Introduction

Why we need activation function ?
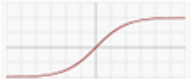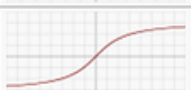

We need activation functions in neural networks for two main reasons:

1. Non-linearity: Without them, networks would only be able to perform **linear** computations, limiting them to learning simple relationships. Activation functions introduce **non-linearity**, allowing them to model complex patterns and solve a wider range of problems.

2. Gradient flow: During training, neural networks adjust their internal weights based on the error. Activation functions allow **gradients** (measures of how errors change with respect to weights) to flow properly through the network, making this learning process possible.

Think of them as essential gates within neurons, deciding which information gets passed forward and shaping the network's ability to learn complex relationships.

This Notebook is all about activation function. How it works and where to use which activation function. And this notebooks is using tensorflow library for code demonstration.

Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x \neq 0 \\ ? & \text{for} \quad x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \dfrac{1}{1 + e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \dfrac{2}{1 + e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \dfrac{1}{x^2 + 1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU) [2]		$f(x) = \begin{cases} \alpha x & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
Exponential Linear Unit (ELU) [3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for} \quad x < 0 \\ x & \text{for} \quad x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for} \quad x < 0 \\ 1 & \text{for} \quad x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \dfrac{1}{1 + e^{-x}}$

Reference material ⬇

- gfg : https://www.geeksforgeeks.org/activation-functions/?ref=lbp
- article : https://www.v7labs.com/blog/neural-networks-activation-functions#why-do-neural-networks-need-an-activation-function
- Google bard and ChatGPT

```python
import tensorflow as tf
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

1. Binary step function

The binary step activation function, also known as the Heaviside step function, is a fundamental but limited tool in the world of neural networks. Here's what you need to know about it:

How it works:

- Imagine a neuron receiving an input value (which can be any number).
- The binary step function applies a threshold to this input value.
- If the input value is **greater than or equal to the threshold**, the neuron is "activated" and outputs a value of **1**.
- If the input value is **less than the threshold**, the neuron is "deactivated" and outputs a value of **0**.

Think of it like a light switch:

- The input value is like the dimmer switch.
- The threshold is like the point where the light turns on.
- Below the threshold, the light is off (output 0).
- Above the threshold, the light is fully on (output 1).

Properties:

- **Simple and easy to understand:** This is the simplest activation function, making it a good starting point for learning.
- **Non-linear:** Unlike linear functions, the binary step function introduces non-linearity, which is crucial for neural networks to learn complex patterns.
- **Not differentiable at 0:** This creates challenges for training neural networks using gradient-based optimization algorithms.
- **Limited output:** Only outputs two values (0 or 1), making it suitable only for **binary classification** problems.

Limitations:

- Due to its limitations, the binary step function is rarely used in modern neural networks. More sophisticated activation functions, like ReLU or sigmoid, are preferred due to their differentiability and wider range of outputs.

In summary:

The binary step activation function is a basic building block in understanding neural networks, but its limitations make it less practical for real-world applications compared to other, more versatile activation functions.

Binary Step Function
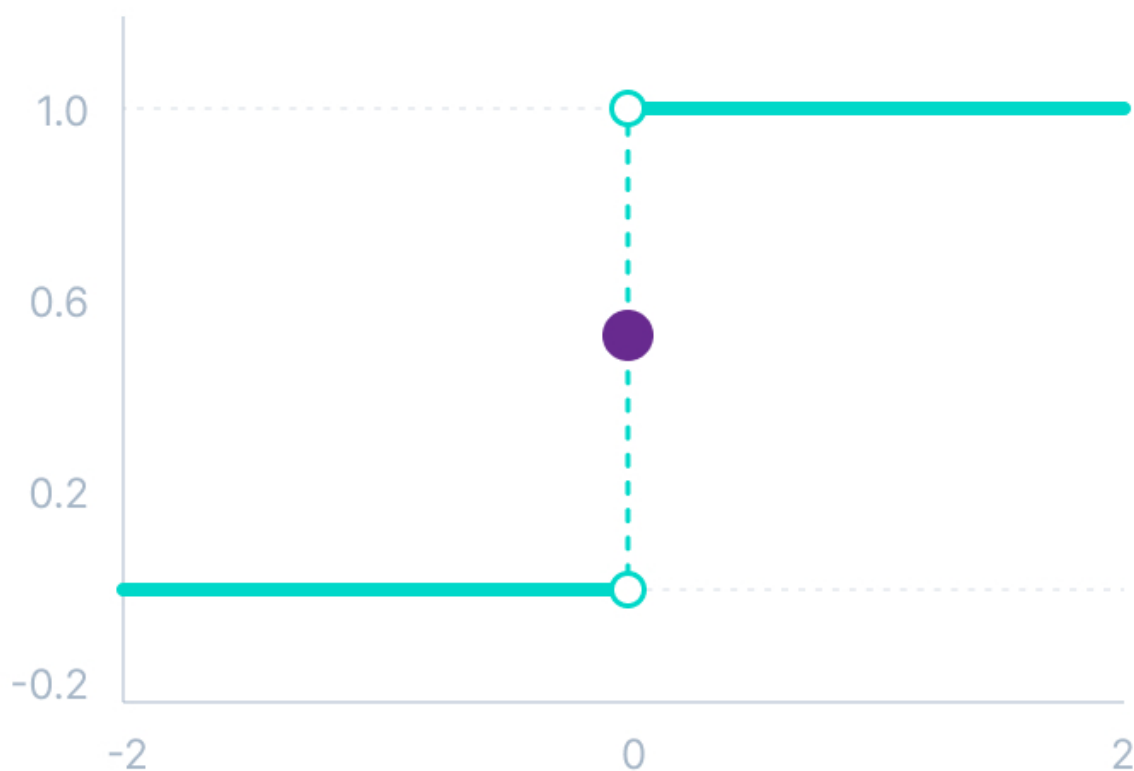
V7 Labs

```
x = tf.constant([-1.0, 0.0, 1.0, 2.0, 3.0 , -7.0 , 10.0 , 5.6 , -0.9])
print("X : " , x.numpy())
# Define the binary step function
def binary_step(x):
    return tf.where(x >= 0, tf.ones_like(x), tf.zeros_like(x))


y = binary_step(x)

print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [0. 1. 1. 1. 1. 0. 1. 1. 0.]
```

2. Linear Activation function

The linear activation function, also known as the identity function or "no activation," is a fundamental concept in neural networks. It's essentially a straight line with a slope of 1 and no intercept, meaning it directly passes the input value to the output without any modification. Here's what you need to know about it:

Function:

- Mathematically, it can be represented as: `f(x) = x`, where x is the input value.
- In simpler terms, the output is simply the input itself, scaled by 1.

Properties:

- **Linear:** As the name suggests, it creates a linear relationship between the input and output. This means any changes in the input are directly reflected in the output.
- **Simple:** It's computationally inexpensive and easy to understand compared to other activation functions.
- **Unbounded:** The output can range from negative infinity to positive infinity, unlike other functions with defined ranges.

Limitations:

- **Limited expressiveness:** Due to its linearity, it cannot learn complex non-linear relationships between the input and output. This significantly limits the types of problems it can solve.
- **Vanishing gradients:** When used in multiple layers with backpropagation, the gradients can become very small or even zero during training, making it difficult for the network to learn effectively.

Use cases:

- Despite its limitations, the linear activation function is still used in some specific scenarios:
    - **Output layer of regression tasks:** When predicting continuous values (e.g., linear regression), a linear output allows for unrestricted predictions across the entire range.
    - **Early layers of some networks:** In certain network architectures, like convolutional neural networks, linear activation may be used in the initial layers for computational efficiency before introducing non-linearity in later layers.

In summary:

The linear activation function is a basic building block of neural networks, offering simplicity and computational efficiency. However, its inherent linearity limits its ability to learn complex relationships and makes it unsuitable for many machine learning tasks. In practice, non-linear activation functions like ReLU, sigmoid, or tanh are preferred for most neural network applications.

Linear Activation Function
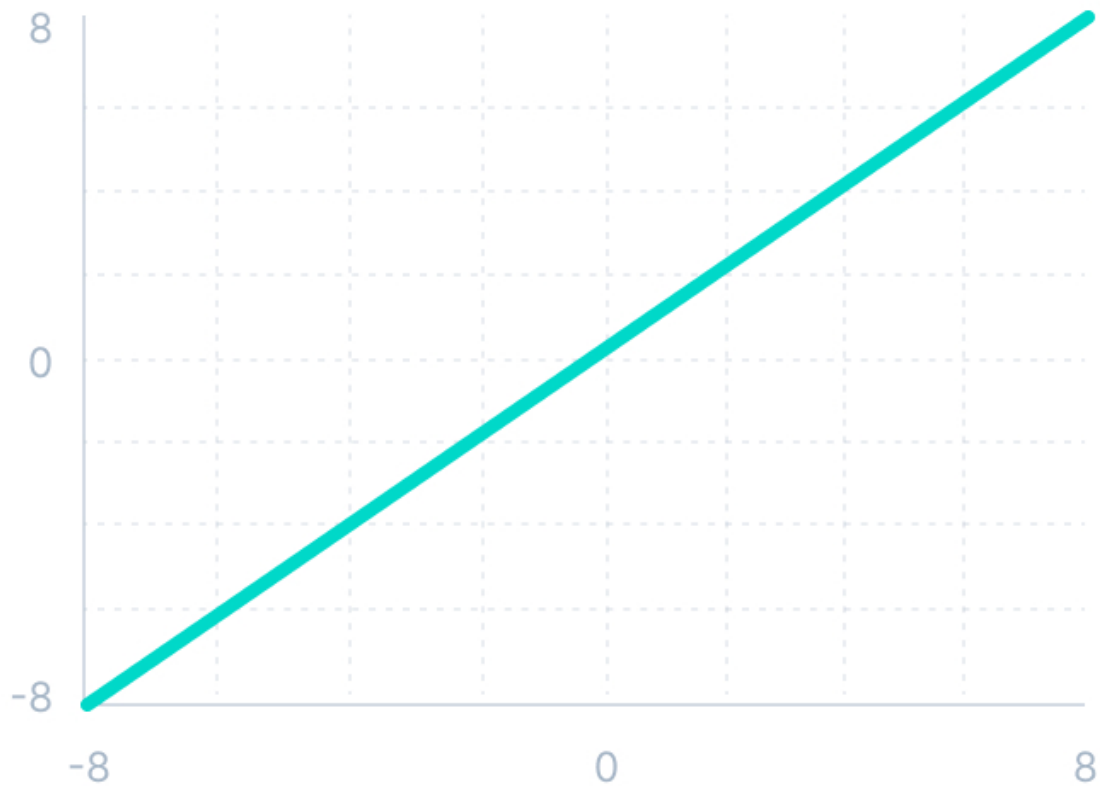
```
print("X : " , x.numpy())
y = tf.keras.activations.linear(x)
print("Y : " , y.numpy())
```

```
X :  [-1.    0.    1.    2.    3.   -7.   10.    5.6 -0.9]
Y :  [-1.    0.    1.    2.    3.   -7.   10.    5.6 -0.9]
```

3. Non-Linear Activation function

Non-linear activation functions solve the following limitations of linear activation functions:

They allow backpropagation because now the derivative function would be related to the input, and it's possible to go back and understand which weights in the input neurons can provide a better prediction. They allow the stacking of multiple layers of neurons as the output would now be a non-linear combination of input passed through multiple layers. Any output can be represented as a functional computation in a neural network.

3.1. Sigmoid/ logistic

The sigmoid, also known as the logistic activation function, is a fundamental function used in artificial neural networks. It plays a crucial role in transforming the weighted sum of inputs within a neuron to an output value. Here's a breakdown of its key characteristics:

Function and Graph:

- Mathematically, the sigmoid function is represented as: `f(x) = 1 / (1 + e^(-x))`.
- Its graph resembles an S-shape, starting near 0 for highly negative inputs, gradually increasing to a plateau near 1 for highly positive inputs, and remaining confined between 0 and 1 for all input values.

Key Properties:

- **Output Range:** The sigmoid function's output is always between 0 and 1, making it suitable for tasks requiring probability-like outputs (e.g., binary classification, where the network predicts the probability of an instance belonging to a particular class).
- **Non-linearity:** This S-shaped curve introduces non-linearity into the network, allowing it to learn complex relationships between inputs and outputs, unlike linear models.
- **Differentiability:** The function is smooth and differentiable, enabling efficient learning through gradient descent optimization algorithms commonly used in training neural networks.

Applications and Limitations:

- **Binary Classification:** Sigmoid excels in problems with two output classes (e.g., spam/not spam, cat/dog), where the output can be interpreted as the probability of belonging to one class.
- **Multi-class Classification:** For problems with more than two classes, a generalization called the softmax function is typically used, which applies the sigmoid concept to each class individually.
- **Limitations:** While effective in its early days, the sigmoid function has some drawbacks:
  - **Vanishing Gradients:** In deep neural networks with many layers, gradients flowing through sigmoid activations can become very small, hindering learning in deeper layers.
  - **Computationally Expensive:** Compared to some newer activation functions, the sigmoid can be computationally slower to evaluate.

Overall:

The sigmoid activation function remains a valuable tool in the neural network toolbox, particularly for understanding the basic concepts of activation functions and their role in introducing non-linearity. However, for complex tasks and deeper networks, more modern activation functions like ReLU or Leaky ReLU are often preferred due to their improved training efficiency and performance.

graph of Sigmoid Function

Sigmoid / Logistic
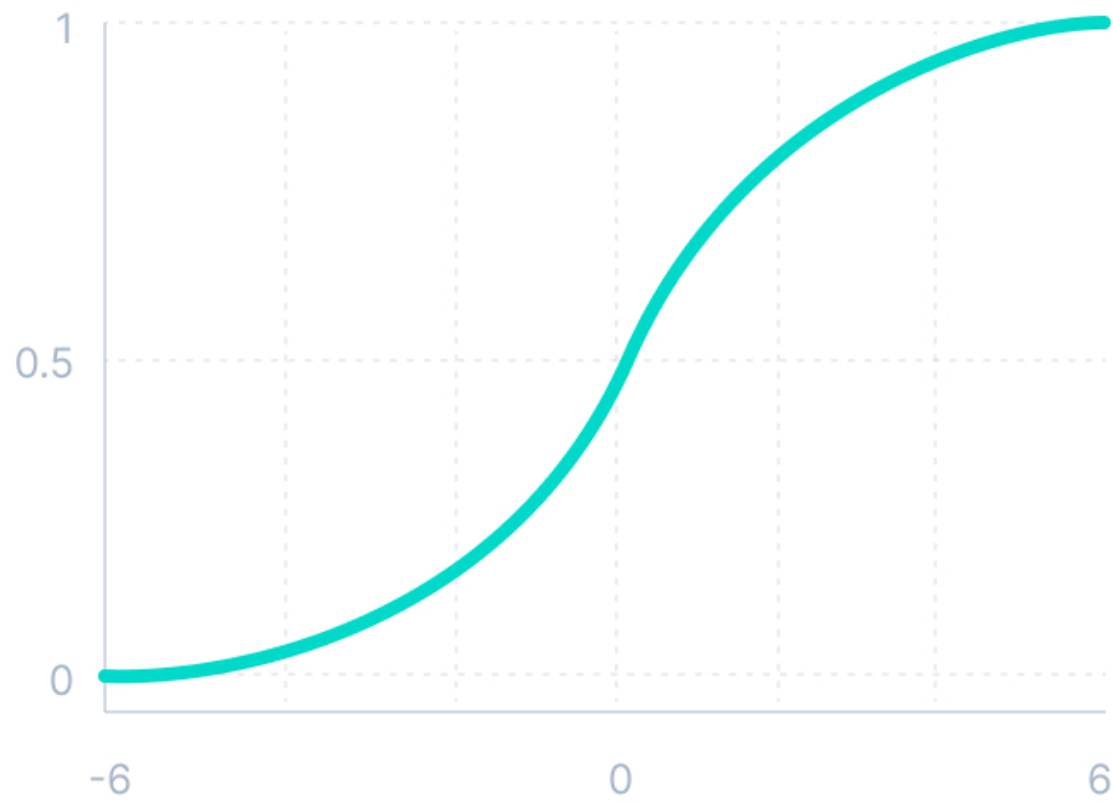
Derivative graph of sigmoid fuction

$$f^1(x) = g(x) = sigmoid(x)*(1-sigmoid(x))$$

V7 Labs
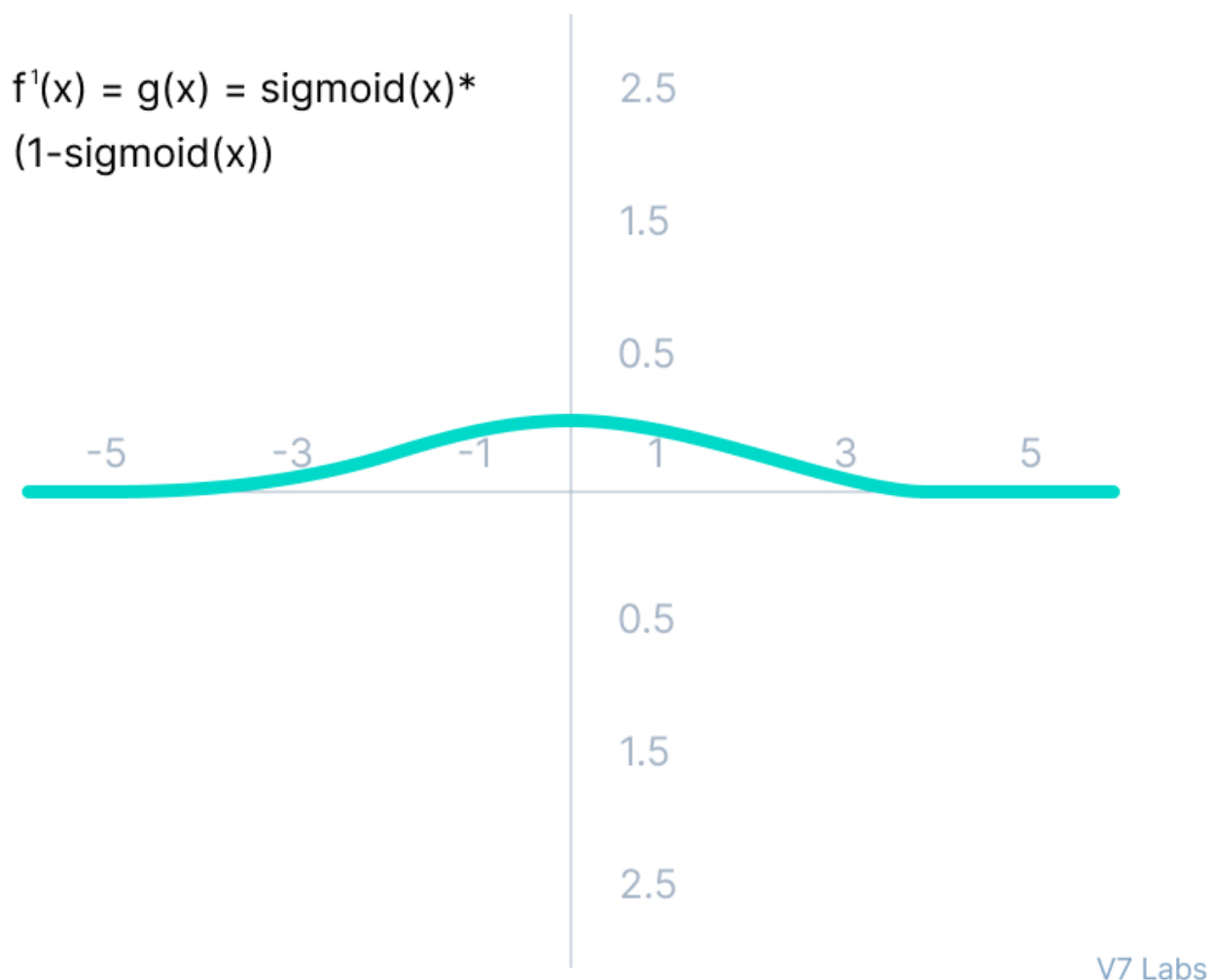
```
print("X : " , x.numpy())
y = tf.keras.activations.sigmoid(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [2.6894143e-01 5.0000000e-01 7.3105860e-01 8.8079709e-01 9.5257413e-01
 9.1105123e-04 9.9995458e-01 9.9631578e-01 2.8905049e-01]
```

3.2. Tanh

The tanh (hyperbolic tangent) activation function is a commonly used activation function in neural networks. Here's a breakdown of its key characteristics and how it works:

Function:

tanh(x) = (e^x - e^(-x)) / (e^x + e^(-x))

Output range: -1 to 1

Visualization: Imagine a bell curve stretched horizontally and squeezed vertically to fit between -1 and 1.

Key characteristics:

- **S-shaped:** Similar to the sigmoid function, tanh has an S-shaped curve, gradually approaching its limits as input values increase or decrease.
- **Zero-centered:** Unlike sigmoid, which outputs values between 0 and 1, tanh outputs values centered around 0, which can be beneficial for some network architectures.
- **Smooth gradients:** Gradients of tanh are smoother than sigmoid, potentially leading to faster and more stable training.

Benefits:

- **Faster convergence:** Zero-centered outputs and smoother gradients can contribute to faster training compared to sigmoid.
- **Improved learning:** Gradients can flow more effectively through the network, aiding learning.
- **Suitable for hidden layers:** The zero-centered nature makes it suitable for hidden layers, as it avoids saturating activations and helps prevent vanishing gradients.

Limitations:

- **Computationally less efficient:** Compared to ReLU, tanh requires more complex calculations, making it slightly slower.
- **Not as interpretable:** The output isn't as easily interpretable as ReLU, where a positive output directly indicates activation.

Applications:

- Often used in hidden layers of neural networks for tasks like:
  - Image recognition (CNNs)
  - Natural language processing (LSTMs, Transformers)
  - Speech recognition
  - And many more

In summary, tanh is a versatile activation function offering advantages like faster convergence and smooth gradients, making it a popular choice for hidden layers in neural networks. However, its computational cost and less interpretable outputs compared to ReLU should be considered.

$$\text{Tanh}(x) = \tanh(x) = \frac{\exp(x) - \exp(-x)}{\exp(x) + \exp(-x)}$$
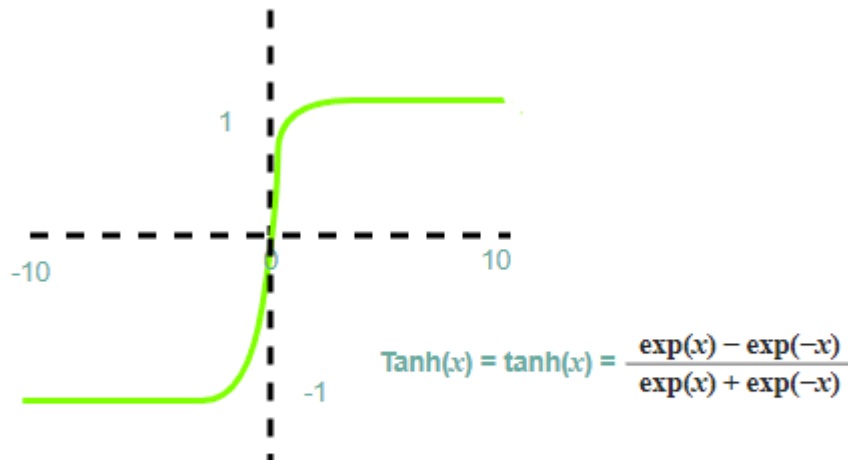
```
print("X : " , x.numpy())
y = tf.keras.activations.tanh(x)
print("Y : " , y.numpy())
```

```
X :  [-1.    0.    1.    2.    3.   -7.   10.    5.6 -0.9]
Y :  [-0.7615942   0.          0.7615942   0.9640276   0.9950547  -0.99999833
  1.          0.99997276 -0.71629786]
```

3.3. RELU

Rectified Linear Unit (ReLU) is a fundamental concept in deep learning, serving as the most widely used activation function due to its simplicity, computational efficiency, and effectiveness in training deep neural networks.

Key characteristics:

- **Piecewise linear:** Represented as f(x) = max(0, x), where for any input x:
    - If x is positive, the output is x itself (unchanged).
    - If x is negative, the output is 0 (rectified to zero).
- **Nonlinearity:** Introduces essential nonlinearity into neural networks, enabling them to learn complex relationships between inputs and outputs that linear models cannot capture.
- **Computationally efficient:** Simple calculation (max(0, x)) makes it faster to compute compared to other activation functions like sigmoid or tanh.
- **Sparsity:** Outputs zeros for negative inputs, which can aid in regularization and reduce overfitting.

Benefits:

- **Faster training:** Efficient computation and sparse outputs contribute to faster training of deep networks.
- **Reduced vanishing gradients:** Unlike sigmoid and tanh, ReLU avoids the vanishing gradient problem, where gradients become very small during backpropagation, making it difficult to train deeper networks.
- **Biological plausibility:** Inspired by the firing behavior of biological neurons, where only positive inputs trigger neuron activation.

Limitations:

- **Dead ReLUs:** Neurons might get stuck in an inactive state with zero output if they consistently receive negative inputs, hindering learning. Variants like Leaky ReLU address this.
- **Not zero-centered:** Can introduce biases into the network's outputs. For tasks requiring symmetric responses to positive and negative inputs, zero-centered activations might be preferred.

Applications:

- Image recognition (CNNs)
- Natural language processing (LSTMs, Transformers)
- Speech recognition
- Reinforcement learning
- And many more deep learning tasks

In summary, ReLU is a powerful and versatile activation function that forms the backbone of countless deep learning models. While it has limitations, its efficiency, nonlinearity, and biological inspiration make it a go-to choice for many applications.
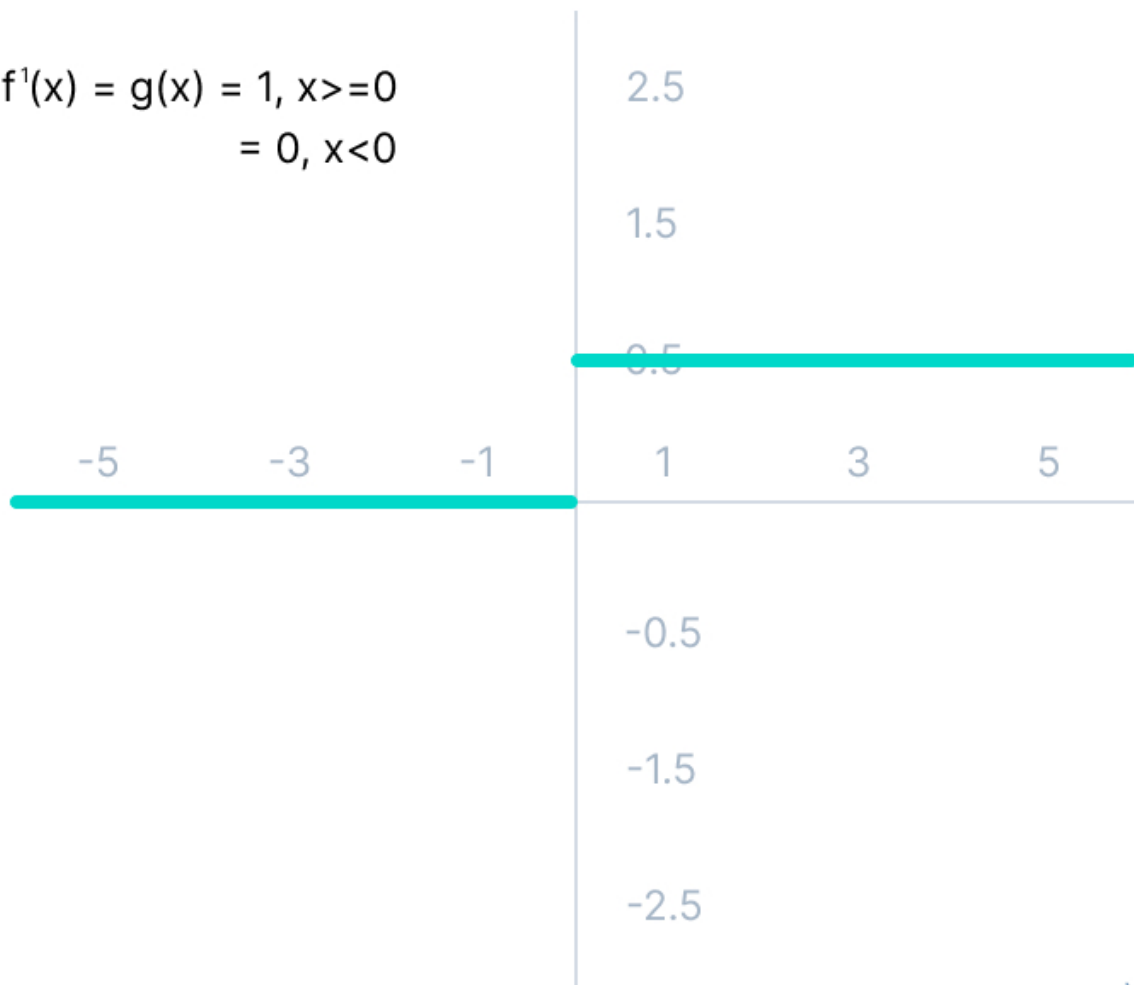
graph of Relu activation function

ReLU
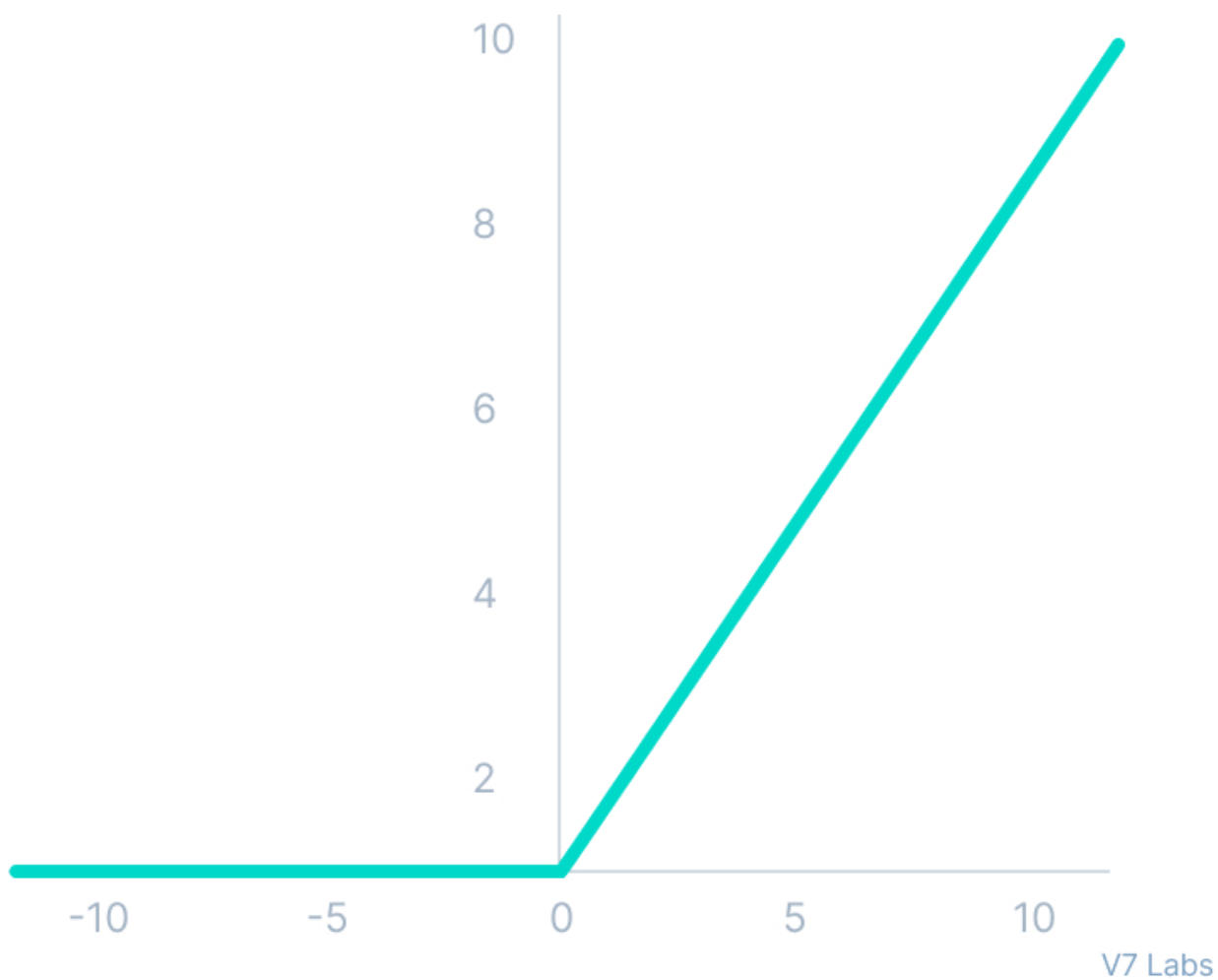
Derivative of Relu

The Dying ReLU problem

$f^1(x) = g(x) = 1, x>=0$
$= 0, x<0$

2.5

1.5

0.5

-5   -3   -1   1   3   5

-0.5

-1.5

-2.5

V7 Labs

```
print("X : " , x.numpy())
y = tf.keras.activations.relu(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [ 0.   0.   1.   2.   3.   0.  10.   5.6  0. ]
```

3.4. Leaky RELU / parametric RELU

Both Leaky ReLU and Parametric ReLU are activation functions that build upon the Rectified Linear Unit (ReLU) by addressing its limitations. Here's a breakdown of each:

Leaky ReLU:

- **Function:** f(x) = max(α * x, x) for x < 0 and f(x) = x for x >= 0, where α is a small positive constant (typically 0.01).
- **Intuition:** Instead of completely zeroing out negative inputs like ReLU, Leaky ReLU allows a small, non-zero gradient to flow through them. This helps prevent "dying ReLUs" where neurons become permanently inactive due to negative inputs.
- **Benefits:**
  - Addresses the "dying ReLU" problem.
  - Maintains the simplicity and efficiency of ReLU.
  - Can improve the stability and speed of training.
- **Limitations:**
  - Still susceptible to vanishing gradients for very negative inputs.
 - The choice of α remains a hyperparameter to be tuned.

Parametric ReLU (PReLU):

- **Function:** f(x) = max(α_i * x, x) for x < 0 and f(x) = x for x >= 0, where α_i is a learnable parameter specific to each channel or feature in the network.
- **Intuition:** PReLU takes Leaky ReLU a step further by making the α parameter learnable during training. This allows the network to automatically adjust the slope for negative inputs based on the specific data and task.
- **Benefits:**
  - More flexible than Leaky ReLU as it can adapt the slope to different features.
  - Can potentially improve performance compared to Leaky ReLU with a fixed α.
- **Limitations:**
  - Introduces additional parameters to the network, increasing complexity.
  - May require more careful tuning and hyperparameter optimization.

Choosing the Right Activation:

- **ReLU:** A good starting point for many tasks due to its simplicity and efficiency.
- **Leaky ReLU:** Preferred when dealing with negative inputs or preventing "dying ReLUs."
- **PReLU:** Consider if you need more flexibility in adapting the slope for different features, but be aware of the added complexity.

Ultimately, the best choice depends on your specific data, network architecture, and desired performance. Experimenting with different activation functions can help you find the optimal one for your task.

$f(v)$

$f(v) = y$

y

$f(v) = ay$
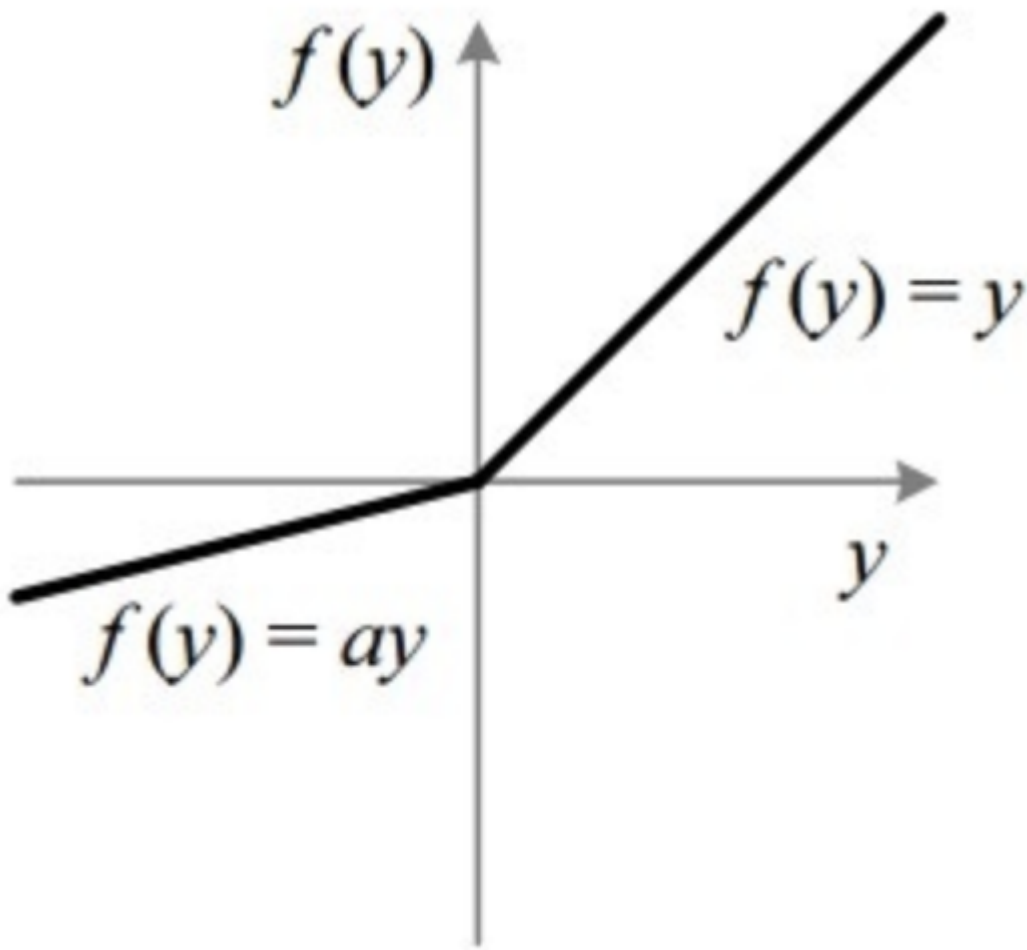
```
print("X : " , x.numpy())
y = tf.keras.activations.relu(x,alpha=0.1)
print("Y : " , y.numpy())
```

```
X :  [-1.    0.    1.    2.    3.   -7.   10.    5.6 -0.9]
Y :  [-0.1   0.    1.    2.    3.   -0.7  10.    5.6  -0.09]
```

3.5. ELU

The ELU (Exponential Linear Unit) activation function is a popular choice in deep learning due to its advantages over ReLU and its variants. Here's a breakdown of its key characteristics:

Function:

f(x) = { α * (exp(x) - 1) for x < 0 x for x >= 0 }

where α is a hyperparameter typically set to 1.

Key characteristics:

- **Smooth and continuous:** Unlike ReLU's sharp transition at zero, ELU transitions smoothly, avoiding issues like dead ReLUs and improving gradient flow.
- **Mean shift towards zero:** Similar to Leaky ReLU, ELU allows negative outputs, which helps center activations around zero. This can improve training speed and stability.
- **Non-saturating:** Unlike sigmoid and tanh, ELU doesn't saturate for large positive inputs, preventing vanishing gradients in deeper layers.
- **Computationally efficient:** Similar to ReLU, ELU is faster to compute than sigmoid or tanh.

Benefits:

- **Faster training:** Smooth gradient flow and centered activations can lead to faster convergence during training.
- **Reduced vanishing gradients:** Non-saturating nature helps mitigate vanishing gradients in deeper layers.
- **Biological plausibility:** Similar to ReLU, ELU shares some features with biological neuron behavior.

Limitations:

- **Hyperparameter tuning:** The α parameter needs to be chosen carefully, unlike ReLU's simplicity.
- **Less interpretable:** Compared to ReLU, the output isn't as easily interpretable due to its exponential nature.

Applications:

- Similar to ReLU, ELU is widely used in deep learning tasks like:
  - Image recognition (CNNs)
  - Natural language processing (LSTMs, Transformers)
  - Speech recognition
  - And many more

In summary, ELU offers a compelling alternative to ReLU, addressing some of its limitations while maintaining computational efficiency. Its smooth behavior, non-saturating nature, and ability to center activations make it a popular choice for various deep learning tasks.
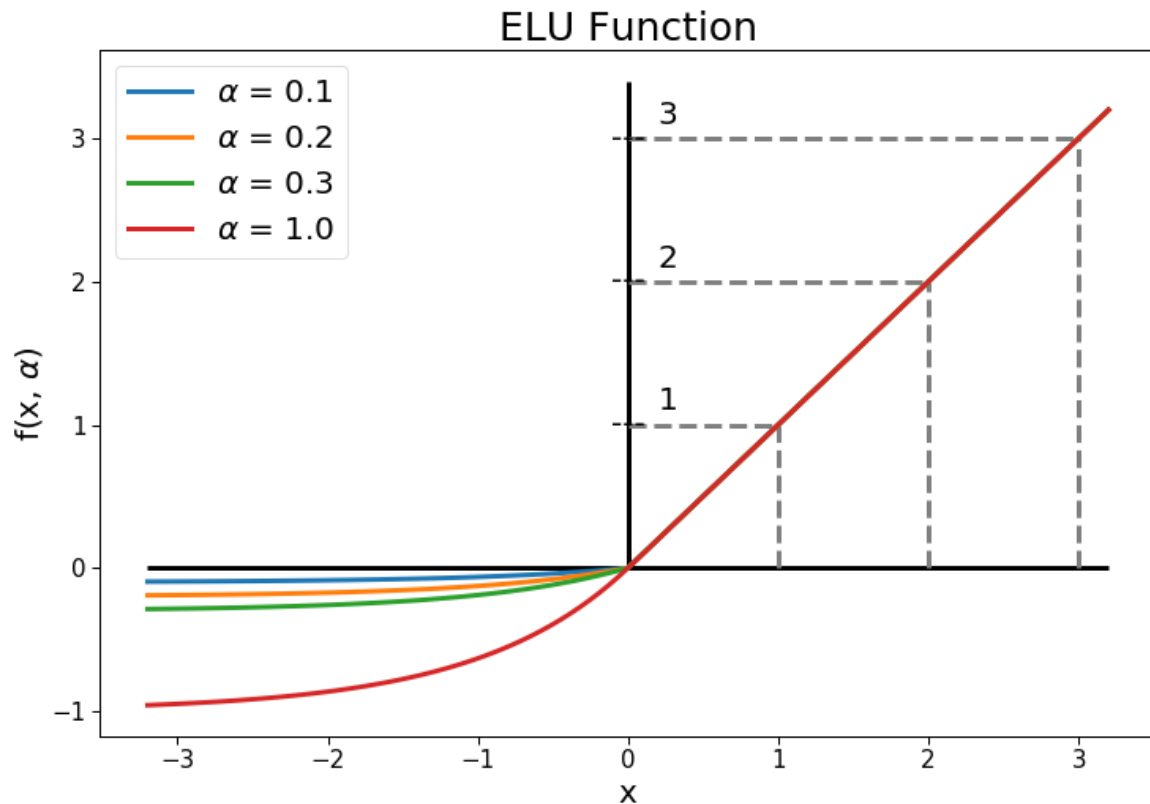
ELU Function

```
print("X : " , x.numpy())
y = tf.keras.activations.elu(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [-0.63212055  0.          1.          2.          3.          -0.9990881
 10.          5.6        -0.59343034]
```

3.6. Softmax

The softmax activation function plays a crucial role in neural networks, especially for **multi-class classification** tasks. Here's a breakdown of its workings and importance:

Function:

Imagine you have a vector of real numbers representing the "raw" outputs of a neural network layer. Softmax transforms this vector into a **probability distribution** across all possible classes. Here's the formula:

```
softmax(z_i) = e^(z_i) / Σ_j e^(z_j)
```

where:

- `z_i` is the input value for class `i` .
- `e^(z_i)` is the exponentiation of `z_i` .
- Σ_j e^(z_j) is the sum of exponentiations of all input values.

Key takeaways:

- **Probabilities sum to 1:** Each output value represents the probability of the input belonging to a specific class. All probabilities add up to 1, ensuring they form a valid probability distribution.
- **Interpretability:** Softmax provides clear interpretations as probabilities, making it easier to understand the network's predictions.
- **Multi-class:** Unlike sigmoid used for binary classification, softmax is designed for tasks with multiple possible classes.

Benefits:

- **Intuitive output:** Probabilities are readily understandable and interpretable.
- **Confident predictions:** The highest probability indicates the most likely class, while lower values represent decreasing confidence.
- **Widely used:** Softmax is the standard activation function for multi-class classification tasks in deep learning.

Limitations:

- **Not directly comparable:** Probabilities from different inputs cannot be directly compared, as they are relative within their own input vectors.
- **Numerical stability:** For large input values, exponentiation can lead to numerical overflow issues.

Applications:

- Image recognition (classifying images into multiple categories)
- Natural language processing (sentiment analysis, text classification)
- Speech recognition (identifying different speakers or commands)
- And many more multi-class classification tasks

In summary, the softmax activation function is a powerful tool for interpreting and understanding the predictions of neural networks in multi-class classification problems. Its ability to transform raw outputs into clear probabilities makes it a valuable asset in various deep learning applications.

$$softmax(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{N} e^{z_j}}$$

3.7. GELU

The GELU activation function, also known as the "Gaussian Error Linear Unit," is a relatively new but increasingly popular activation function in deep learning. It combines some of the desirable properties of other commonly used activations like ReLU and ELU, making it a compelling choice for many tasks.

Here's what you need to know about GELU:

Function:

f(x) = x * Φ(x), where Φ(x) is the standard Gaussian cumulative distribution function (CDF).

Key characteristics:

- **Smooth and continuous:** Like ELU, GELU offers a smooth transition around zero, avoiding dead ReLU problems and improving gradient flow.
- **Non-negative:** Similar to ReLU, GELU only outputs non-negative values, which can be beneficial for certain tasks.
- **Non-saturating:** As with ELU, GELU doesn't saturate for large positive inputs, mitigating vanishing gradients.
- **Computationally efficient:** Although slightly more complex than ReLU, GELU remains efficient compared to sigmoid or tanh.

Benefits:

- **Fast training:** The smooth gradient flow and non-saturating nature can lead to faster convergence during training.
- **Reduced vanishing gradients:** This is advantageous for training deeper neural networks.

- **Potentially better performance:** Compared to ReLU and ELU, GELU has shown competitive or even superior performance in some tasks like natural language processing (NLP) and speech recognition.

Limitations:

- **Slightly more complex:** Requires calculating the Gaussian CDF, which adds a bit of computational overhead compared to ReLU.
- **Less interpretable:** The output is less straightforward to interpret due to the Gaussian CDF component.

Applications:

- GELU is increasingly being used in various deep learning tasks, including:
    - NLP (language models, machine translation)
    - Speech recognition
    - Computer vision (image classification, object detection)
    - And more

In summary, GELU offers a balance between smoothness, non-negativity, computational efficiency, and performance, making it a valuable choice for many deep learning applications. While it might not be the simplest option, its potential benefits are attracting growing interest within the field.

$$g = 0.5x \left(1 + \frac{2}{\sqrt{\pi}} \int_0^{\frac{x}{\sqrt{2}}} e^{-t^2} dt \right)$$
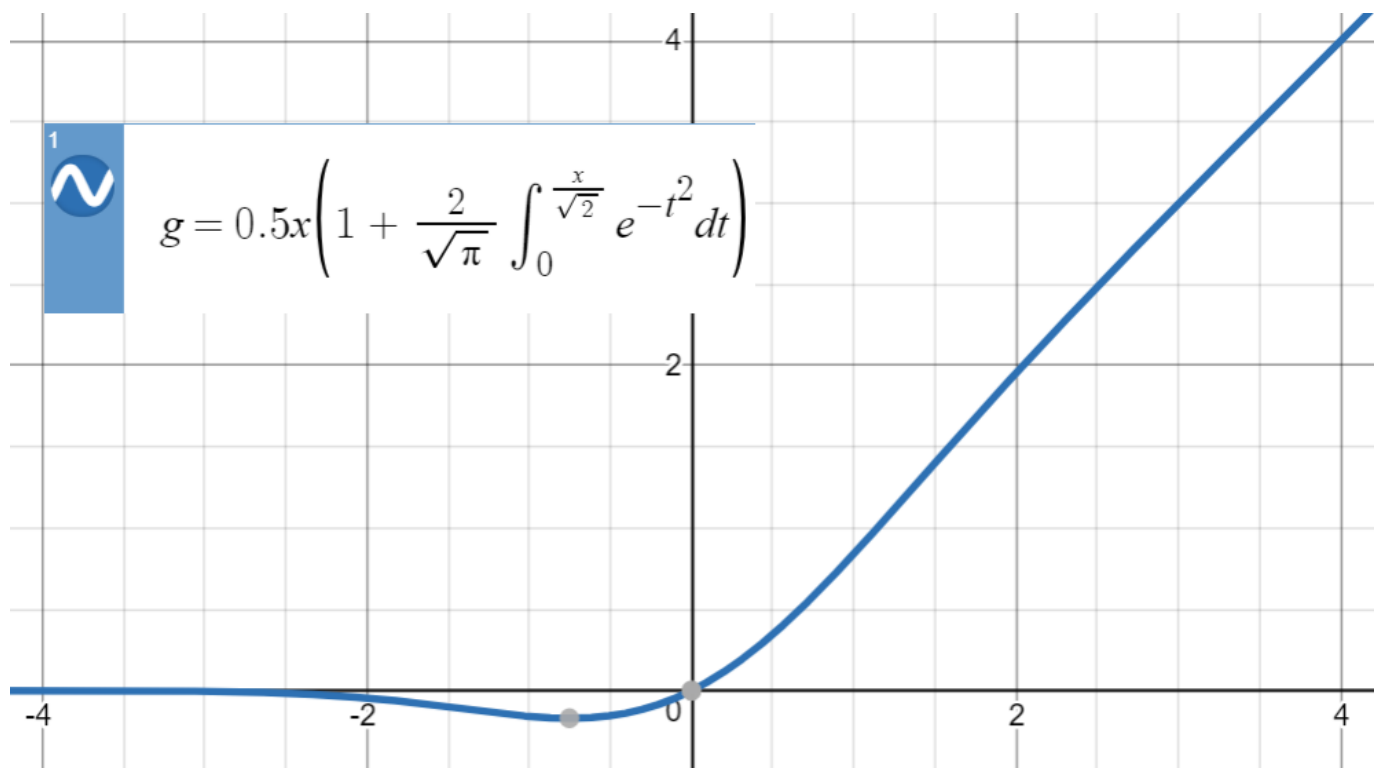
```
print("X : " , x.numpy())
y = tf.keras.activations.gelu(x)
print("Y : " , y.numpy())
```

```
X : [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y : [-0.15865526  0.          0.8413447   1.9544997   2.9959502  -0.
 10.          5.6         -0.16565411]
```

3.8. SELU

The SELU (Scaled Exponential Linear Unit) activation function is another interesting option in the world of neural network activations, building upon the strengths of both ELU and ReLU while addressing some of their limitations. Here's a rundown of its key features:

Function:

f(x) = { λ * α * (exp(x) - 1) for x < 0 λ * x for x >= 0 }

where:

- α ≈ 1.6733
- λ ≈ 1.0507

Key characteristics:

- **Self-normalizing:** A unique feature of SELU is its ability to automatically normalize the activations in a network to have a mean of zero and a variance of one. This can improve training speed and stability, potentially reducing the need for explicit batch normalization layers.
- **Smooth and continuous:** Similar to ELU, SELU has a smooth transition at zero, avoiding dead ReLUs and improving gradient flow.
- **Non-saturating:** Like ELU, it avoids saturating for large positive inputs, mitigating vanishing gradients.
- **Computationally efficient:** Like ReLU and ELU, it's fast to compute.

Benefits:

- **Faster training:** Self-normalization can lead to faster convergence and potentially reduce the need for batch normalization.
- **Reduced vanishing gradients:** Non-saturating nature helps prevent vanishing gradients in deep networks.
- **Stable training:** Smooth gradient flow and centered activations can contribute to more stable training.

Limitations:

- **Fixed hyperparameters:** Unlike ReLU, its hyperparameters (α and λ) are fixed, reducing flexibility.
- **Less interpretable:** Similar to ELU, its output is less easily interpretable than ReLU due to the exponential component.

Applications:

- SELU is well-suited for various deep learning tasks, particularly where faster training and stable gradients are crucial, like:
    - Image recognition (CNNs)
    - Natural language processing (LSTMs, Transformers)
    - Speech recognition
    - And more

In conclusion, SELU offers a compelling alternative with its self-normalizing property and smooth behavior, potentially leading to faster and more stable training. While it lacks the full flexibility of Leaky ReLU or PReLU, it can be a strong contender in many deep learning applications.
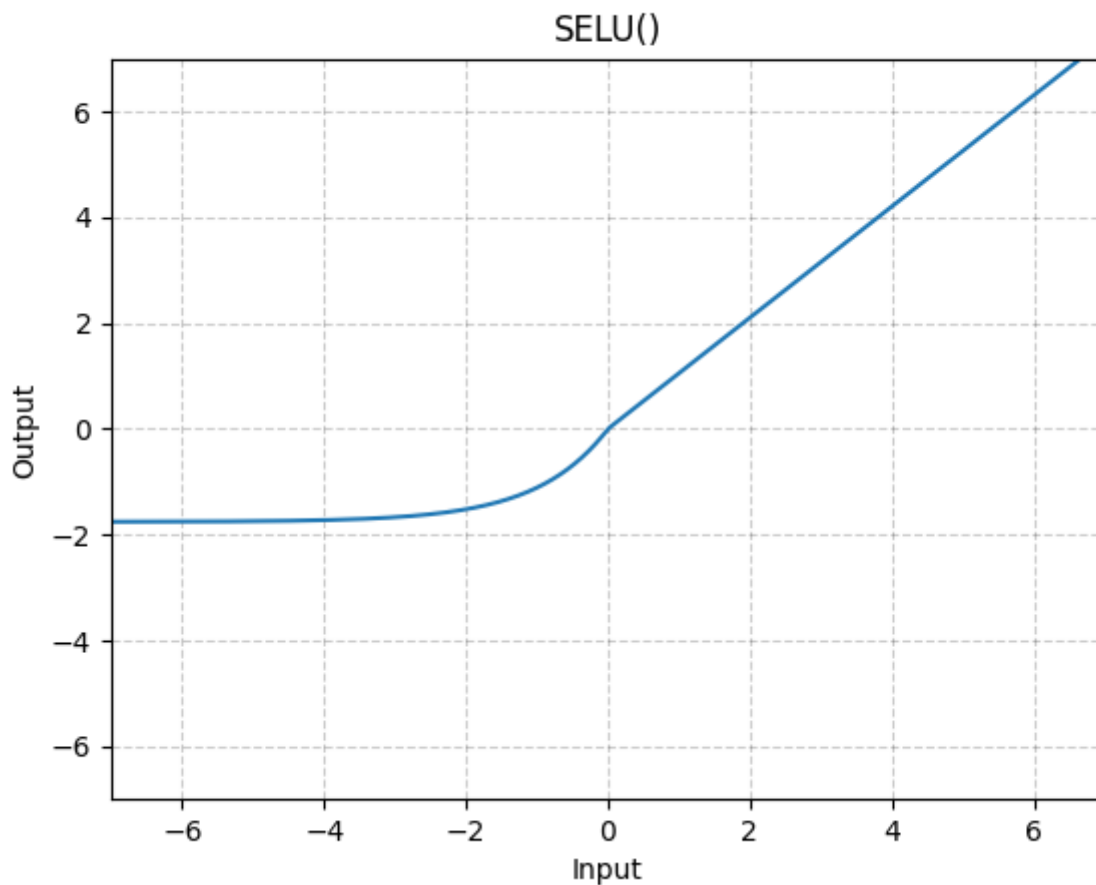
SELU()


```
print("X : " , x.numpy())
y = tf.keras.activations.selu(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [-1.1113307  0.          1.050701   2.101402   3.152103  -1.7564961
 10.50701    5.8839254 -1.0433095]
```

3.9. Swish

The Swish activation function, also known as **SiLU** (Smoothly Clipped Linear Unit), is a recent addition to the toolbox of deep learning practitioners. It offers several advantages over commonly used activation functions like ReLU, making it a popular choice for various tasks. Here's a breakdown of its key characteristics:

Function:

f(x) = x * sigmoid(β * x)

where β is typically set to 1, but can be a learnable parameter. When β = 1, Swish becomes identical to SiLU.

Key characteristics:

- **Smooth and non-monotonic:** Unlike ReLU's sharp transition at zero, Swish has a smooth, S-shaped curve that gradually increases towards a positive value. This avoids dead ReLUs and improves gradient flow.
- **Non-zero mean:** Similar to ELU, Swish allows negative outputs, which helps center activations around zero. This can improve training speed and stability.
- **Bounded:** Swish is bounded above (around 1) and below (around -1), preventing exploding gradients and vanishing gradients.
- **Computationally efficient:** Similar to ReLU, Swish is faster to compute than sigmoid or tanh.

Benefits:

- **Faster training:** Smooth gradient flow and centered activations can lead to faster convergence during training.
- **Reduced vanishing gradients:** Bounded nature helps mitigate vanishing gradients in deeper layers.
- **Potentially better performance:** In some tasks, Swish has been shown to outperform ReLU and other activation functions.

Limitations:

- **Hyperparameter tuning:** If β is learnable, it adds an extra parameter to tune.
- **Less interpretable:** Compared to ReLU, the output isn't as easily interpretable due to its smooth and non-monotonic nature.

Applications:

- Similar to ReLU and ELU, Swish is used in various deep learning tasks like:
    - Image recognition (CNNs)
    - Natural language processing (LSTMs, Transformers)
    - Speech recognition
    - And many more

Overall, Swish is a promising activation function with a good balance of smoothness, non-monotonicity, boundedness, and computational efficiency. While it introduces some complexity with

potential hyperparameter tuning, its benefits have led to its increasing adoption in deep learning models.
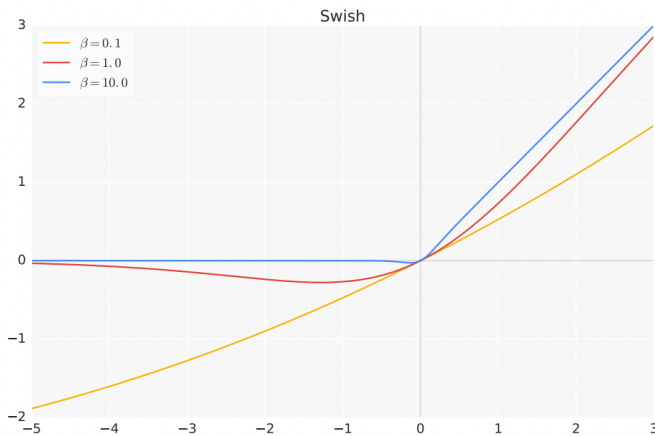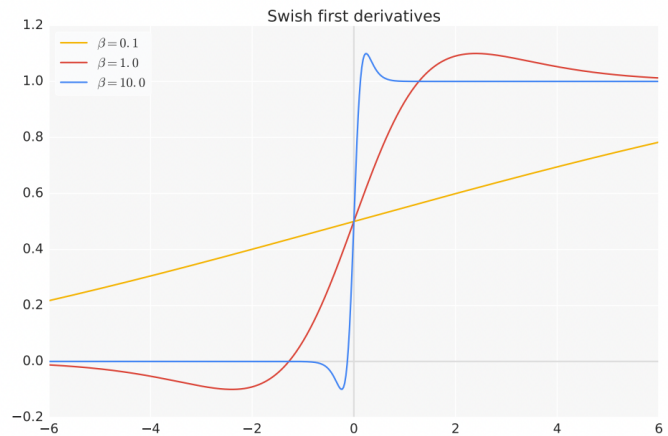

Figure 4: The Swish activation function.

Figure 5: First derivatives of Swish.

```
print("X : " , x.numpy())
y = tf.keras.activations.swish(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [-2.6894143e-01  0.0000000e+00  7.3105860e-01  1.7615942e+00
  2.8577223e+00 -6.3773585e-03  9.9995461e+00  5.5793681e+00
 -2.6014543e-01]
```

3.10. Softplus

The softplus activation function, also known as **smooth ReLU**, is a popular choice in deep learning due to its smooth, non-linear nature and several advantages over traditional options like ReLU. Here's a breakdown of its key characteristics:

Function:

$f(x) = \ln(1 + \exp(x))$

Key characteristics:

- **Smooth and continuous:** Unlike ReLU's sharp transition at zero, softplus transitions smoothly, avoiding issues like dead ReLUs and improving gradient flow.
- **Non-negative outputs:** Always produces positive outputs, which can be beneficial for tasks requiring positive values like probability distributions or variances.
- **Non-saturating:** Similar to ELU, softplus doesn't saturate for large positive inputs, preventing vanishing gradients in deeper layers.

- **Computationally efficient:** While not as fast as ReLU, softplus is still relatively efficient due to the use of logarithmic and exponential functions.

Benefits:

- **Faster training:** Smooth gradient flow can lead to faster convergence during training compared to ReLU.
- **Reduced vanishing gradients:** Non-saturating nature helps mitigate vanishing gradients in deeper layers.
- **Positive outputs:** Suitable for tasks requiring non-negative values like variances or probabilities.
- **Biological plausibility:** Shares similarities with the firing behavior of biological neurons.

Limitations:

- **Not zero-centered:** Outputs are always positive, which might not be ideal for tasks requiring symmetric responses to positive and negative inputs.
- **Less interpretable:** Compared to ReLU, the output behavior is less interpretable due to the logarithmic function involved.

Applications:

- Softplus is often used in tasks involving positive values, such as:
  - Variational inference
  - Reinforcement learning (e.g., actor networks)
  - Generative models
  - And other deep learning applications where smooth, non-negative activations are desired

In summary, the softplus activation function offers a smooth, non-saturating alternative to ReLU with the benefit of always producing positive outputs. While it has limitations in interpretability and zero-centering, its advantages make it a valuable choice for specific applications.

Softplus function
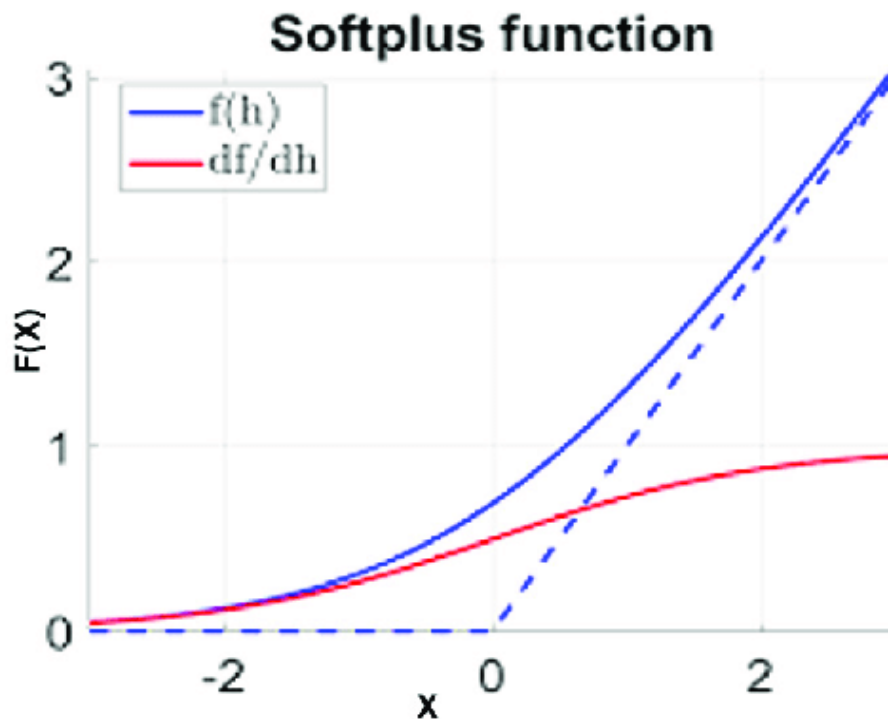
```
print("X : " , x.numpy())
y = tf.keras.activations.softplus(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [3.1326169e-01 6.9314718e-01 1.3132616e+00 2.1269281e+00 3.0485871e+00
 9.1146637e-04 1.0000046e+01 5.6036911e+00 3.4115386e-01]
```

3.11. Softsign

The softsign activation function is an alternative to the more commonly used ReLU or tanh functions in neural networks. It shares some similarities with both, but also has its own unique properties. Here's a breakdown:

Function:

$f(x) = x / (1 + |x|)$

Key characteristics:

- **Range:** Outputs are between -1 and 1, similar to tanh.

- **Smooth and continuous:** Unlike ReLU's sharp transition at zero, softsign transitions smoothly, avoiding issues like dead ReLUs.
- **Non-saturating:** Similar to ELU, softsign doesn't saturate for large positive or negative inputs, preventing vanishing gradients in deeper layers.
- **Zero-centered:** Outputs are centered around zero, which can improve training speed and stability.
- **Computationally efficient:** Similar to ReLU and ELU, softsign is faster to compute than sigmoid or tanh.

Benefits:

- **Smooth gradient flow:** Its smooth transition helps alleviate vanishing gradients in deep networks.
- **Zero-centered outputs:** This can improve training speed and stability.
- **Non-saturating:** Prevents vanishing gradients in deeper layers.

Limitations:

- **Less interpretable:** Compared to ReLU, the output isn't as easily interpretable due to its non-linear nature.
- **Less common:** Not as widely used as ReLU or tanh, so finding resources and best practices might be harder.

Applications:

- While not as widely used as other activations, softsign can be suitable for tasks where:
  - Smooth gradient flow is crucial (e.g., deep networks).
  - Zero-centered outputs are beneficial (e.g., avoiding bias towards positive or negative values).
  - Computational efficiency is important.

In summary, softsign offers a smooth, non-saturating, and zero-centered activation function that can be useful in specific scenarios. However, its less common use and slightly lower interpretability compared to other options like ReLU or ELU should be considered when choosing an activation function for your neural network.
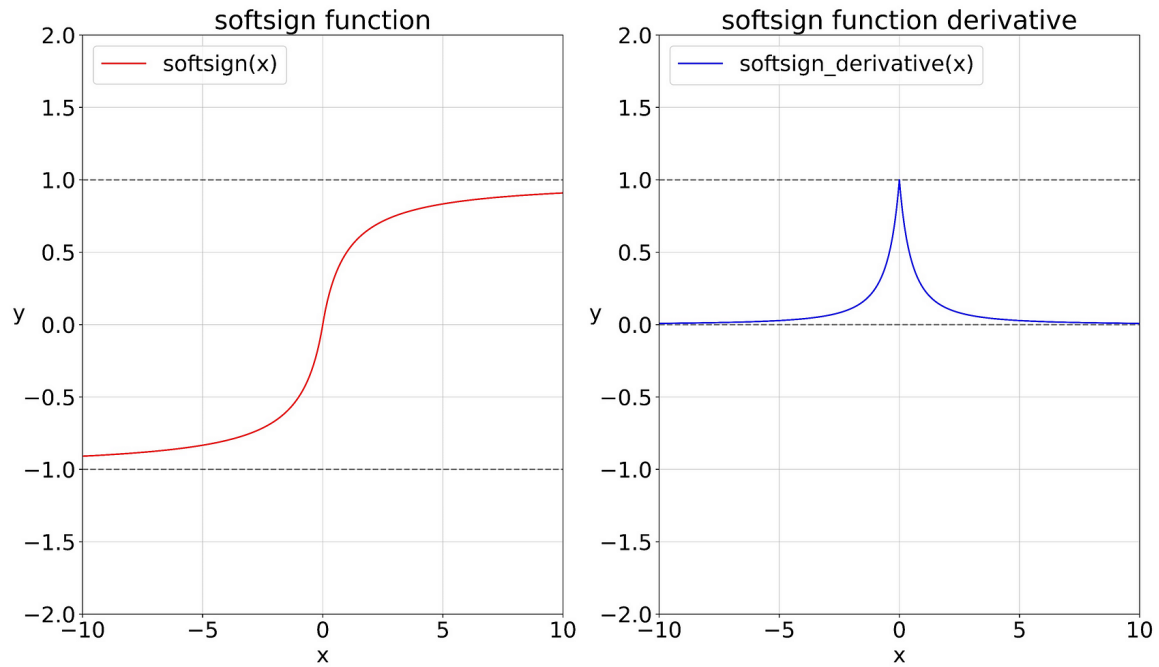
```
print("X : " , x.numpy())
y = tf.keras.activations.softsign(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [-0.5         0.          0.5         0.6666667   0.75       -0.875
  0.90909094  0.8484849  -0.4736842 ]
```

3.12. Mish

The Mish activation function, introduced in 2019, is a relatively new addition to the toolkit of deep learning practitioners. It combines several desirable properties, making it a promising choice for various tasks. Here's a breakdown:

Function:

Mish(x) = x * tanh(ln(1 + exp(x)))

Key characteristics:

- **Smooth and continuous:** Like ELU, Mish has a smooth transition at zero, avoiding dead neuron issues and improving gradient flow.
- **Non-monotonic:** Unlike ReLU and ELU, Mish has a small dip below zero, allowing it to preserve some negative information and potentially improve performance on tasks where negative inputs are relevant.

- **Unbounded above, bounded below:** Similar to ReLU, Mish outputs approach but never reach positive infinity, preventing saturation and vanishing gradients. It has a lower bound of around -0.31.
- **Self-regularization:** The built-in `tanh` component is believed to have a self-regularizing effect, potentially reducing overfitting.

Benefits:

- **Faster training:** Smoothness and non-monotonicity can contribute to faster training compared to ReLU and its variants.
- **Improved performance:** In some cases, Mish can lead to better accuracy compared to other activation functions.
- **Potentially less prone to overfitting:** Self-regularization property might reduce overfitting issues.

Limitations:

- **Relatively new:** Less research and practical experience compared to established options like ReLU.
- **Slightly more complex computation:** Requires more calculations than ReLU but still faster than sigmoid or tanh.

Applications:

- Mish is gaining traction in various tasks, including:
    - Image recognition (CNNs)
    - Natural language processing (LSTMs, Transformers)
   - Speech recognition
    - And more

Summary:

Mish is a promising activation function with interesting properties like smoothness, non-monotonicity, self-regularization, and potential performance improvements. While still relatively new, its characteristics make it worth considering for your deep learning projects. Remember to evaluate its suitability for your specific task and compare it to other options based on your needs.
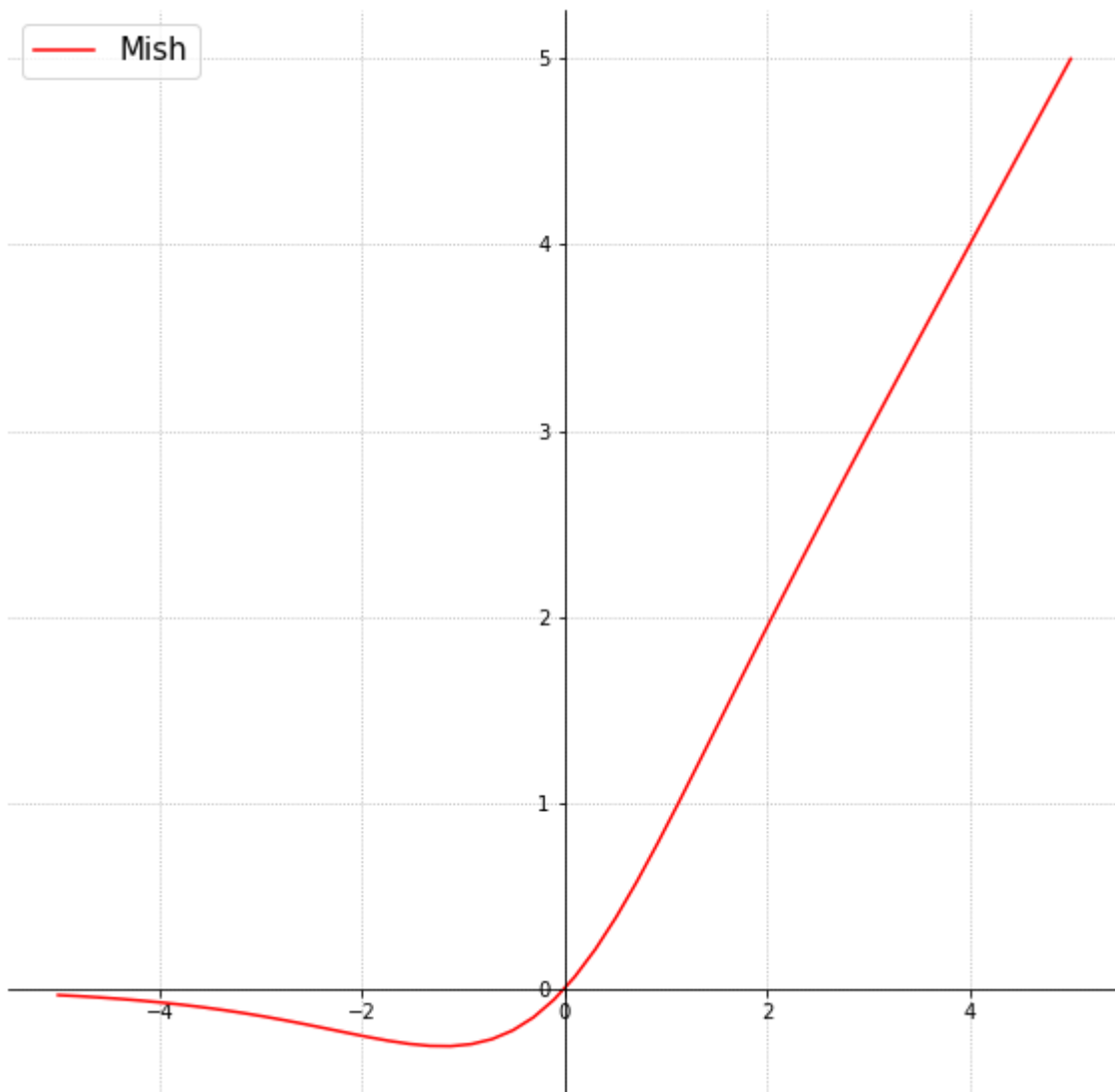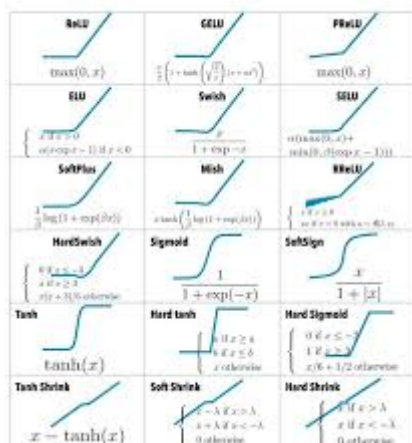
```
print("X : " , x.numpy())
y = tf.keras.activations.mish(x)
print("Y : " , y.numpy())
```

```
X :  [-1.   0.   1.   2.   3.  -7.  10.   5.6 -0.9]
Y :  [-3.0340144e-01  0.0000000e+00  8.6509836e-01  1.9439590e+00
  2.9865348e+00 -6.3802623e-03  1.0000000e+01  5.5998483e+00
 -2.9565641e-01]
```

Additional Graphs

Common Activation Functions / Common Activation Functions Derivative
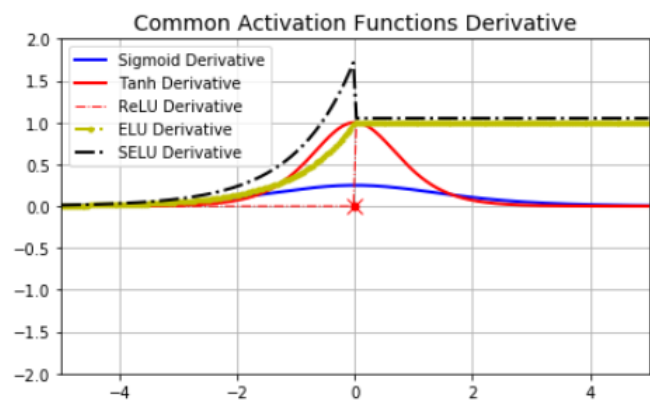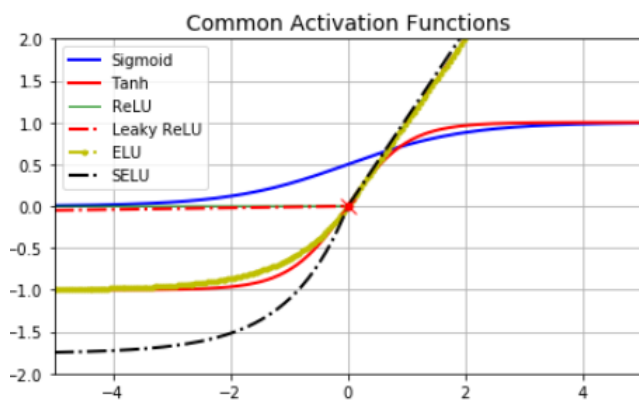
Neural Network Activation Functions: a small subset!

Some Important Questions

4. How to choose the right Activation Function?

Choosing the right activation function for your neural network can be crucial for its performance. Here's a breakdown of factors to consider:

1. Problem Type:

- **Regression:** For continuous output values, use a linear activation function (e.g., identity) to maintain the data's original scale.
- **Classification:** Use functions that output probabilities between 0 and 1, such as sigmoid for single-class or softmax for multi-class problems.

2. Network Architecture:

- **Hidden Layers:** ReLU with its simplicity and computational efficiency is the default choice for most hidden layers. Leaky ReLU or ELU can address "dying neurons" issues.
- **Output Layer:** Adapt based on the problem type (e.g., sigmoid for single-class, softmax for multi-class).

3. Gradient Flow:

- Avoid activation functions that saturate (e.g., sigmoid, tanh) for deeper networks, as vanishing gradients hinder learning. ReLU, Leaky ReLU, ELU, and parametric options promote better gradient flow.

4. Performance and Interpretability:

- Experiment with different options and compare their impact on model performance (accuracy, loss).
- Consider the interpretability of outputs if needed (e.g., ReLU's binary nature makes it easier to interpret compared to ELU's smoothness).

5. Computational Cost:

- If efficiency is critical, prioritize ReLU and its variants due to their simple calculations.

Additional Tips:

- Start with common choices like ReLU for hidden layers and softmax for the output layer, then fine-tune based on your specific problem.
- Consider using parametric activation functions (e.g., PReLU) if you need more flexibility in adjusting behavior for different features.
- Don't hesitate to experiment and compare different options to find the one that works best for your case.

Remember, there's no single "best" activation function. The optimal choice depends on your specific data, network architecture, and desired outcomes.

5. Problems

While activation functions are crucial for the success of neural networks, they do come with some inherent challenges and potential drawbacks. Here are some key problems associated with activation functions:

Vanishing Gradients: This problem occurs when gradients become very small or zero as they flow through the network during backpropagation, making it difficult to update weights in earlier layers and hindering learning. This is particularly problematic with functions like sigmoid and tanh that saturate (reach a constant value) for large inputs.

Dying ReLUs: This issue specifically affects the ReLU activation function, where neurons remain permanently inactive ("dead") due to consistently receiving negative inputs, reducing the network's effective capacity. While Leaky ReLU and ELU address this, they introduce other complexities.

Non-zero Mean Outputs: Some activation functions (e.g., ReLU) can introduce a bias by shifting the average output towards positive values. This can affect the learning process and performance in tasks requiring symmetric behavior for both positive and negative inputs.

Interpretability: Complex activation functions like ELU might make it harder to understand how the network's internal representations evolve and contribute to the final output, limiting our ability to interpret its decision-making process.

Computational Cost: While ReLU and Leaky ReLU excel in efficiency, complex activation functions like PReLU introduce additional parameters that increase the computational cost of training and inference.

Finding the Right Fit: Choosing the optimal activation function can be challenging due to the interplay between these issues and the specific characteristics of your data and network architecture. Experimentation and careful consideration are crucial for achieving optimal performance.

Additional Considerations:

- **New Activation Functions:** Research continues to explore novel activation functions aimed at addressing these challenges and potentially offering improved performance.
- **Ensemble Methods:** Combining multiple activation functions in different layers or network branches can leverage their strengths and mitigate their limitations.

By understanding these problems and considering the specific requirements of your project, you can make informed decisions when choosing and fine-tuning activation functions for your deep learning models.