Assignment - 3

Student Name: Vivek Kumar
UID: 21BCS8129
Branch: BE-CSE (LEET)
Section/Group: 809/A
Semester: 4th
Date of Performance: 04/05/2022
Subject Name: Principles of AI
Subject Code: 20CST-258

1. Aim/Overview of the practical:

Define Activation function?

2. Theories:

The activation function defines the output of a neuron / node given an input or set of input (output of multiple neurons). It's the mimic of the stimulation of a biological neuron.

The output of the activation function to the next layer (in shallow neural network: input layer and output layer, and in deep network to the next hidden layer) is called forward propagation (information propagation). It's considered as a non-linearity transformation of a neural network.

There is a list of activation functions commonly used:

- **Binary**
- **Linear**
- **Sigmoid**
- **Tanh**
- **ReLU**
- **Leaky ReLU (LReLU)**
- **Parametric ReLU (PReLU)**
- **Exponential Linear Unit (eLU)**
- **ReLU-6**
- **Softplus**
- **Softsign**
- **Softmax**
- **Swish**

Binary

The binary activation function is the simpliest. It's based on binary classifier, the output is 0 if values are negatives else 1. See this activation function as a threshold in binary classification.

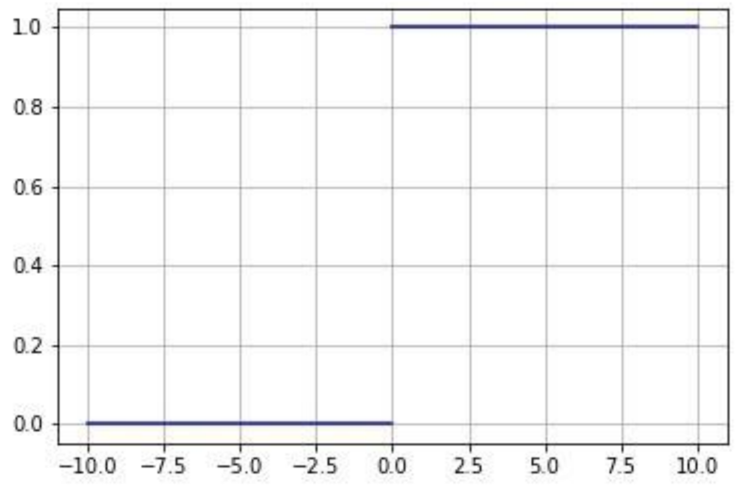
The code for a binary activation function is:

```
def binary_active_function(x):
    return 0 if x < 0 else 1
```

What is the output of this function ?

```
for i in [-5, -3, -1, 0, 2, 5]:
    print(binary_active_function(i))output:
    0
    0
    0
    1
    1
    1
```

Or visualy:



Binary activation function

Linear activation function

The next step after the binary function is to use a linear function instead of a step. The output is proportional to the input.

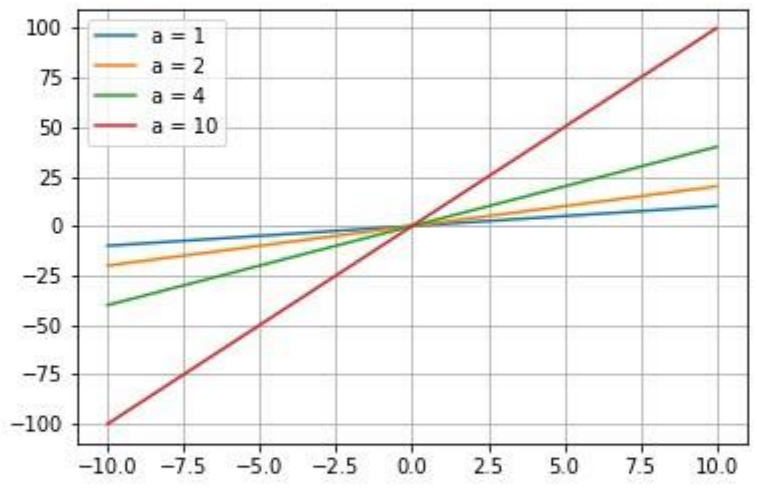
The corresponding code is:

```
def linear_active_function(a, x):
    return a*x
```

We can compute it for different values of "*a*":

```
$ x = numpy.linspace(-10, 10, 5000)
$ y_1 = [linear_active_function(1, i) for i in x] # a = 1
$ y_2 = [linear_active_function(2, i) for i in x] # a = 2
$ y_1
> [-10.0, -9.9, -9.8, -9.7, ..., 9.7, 9.8, 9.9, 10.0]
```

If we plot the results for a = 1, 2, 4 and 10:



Sigmoid:
Sigmoid is the most used activation function with *ReLU* and *tanh*. It's a non-linear activation function also called *logistic function*. The output of this activation function vary between 0 and 1. All the output of neurons will be positive.

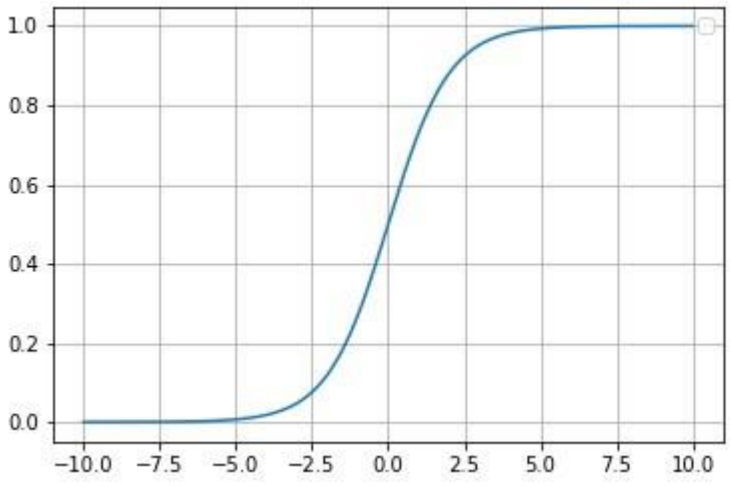
The corresponding code is as follow:

```
def sigmoid_active_function(x):
    return 1./(1+numpy.exp(-x))
```

A quick computation:

```
$ x = numpy.linspace(-10, 10, 5000)
$ y = [sigmoid_active_function(i) for i in x]
$ y
> [4.5397868702434395e-05, 4.5854103946941324e-05, ... , 0.9999532196250409,
0.9999536850759906, 0.9999541458960531]
```

If we plot the results:



Sigmoid activation function

Tanh:

The tangent hyperbolic function (tanh) is similar to the sigmoïd function in the way that their form are similar. Tanh is symmetric in 0 and the values are in the range -1 and 1. As the sigmoid they are very sensitive in the central point (0, 0) but they saturate for very large number (positive and negative). This symmetry make them better than the sigmoid function.

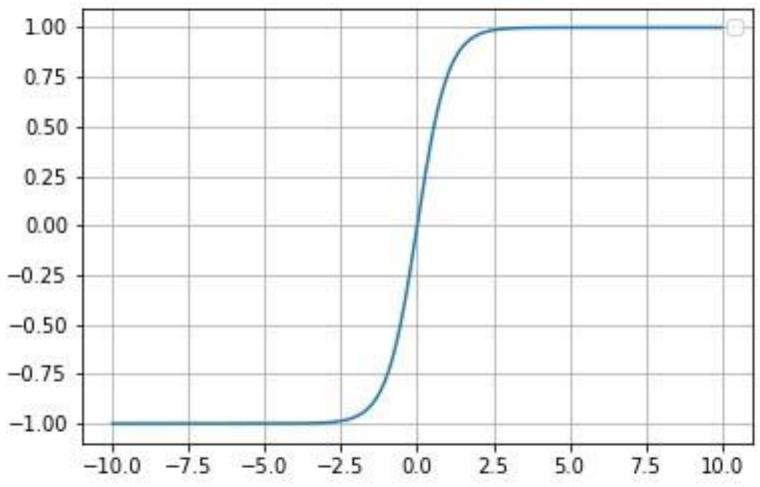
The corresponding code to apply a tanh function is:

```
def tanh_active_function(x):
    return 2*sigmoid_active_function(2*x)-1
```

Compute the *y* values:

```
$ x = numpy.linspace(-10, 10, 5000)
$ y = [tanh_active_function(i) for i in x]
$ y
> [-0.9999999958776927, -0.9999999957944167, ... , 0.9999999956227836,
0.9999999957094583, 0.9999999957944166]
```

And the corresponding result:



tanh activation function

ReLU

The *REctified Linear Unit* was develop to avoid the saturation with big positive numbers. The non-linearity permit to conserve and learn the patterns inside the data and the linear part (>0 — also called piecewise linear function) make them easily interpretable.

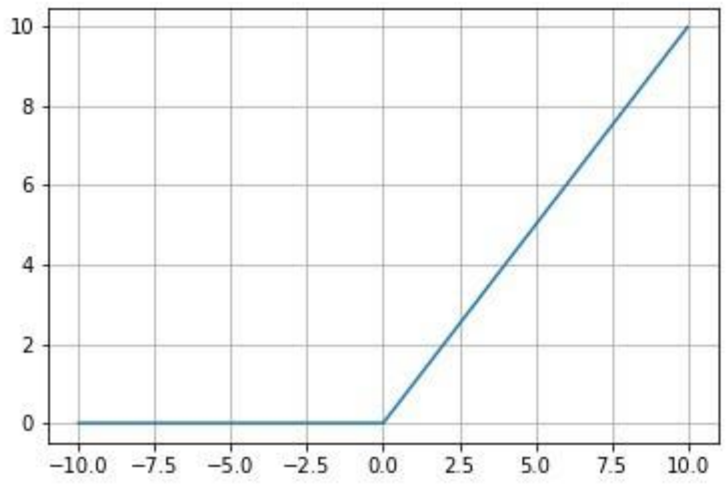
The function below shows how to implement the ReLU function:

```
def relu_active_function(x):
    return numpy.array([0, x]).max()
```

y computation:

```
$ x = numpy.linspace(-10, 10, 5000)
$ y = [relu_active_function(i) for i in x]
$ y
> [0.0, 0.0, ... , 9.97, 9.98, 9.99]
```

The results:



ReLU activation function

Leaky ReLU:
This activation function is a modification of the ReLU activation function to avoid the "dying problem". The function return a linear slope where a=0.01 which permit to keep neurons activated with a gradient flow.

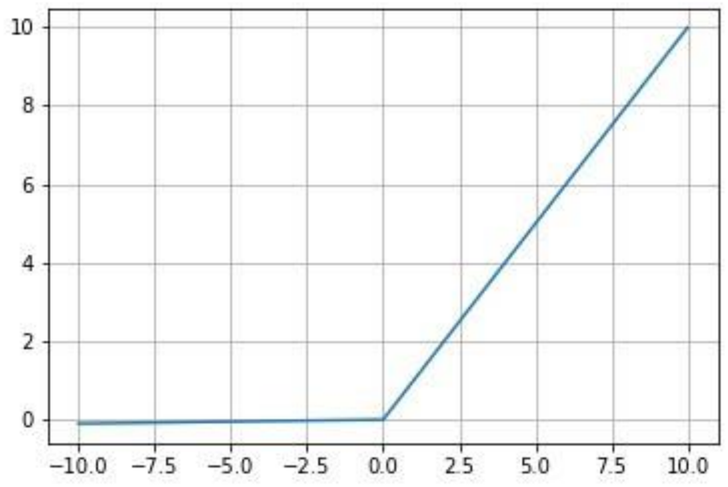
See the code below:

```
def leaky_relu_active_function(x):
    return 0.01*x if x < 0 else x
```

Compute the y axis to plot the results:

```
$ x = numpy.linspace(-10, 10, 5000)
$ y = [leaky_relu_active_function(i) for i in x]
$ y
> [-0.1, -0.0999, ... , 9.97, 9.98, 9.99]
```

Plot the results:



Leaky ReLU activation function

Parametric ReLU:
After the Leaky ReLU there is another activation function created to avoid the “dying ReLU problem”, the parametric or parametrised ReLU. The coefficient a is not lock at 0.01 (Leaky ReLU) but it free to estimate. It’s a generalization of the ReLU, the algorithm learn the rectifier parameter.

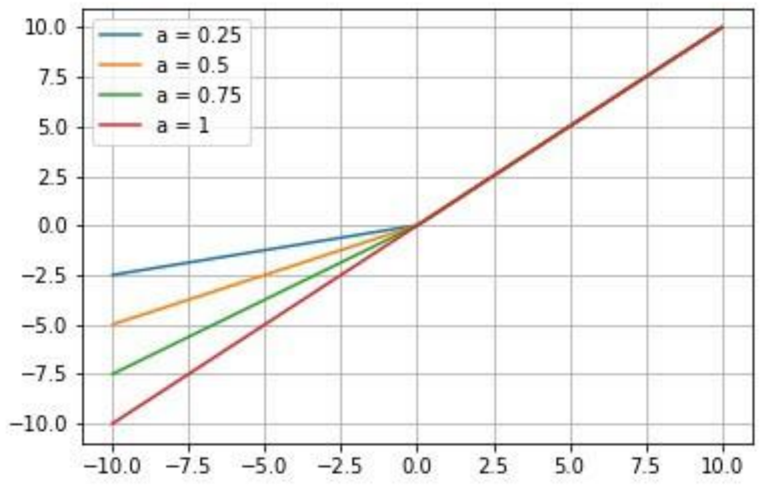
The code:

```
def parametric_relu_active_function(a, x):
    return a*x if x < 0 else x
```

Compute the results for different *a* values:

```
$ x   = numpy.linspace(-10, 10, 5000)
$ y_1 = [parametric_relu_active_function(0.25, i) for i in x]
$ y_2 = [parametric_relu_active_function(0.5, i) for i in x]
$ y_3 = [parametric_relu_active_function(0.75, i) for i in x]
$ y_4 = [parametric_relu_active_function(1, i) for i in x]
$ y_1
> [-2.5, -2.4975, ... , 9.97, 9.98, 9.99]
```

Plot the results for a = 0.25, 0.5, 0.75, 1:



Parametric ReLU activation function

If a = 0 the parametric ReLU is equivalent to the ReLU activation function. If a=0.01 the parametric ReLU correspond to the Leaky ReLU.

Exponential Linear Unit (eLU):
eLU is another variation of the ReLU function. The negative part of the function is handled by the exponential function with a slow smooth.

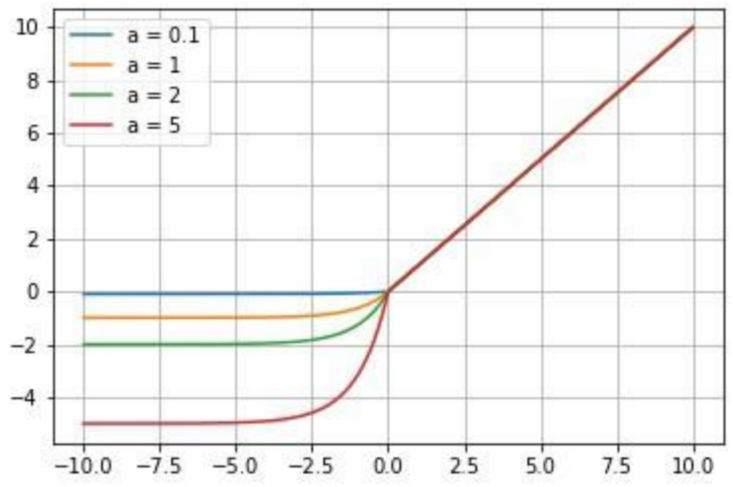
The corresponding function:

```
def elu_active_function(a, x):
    return a*(numpy.exp(x)-1) if x < 0 else x
```

y computation:

```
$ x   = numpy.linspace(-10, 10, 5000)
$ y_1 = [elu_active_function(0.1, i) for i in x]
$ y_2 = [elu_active_function(1, i) for i in x]
$ y_3 = [elu_active_function(2, i) for i in x]
$ y_4 = [elu_active_function(5, i) for i in x]
$ y_1
> [-0.09999546000702375, -0.09999541437933579, ... , 9.97, 9.98, 9.99]
```

Plot the results for a = 0.1, 1, 2, 4:



eLU activation function

ReLU-6:
Another variation of the ReLU function is the ReLU-6, 6 is an arbitrary parameter fixed by hand. The advantage is to shape the output for large positive number to the 6 value.

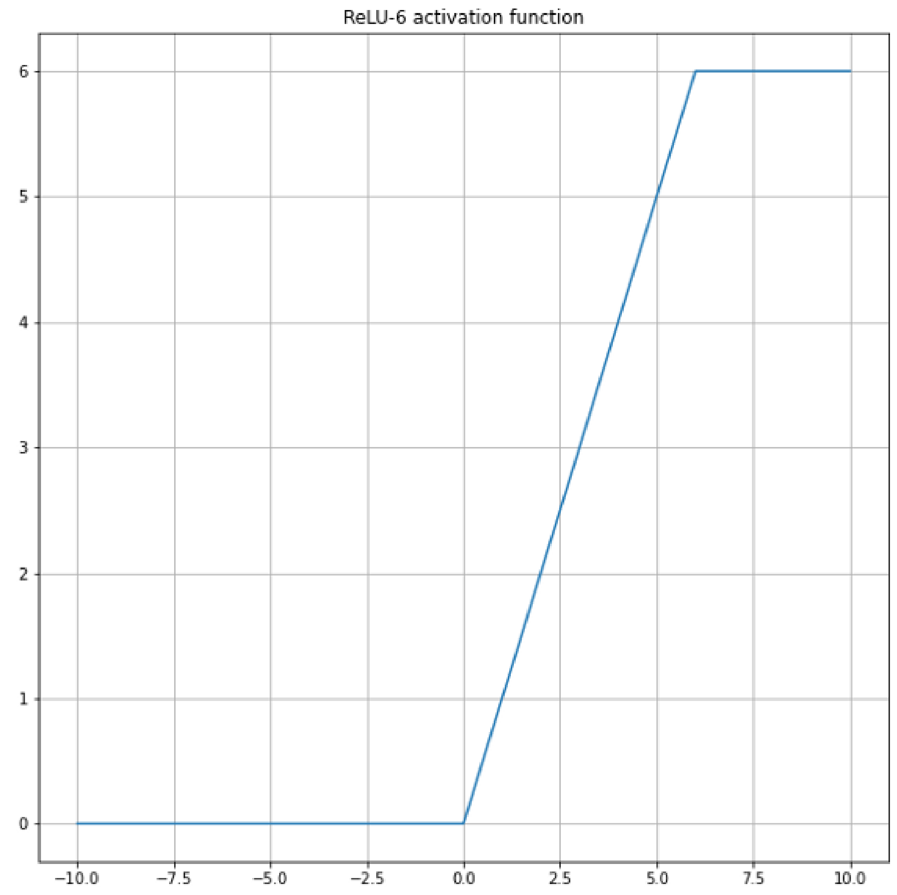
The corresponding code:

```
def relu_6_active_function(x):
    return numpy.array([0, x]).max() if x<6 else 6
```

The y computation:

```
$ y = [relu_6_active_function(i) for i in x]
```

Plot the results:



ReLU-6 activation function

Softplus:
The softplus activation function is an alternative of sigmoid and tanh functions. This functions have limits (upper, lower) but softplus is in the range (0, +inf).

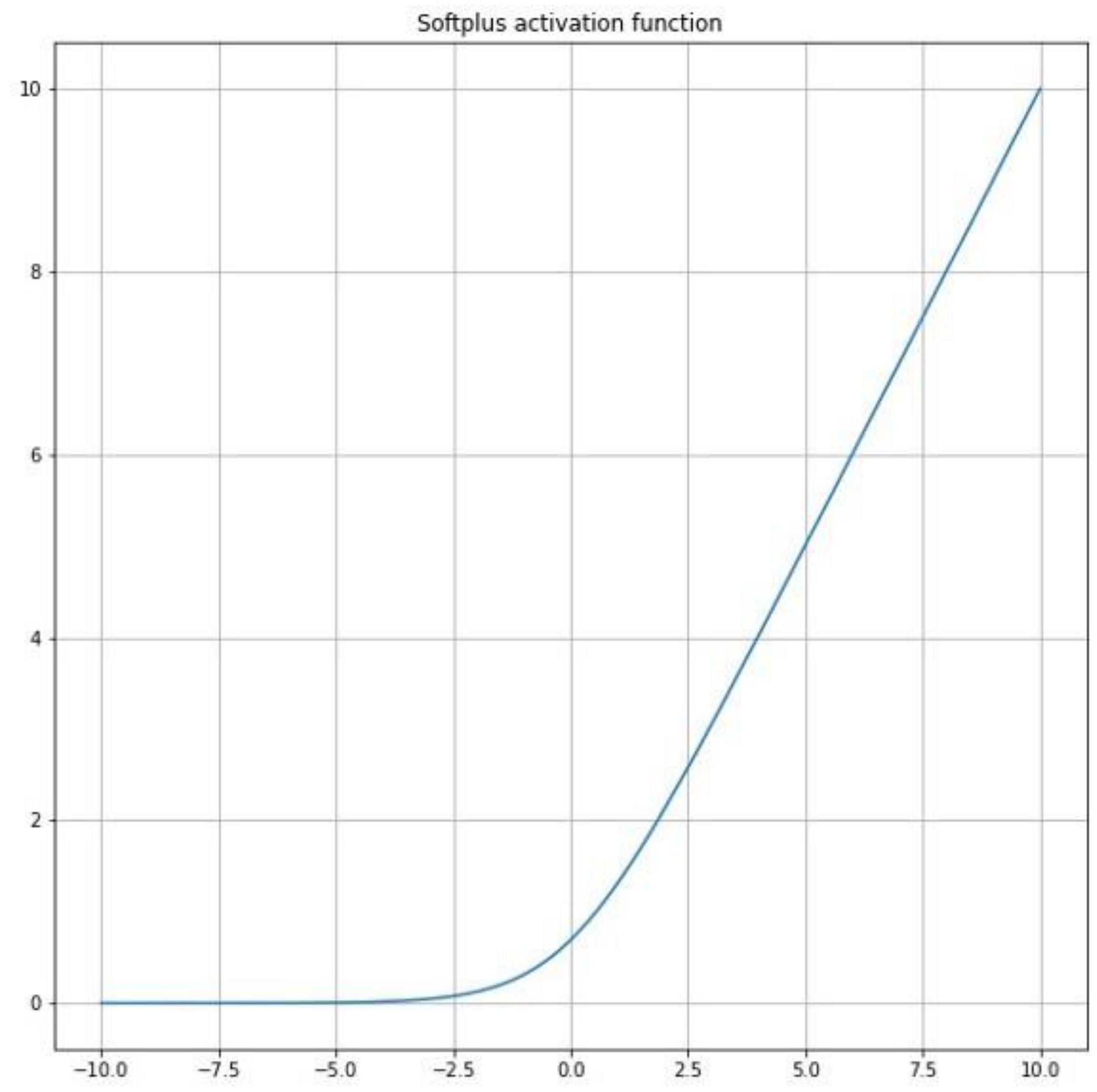
The corresponding code:

```
def softplus_active_function(x):
    return math.log(1+numpy.exp(x))
```

y computation:

$ y = [softplus_active_function(i) for i in x]

Plot the results:



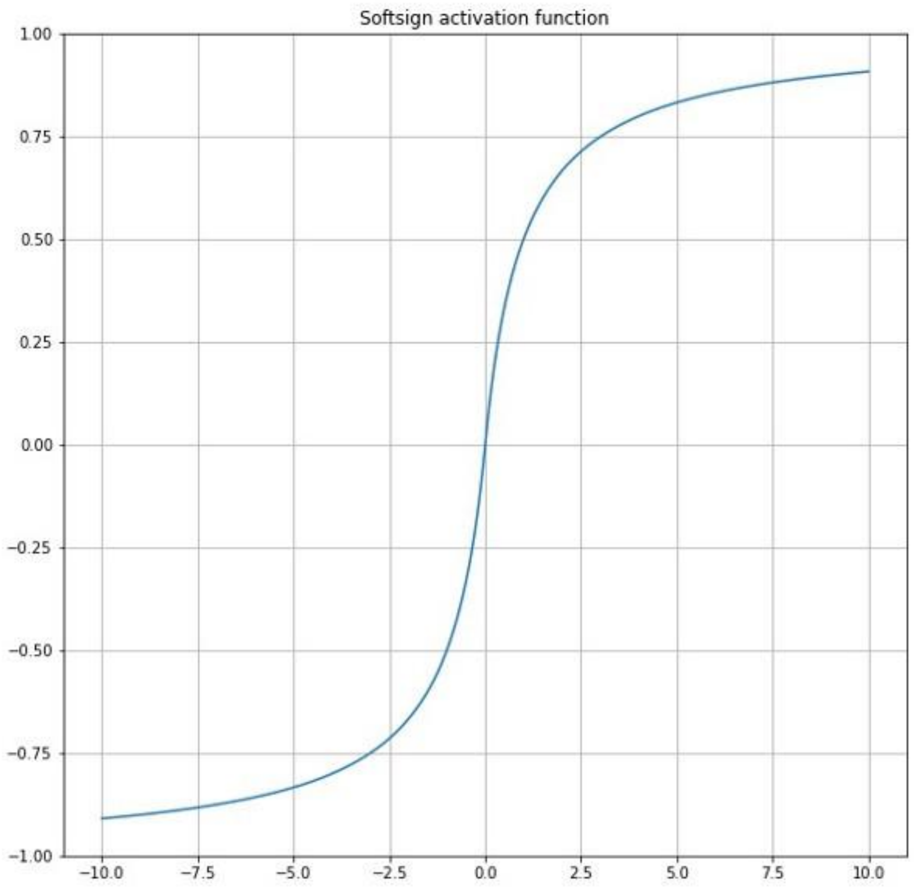
Softplus activation function

Softsign:
This activation function is a variation of tanh but is not very used in practice. tanh and softsign functions are closely related, tanh converges exponentially whereas softsign converges polynomially.

The corresponding code:

```
def softsign_active_function(x):
    return x / (1 + abs(x) )$ y = [softsign_active_function(i) for i in x]
```

Plot the results:



Softsign activation function

Softmax:
The softmax activation function is different from the other because it compute the probability distribution. The sum of the output is equal to 1.

The corresponding code:

```
def softmax_active_function(x):
    return numpy.exp(x)/numpy.sum(numpy.exp(x))
```

Compute the output is different because it’s a probability distribution taking into account the sum of exponential, the function needed all the *x* points to compute the output *y*.

```
$ x = [0.8, 1.2, 2.4, 4.6]
$ y = softmax_active_function(x)
$ y
> [0.01917691, 0.02860859, 0.09498386, 0.85723064]
$ numpy.sum(y)
> 1.0
```

Swish:
Swish is the newer activation function, published by Google in 2017 it improves the performances of ReLU on deeper models. This function is a variation of sigmoid function because it can be expressed by: x*sigmoid(x).

Swish has the properties of one-sided boundedness at zero, smoothness, and non-monotonicity, which may play a role in the observed efficacy of Swish and similar activation functions. SWISH: A SELF-GATED ACTIVATION FUNCTION, Prajit Ramachandran∗, Barret Zoph, Quoc V. Le, 2017

The corresponding codes:

```
def swish_active_function(x):
    return x/(1+numpy.exp(-x))
```

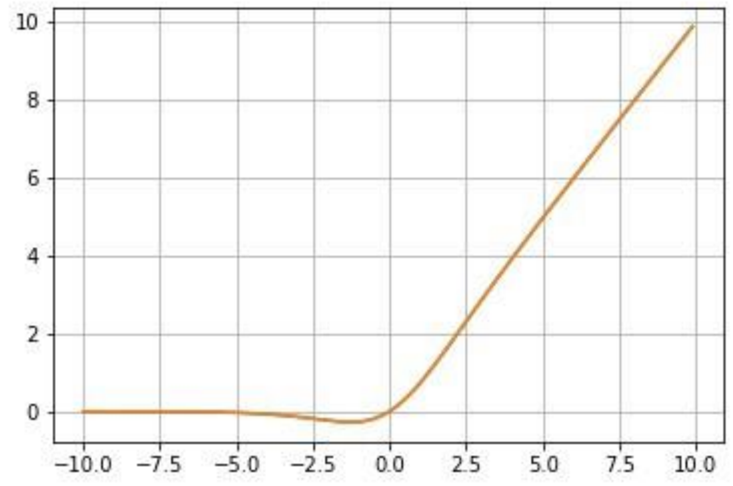
Or:

```
def swish_active_function(x):
    return x*sigmoid_active_function(x)
```

Compute the values:

```
$ x = numpy.linspace(-10, 10, 5000)
$ y = [swish_active_function(i) for i in x]
$ y
> [-0.0004539786870243439, -0.0004967044303692657, ..., 9.699405586525717,
9.799456604457717, 9.89950329556963]
```

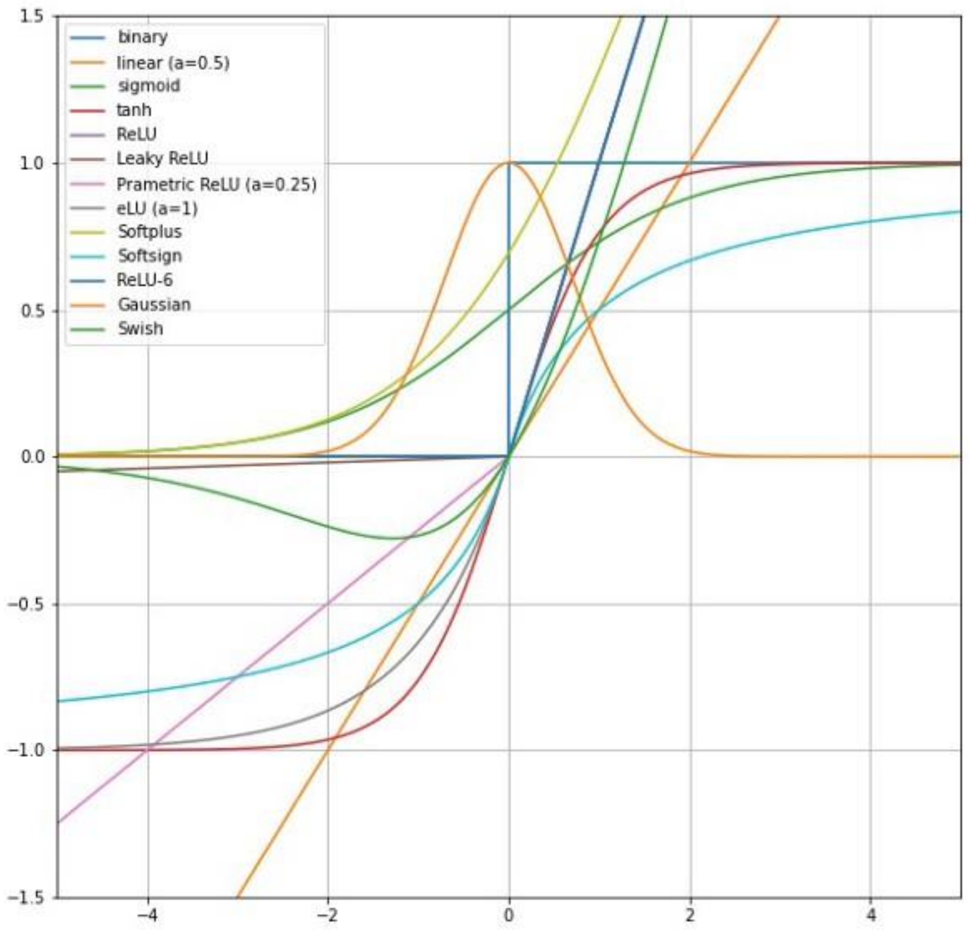
Plot the results:



Swish activation function

Conclusion:
Here are presented some activation functions (most popular) with their code and representation. Hope this visualisation could permit to everyone to understand the output of neurons and determined which function is better for the problem they work on.

The last plot corresponds to the activation functions stack in one graphic.



Evaluation Grid (To be created as per the SOP and Assessment guidelines by the faculty):

Sr. No.	Parameters	Marks Obtained	Maximum Marks
1.			
2.			
3.			