Activation functions

Saket Kumar
8 min read · 22 hours ago

Activation functions are used in neural networks for several crucial reasons:

Non-Linearity:

- **Purpose:** They introduce non-linearity into the network, allowing it to learn and model complex patterns and relationships in the data.
- **Reason:** Without activation functions, a neural network would be equivalent to a linear model, regardless of the number of layers.

Flexibility:

- **Purpose:** They enable the network to adapt and learn various types of data distributions and features.
- **Reason:** Different activation functions (like ReLU, Sigmoid, Tanh) provide different types of non-linear mappings.

Gradient Propagation:

- **Purpose:** They facilitate the backpropagation process by allowing gradients to flow through the network.
- **Reason:** Activation functions help in distributing gradients during training, although some functions may lead to vanishing or exploding gradients.

Control Output Range:

- **Purpose:** They control the range of output values from each neuron.
- **Reason:** For example, Sigmoid restricts outputs between 0 and 1 (useful for probabilities), while Tanh ranges from -1 to 1 (centered around zero).

Introducing Thresholds:

- **Purpose:** They introduce thresholds that can help in feature extraction and pattern recognition.
- **Reason:** Functions like ReLU activate only positive values, acting as a threshold mechanism for data transformation.

> *Sigmoid (Logistic) Function*

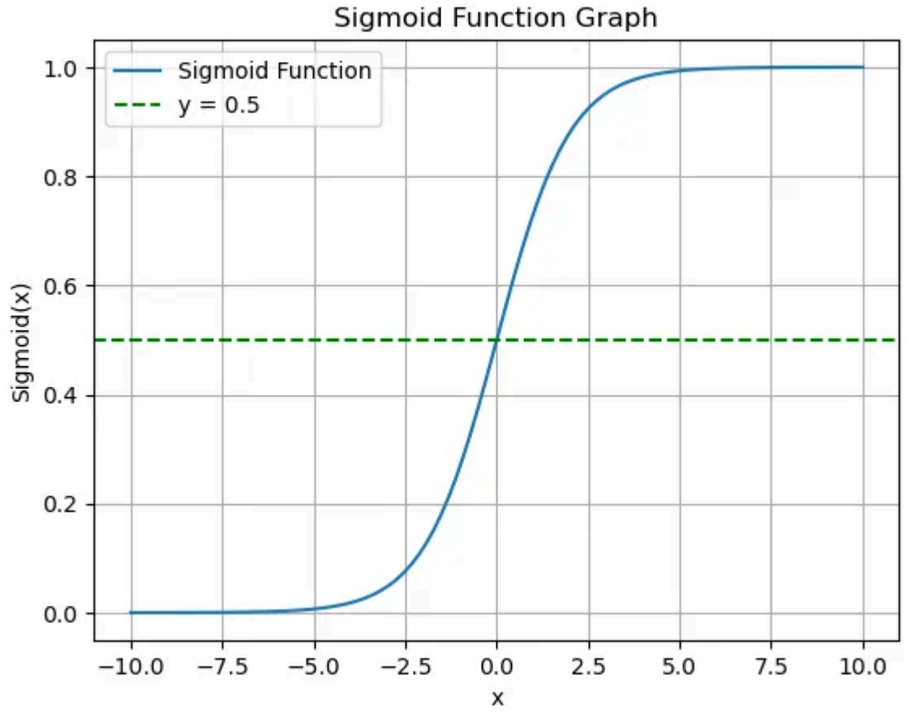
Formula: $\sigma(x) = \frac{1}{1+e^{-x}}$

Range: (0, 1)

Use Case: Often used in binary classification problems. It squashes input values between 0 and 1, making it suitable for interpreting outputs as probabilities.

Limitations: Can cause vanishing gradient problems, where gradients become too small for effective learning.

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

# Sigmoid function
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = sigmoid(x)

# Plotting the sigmoid graph
plt.plot(x, y, label='Sigmoid Function')

# Adding a horizontal green line at y = 0.5
plt.axhline(y=0.5, color='green', linestyle='--', label='y = 0.5')

# Adding title and labels
plt.title('Sigmoid Function Graph')
plt.xlabel('x')
plt.ylabel('Sigmoid(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```



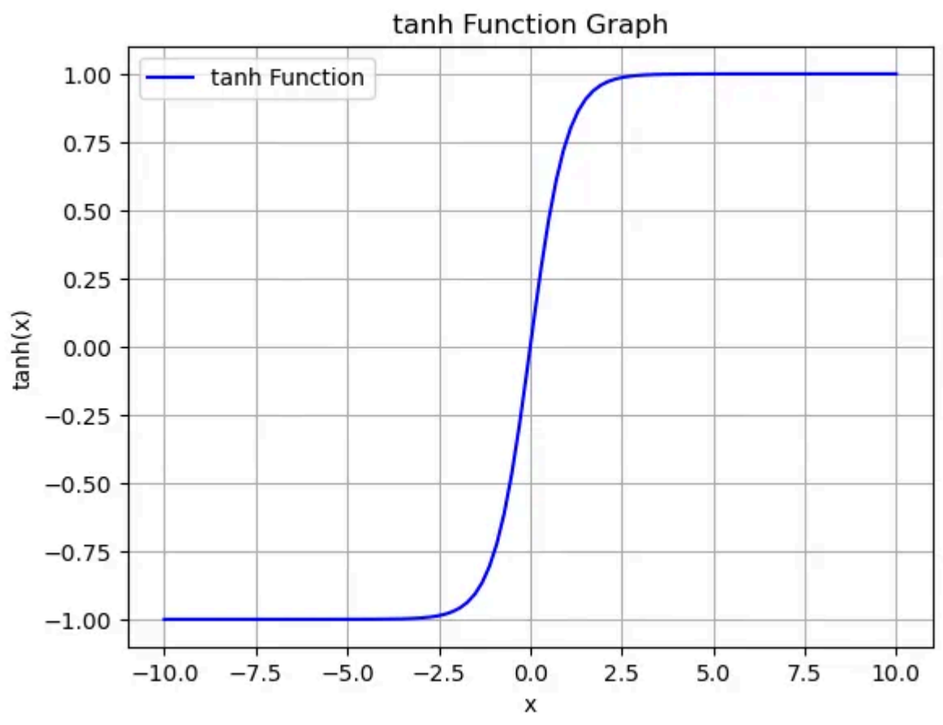
Tanh (Hyperbolic Tangent)

Formula: $\tanh(x) = \frac{2}{1+e^{-2x}} - 1$

- **Range:** (-1, 1)
- **Use Case:** Similar to Sigmoid but outputs are centered around zero, which helps in making gradients flow better during backpropagation.
- **Limitations:** Like Sigmoid, it can also suffer from vanishing gradients.

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

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

```
# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = tanh(x)

# Plotting the tanh graph
plt.plot(x, y, label='tanh Function', color='blue')

# Adding title and labels
plt.title('tanh Function Graph')
plt.xlabel('x')
plt.ylabel('tanh(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```

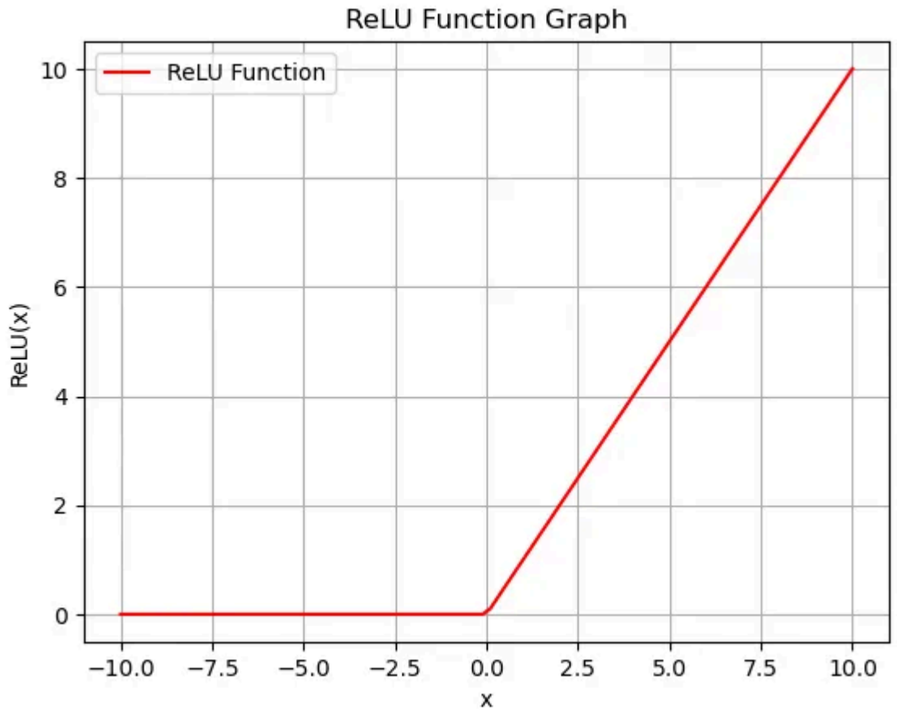


> *ReLU (Rectified Linear Unit)*

Formula: $f(x) = \max(0, x)$

- **Range:** [0, ∞)
- **Use Case:** One of the most widely used activation functions in deep learning. It introduces non-linearity and is computationally efficient.
- **Limitations:** ReLU can cause “dead neurons” where neurons stop activating for all inputs, especially when inputs are negative (dying ReLU problem).

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

# ReLU function
def relu(x):
    return np.maximum(0, x)

# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = relu(x)

# Plotting the ReLU graph
plt.plot(x, y, label='ReLU Function', color='red')

# Adding title and labels
plt.title('ReLU Function Graph')
plt.xlabel('x')
plt.ylabel('ReLU(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```



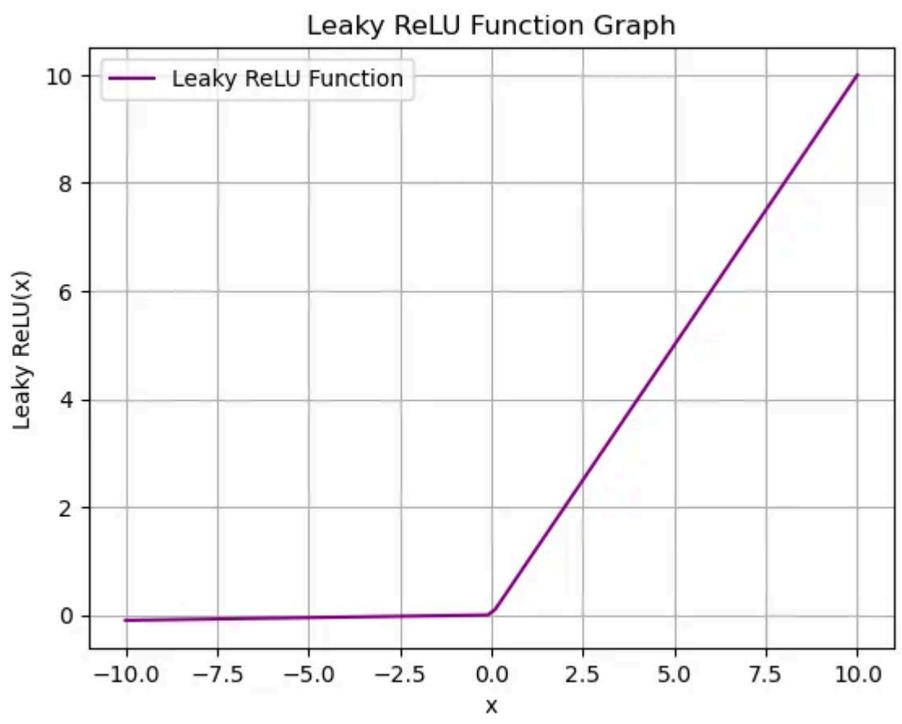
Leaky ReLU

$$\text{Formula: } f(x) = \max(0.01x, x)$$

- **Range:** (-∞, ∞)
- **Use Case:** An improvement over ReLU. It allows a small gradient for negative inputs, reducing the dying ReLU problem.
- **Limitations:** The small slope for negative values is still arbitrarily chosen.

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

# Leaky ReLU function
def leaky_relu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)
```

```
# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = leaky_relu(x)

# Plotting the Leaky ReLU graph
plt.plot(x, y, label='Leaky ReLU Function', color='purple')

# Adding title and labels
plt.title('Leaky ReLU Function Graph')
plt.xlabel('x')
plt.ylabel('Leaky ReLU(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```



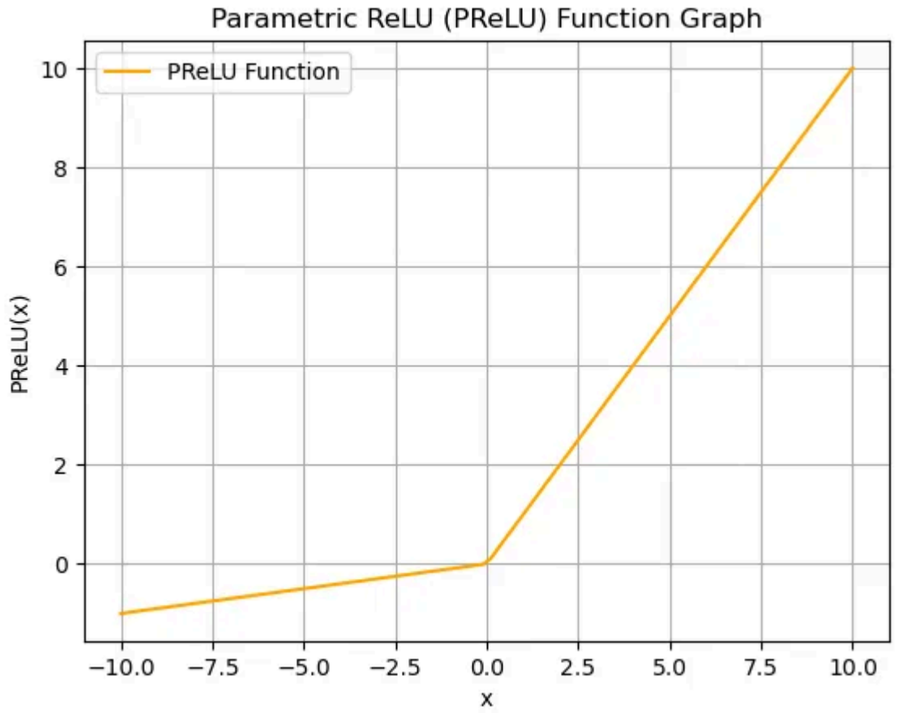
> *Parametric ReLU (PReLU)*

Formula: $f(x) = \max(\alpha x, x)$

where α\alphaα is learned during training.

- **Range:** (-∞, ∞)
- **Use Case:** A further improvement over Leaky ReLU. The slope for negative inputs (α) is learned automatically.
- **Limitations:** Adds a small computational cost due to the learning of α.

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

# Parametric ReLU function
def prelu(x, alpha=0.1):
    return np.where(x > 0, x, alpha * x)

# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = prelu(x)

# Plotting the PReLU graph
plt.plot(x, y, label='PReLU Function', color='orange')

# Adding title and labels
plt.title('Parametric ReLU (PReLU) Function Graph')
plt.xlabel('x')
plt.ylabel('PReLU(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```



ELU (Exponential Linear Unit)

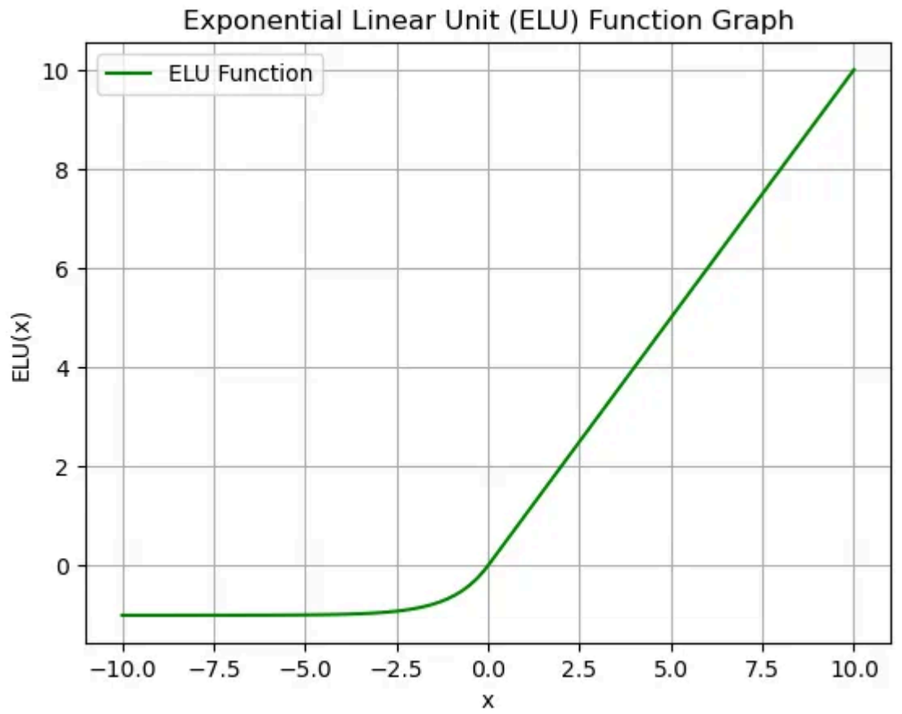
Formula:

$$f(x) = \begin{cases} x & \text{if } x \geq 0 \\ \alpha(e^x - 1) & \text{if } x < 0 \end{cases}$$

- **Range:** (-α, ∞)
- **Use Case:** Overcomes the vanishing gradient problem for negative inputs by having non-zero outputs. ELU can lead to faster and more robust learning.
- **Limitations:** More computationally expensive than ReLU.

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

```python
# ELU function
def elu(x, alpha=1.0):
    return np.where(x > 0, x, alpha * (np.exp(x) - 1))

# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = elu(x)

# Plotting the ELU graph
plt.plot(x, y, label='ELU Function', color='green')

# Adding title and labels
plt.title('Exponential Linear Unit (ELU) Function Graph')
plt.xlabel('x')
plt.ylabel('ELU(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```

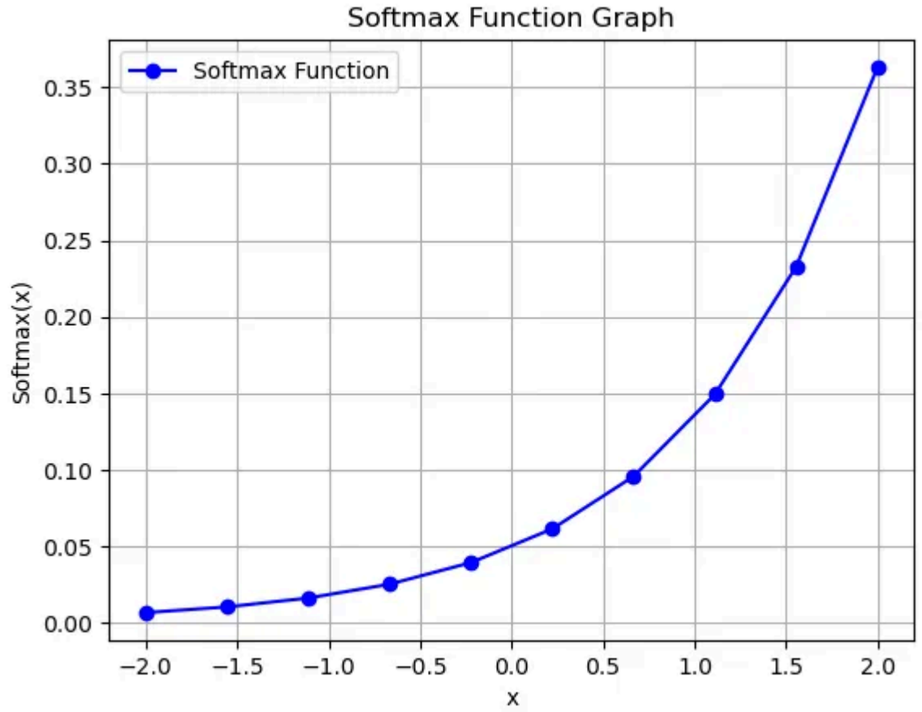


> *Softmax*

Formula: $f(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}}$

- **Range:** (0, 1)
- **Use Case:** Commonly used in the output layer for multi-class classification problems. It converts logits into probability distributions.
- **Limitations:** May lead to problems when dealing with large logits, causing numerical instability.

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

# Softmax function
def softmax(x):
    exp_x = np.exp(x - np.max(x))  # Subtract max to prevent overflow
    return exp_x / exp_x.sum()

# Generate an array of x values (input)
x = np.linspace(-2, 2, 10)  # Using a smaller range to visualize better

# Compute the corresponding y values (output)
y = softmax(x)

# Plotting the Softmax graph
plt.plot(x, y, 'bo-', label='Softmax Function', color='blue')

# Adding title and labels
plt.title('Softmax Function Graph')
plt.xlabel('x')
plt.ylabel('Softmax(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```



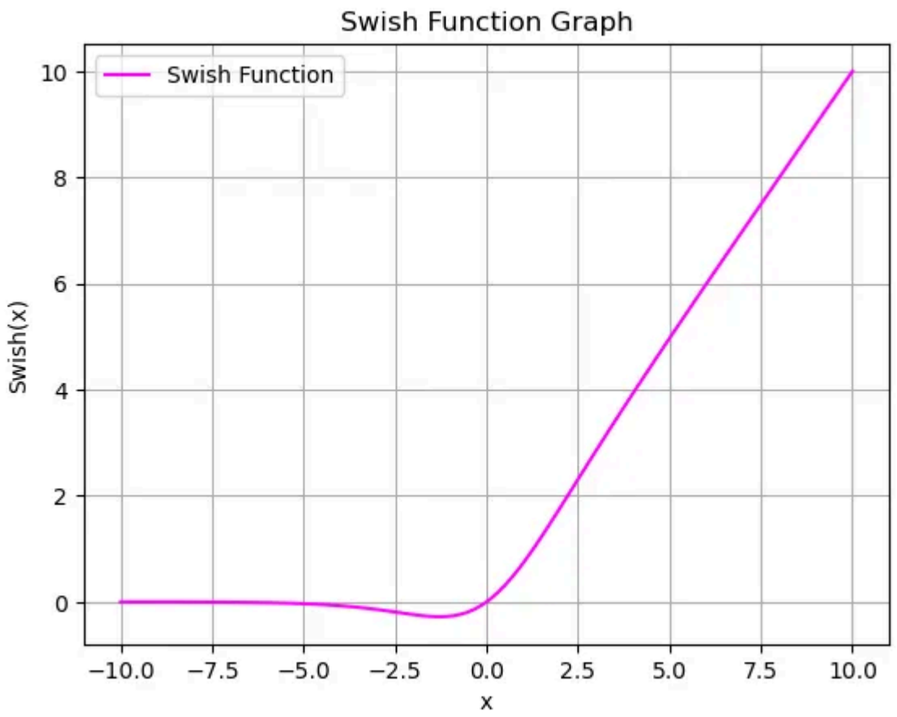
Swish

Formula: $f(x) = x \cdot \text{sigmoid}(x)$

- **Range:** (-0.28, ∞)
- **Use Case:** Developed by Google researchers, Swish shows better performance than ReLU in many deep learning tasks. It's smooth and non-monotonic.
- **Limitations:** More computationally intensive than ReLU.

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

# Swish function
def swish(x, beta=1.0):
    return x * sigmoid(beta * x)
```

```
# Sigmoid function for Swish calculation
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = swish(x)

# Plotting the Swish graph
plt.plot(x, y, label='Swish Function', color='magenta')

# Adding title and labels
plt.title('Swish Function Graph')
plt.xlabel('x')
plt.ylabel('Swish(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```

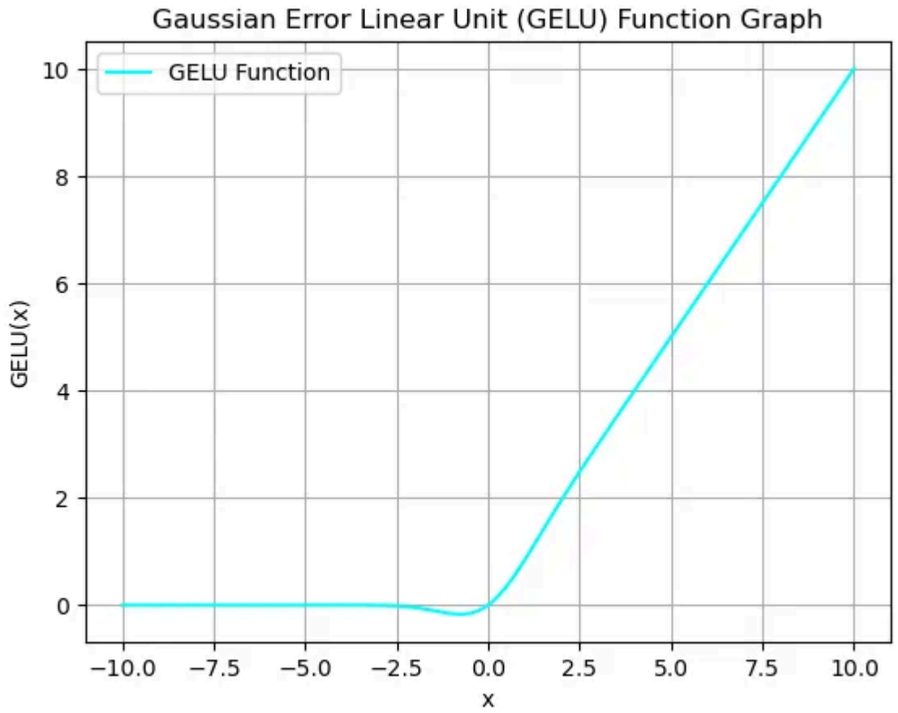


> *GELU (Gaussian Error Linear Unit)*

Formula: $f(x) = 0.5x(1 + \tanh(\sqrt{\frac{2}{\pi}}(x + 0.044715x^3)))$

- **Range:** (-∞, ∞)
- **Use Case:** Popular in transformer-based models like BERT. It has a smoother output than ReLU and allows for small negative inputs.
- **Limitations:** More complex computation compared to simpler activations like ReLU.

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import erf  # Error function from SciPy

# GELU function
def gelu(x):
    return 0.5 * x * (1 + erf(x / np.sqrt(2)))

# Generate an array of x values (input)
x = np.linspace(-10, 10, 100)

# Compute the corresponding y values (output)
y = gelu(x)

# Plotting the GELU graph
plt.plot(x, y, label='GELU Function', color='cyan')

# Adding title and labels
plt.title('Gaussian Error Linear Unit (GELU) Function Graph')
plt.xlabel('x')
plt.ylabel('GELU(x)')

# Adding grid and legend
plt.grid(True)
plt.legend()

# Display the plot
plt.show()
```



Vanishing Gradient Problem

Activation Functions Affected:

- **Sigmoid Function**: The sigmoid function outputs values between 0 and 1. For very large or very small input values, the gradient of the sigmoid function becomes very small (close to zero), causing the gradients to vanish during backpropagation. This can make training deep networks difficult.
- **Tanh Function**: Similar to the sigmoid function, the tanh function outputs values between -1 and 1. For very large or very small inputs, the gradients can become very small, leading to the vanishing gradient problem.

Why It Happens:

- The gradients of these functions are derivatives of the activation function. When these derivatives are very small, the gradients propagated back through the network become very small, leading to slow or stalled learning.

Exploding Gradient Problem

Activation Functions Affected:

- **ReLU Function**: While ReLU itself doesn't directly cause exploding gradients, if the weights are initialized poorly or the network is very deep, the gradients can grow exponentially, causing exploding gradients. This is more of an issue with network architecture and initialization rather than the ReLU function itself.
- **Leaky ReLU and PReLU**: Similar to ReLU, these functions can also contribute to exploding gradients if the network is poorly initialized or not regularized.

Why It Happens:

- Exploding gradients occur when large weight updates cause gradients to become very large, leading to numerical instability and sometimes causing weights to grow exponentially.

Derivative range of popular activation functions:

- **Sigmoid:** Ranges from 0 to 0.25.
- **Tanh:** Ranges from 0 to 1.
- **ReLU:** Ranges from 0 to 1.
- **Leaky ReLU:** Ranges from α to 1.
- **PReLU:** Ranges from α to 1.

Computationally Expensive:

Functions that require more complex computations, like Tanh or Swish, slow down the training process due to increased computational overhead. This can become significant when training large models or when the computational resources are limited.

Normalization Effects:

- **Purpose:** Some activation functions can affect the normalization of data within the network.
- **Examples:** Tanh centers the data around zero, which can help with normalization, while ReLU may lead to data sparsity.

Learnable Parameters:

- **Purpose:** Some activation functions include learnable parameters that can be adjusted during training.
- **Examples:** PReLU includes learnable parameters for negative inputs, which can improve performance.

Written by Saket Kumar

10 Followers

Edit profile