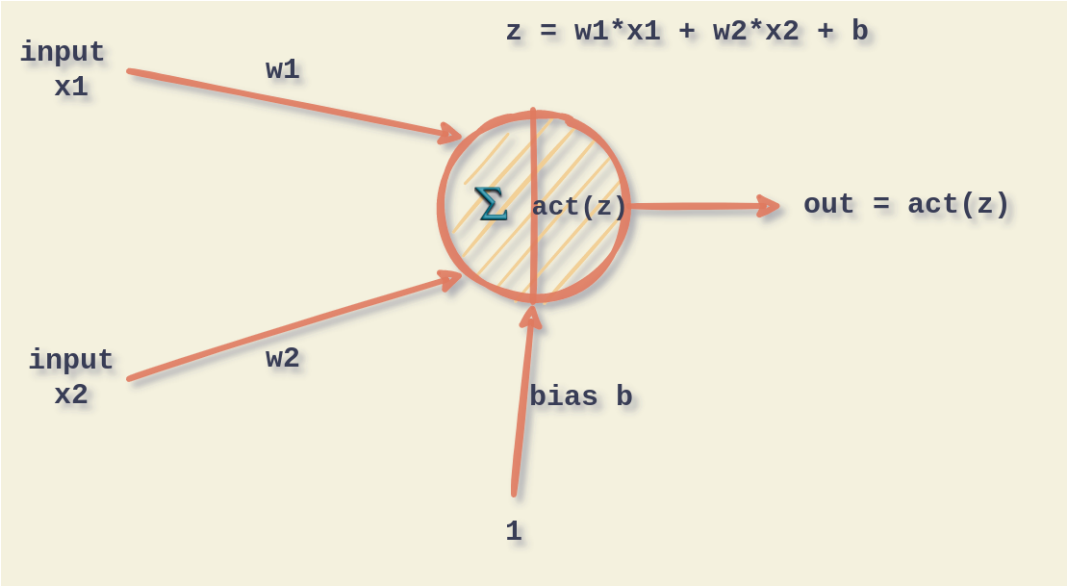
Activation Function

Activation functions helps to determine the output of a neural network. These type of functions are attached to each neuron in the network, and determines whether it should be activated or not, based on whether each neuron's input is relevant for the model's prediction.

> Activation function also helps to normalize the output of each neuron to a range between 1 and 0 or between -1 and 1.



In a neural network, inputs are fed into the neurons in the input layer. Each neuron has a weight, and multiplying the input number with the weight gives the output of the neuron, which is transferred to the next layer.

The activation function is a mathematical "gate" in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold.

> Neural networks use non-linear activation functions, which can help the network learn complex data, compute and learn almost any function representing a question, and provide accurate predictions.

In [1]:
```
# python helper functions for plotting graphs -
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import tensorflow as tf
plt.style.use('fivethirtyeight')

x = np.linspace(-10, 10, 100)

def plot_graph(x, y,
               ALPHA=0.6,
               label_x = r"$x \rightarrow$", label_y=r"$act(x) \rightarrow$",
               title=None,
               LABEL=None):
    plt.figure(figsize=(7,5))
    plt.axhline(y=0, color="black", linestyle="--", lw=2)
    plt.axvline(x=0, color="black", linestyle="--", lw=2)
    plt.xlabel(label_x)
    plt.ylabel(label_y)
    plt.title(title)

    if LABEL != None:
        plt.plot(x, y, alpha=ALPHA, label=LABEL);
        plt.legend(fontsize=14)

    else:
        plt.plot(x, y, alpha=ALPHA);

# def derivative(f, x, eps=0.000001):
#     return (f(x + eps) - f(x - eps))/(2 * eps)

def derivative(f, x, delta_x=1e-6):
    return (f(x + delta_x) - f(x))/(delta_x)
```

In [2]:
```
3e-1
```

Out[2]:
```
0.3
```

Commonly used activation functions

Sigmod function

The function formula and graph are as follows

$$\sigma(x) = \frac{1}{1 + e^{-x}}$$

$$where\ \sigma(x) \in (0, 1),$$
$$and\ x \in [-\infty, +\infty]$$

Python implimentaion -

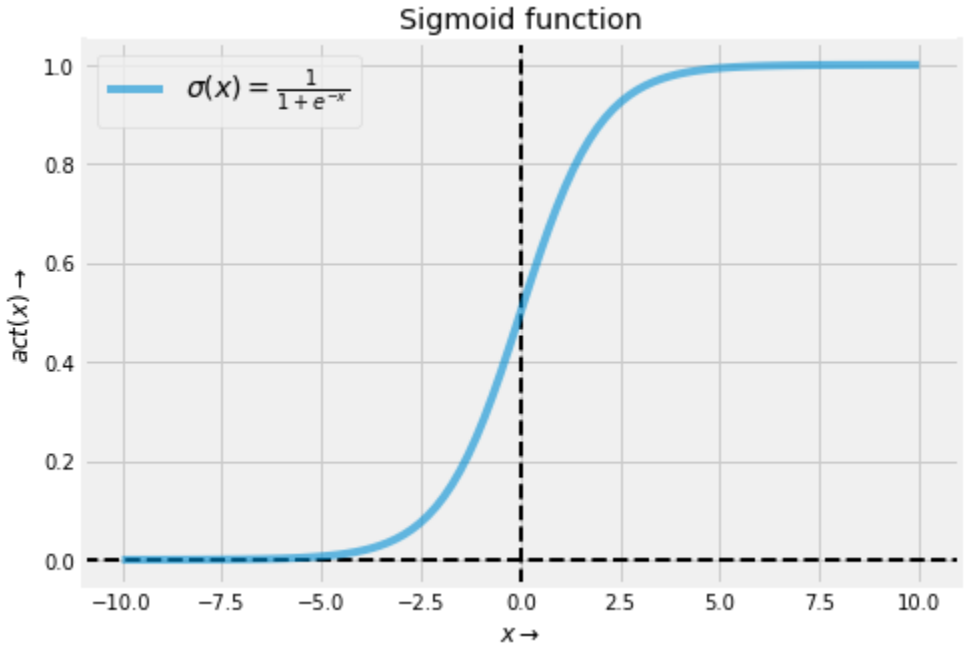
In [ ]:
```
def sigmoid(x):
    return tf.keras.activations.sigmoid(x)
```

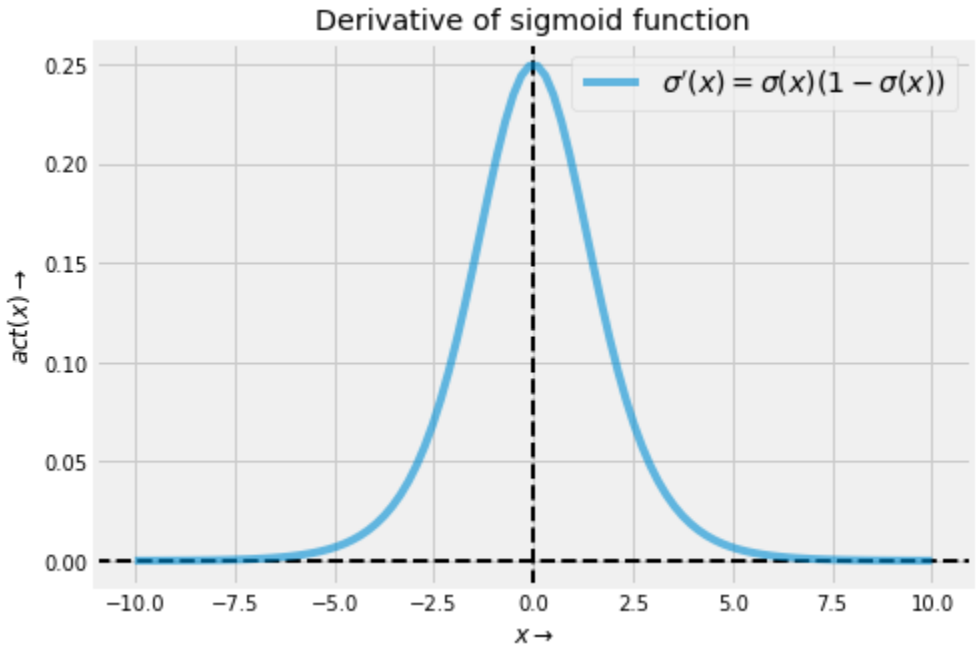
Alternatively -

```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

In [ ]:
```
plot_graph(x, sigmoid(x), title="Sigmoid function",
           LABEL=r"$\sigma(x) = \frac{1}{1 + e^{-x}}$")

plot_graph(x, derivative(sigmoid, x), title="Derivative of sigmoid function",
           LABEL=r"$\sigma^\prime(x) = \sigma(x)(1-\sigma(x))$")
```





- The Sigmoid function is the most frequently used activation function in the beginning of deep learning.
- It is a smoothing function that is easy to derive.
- In the sigmoid function, we can see that its output is in the open interval (0,1). We can think of probability, but in the strict sense, don't treat it as probability. The sigmoid function was more popular once.
- It can be thought of as the firing rate of a neuron. In the middle where the slope is relatively large, it is the sensitive area of the neuron. On the sides where the slope is very gentle, it is the neuron's inhibitory area.

The function itself has certain defects:-

1. When the input is slightly away from the coordinate origin, the gradient of the function becomes very small, almost zero. In the process of neural network backpropagation, we all use the chain rule of differential to calculate the differential of each weight w. When the backpropagation passes through the sigmod function, the differential on this chain is very small. Moreover, it may pass through many sigmod functions, which will eventually cause the weight w to have little effect on the loss function, which is not conducive to the optimization of the weight. This The problem is called gradient saturation or gradient dispersion.
2. The function output is not centered on 0, which will reduce the efficiency of weight update.
3. The sigmod function performs exponential operations, which is slower for computers.

Advantages of Sigmoid Function : -

1. Smooth gradient, preventing “jumps” in output values.
2. Output values bound between 0 and 1, normalizing the output of each neuron.
3. Clear predictions, i.e very close to 1 or 0.

Sigmoid has three major disadvantages:

- Prone to gradient vanishing
- Function output is not zero-centered
- Power operations are relatively time consuming

Hyperbolic tangent activation function

The tanh function formula and curve are as follows

$$tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$$

$$where \ \tanh(x) \in (-1, 1),$$
$$and \ x \in [-\infty, +\infty]$$

Python implimentaion -

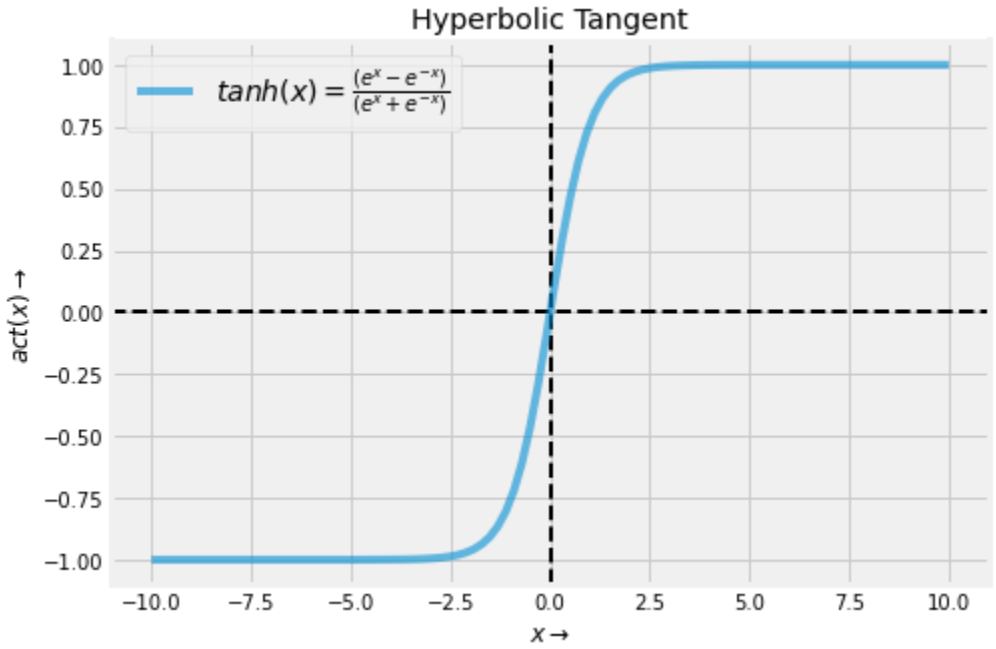
In [ ]:
```
def tanh(x):
    return tf.keras.activations.tanh(x)
```

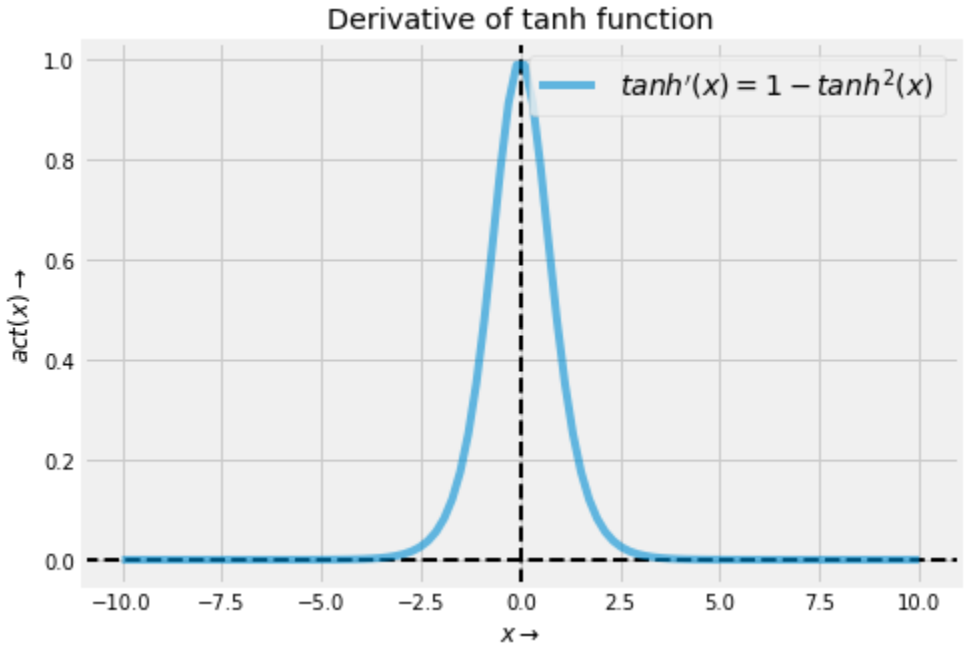
Alternatively -

```
    def tanh(x):
        return np.tanh(x)
```

In [ ]:
```
plot_graph(x, tanh(x), title="Hyperbolic Tangent",
           LABEL=r"$tanh(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}$");

plot_graph(x, derivative(tanh, x), title="Derivative of tanh function",
           LABEL=r"$tanh^\prime(x) = 1 - tanh^2(x)$")
```





Tanh is a hyperbolic tangent function. The curves of tanh function and sigmod function are relatively similar. Let ’s compare them. First of all, when the input is large or small, the output is almost smooth and the gradient is small, which is not conducive to weight update. The difference is the output interval.

The output interval of tanh is (-1, 1), and the whole function is 0-centric, which is better than sigmod.

In general binary classification problems, the tanh function is used for the hidden layer and the sigmoid

Rectified linear unit activation function(ReLU)

ReLU function formula and curve are as follows

$$ReLU(x) = max(x, 0)$$

$$where\ ReLU(x) \in (0, x),$$
$$and\ x \in [-\infty, +\infty]$$

Python implimentation -

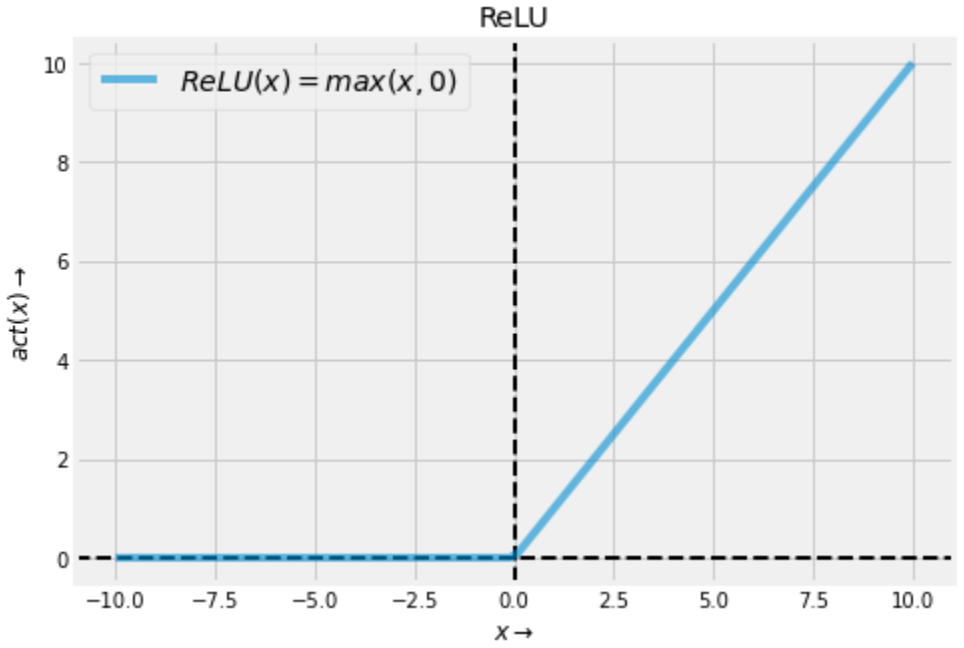
In [ ]:
```
def relu(x):
    return tf.keras.activations.relu(x, alpha=0.0, max_value=None, threshold=0)
```

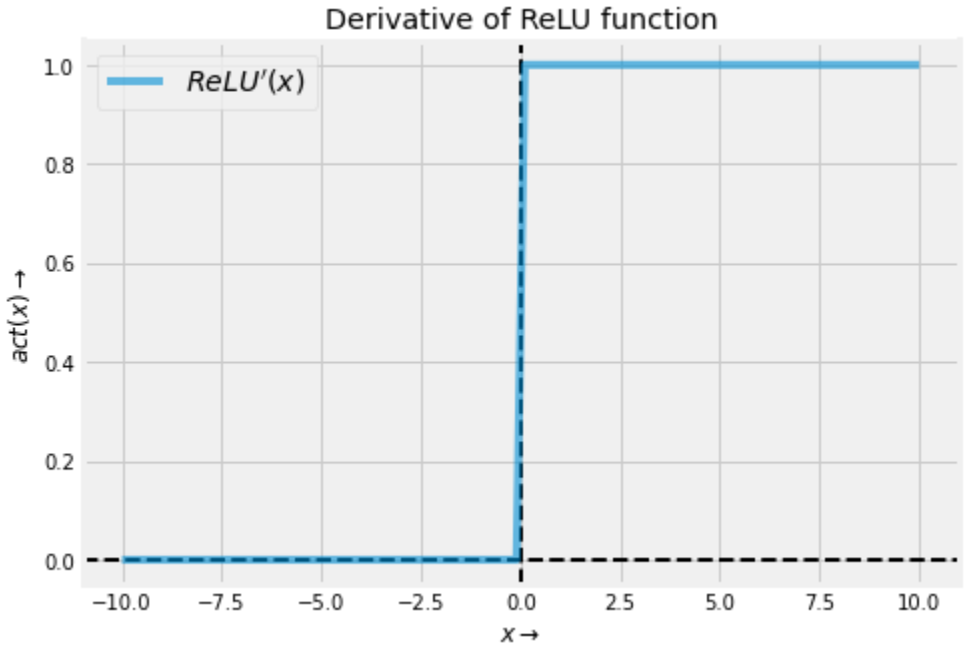
Alternatively

```
def relu(x):
    return np.where(x>=0, x, 0)
```

In [ ]:
```
plot_graph(x, relu(x), title="ReLU",
           LABEL=r"$ReLU(x)= max(x,0)$");

plot_graph(x, derivative(relu, x), title="Derivative of ReLU function",
           LABEL=r"$ReLU^\prime(x)$")
```





The ReLU function is actually a function that takes the maximum value. Note that this is not fully interval-derivable, but we can take sub-gradient, as shown in the figure above. Although ReLU is simple, it is an important achievement in recent years.

The ReLU (Rectified Linear Unit) function is an activation function that is currently more popular. Compared with the sigmod function and the tanh function, it has the following advantages:

1. When the input is positive, there is no gradient saturation problem.

2. The calculation speed is much faster. The ReLU function has only a linear relationship (before and after 0 i.e. conditionally linear or piece wise linear). Whether it is forward or backward, it is much faster than sigmod and tanh. (Sigmod and tanh need to calculate the exponent, which will be slower.)

Of course, there are disadvantages:

1. When the input is negative, ReLU is completely inactive, which means that once a negative number is entered, ReLU will die. In this way, in the forward propagation process, it is not a problem. Some areas are sensitive and some are insensitive. But in the backpropagation process, if you enter a negative number, the gradient will be completely zero, which has the same problem as the sigmod function and tanh function.

Leaky ReLU function

$$leaky\_relu(x, \alpha) = \begin{cases} x & x \geq 0 \\ \alpha x & x < 0 \end{cases}$$

$$where\ x \in [-\infty, +\infty]$$

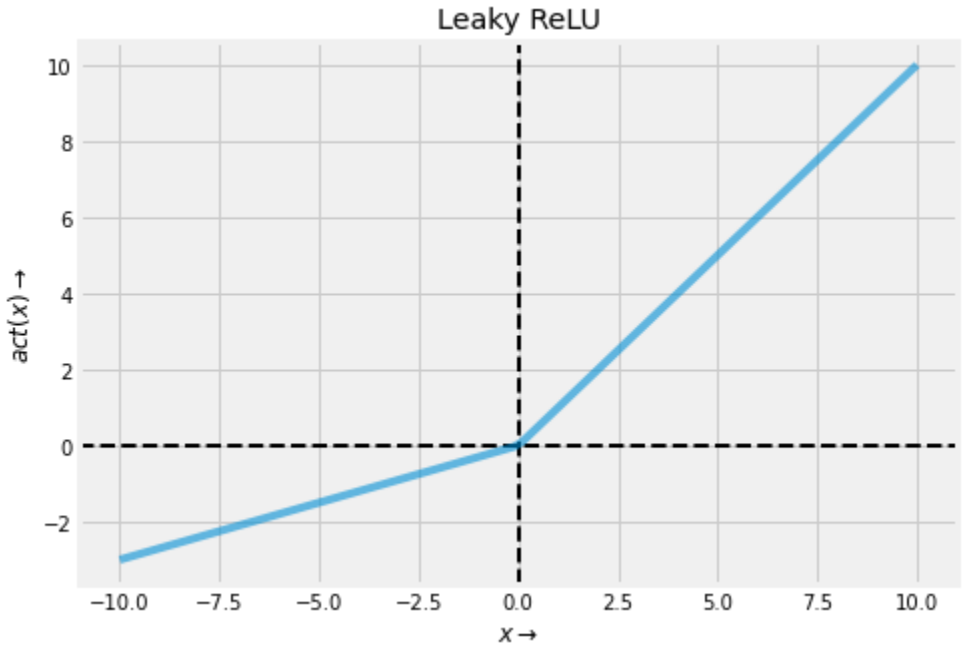
Python implimentation -

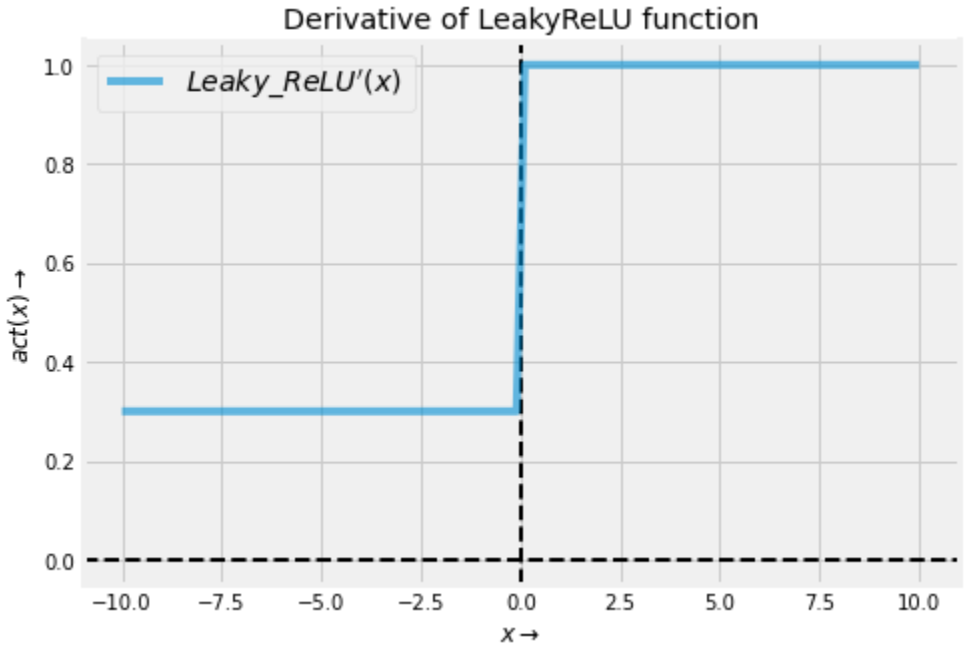
In [ ]:
```
def leaky_relu(x, alpha=0.3):
    return np.where(x>=0, x, alpha*x)
```

In [ ]:
```
# tf.keras.layers.LeakyReLU
```

In [ ]:
```
plot_graph(x, leaky_relu(x), title="Leaky ReLU");

plot_graph(x, derivative(leaky_relu, x), title="Derivative of LeakyReLU function",
           LABEL=r"$Leaky\_ReLU^\prime(x)$")
```





In order to solve the Dead ReLU Problem, people proposed to set the first half of ReLU 0.01x instead of 0. Another intuitive idea is a parameter-based method, $ParametricReLU: f(x) = max(\alpha x, x)$, WHERE α can be learned from back propagation. In theory, Leaky ReLU has all the advantages of ReLU, plus there will be no problems with Dead ReLU, but in actual operation, it has not been fully proved that Leaky ReLU is always better than ReLU.

ELU (Exponential Linear Units) function

$$elu(x, \alpha) = \begin{cases} x & x \geq 0 \\ \alpha \cdot (e^x - 1) & x < 0 \end{cases}$$

$$\alpha = scaler\ slope\ of\ negative\ section$$

Python implimentation -

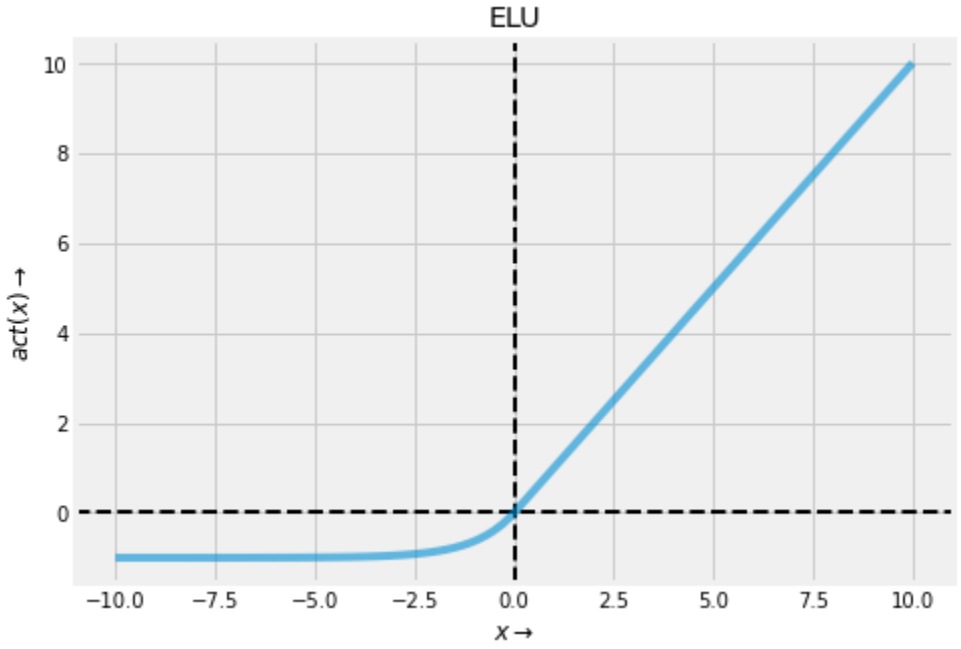
In [ ]:
```
def elu(x, ALPHA=1.0):
    return tf.keras.activations.elu(x, alpha=ALPHA)
```

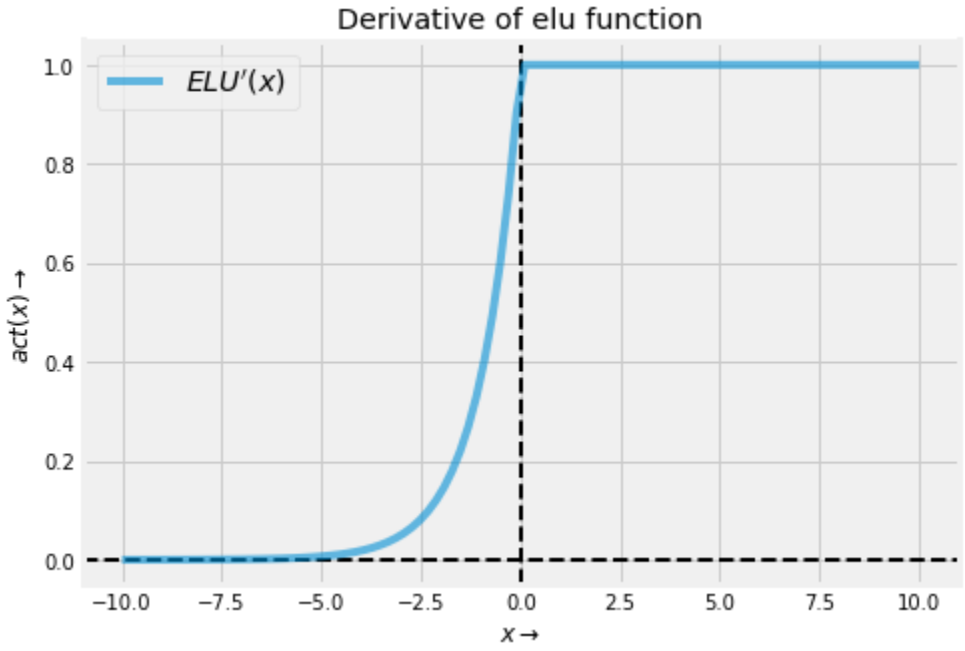
Alternatively

```
def elu(x, ALPHA=1.0):
    return np.where(x>0, x, (ALPHA * (np.exp(x) - 1)))
```

In [ ]:
```
plot_graph(x, elu(x), title="ELU");

plot_graph(x, derivative(elu, x), title="Derivative of elu function",
           LABEL=r"$ELU^\prime(x)$")
```





ELU is also proposed to solve the problems of ReLU. Obviously, ELU has all the advantages of ReLU, and:

- No Dead ReLU issues
- The mean of the output is close to 0, zero-centered

One small problem is that it is slightly more computationally intensive. Similar to Leaky ReLU, although theoretically better than ReLU, there is currently no good evidence in practice that ELU is always better than ReLU.

SELU:

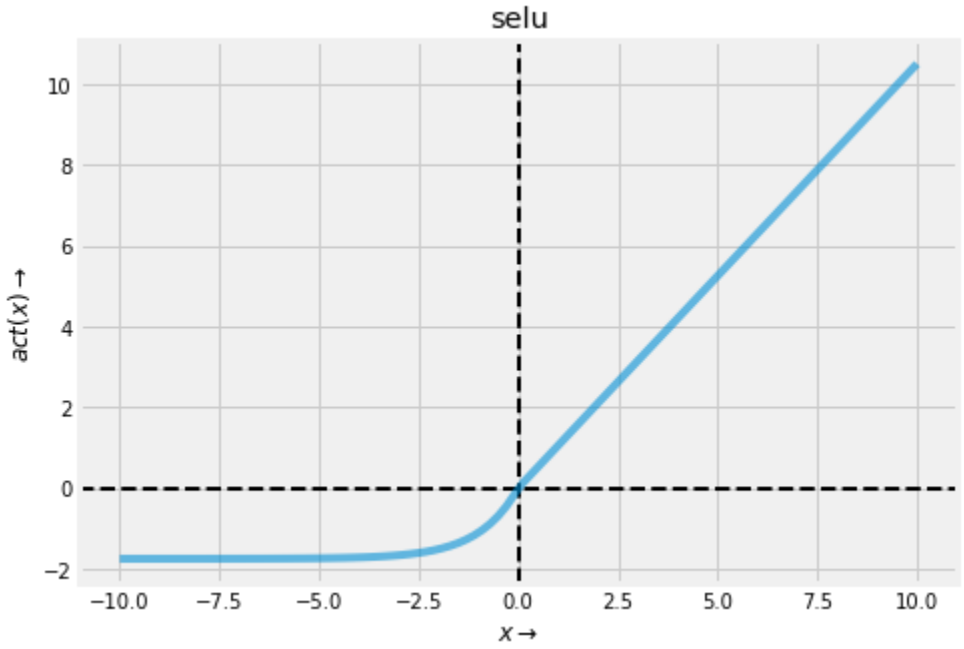
scale * elu(x, alpha)

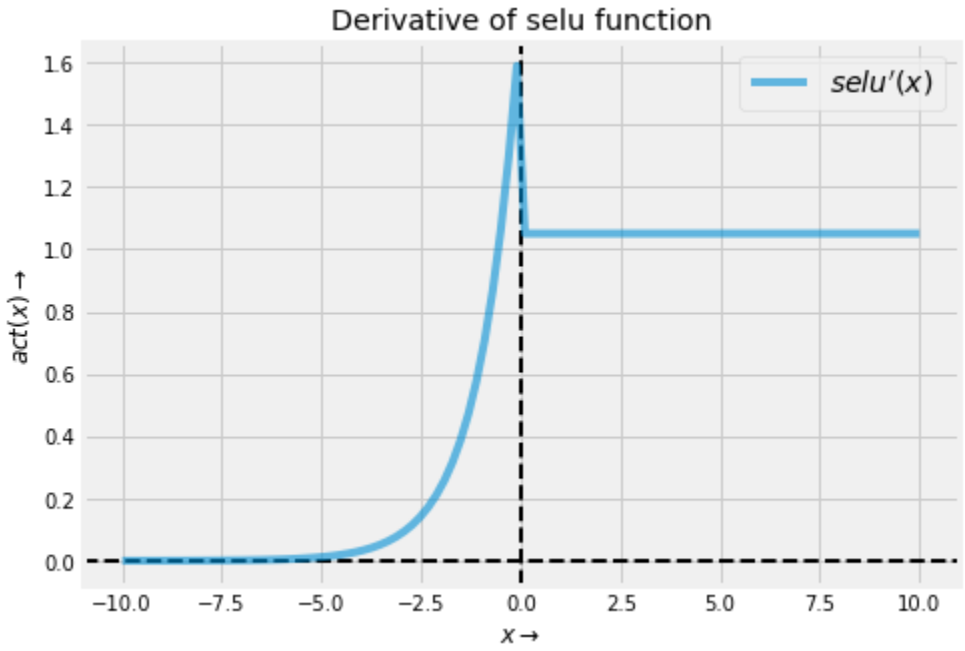
reference (https://www.tensorflow.org/api_docs/python/tf/keras/activations/selu)

In [ ]:
```
def selu(x, ALPHA=1.0):
    return tf.keras.activations.selu(x)
```

In [ ]:
```
plot_graph(x, selu(x), title="selu");

plot_graph(x, derivative(selu, x), title="Derivative of selu function",
           LABEL=r"$selu^\prime(x)$")
```





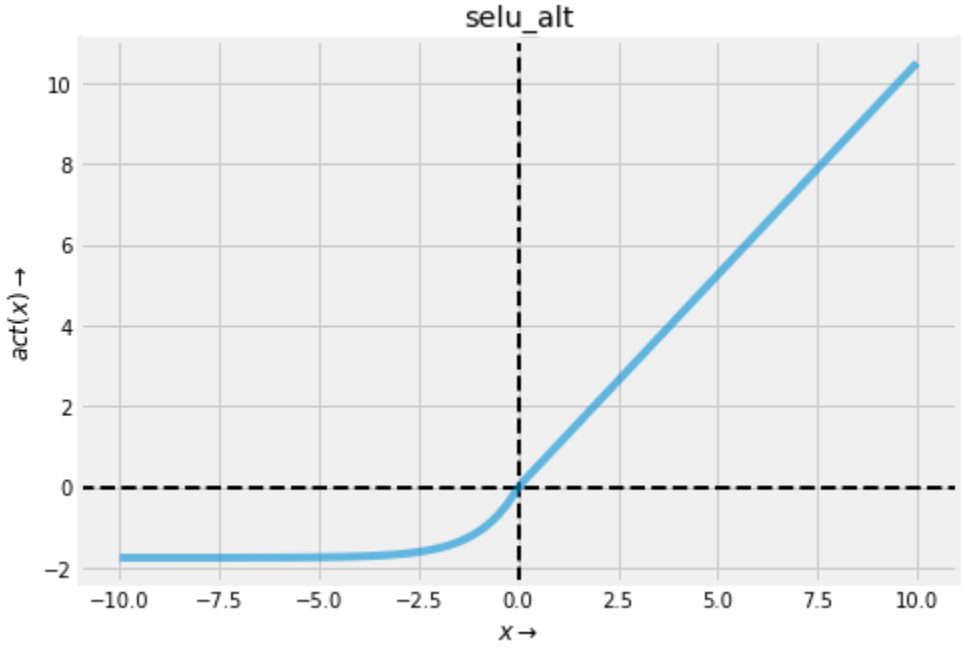
In [ ]:

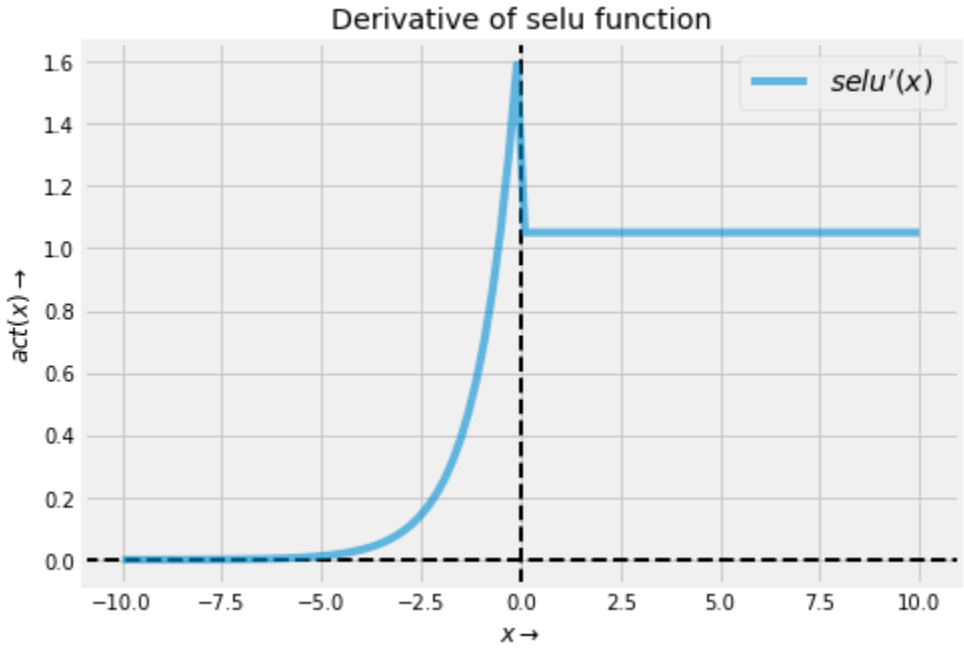
```
def selu_alt(x, ALPHA=1.67326324, scale=1.05070098):
    return scale * tf.keras.activations.elu(x, alpha=ALPHA)
```

In [ ]:

```
plot_graph(x, selu_alt(x), title="selu_alt");

plot_graph(x, derivative(selu_alt, x), title="Derivative of selu function",
           LABEL=r"$selu^\prime(x)$")
```





PRelu (Parametric ReLU)

PReLU is also an improved version of ReLU. In the negative region, PReLU has a small slope, which can also avoid the problem of ReLU death. Compared to ELU, PReLU is a linear operation in the negative region. Although the slope is small, it does not tend to 0, which is a certain advantage.

$$f(y_i) = \begin{cases} y_i & y_i > 0 \\ \alpha_i \cdot y_i & y_i \leq 0 \end{cases}$$

We look at the formula of PReLU. The parameter α is generally a number between 0 and 1, and it is generally relatively small, such as a few zeros. When α = 0.01, we call PReLU as Leaky Relu , it is regarded as a special case PReLU it.

Above, y_i is any input on the ith channel and a_i is the negative slope which is a learnable parameter.

- if $\alpha_i = 0$, f becomes ReLU
- if $\alpha_i > 0$, f becomes leaky ReLU

Softmax activation function

$$S(x_j) = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}, where\ j = 1, 2, \cdots, K$$

for an arbitrary real vector of length K, Softmax can compress it into a real vector of length K with a value in the range (0, 1), and the sum of the elements in the vector is 1.

It also has many applications in Multiclass Classification and neural networks. Softmax is different from the normal max function: the max function only outputs the largest value, and Softmax ensures that smaller values have a smaller probability and will not be discarded directly. It is a "max" that is "soft".

The denominator of the Softmax function combines all factors of the original output value, which means that the different probabilities obtained by the Softmax function are related to each other. In the case of binary classification, for Sigmoid, there are:

$$p(y = 1|x) = \frac{1}{1 + e^{-\theta^T x}}$$

$$p(y = 0|x) = 1 - p(y = 1|x) = \frac{e^{-\theta^T x}}{1 + e^{-\theta^T x}}$$

For Softmax with K = 2, there are:

$$p(y = 1|x) = \frac{e^{\theta_1^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{1}{1 + e^{(\theta_0^T - \theta_1^T)x}} = \frac{1}{1 + e^{-\beta x}}$$

$$p(y = 0|x) = \frac{e^{\theta_0^T x}}{e^{\theta_0^T x} + e^{\theta_1^T x}} = \frac{e^{(\theta_0^T - \theta_1^T)x}}{1 + e^{(\theta_0^T - \theta_1^T)x}} = \frac{e^{-\beta x}}{1 + e^{-\beta x}}$$

Among them:

$$\beta = -(\theta_0^T - \theta_1^T)$$

can be seen that in the case of binary classification, Softmax is degraded to Sigmoid.

In [ ]:
```
tf.keras.activations.softmax(tf.constant([[0.3,0.3,0.4,0.5,0.7]]), axis=-1)
```

Out[18]:
```
<tf.Tensor: shape=(1, 5), dtype=float32, numpy=
array([[0.17186859, 0.17186859, 0.18994418, 0.20992078, 0.2563978 ]],
      dtype=float32)>
```

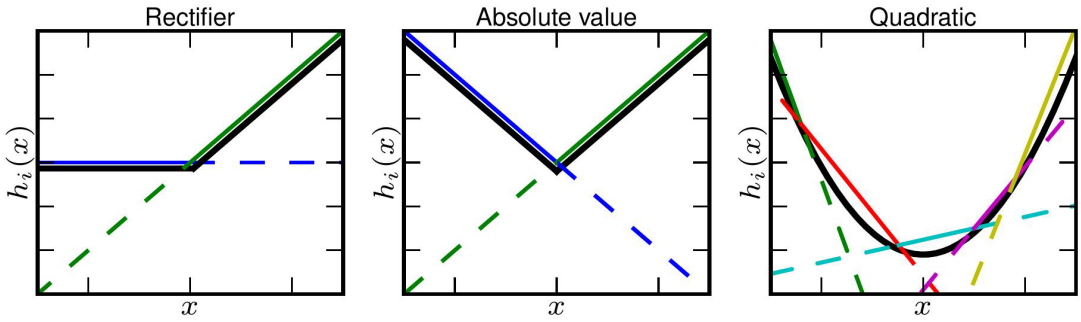
In [ ]:
```
# np.sum(np.exp(x) / tf.reduce_sum(np.exp(x)))
```

In [ ]:
```
import numpy as np
1 / (1 + np.exp(-10))
```

Out[20]:
```
0.9999546021312976
```

Maxout

The Maxout Unit is a generalization of the ReLU and the leaky ReLU functions. It is a piecewise linear function that returns the maximum of the inputs, designed to be used in conjunction with dropout. Both ReLU and leaky ReLU are special cases of Maxout.



$$f(x) = max(w_1^T x + b_1, w_2^T x + b_2)$$

The main drawback of Maxout is that it is computationally expensive as it doubles the number of parameters for each neuron.

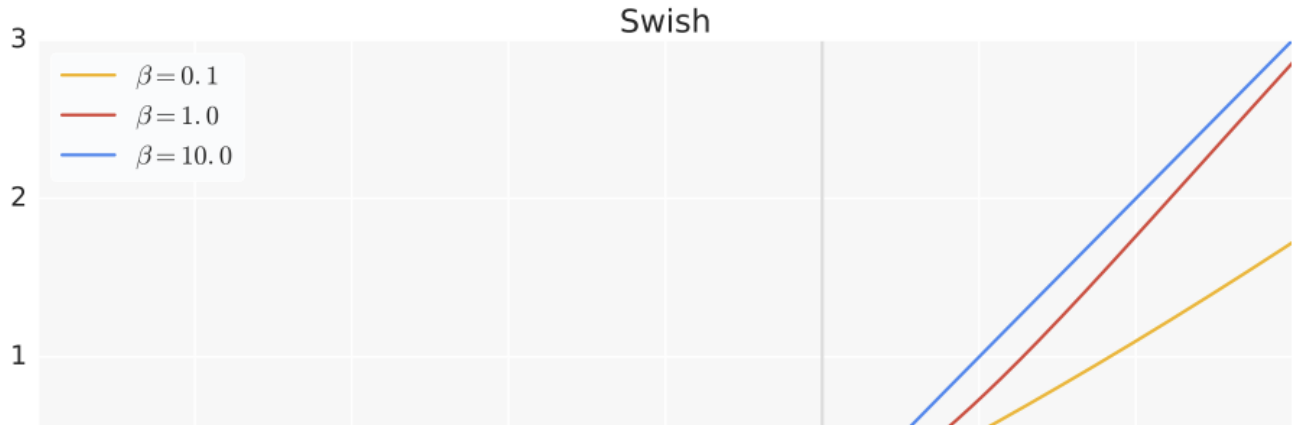
reference (https://paperswithcode.com/method/maxout)

Swish

Swish is an activation function,

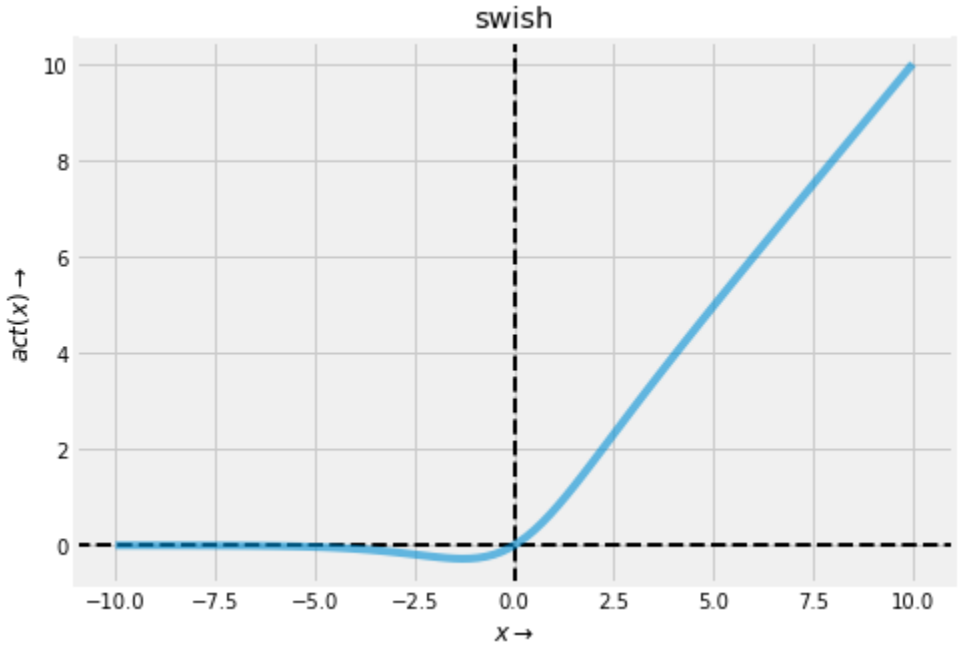
$f(x) = x.\sigma(\beta.x)$, where β a learnable parameter.

Nearly all implementations do not use the learnable parameter , in which case the activation function is $f(x) = x.\sigma(x)$ ("Swish-1").

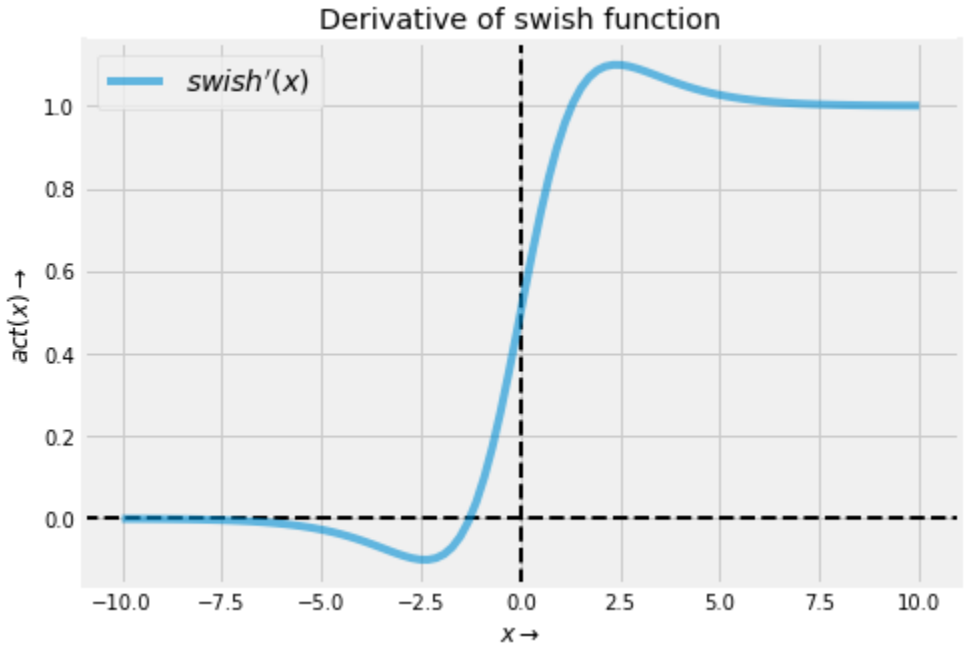


In [ ]:
```
def swish(z):
    return tf.keras.activations.swish(z)
```

In [ ]:
```
plot_graph(x, swish(x), title="swish");
```

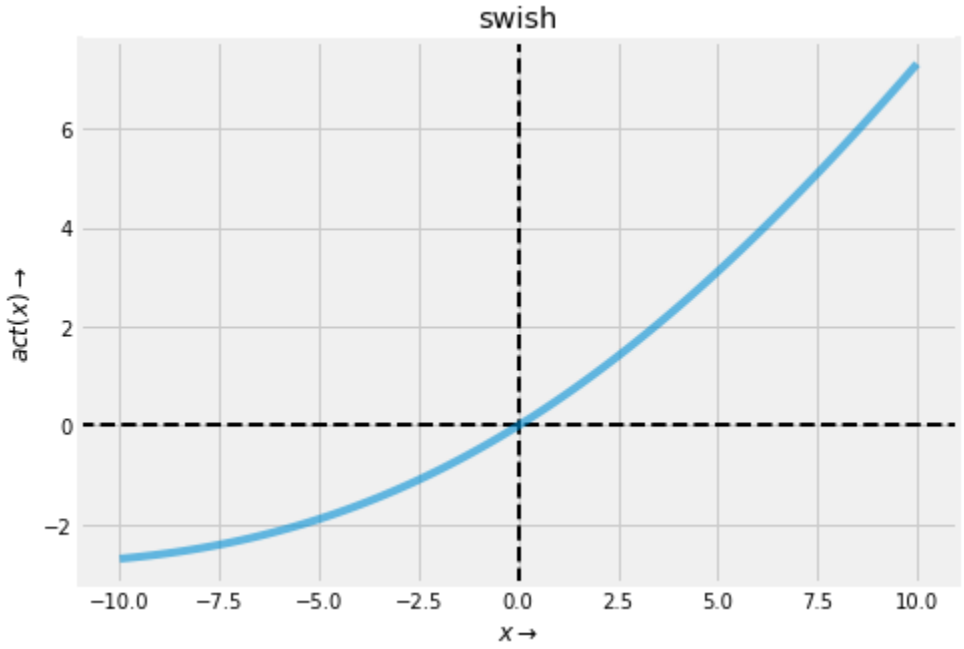


In [ ]:
```
plot_graph(x, derivative(swish, x), title="Derivative of swish function",
           LABEL=r"$swish^\prime(x)$")
```



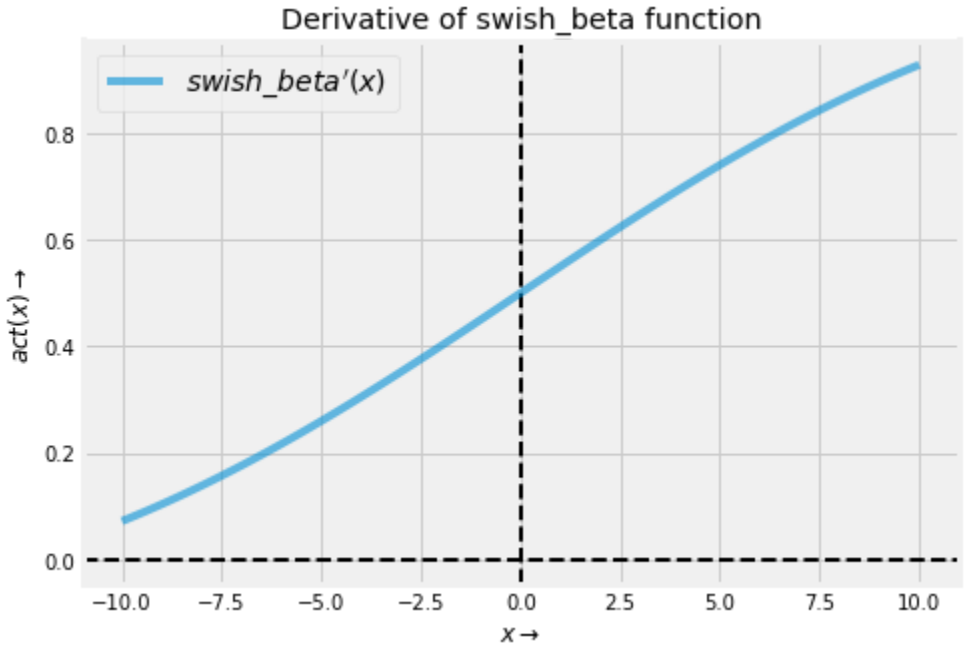
In [ ]:
```
def swish_beta(z, beta=0.1):
    return z * sigmoid(z * beta)
```

In [ ]:
```
plot_graph(x, swish_beta(x), title="swish");
```



In [ ]:

```
plot_graph(x, derivative(swish_beta, x), title="Derivative of swish_beta function",
           LABEL=r"$swish\_beta^\prime(x)$")
```



Softplus

Softplus is an activation function

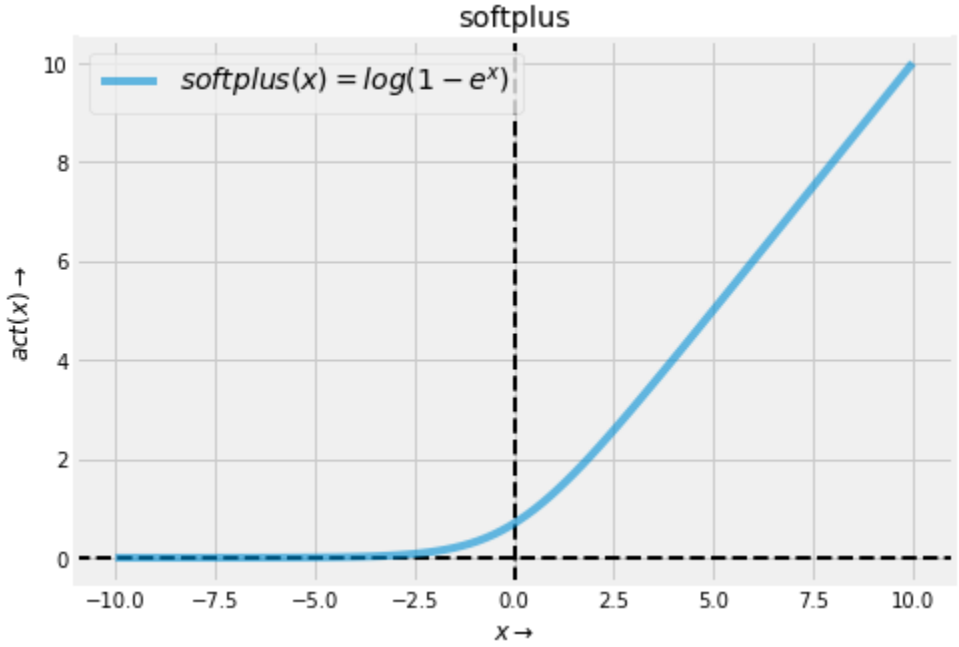
$$f(x) = log(1 + e^x)$$

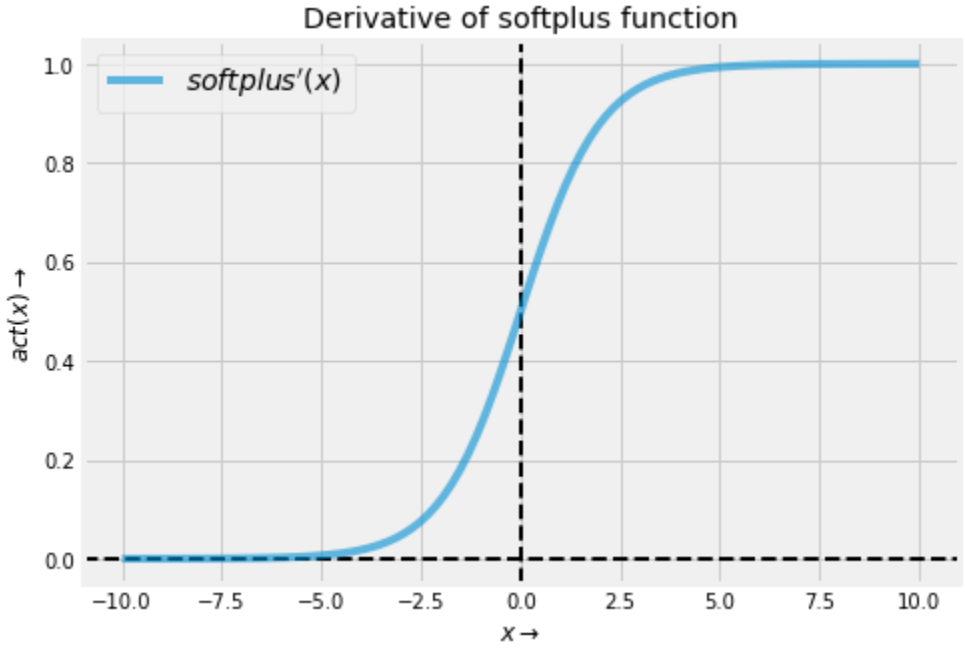
It can be viewed as a smooth version of ReLU.

In [ ]:
```
def softplus(z):
    return tf.keras.activations.softplus(z)
```

In [ ]:
```
plot_graph(x, softplus(x), title="softplus",
           LABEL=r"$softplus(x)= log(1-e^x)$");

plot_graph(x, derivative(softplus, x), title="Derivative of softplus function",
           LABEL=r"$softplus^\prime(x)$")
```





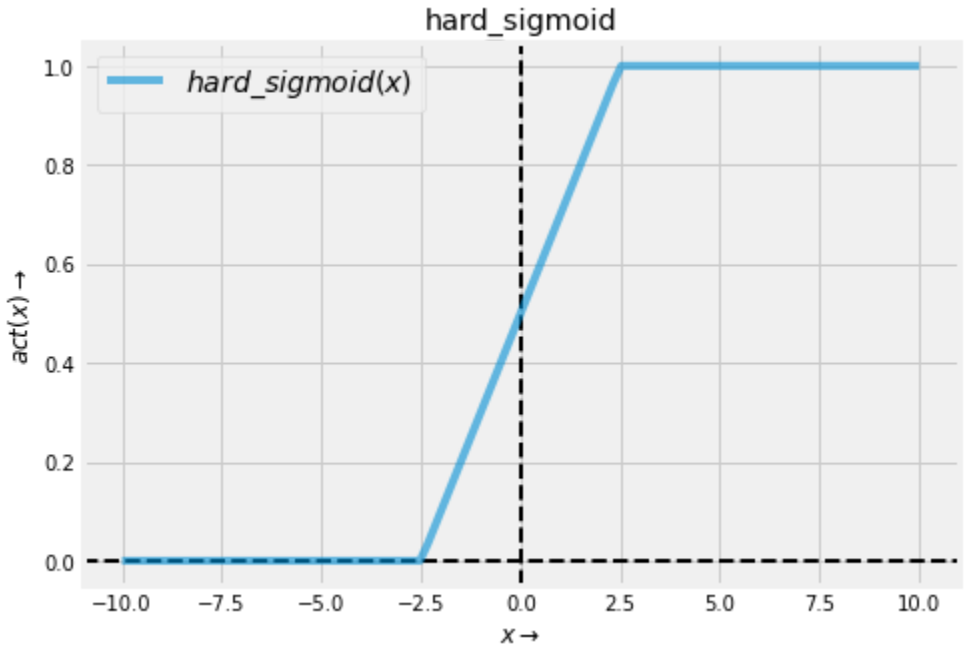
$$hard\_sigmoid(x) = \begin{cases} 0 & x < -2.5 \\ 1 & x > 2.5 \\ 0.2 \times x + 0.5 & -2.5 \leq x \leq 2.5 \end{cases}$$

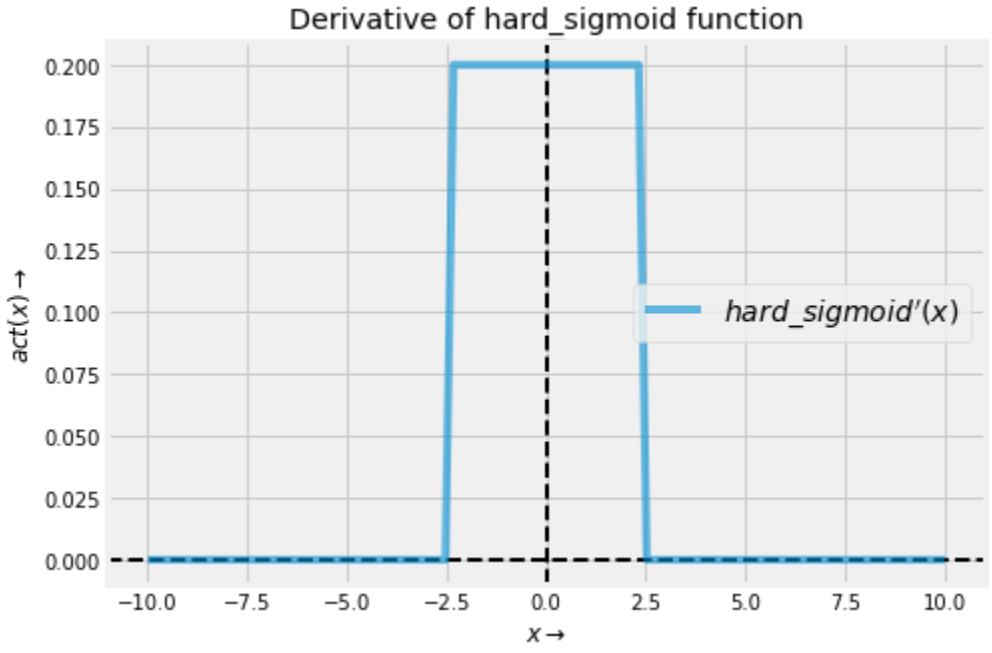
In [ ]:
```
def hard_sigmoid(z):
    z = tf.constant(z) # Check reference
    return tf.keras.activations.hard_sigmoid(z)
```

reference (https://stackoverflow.com/questions/57555407/numpy-dtype-object-has-no-attribute-base-dtype-in-keras)

In [ ]:
```
plot_graph(x, hard_sigmoid(x), title="hard_sigmoid",
           LABEL=r"$hard\_sigmoid(x)$");

plot_graph(x, derivative(hard_sigmoid, x), title="Derivative of hard_sigmoid function",
           LABEL=r"$hard\_sigmoid^\prime(x)$")
```





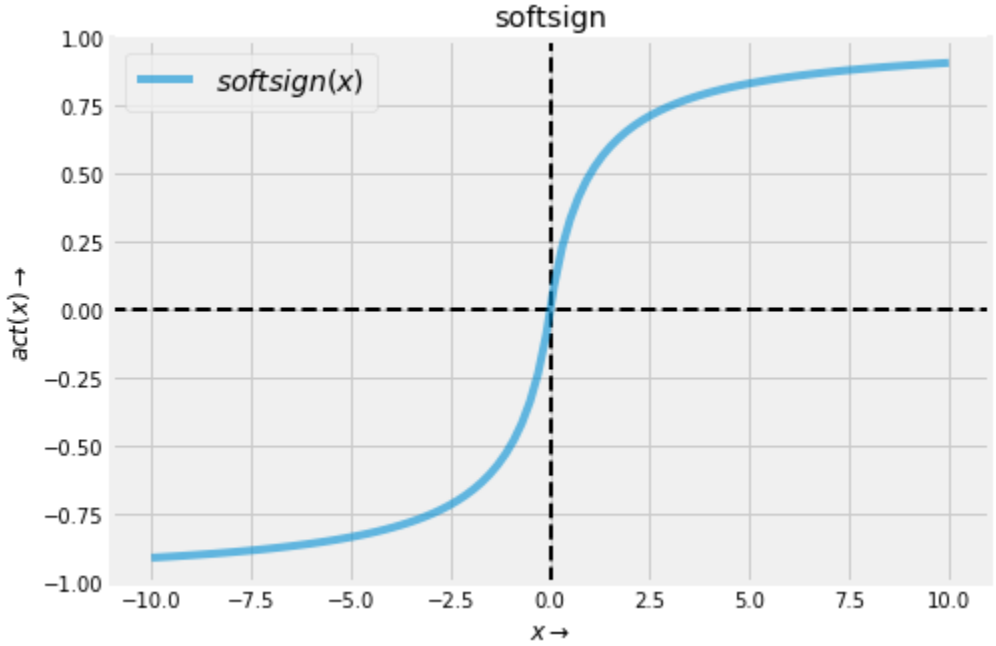
Softsign

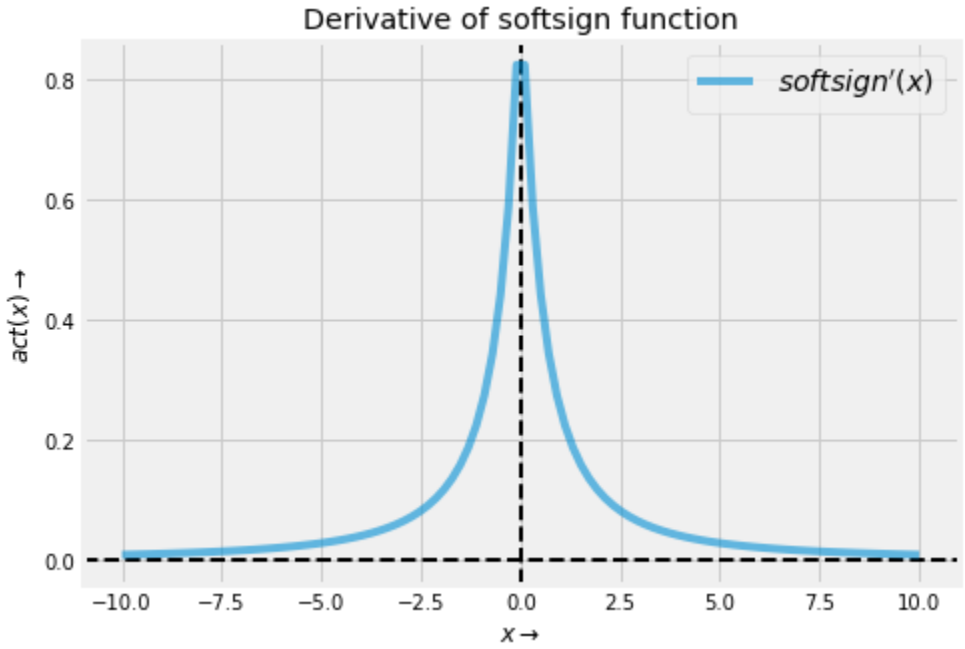
$$softsign(x) = \frac{x}{(|x| + 1)}$$

In [ ]:
```
def softsign(z):
    return tf.keras.activations.softsign(z)
```

In [ ]:
```
plot_graph(x, softsign(x), title="softsign",
           LABEL=r"$softsign(x)$");

plot_graph(x, derivative(softsign, x), title="Derivative of softsign function",
           LABEL=r"$softsign^\prime(x)$")
```





----------NOTE----------

Generally speaking, these activation functions have their own advantages and disadvantages. There is no statement that indicates which ones are not working, and which activation functions are good. All the good and bad must be obtained by experiments.