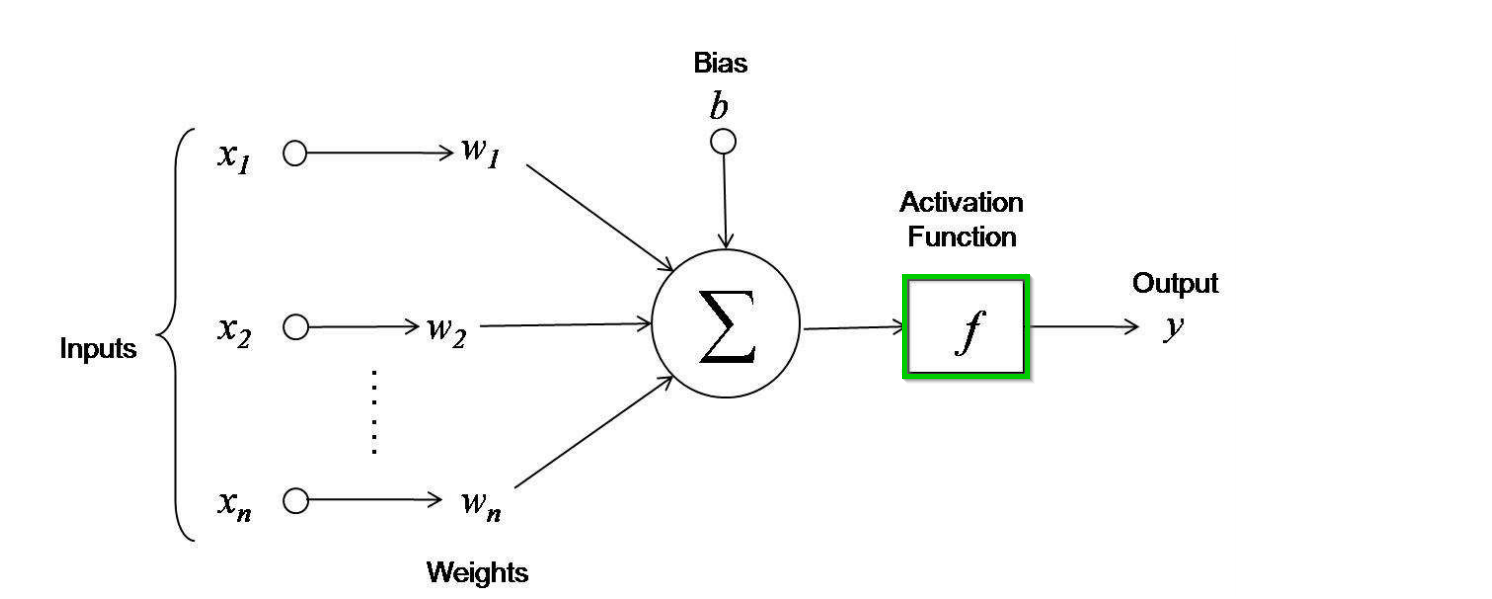
Evolution of Activation Functions – Part 1

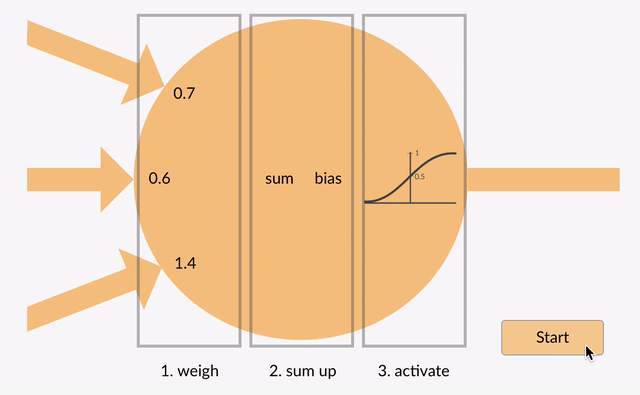
By now, we all are familiar with neural networks and its architecture (input layer, hidden layer, output layer) but one thing that I’m continuously asked is - ‘why do we need activation functions?’ or ‘what will happen if we pass the output to the next layer without an activation function’ or ‘Is nonlinearities really needed by the neural networks?’

To answer the above questions, let us take a step back and understand what happens inside a neuron:



Inside a neuron, each input gets multiplied with the weights *(x \* w)* and summed up *∑(x \* w),* then a bias is added *∑(x \* w) + b.*This output is passed to an activation function. Mathematically, **y = σ (*∑(x \* w) + b*)** where σ is any activation function.

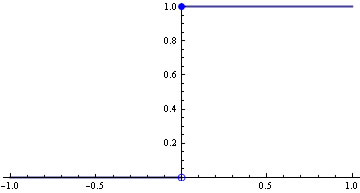
Visually it looks something like below-



The purpose of an activation function is to see the output of ***∑(x \* w) + b*** and decide if this neuron has to be activated and pass the output to the next layer or not.

To achieve this, one approach that can be used is threshold. As ***∑(x \* w) + b*** can vary from–inf to +inf, we can select a threshold value and based on the threshold, we can activate the neuron. For e.g. let’s suppose our threshold is 0, if the output ***∑(x \* w) + b*** is greater than 0, we can activate the neuron else not. This is what we call as a Step function:

**Step Function**

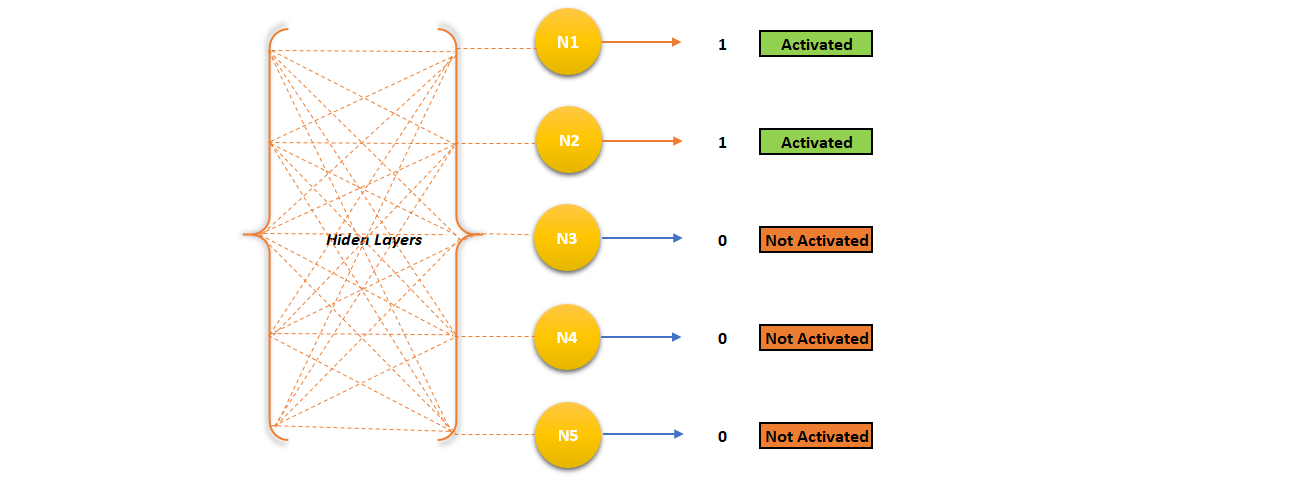


This step function yield two values (0 and 1) based on the threshold (0 in this case)

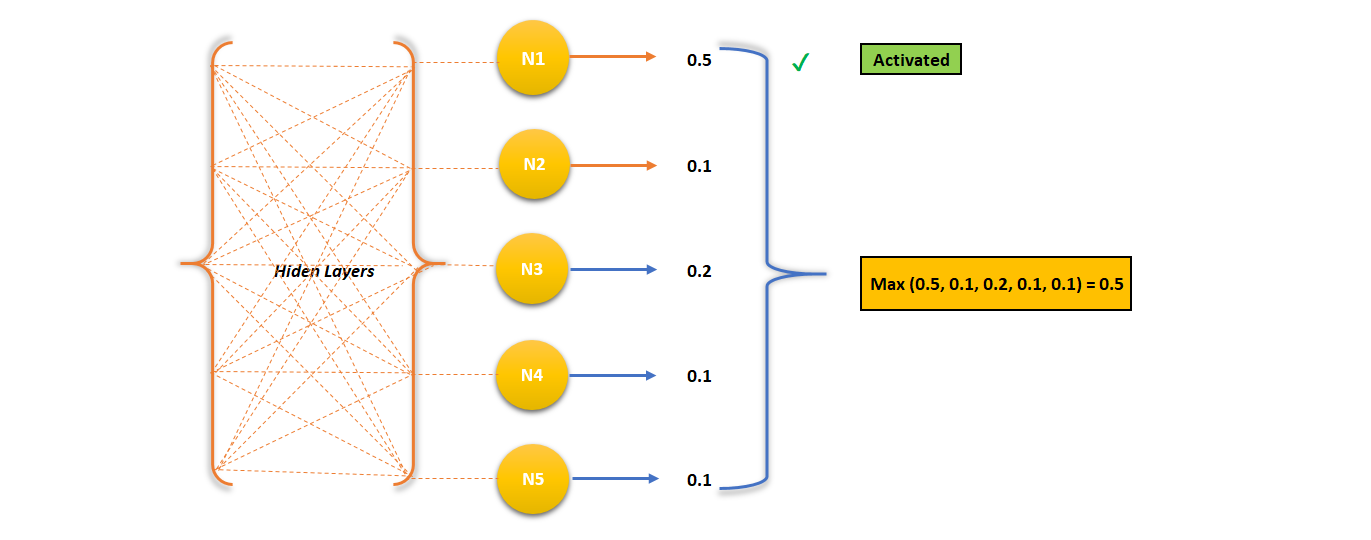
So, y = 1 if ***∑(x \* w) + b*** > 0 which means the neuron is activated.

And y=0 if ***∑(x \* w) + b*** < 0 which means the neurons is not activated.

This activation function does our job well in case of binary classification problem where we need binary outputs (say 0 or 1, yes or no). But this function has a downside, it fails miserably when it comes to multi-label classification.



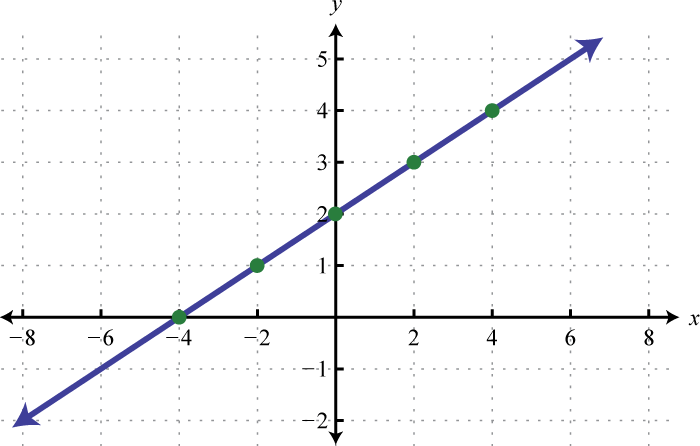
Let’s suppose we have a multi-label classification problem where we have to classify our input into 5 categories. We have stacked 5 neurons together for the classification. The ideal scenario in this case would be one out of 5 neurons (N1, N2, N3, N4, and N5) should be activated and rest of the neurons should be 0 – not activated. This way, we can classify the input instance into the activated neuron category. But, there is a high possibility that more than one neurons are activated as their function value ***∑(x \* w) + b*** is greater than 0 (our threshold). In such cases, how do we decide which neurons to consider for classification? Should we classify the input instance to N1 category or to the N2 category?



To solve this problem, we made the activation function continuous. So, instead of getting binary outputs like 0 and 1, we can get continuous outputs say 20% activated or 30% activated (activation rate or firing rate) and then we can take max or softmax of it to select the most activated neuron.

The simplest continuous function we can think of is linear function

2. **Linear Function**



We can choose a simple linear function as our activation function –

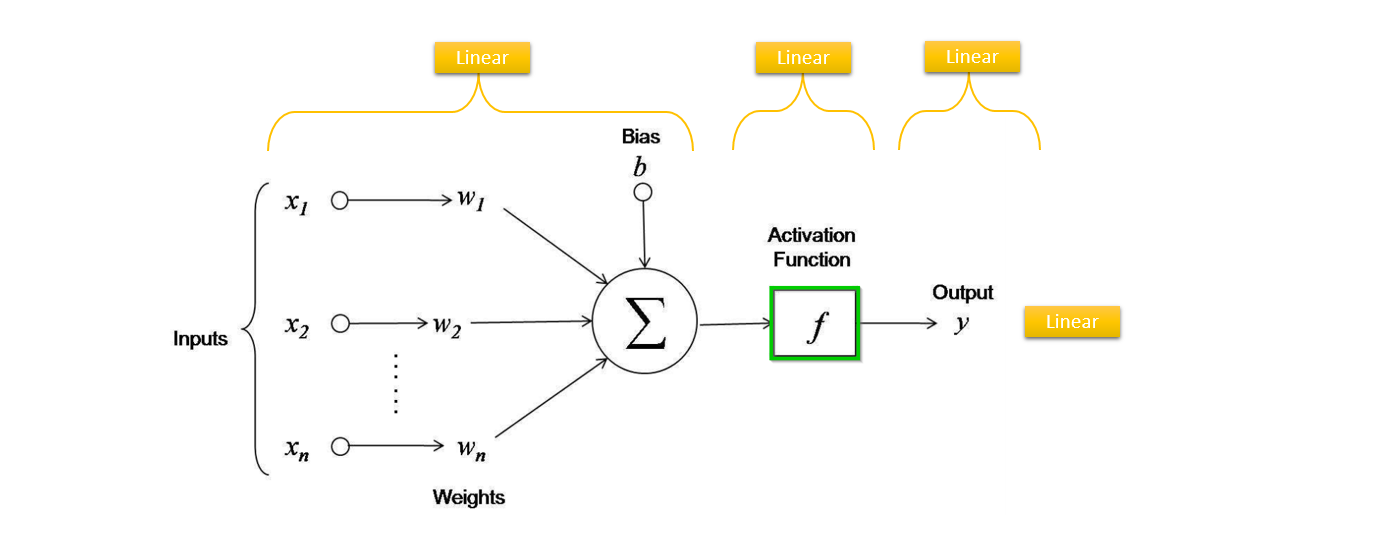
*y = c\*x*

Now, our activation function can output continuous values which will be directly proportional to the input. No more binary outputs like 0 or 1.

But there is a two major problem of using a linear function as an activation-

1. As we use gradient descent for training neural network, the gradient of this activation *c\*x* i.e. derivative of this activation, is constant (*c*).

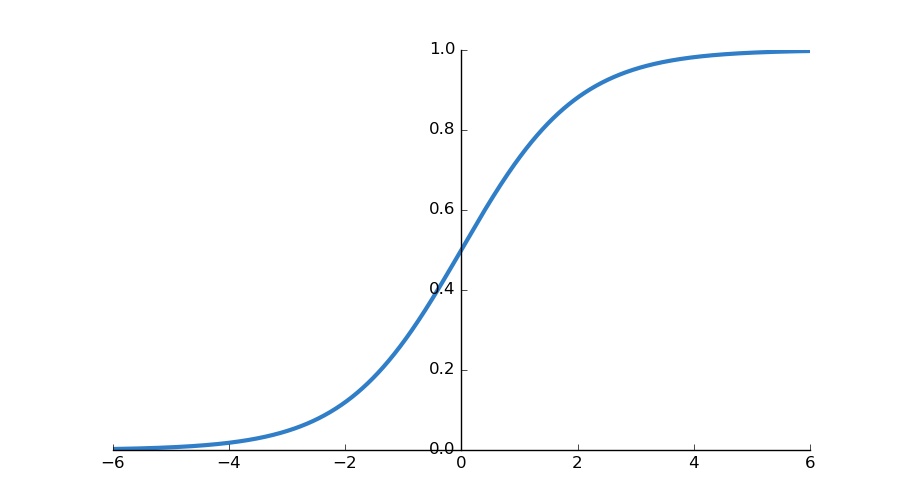
What does that mean? It means that the weight update during back propagation will be constant and independent of the input. So, if there is an error in prediction, the changes made by back propagation is constant and does not depend on the input (*delta x*). That’s not good. There is another problem.

1. If we are using a linear activation function, the whole process inside a neuron becomes linear. So, the output out of a neuron using a linear activation would also be linear. If this output passes to the next layer, again we multiply it with weights, add bias to it and fires based on another linear activation function. 

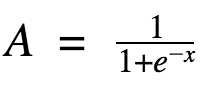
No matter how many layers we add, if all the intermediate layers are linear in nature, the final activation of last layer will also mimic a linear relationship with the input of the first layer. But, what’s the problem with linearity? The problem with linearity is we fail to learn the complex relationship in our data. As we know neural network are universal function approximator. Its job is to approximate/ build any complex function that maps input to the output. If the neural network architecture uses a linear activation function, the model will learn only simple (linear) relationship and we will fail to find complex relationship prevalent in our data. Thus, we need nonlinearities in the neural networks.

So, if we can’t use a linear activation function, let’s go with a non-linear activation - Sigmoid

3. **Sigmoid**



A sigmoid function transforms the input as follows.



The first thing to note in sigmoid is - nonlinear nature. So, the gradient of this function would be smooth and not fixed and it will induce non-linearity which will help us to find the complex (nonlinear) relationship, too, in the data.

A sigmoid function takes the input and squash it between 0 and 1 which is a good thing as an activation function because, unlike linear activation, it won’t blow up the activation when input is too high.

It solves the problem of fixed gradient and linearity, we had in the linear activation function, but it also has its downside.

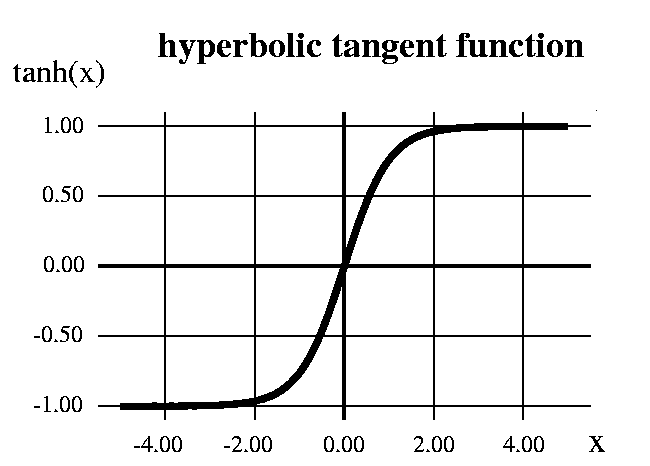
* A sigmoid has a steep curve between -2 to +2, which means even a small change in the value of input (x) in that region, will cause the value of y to change significantly. That means it tries to bring the y value to either side of the curve (above x=2 and below x=-2) which is good, from a binary classifier point of view. Making clear distinctions on prediction.

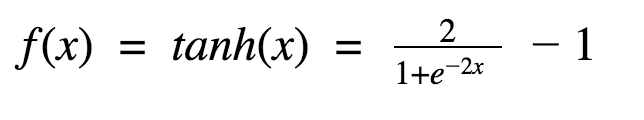
But, look at the region towards the end of the function (above x=4 and below x=-4). The function is almost flat. That means, the gradient in that region would be very small. What does that mean? It means that the y value tend to respond very less to changes in x. Once the gradient is too small or has vanished, the gradient update ceases and network fails to learn further. This problem is called “vanishing gradient”.

* A sigmoid activation is not zero centered – 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, these weights can only all decrease or all increase *together* for a given input pattern. As a result, the gradient updates go too far in different directions which makes optimization harder.

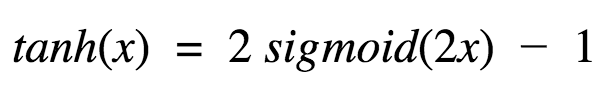
So, do we have an activation function which solves above problem for us? Yes, enters *tanh.*

4. **Tanh**





 A tanh function takes any real value as input and squash it between -1 to +1. This function looks very similar to a sigmoid function. Infact, it is a shifted sigmoid function.



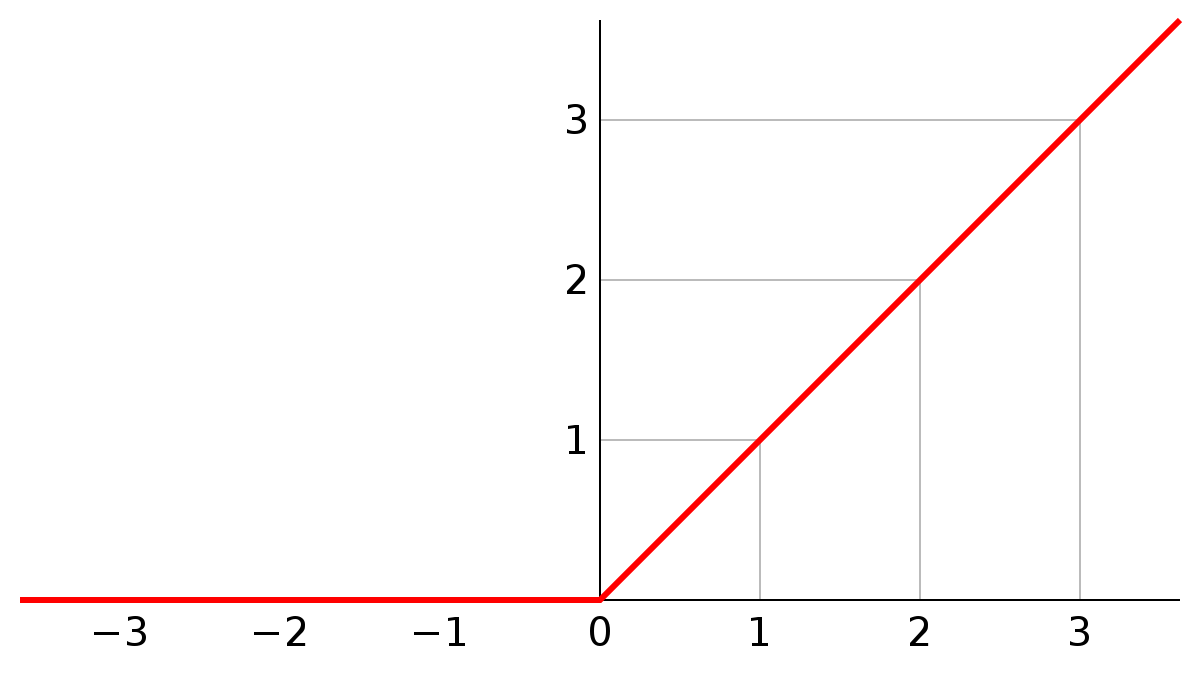
As it is similar to sigmoid function, so it retains the property of nonlinearity. It ouputs value between -1 to +1, so no problem of activation blowing up. However, its output is always zero-centered which helps since the neurons in the later layers of the network would be receiving inputs that are zero-centered. Hence, in practice, tanh activation functions are preferred in hidden layers over sigmoid.

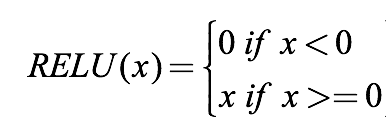
But still *tanh* suffers from vanishing gradient problem. Now, enters ReLU

5. **ReLU**

ReLU – Rectified Linear Unit is presented as

*y = max(0, x)*





If you first look at ReLU, it appears to be a linear in nature. But it is a nonlinear function which thresholds at 0 i.e. all the negative inputs are set to 0. ReLU has an upper hand over tanh or sigmoid in many ways:

1. It’s less computationally expensive. It is an important thing to consider while building deep neural networks.
2. Sparsity – *Sigmoid* and *tanh* activation function would almost always gets fired in a neural network. That means, all the activations are getting computed and processed in calculating the final output of the network. It’s a good thing if we consider all the activation before the final output but only when our architecture is small. When we build a very deep neural network, ideally we want only a section of neurons to fire and contribute to final output of the network. The ReLU does just that. It sets considerable amount of activation to 0 because of its nature of the curve. This sparse activation leads to fewer neurons to fire making the network lighter and computation faster.

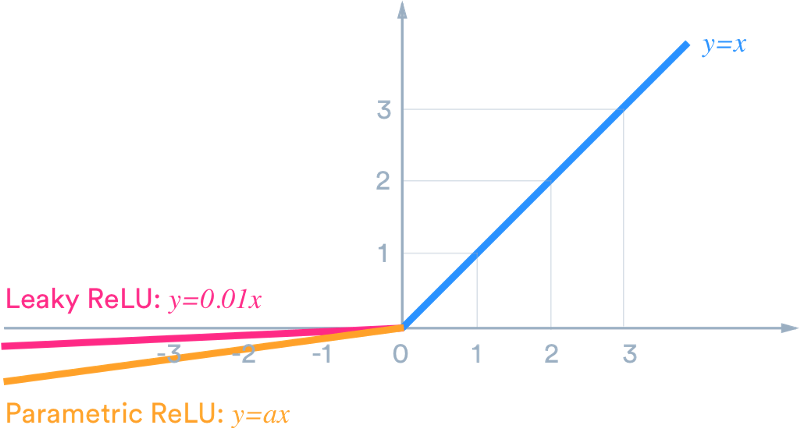
But not even ReLU comes without the downside –

Range of ReLU is (0,+inf). This can blow up the activation for high positive values.

Moreover, if you observe, it has a horizontal line for the negative inputs. That means gradient (derivative of that region) for all the –ve input will be set to 0 which means no weight update during backpropagation – making major part of the network passive. This is called ***‘Dying ReLU’*** problem.

To mitigate this problem, we have ‘Leaky ReLU’

6. **Leaky ReLU / Parametric ReLU**



Leaky ReLU solves the Dying ReLU problem by having a small negative slope. Instead of thresholding at 0. i.e. setting the negative input to 0, we assign the negative input a value which is a linear combination of the input (*0.01x*). By assigning a small value to the negative input, unlike ReLU, the gradient of that region becomes non-zero, thereby, making the network active and taking part in learning process.

This idea can be extended further by making a small change. Instead of assigning the coefficient of the negative part, a fixed value say (0.01), we can learn the multiplier (α) and treat it as an additional hyperparameter in our process. This is known as Parametric ReLU. In practice, it is believed that this performs better than Leaky ReLU.

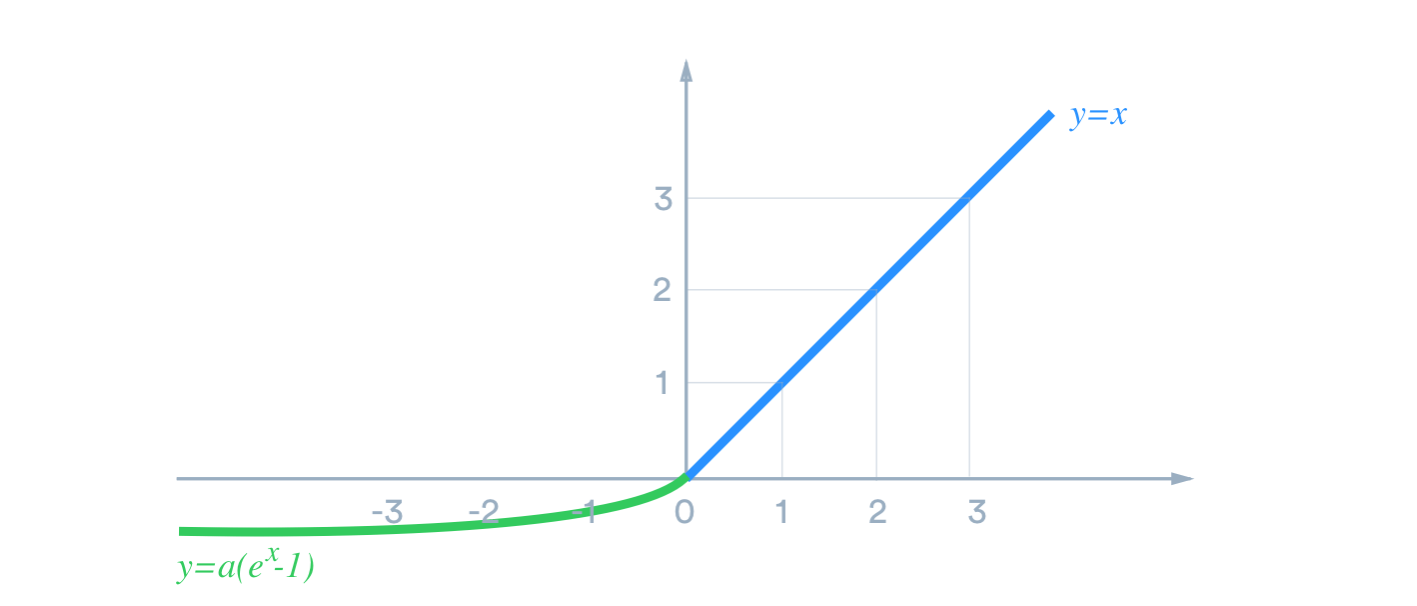
So, Leaky/Parameterized ReLU solves two problems-

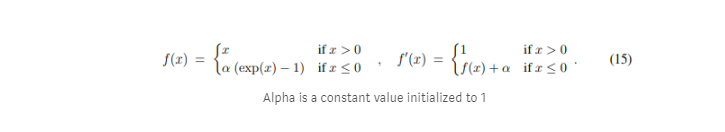
1. It fixes the “dying ReLU” problem, as it doesn’t have zero-slope parts.
2. The problem with ReLU is that they have non-negative activation hence their mean activation is greater than zero. A positive mean introduces a bias for the next layer which can slow down the learning. If the mean value of activation is 0 we get a faster learning. Since Leaky ReLU can have negative values it pushes the mean of the activations closer to zero. Having mean activations closer to zero also causes the faster learning and convergence.

Wait! There is one more entry to ReLU’s family – ELU – Exponential Linear Unit.

7. **ELU**

Similar to leaky ReLU, ELU has a small slope for negative values. Instead of a straight line, it uses a log curve like the following:





*F(x) is the forward pass and its derivative is F`(x) for calculating its backward gradients.*

The only thing that you need to know is that the derivative of the exponent function is the exponent itself.

In plain English, it acts like a ReLU unit if x is positive, but for negative values it is a nonlinear function bounded by a fixed value **-1**, for **α=1**. This behavior helps to push the mean activation of neurons closer to zero and preserve nonlinearity which is beneficial for learning and it helps to learn representations that are more robust to noise and also

It is designed to combine the good parts of ReLU and leaky ReLU — while it doesn’t have the dying ReLU problem, preserves nonlinearity, it saturates for large negative values, allowing them to be essentially inactive and robust to noise.

*Which one to use, in practice:*

* Use *ReLU*. Be careful with your learning rates
* Try out *Leaky ReLU / ELU*
* Try out *tanh* but don’t expect much.
* Don’t use *sigmoid*.

*I will cover the remaining activation functions in part-2 of this series*