with Convolutional Neural networks. It is easy to compute and does not saturate and does not cause the Vanishing Gradient Problem. It has just one issue of not being zero centred. It suffers from **"dying ReLU"** problem. Since the output is zero for all negative inputs. It causes some nodes to completely die and not learn anything.

Another problem with ReLU is of exploding the activations since it higher limit is, well, inf. This sometimes leads to unusable nodes.

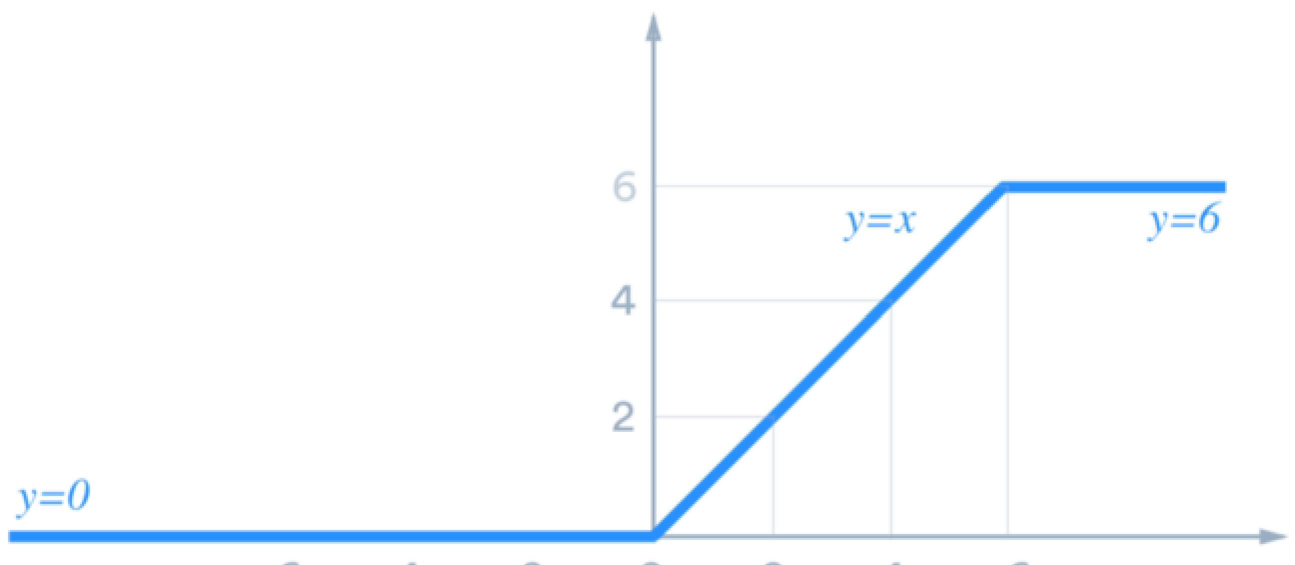
- **Leaky ReLU and Parametric ReLU**: It is defined as **f(x) = max(αx, x)**



the figure is for α = 0.1

Here α is a hyperparameter generally set to **0.01**. Clearly, Leaky ReLU solves the **"dying ReLU"** problem to some extent. Note that, if we set α as 1 then Leaky ReLU will become a linear function f(x) = x and will be of no use. Hence, the value of **α is never set close to 1.** If we set **α** as a hyperparameter for each neuron separately, we get **parametric ReLU** or **PReLU**.

- **ReLU6**: It is basically ReLU restricted on the positive side and it is defined as **f(x) = min(max(0,x),6)**



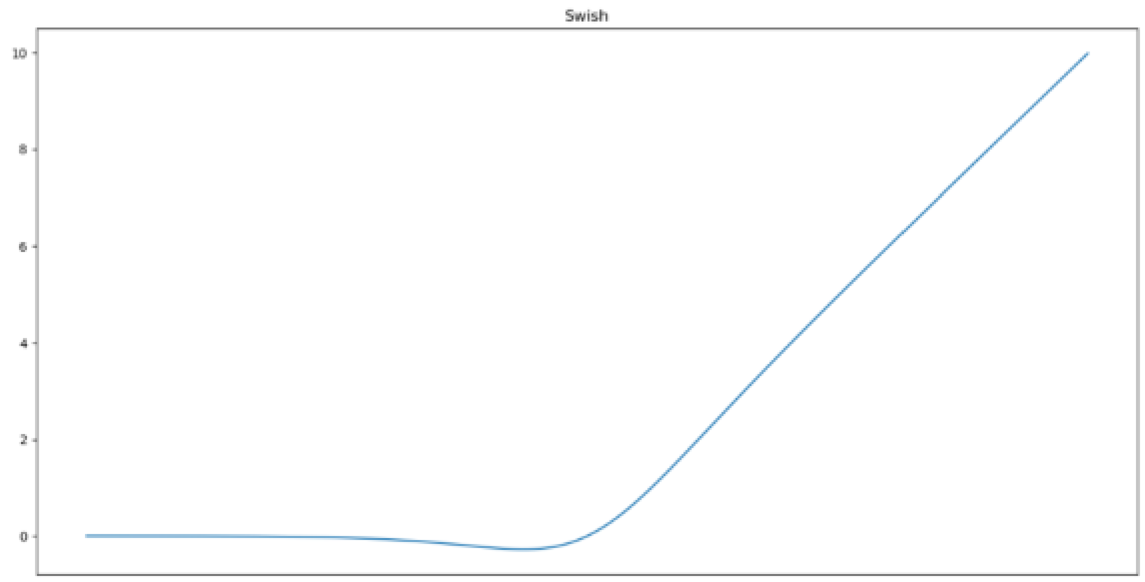
RELU — 6

This helps to stop blowing up the activation thereby stopping the gradients to explode(going to inf) as well another of the small issues that occur with normal ReLUs.

**The idea that comes into one's mind is why not combine ReLU6 and a LeakyReLU to solve all known issues that we have with previous activation functions. Popular DL frameworks do not provide an implementation of such an activation function but I think this would be a good idea.**

## Notable non-linear activations coming out of latest research

- **Swish**: This was proposed in 2017 by Ramachandran et.al. It is defined as **f(x) = x*sigmoid(x)**.

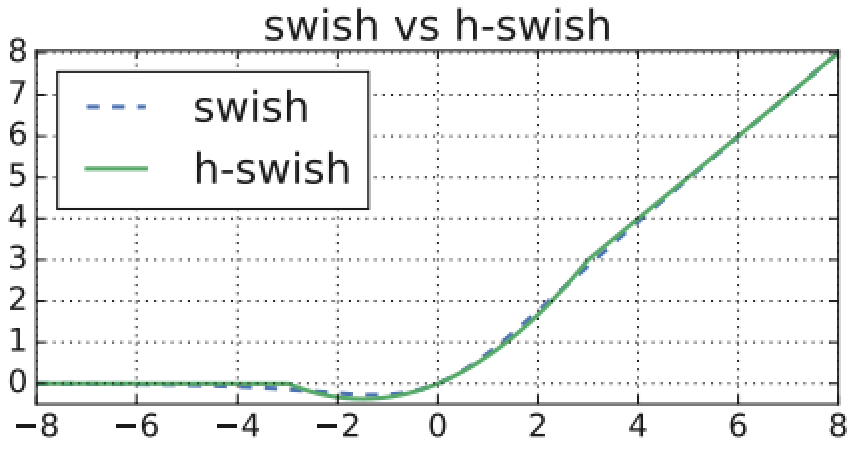


$$f(x) = x * sigmoid(x) \\ = x * (1 + e^{-x})^{-1}$$

It is slightly better in performance as compared to ReLU since its graph is quite similar to ReLU. However, because it does not change abruptly at a point as ReLU does at x = 0, this makes it easier to converge while training.

But, the drawback of Swish is that it is computationally expensive. To solve that we come to the next version of Swish.

- **Hard-Swish or H-Swish**: This is defined as:



$$\text{h-swish}[x] = x\frac{\text{ReLU6}(x+3)}{6}$$

The best part is that it is almost similar to swish but it is less expensive computationally since it replaces sigmoid (exponential function) with a ReLU (linear type).

## How to use them in deep neural networks?

- **Tanh and sigmoid cause huge vanishing gradient problems**. Hence, they should not be used.
- **Start with ReLU in your network**. Activation layer is added after the weight layer (something like CNN, RNN, LSTM or linear dense layer) as discussed above in the article. If you think the model has stopped learning, then you can replace it with a LeakyReLU to avoid the Dying ReLU problem. However, the Leaky ReLU will increase the computation time a little bit.
- **If you also have Batch-Norm layers in your network, that is added before the activation function making the order CNN-Batch Norm-*Act***. Although the order of Batch-Norm and Activation function is a topic of debate and some say that the order doesn't matter, I use the order mentioned above just to follow the original Batch-Norm paper.
- Activation functions work best in their default hyperparameters that are used in popular frameworks such as Tensorflow and Pytorch. However, one can fiddle with the negative slope in **LeakyReLU and set it to 0.02** to expedite learning.