20MAI0077 - Vivek Dadhich

Github link - https://github.com/vivek20dadhich/CSE6037-Deep-Learning-and-its-Applications-Lab-Assignments (https://github.com/vivek20dadhich/CSE6037-Deep-Learning-and-its-Applications-Lab-Assignments)

Activation functions implementation in python

In [1]:

```
1 import matplotlib.pyplot as plt
2 import numpy as np
```

Sigmoid function

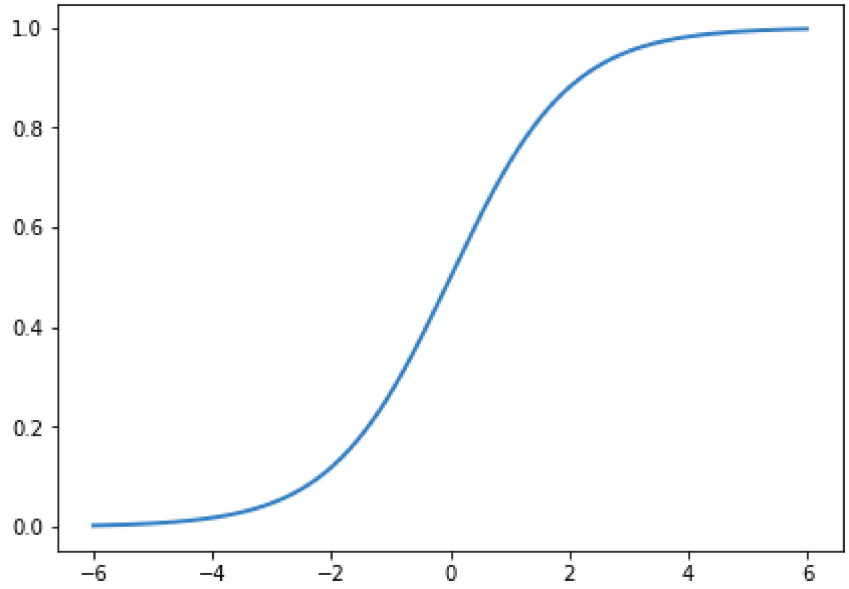
f(x)=1/(1+exp(-x) the function range between (0,1)

In [2]:

```
def sigmoid(x):
    s=1/(1+np.exp(-x))
    ds=s*(1-s)
    return s,ds
x=np.arange(-6,6,0.01)
sigmoid(x)

# Create and show plot
fig, ax = plt.subplots(figsize=(7, 5))
ax.plot(x,sigmoid(x)[0], color="#307EC7", linewidth=2)
fig.show()
```

```
<ipython-input-2-ac80167ca9cd>:11: UserWarning: Matplotlib is currently usin
g module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so ca
nnot show the figure.
  fig.show()
```



tanh or hyperbolic

a=(e^z-e^(-z))/(e^z+e^(-z)

In [3]:

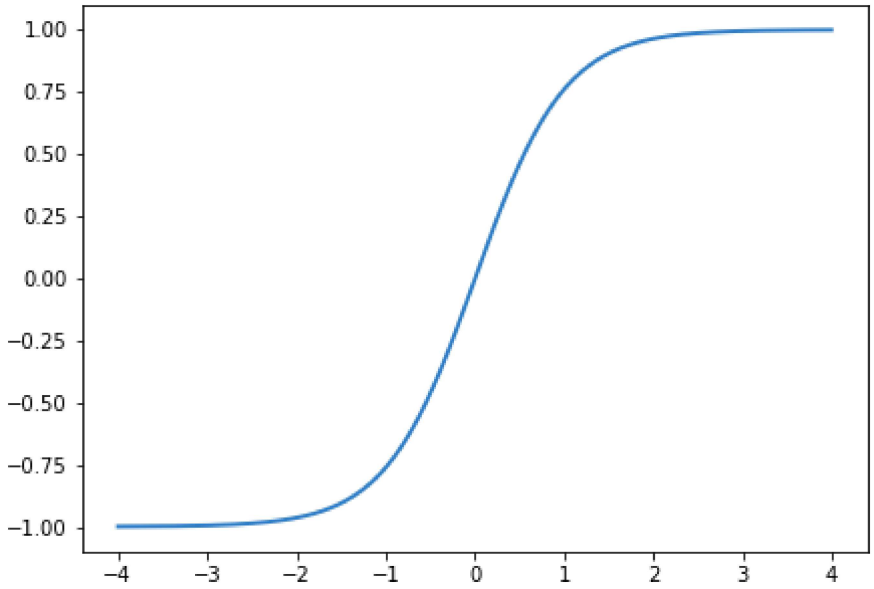
```
#Observations:
#It’s output is zero centered because its range in between -1 to 1 i.e -1 < output < 1
#Hence optimization is easier in this method hence in practice it is always preferred o
#But still it suffers from Vanishing gradient problem.
```

In [4]:

```
def tanh(x):
    t=(np.exp(x)-np.exp(-x))/(np.exp(x)+np.exp(-x))
    dt=1-t**2
    return t,dt
z=np.arange(-4,4,0.01)
tanh(z)[0].size,tanh(z)[1].size

fig_tanh, ax = plt.subplots(figsize=(7, 5))
# Create and show plot
ax.plot(z,tanh(z)[0], color="#307EC7", linewidth=2)
fig_tanh.show()
```

```
<ipython-input-4-e598b036cfe9>:11: UserWarning: Matplotlib is currently usin
g module://ipykernel.pylab.backend_inline, which is a non-GUI backend, so ca
nnot show the figure.
  fig_tanh.show()
```



ReLu

A(x) = max(0,x). It gives an output x if x is positive and 0 otherwise

In [5]:

```
#ReLu is less computationally expensive than tanh and sigmoid because it involves simpl
#At a time only a few neurons are activated making the network sparse making it efficie
#It avoids and rectifies vanishing gradient problem . Almost all deep learning Models u
```

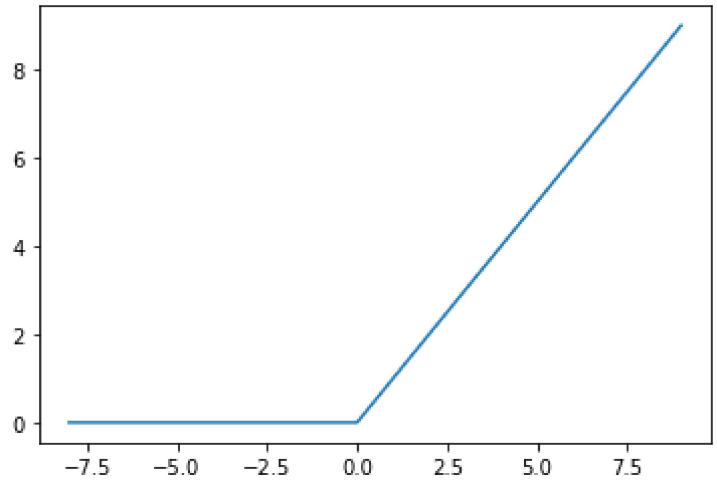
In [6]:

```
# rectified linear function
def relu(x):
    return max(0.0, x)

# define a series of inputs
series_in = [x for x in range(-8, 10)]

# calculate outputs for our inputs
series_out = [relu(x) for x in series_in]

# line plot of raw inputs to rectified outputs
plt.plot(series_in, series_out)
plt.show()
```



Leaky ReLu

The Leaky ReLu function is an improvisation of the regular ReLu function. To address the problem of zero gradient for negative value, Leaky ReLu gives an extremely small linear component of x to negative inputs.

Mathematically we can express Leaky ReLu as:

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

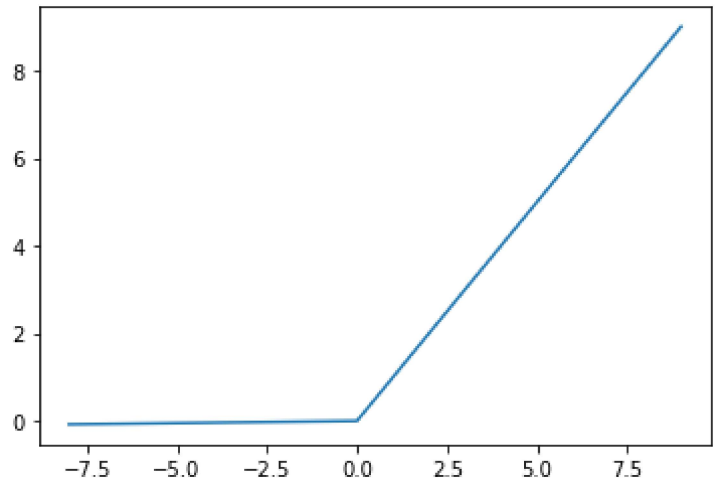
In [7]:

```
def leaky_relu(x):
    if x>0:
        return x
    else:
        return 0.01*x

# define a series of inputs
series_in_leaky = [x for x in range(-8, 10)]

# calculate outputs for our inputs
series_out_leaky = [leaky_relu(x) for x in series_in_leaky]

# line plot of raw inputs to rectified outputs
plt.plot(series_in_leaky, series_out_leaky)
plt.show()
```



Softmax

This function is great for classification problems, especially if you're dealing with multi-class classification problems, as it will report back the "confidence score" for each class. Since we're dealing with probabilities here, the scores returned by the softmax function will add up to 1.

In [8]:

```
#declare an array which imitates the output layer of a neural network:
output_layer = np.array([1.3, 5.1, 2.2, 0.7, 1.1])

#exponentiate each of the elements of the output layer:
exponentiated = np.exp(output_layer)

#divide each element by exponentiated sum and store results in another array:
probabilities = exponentiated / np.sum(exponentiated)
probabilities #these are the target class probabilities obtained from the output layer
```

Out[8]:

```
array([0.02019046, 0.90253769, 0.04966053, 0.01108076, 0.01653055])
```

In [9]:

```
#the sum of the probabilities should equal to 1
probabilities.sum()
```

Out[9]:

```
1.0
```

In [ ]:

```

```