Table of Content

When to use which Activation Function in a Neural Network?

Specifically, it depends on the problem type and the value range of the expected output. For example, to predict values that are larger than 1, tanh or sigmoid are not suitable to be used in the output layer, instead, ReLU can be used.

On the other hand, if the output values have to be in the range (0,1) or (-1, 1) then ReLU is not a good choice, and sigmoid or tanh can be used here. While performing a classification task and using the neural network to predict a probability distribution over the mutually exclusive class labels, the softmax activation function should be used in the last layer. However, regarding the hidden layers, as a rule of thumb, use ReLU as an activation for these layers.

In the case of a binary classifier, the Sigmoid activation function should be used. The sigmoid activation function and the tanh activation function work terribly for the hidden layer. For hidden layers, ReLU or its better version leaky ReLU should be used. For a multiclass classifier, Softmax is the best-used activation function. Though there are more activation functions known, these are known to be the most used activation functions.

What is an Activation Function?

An activation function is a very important feature of an artificial neural network that is used to determine what to do with the neurons, i.e., whether they will be activated or not. In artificial neural networks, the activation function defines the output of that node given an input set or just a single input.

One of the reasons for evincing interest in us is the hope to understand our mind, which emerges from neural processing in our brain. Advances in machine learning have been achieved in recent years by combining massive data sets and deep learning techniques.is yet another reason.

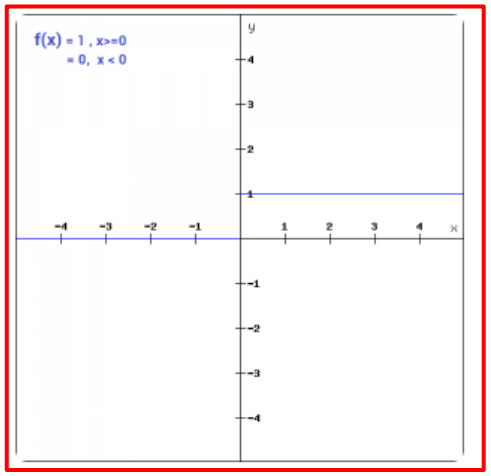
Types of Activation Functions

- Linear Activation Function
- Non-Linear Activation Function
  - Sigmoid Activation Function
  - Tanh Activation Function
  - ReLU Activation Function
    - Leaky ReLU
    - Parametric ReLU

Popular types of activation functions and when to use them

1. Binary Step Function
The first thing that comes to our mind when we have an activation function would be a threshold-based classifier i.e. whether or not the neuron should be activated based on the value from the linear transformation.
In other words, if the input to the activation function is greater than a threshold, then the neuron is activated, else it is deactivated, i.e. its output is not considered for the next hidden layer. Let us look at it mathematically-

f(x) = 1, x>=0 = 0, x<0



This is the simplest activation function, which can be implemented with a single if-else condition in python

```
def binary_step(x):
 if x<0: return 0
else: return 1
binary_step(5), binary_step(-1)
```

Output:

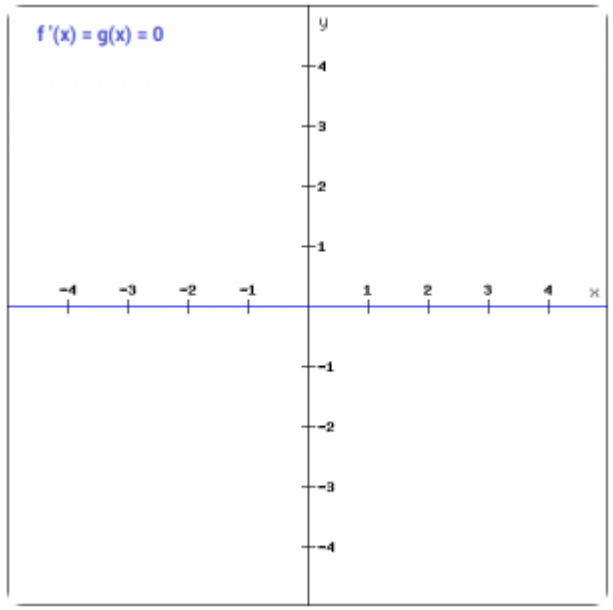
(1,0)
The binary step function can be used as an activation function while creating a binary classifier. As you can imagine, this function will not be useful when there are multiple classes in the target variable. That is one of the limitations of binary step function.

Moreover, the gradient of the step function is zero which causes a hindrance in the back propagation process. That is, if you calculate the derivative of f(x) with respect to x, it comes out to be 0.

f'(x) = 0, for all x

Moreover, the gradient of the step function is zero which causes a hindrance in the back propagation process. That is, if you calculate the derivative of f(x) with respect to x, it comes out to be 0.

f'(x) = 0, for all x

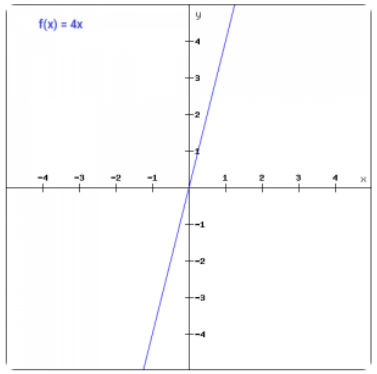


Gradients are calculated to update the weights and biases during the backprop process. Since the gradient of the function is zero, the weights and biases don’t update.

2. Linear Function

We saw the problem with the step function, the gradient of the function became zero. This is because there is no component of x in the binary step function. Instead of a binary function, we can use a linear function. We can define the function as-

f(x)=ax



Here the activation is proportional to the input.The variable 'a' in this case can be any constant value. Let's quickly define the function in python:

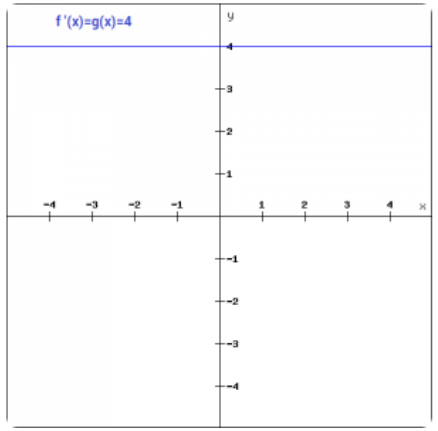
```
def linear_function(x):
    return 4*x
linear_function(4), linear_function(-2)
```

Output:

(16, -8)

What do you think will be the derivative is this case? When we differentiate the function with respect to x, the result is the coefficient of x, which is a constant.

f'(x) = a

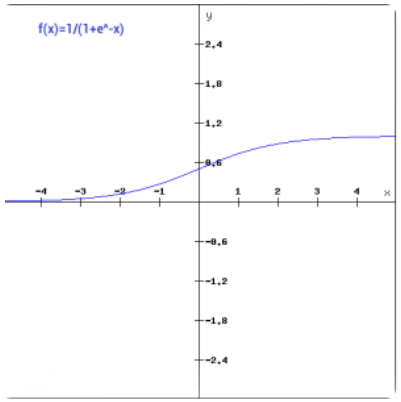


Although the gradient here does not become zero, but it is a constant which does not depend upon the input value x at all. This implies that the weights and biases will be updated during the backpropagation process but the updating factor would be the same. In this scenario, the neural network will not really improve the error since the gradient is the same for every iteration. The network will not be able to train well and capture the complex patterns from the data. Hence, linear function might be ideal for simple tasks where interpretability is highly desired.

3. Sigmoid

The next activation function that we are going to look at is the Sigmoid function. It is one of the most widely used non-linear activation function. Sigmoid transforms the values between the range 0 and 1. Here is the mathematical expression for sigmoid-

f(x) = 1/(1+e^-x)



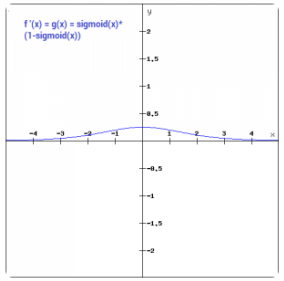
A noteworthy point here is that unlike the binary step and linear functions, sigmoid is a non-linear function.
This essentially means -when I have multiple neurons having sigmoid function as their activation function,the output
is non linear as well. Here is the python code for defining the function in python-

```
import numpy as np
def sigmoid_function(x):
   z = (1/(1 + np.exp(-x)))
   return z
sigmoid_function(7),sigmoid_function(-22)
```

Output:

Output
(0.9990889488055994, 2.7894680920908113e-10)
Additionally, as you can see in the graph above, this is a smooth S-shaped function and is continuously differentiable. The derivative of this function comes out to be (sigmoid(x)*(1-sigmoid(x)). Let's look at the plot of it's gradient.

f'(x) = sigmoid(x)*(1-sigmoid(x))

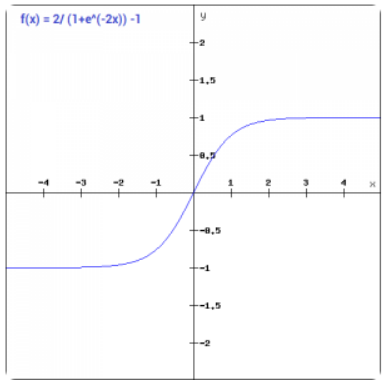


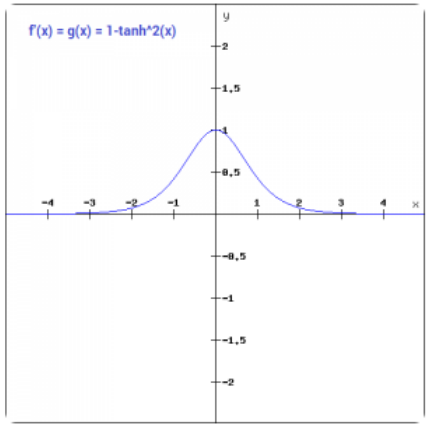
The gradient values are significant for range -3 and 3 but the graph gets much flatter in other regions. This implies that for values greater than 3 or less than -3, will have very small gradients. As the gradient value approaches zero, the network is not really learning.
Additionally, the sigmoid function is not symmetric around zero. So output of all the neurons will be of the same sign. This can be addressed by scaling the sigmoid function which is exactly what happens in the tanh function. Let's read on.

4. Tanh

The tanh function is very similar to the sigmoid function. The only difference is that it is symmetric around the origin. The range of values in this case is from -1 to 1. Thus the inputs to the next layers will not always be of the same sign. The tanh function is defined as-

tanh(x)=2sigmoid(2x)-1





The gradient of the tanh function is steeper as compared to the sigmoid function. You might be wondering, how will we decide which activation function to choose? Usually tanh is preferred over the sigmoid function since it is zero centered and the gradients are not restricted to move in a certain direction.

In order to code this is python, let us simplify the previous expression.

tanh(x) = 2sigmoid(2x)-1
tanh(x) = 2/(1+e^(-2x)) -1
And here is the python code for the same:

```
def tanh_function(x):
    z = (2/(1 + np.exp(-2*x))) -1
    return z
tanh_function(0.5), tanh_function(-1)
```

Output:
(0.4621171572600098, -0.7615941559557646)
As you can see, the range of values is between -1 to 1. Apart from that, all other properties of tanh function are the same as that of the sigmoid function.
Similar to sigmoid, the tanh function is continuous and differentiable at all points.

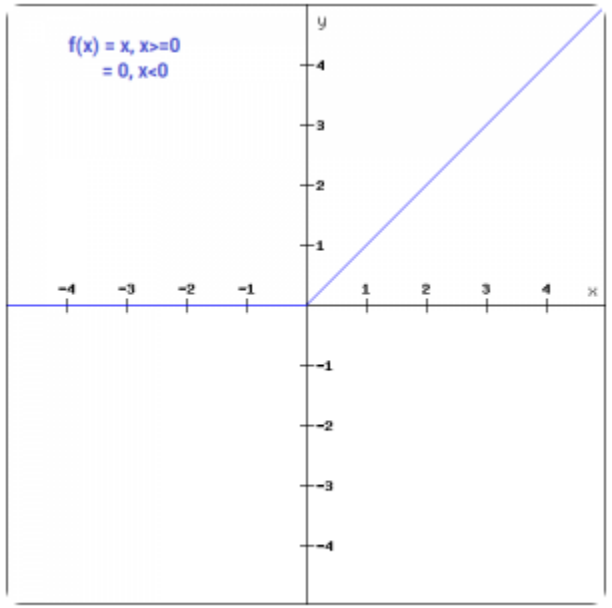
Let's have a look at the gradient of the tan h function.

5. ReLU

The ReLU function is another non-linear activation function that has gained popularity in the deep learning domain. ReLU stands for Rectified Linear Unit. The main advantage of using the ReLU function over other activation functions is that it does not activate all the neurons at the same time.

This means that the neurons will only be deactivated if the output of the linear transformation is less than 0. The plot below will help you understand this better-

f(x)=max(0,x)



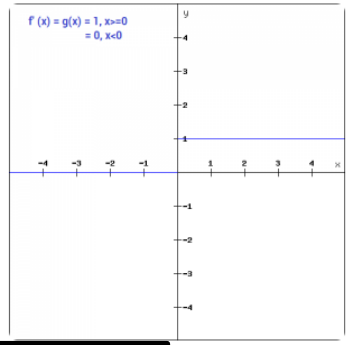
For the negative input values, the result is zero, that means the neuron does not get activated.
Since only a certain number of neurons are activated, the ReLU function is far more computationally
efficient when compared to the sigmoid and tanh function. Here is the python function for ReLU:

```
def relu_function(x):
    if x<0:
        return 0
    else:
        return x
relu_function(7), relu_function(-7)
```

Output:

(7, 0)
Let's look at the gradient of the ReLU function.

f'(x) = 1, x>=0
= 0, x<0

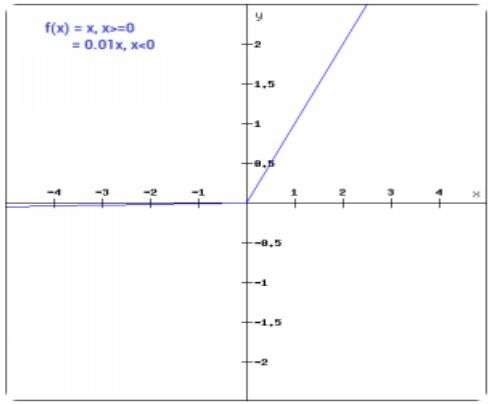


If you look at the negative side of the graph, you will notice that the gradient value is zero. Due to this reason, during the backpropogation process, the weights and biases for some neurons are not updated. This can create dead neurons which never get activated. This is taken care of by the 'Leaky' ReLU function.

6. Leaky ReLU

Leaky ReLU function is nothing but an improved version of the ReLU function. As we saw that for the ReLU function, the gradient is 0 for x<0, which would deactivate the neurons in that region. Leaky ReLU is defined to address this problem. Instead of defining the Relu function as 0 for negative values of x, we define it as an extremely small linear component of x. Here is the mathematical expression-

f(x)= 0.01x, x<0
= x, x>=0



Since Leaky ReLU is a variant of ReLU, the python code can be implemented with a small modification-

```
def leaky_relu_function(x):
    if x<0:
        return 0.01*x
    else:
        return x
leaky_relu_function(7), leaky_relu_function(-7)
```

Output:

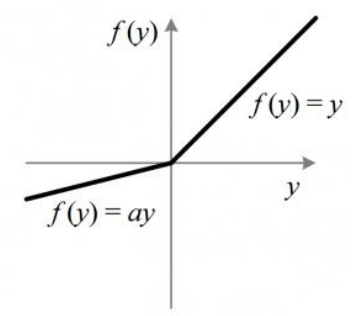
(7, -0.07)

Apart from Leaky ReLU, there are a few other variants of ReLU, the two most popular are – Parameterised ReLU function and Exponential ReLU.

7. Parameterised ReLU

This is another variant of ReLU that aims to solve the problem of gradient's becoming zero for the left half of the axis. The parameterised ReLU, as the name suggests, introduces a new parameter as a slope of the negative part of the function. Here's how the ReLU function is modified to incorporate the slope parameter-

f(x) = x, x>=0
= ax, x<0



When the value of a is fixed to 0.01, the function acts as a Leaky ReLU function. However, in case of a parameterised ReLU function, 'a' is also a trainable parameter. The network also learns the value of 'a' for faster and more optimum convergence.

The derivative of the function would be same as the Leaky ReLu function, except the value 0.01 will be replcaed with the value of a.

An activation function in neural networks determines the output of a neuron given its input. It introduces non-linearity, enabling the network to learn complex relationships. Common types include:

1. Sigmoid: S-shaped curve, suitable for binary classification but can suffer from vanishing gradient.
2. Tanh (Hyperbolic Tangent): Similar to sigmoid but centered around zero, still susceptible to vanishing gradient.
3. ReLU (Rectified Linear Activation): Replaces negative inputs with zero, addressing vanishing gradient and promoting faster training.
4. Leaky ReLU: Allows small negative values, avoiding the “dying ReLU” problem.
5. PReLU (Parametric ReLU): Generalizes Leaky ReLU by making the slope learnable.
6. ELU (Exponential Linear Unit): Smoothly handles negative inputs, preventing dead neurons and improving convergence.
7. Softmax: Converts a vector of scores into a probability distribution, often used in multi-class classification.