GitHub Repository : [Blaming ML/ml-tutorial-Sreenivasulu .ipynb at main · sreenu2003v/ML]

Impact of Activation Functions in Deep Learning

Module 1: Introduction

Imagine this - You're creating a strong network of learning to identify handwritten digits. Your network consists of several layers of neurons, each of which maps data before sending it on. But if you didn't have activation functions, each layer would only perform a linear mapping—regardless of how many layers you stack, the network would never be able to learn about complex patterns. It would be similar to a linear regression model, and it would perform badly on tasks like reading handwritten digits or speech interpretation.

That is where activation functions enter the scene. They provide non-linearity, enabling neural networks to pick up on difficult patterns, detect objects in pictures, and even create human language-like text. Activation functions are the unseen powerhouse of deep learning's capabilities, deciding whether a neuron fires or not. Their decision affects anything from training duration to model performance, and the right one may be the distinction between a model that converges and one that never does.

In this tutorial, I'll cover why activation functions are important, discuss their variety, and pit them against one another—equipping you with the knowledge to make informed decisions in your own deep learning endeavor.

1.1 What Are Activation Functions?

Activation function is a mathematical function applied to the output of each neuron in a neural network. Its sole purpose is to introduce non-linearity so that the model learns sophisticated patterns not otherwise learnable through linear transformations alone.

Lacking activation functions, a multi-layered neural network would not be distinct from a linear regression model, significantly diminishing its capability of learning complex relationships between data.

1.2 Key Roles of Activation Functions:

Introduces non-linearity: Enables neural networks to be learned with non-linearly separable data.

Enable learning from intricate details and structures: Supports the model in learning deeper patterns.

Control gradient flow at backpropagation: Provides training updates to pass smoothly.

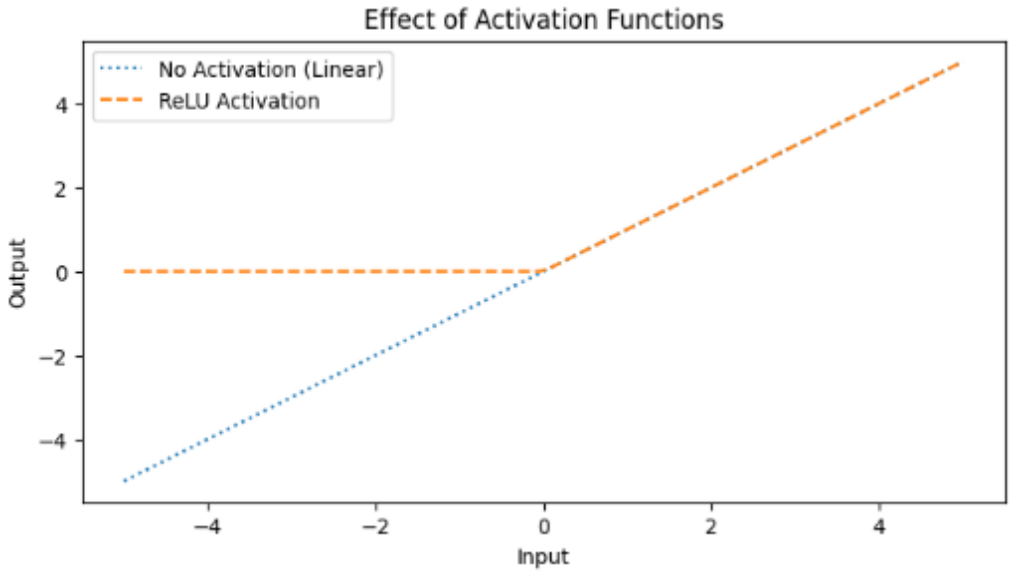
Affects training speed and convergence: Impacts how fast the model will learn and whether it converges well.

1.3 Quick Visual Demonstration: Why Activation Functions Matter

To see why activation functions are important, let's compare a neural network with and without an activation function.

As seen below, below is a comparison of a linear function (linear function without activation) to the Rectified Linear Unit (ReLU), the most preferred among the activation functions for use in deep learning.

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

x = np.linspace(-5, 5, 100)
linear_output = x  # No activation function
relu_output = np.maximum(0, x)  # ReLU activation

plt.figure(figsize=(8,4))
plt.plot(x, linear_output, label="No Activation (Linear)", linestyle="dotted")
plt.plot(x, relu_output, label="ReLU Activation", linestyle="dashed")
plt.legend()
plt.title("Effect of Activation Functions")
plt.xlabel("Input")
plt.ylabel("Output")
plt.show()
```



Explanation of the Plot:

Without an activation function (blue dotted line):

- The output is simply a **linear transformation** of the input, meaning the model cannot learn **non-linear patterns**.
- No matter how many layers we stack, the entire network will still behave like a **single linear equation**, limiting its ability to recognize complex relationships.

With ReLU activation (red dashed line):

- The function outputs **0** for negative values and **x** for positive values, introducing **non-linearity**.
- This allows the neural network to learn more **complex structures and representations**, making it suitable for deep learning applications.

Module 2: Types of Activation Functions in Deep Learning

It is important to know the general context of activation functions and their position within neural networks before going into more detail about the concept. Activation functions can be grouped into three categories, which serve different functions in deep learning.

2. 1 Linear Activation Function

Definition

A **linear activation function** simply applies a weighted sum transformation without introducing any non-linearity:

$$f(x) = ax$$

where a is a constant.

Strengths

1. Simple and easy to compute.
2. Works well for **linear regression** problems.

Weaknesses

1. **No non-linearity**, meaning no matter how many layers are added, the network behaves like a single linear transformation.
2. **Cannot learn complex patterns** like curves, edges, or abstract features in images.

Why Not Use It in Deep Learning?

Deep learning relies on **stacking multiple layers**, but a linear function does not allow for meaningful feature transformations, making it ineffective for complex tasks.

2.2 Non-Linear Activation Functions

Non-linear activation functions are **critical in deep learning** as they enable the model to learn **complex and hierarchical patterns**. These include:

(a) Sigmoid Activation Function (Logistic Function)

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

1. Used in **binary classification**.
2. Suffers from **vanishing gradients** in deep networks.
3. **Best Use Case:** Output layer of **binary classification models**.

(b) Hyperbolic Tangent (Tanh) Activation Function

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

1. Outputs are **zero-centered (-1 to 1)**, improving training.
2. Still suffers from **vanishing gradients** in deep networks.
3. **Best Use Case:** Used in hidden layers **but replaced by ReLU** in modern deep networks.

(c) Rectified Linear Unit (ReLU)

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

1. **Solves vanishing gradient problems** for positive values.
2. **Fast computation** (no exponentiation).
3. Can suffer from **dying neurons** (outputs zero for negative values).
4. **Best Use Case: Most widely used** in deep learning hidden layers.

(d) Leaky ReLU

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

1. Fixes **dying ReLU problem** by allowing a small slope for negative values.
2. Still non-zero-centered.
3. **Best Use Case:** Hidden layers of deep networks, **when ReLU neurons die**

(e) Swish Activation Function (Used in EfficientNet)

$$f(x) = x \cdot \sigma(x)$$

1. Outperforms ReLU in **some deep learning applications**.
2. More computationally expensive.
3. **Best Use Case: State-of-the-art deep learning models**.

2.3 Specialized Activation Functions

These activation functions are used in **specific tasks**:

- **Softmax:** Used in the **output layer of multi-class classification networks**.
- **GELU (Gaussian Error Linear Unit):** Used in **transformers and NLP models**.
- **ELU (Exponential Linear Unit):** An improved version of ReLU that reduces bias shifts.

2. 4 Choosing the Right Activation Function

Activation Function	Best Use Case	Avoid When
Sigmoid	Binary classification (output layer)	Deep networks (vanishing gradients)
Tanh	Hidden layers of simple networks	Vanishing gradient problem
ReLU	Most deep learning models	Dying neurons issue
Leaky ReLU	Fixing dying ReLU neurons	Requires additional hyperparameter tuning
Swish	High-performance networks	Expensive computations

Module 3: Linear Activation Functions

3.1 Mathematical Formula

$$f(x) = ax$$

where a is a constant.

Benefits of the Linear Activation Function

- **Simple and easy to compute** – No complex operations involved.
- **Works well for linear regression problems** – When the relationship between input and output is purely linear.
- **Can be useful in specialized cases** – For example, in output layers where unbounded values are required.

Limitations of the Linear Activation Function

- **No non-linearity** → If multiple linear layers are stacked, the network behaves like a **single linear transformation**.
- **Cannot learn complex patterns** → Cannot capture non-linear relationships like curves, edges, or abstract features.
- **Limited representation power** → Deep networks become useless without non-linearity.

3. 2 Why is Linear Activation NOT Used in Deep Learning?

A neural network **with only linear activation functions** behaves like a **linear regression model**, regardless of how many layers it has.

- **Example:**

Let's assume we have a multi-layer neural network:

$$y = W_3(W_2(W_1X + b_1) + b_2) + b_3$$

Since all transformations are linear, we can simplify this to:

$$y = W'X + b'$$

Thus, no matter how many layers we add, the network **collapses into a single linear equation**, meaning it **cannot model complex relationships in data**.

This is why **deep learning relies on non-linear activation functions** like **ReLU, Sigmoid, and Tanh**.

3.3 Real-World Use Cases of Linear Activation Function

Although linear activation is not commonly used in deep learning, it is still useful in **some cases**:

Regression Tasks (Continuous Outputs)

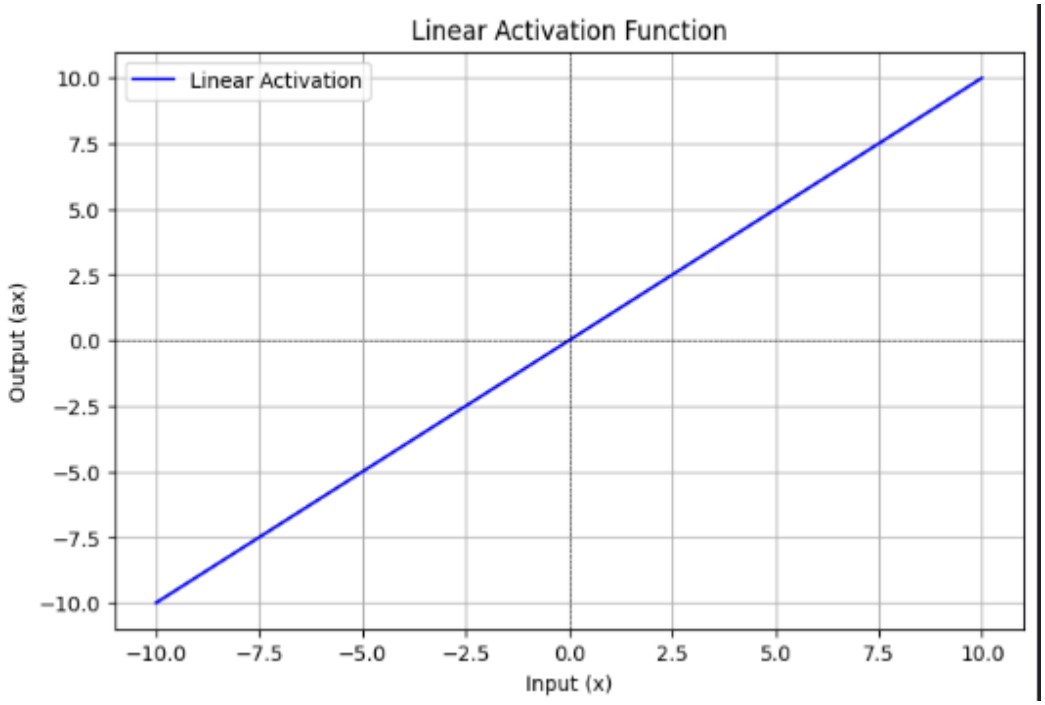
- Used in the **output layer of regression networks** where predictions are unbounded (e.g., **stock price prediction, house price estimation**).
- Example: A **neural network predicting temperature** in Celsius doesn't require values to be constrained between 0 and 1.

Linear Regression Models

- If we only need to **fit a straight line** to data, a simple **linear activation function** is sufficient.

3.4 Implementing Linear Activation and Visualization

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

# Define the linear activation function
def linear(x, a=1):
    return a * x  # Simple linear transformation

# Generate input values
x = np.linspace(-10, 10, 100)
y = linear(x)

# Plot the linear function
plt.figure(figsize=(8,5))
plt.plot(x, y, label="Linear Activation", color="blue")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Output (ax)")
plt.title("Linear Activation Function")
plt.legend()
plt.grid()
plt.show()
```



Observations from the Plot

- The **output is always a straight line**, meaning **no non-linearity** is introduced.
- The function **cannot capture complex patterns** beyond simple proportional relationships.

Module 4: Non-Linear Activation Functions in Deep Learning

Since the **Linear Activation Function** is limited in deep learning due to its inability to model complex relationships, we rely on **non-linear activation functions**. These functions allow neural networks to learn intricate patterns, making deep learning successful in fields like **image recognition, NLP, and autonomous systems**.

Why Non-Linear Activation Functions?

1. They allow deep neural networks to learn and model **non-linear relationships**.
2. Enable **hierarchical feature extraction** at different network layers.
3. Prevent the entire network from **collapsing into a single linear function**.

4.1 Types of Non-Linear Activation Functions

The most commonly used **non-linear activation functions** include:

1. **Sigmoid Activation Function** (Good for binary classification, but has vanishing gradient issues).

2. **Tanh (Hyperbolic Tangent) Activation Function** (Better than Sigmoid, but still has vanishing gradient issues).
3. **ReLU (Rectified Linear Unit)** (Most widely used in modern deep learning).
4. **Leaky ReLU** (Fixes the dying neuron problem in ReLU).
5. **Swish Activation Function** (A self-gated function used in advanced architectures).
6. **Softmax Activation Function** (For multi-class classification tasks).

Each of these functions has **specific advantages, limitations, and real-world applications**, which we will cover **one by one** in the following sections.

4.2 Sigmoid Activation Function (Logistic Function)

The **sigmoid activation function** is one of the earliest activation functions used in neural networks. It is particularly useful in **binary classification tasks** because it maps any input to a range between **0 and 1**, making it ideal for **probability estimation**.

Mathematical Formula:

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

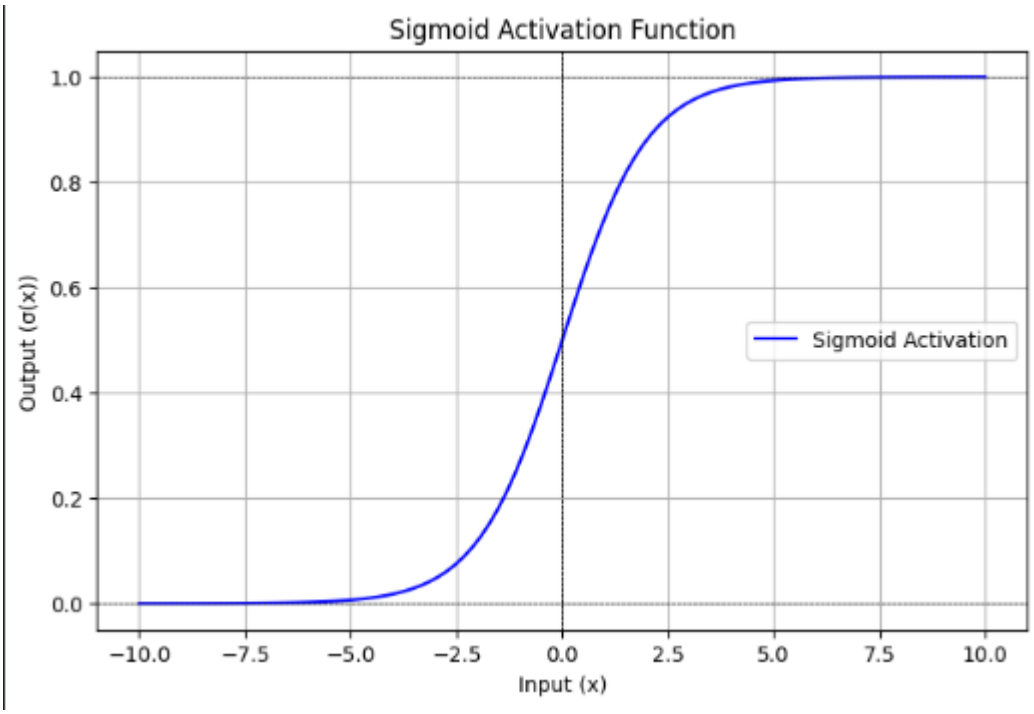
Where:

- x is the input to the neuron.
- e is the Euler's number (approximately 2.718).
- The function output is always in the range **(0,1)**.

🔹 How Does Sigmoid Work?

- If x is **large and positive**, $\sigma(x)$ approaches **1**.
- If x is **large and negative**, $\sigma(x)$ approaches **0**.
- If $x = 0$, then $\sigma(x) = 0.5$.

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

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

# Generate input values
x = np.linspace(-10, 10, 100)
y = sigmoid(x)

# Plot the sigmoid function
plt.figure(figsize=(8,5))
plt.plot(x, y, label="Sigmoid Activation", color="blue")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axhline(1, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Output (σ(x))")
plt.title("Sigmoid Activation Function")
plt.legend()
plt.grid()
plt.show()
```



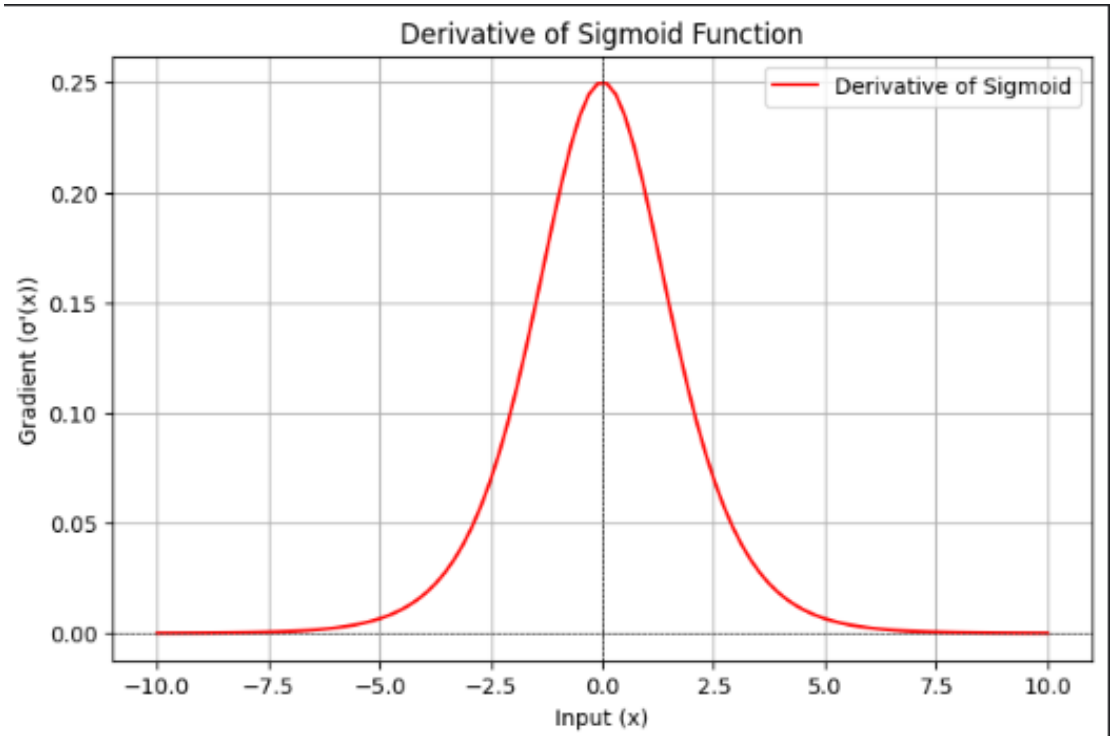
Observations from the Plot

- The function asymptotically approaches 0 and 1 but never exactly reaches them.

- The steepest slope is around x=0x = 0x=0, meaning neurons are most sensitive to changes near zero.

Derivative of Sigmoid Function

```python
# Compute derivative of sigmoid
def sigmoid_derivative(x):
    return sigmoid(x) * (1 - sigmoid(x))

y_derivative = sigmoid_derivative(x)

# Plot the derivative of the sigmoid function
plt.figure(figsize=(8,5))
plt.plot(x, y_derivative, label="Derivative of Sigmoid", color="red")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Gradient (σ'(x))")
plt.title("Derivative of Sigmoid Function")
plt.legend()
plt.grid()
plt.show()
```



Observations from the Plot

- The gradient is highest around x=0x = 0x=0.
- For large ∣x∣|x|∣x∣, the gradient approaches zero, confirming the vanishing gradient issue.

4. 3 Tanh (Hyperbolic Tangent) Activation Function

The **Tanh (Hyperbolic Tangent) Activation Function** is an improvement over the **Sigmoid Activation Function** because it is **zero-centered**, which helps in faster and more efficient training of deep neural networks.

Mathematical Formula:

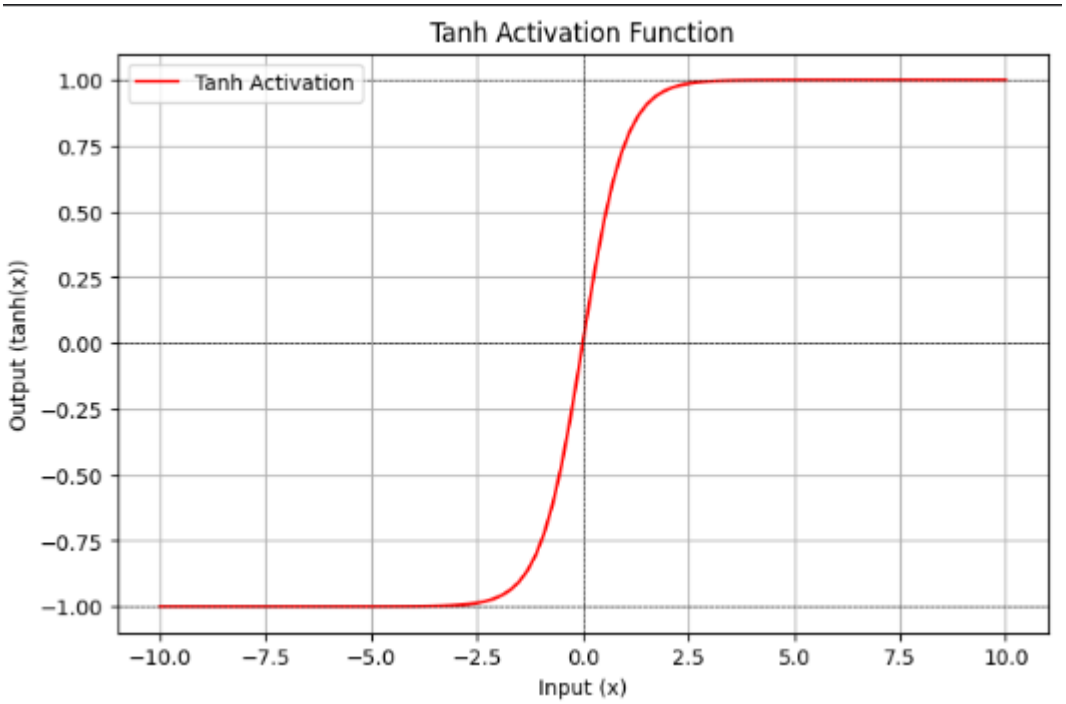
$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

- Unlike **Sigmoid**, which outputs values between **0 and 1**, Tanh outputs values between **-1 and 1**.
- This makes **Tanh better than Sigmoid**, as it allows **both positive and negative activations**, reducing the **bias shift issue** seen in Sigmoid.

🔹 How Does Tanh Work?

- If x is **large and positive**, $\tanh(x)$ approaches **1**.
- If x is **large and negative**, $\tanh(x)$ approaches **-1**.
- If $x = 0$, then $\tanh(x) = 0$.

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

# Define the Tanh function
def tanh(x):
    return np.tanh(x)

# Generate input values
x = np.linspace(-10, 10, 100)
y = tanh(x)

# Plot the Tanh function
plt.figure(figsize=(8,5))
plt.plot(x, y, label="Tanh Activation", color="red")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axhline(1, color="black", linewidth=0.5, linestyle="dashed")
plt.axhline(-1, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Output (tanh(x))")
plt.title("Tanh Activation Function")
plt.legend()
plt.grid()
plt.show()
```

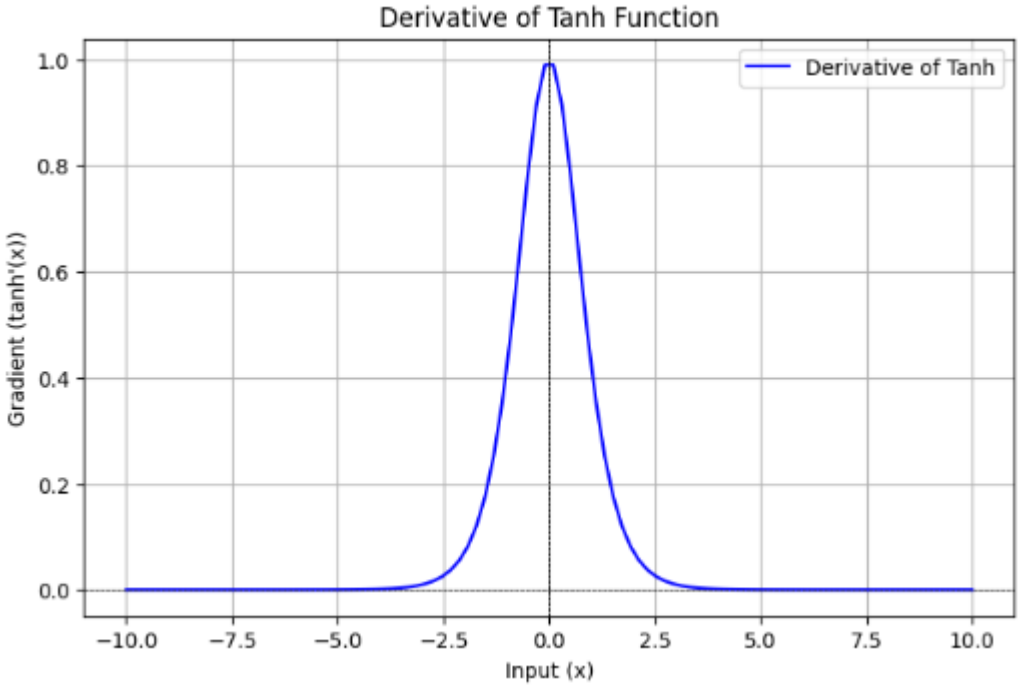


Observations from the Plot

- The function **outputs values between -1 and 1**.
- **Compared to Sigmoid, Tanh is zero-centered**, meaning **weight updates are more balanced**.
- The steepest slope occurs around **x=0x = 0x=0**, leading to better sensitivity to small changes in input.

Derivative of TanH Function

```
# Compute derivative of Tanh
def tanh_derivative(x):
    return 1 - tanh(x)**2

y_derivative = tanh_derivative(x)

# Plot the derivative of the Tanh function
plt.figure(figsize=(8,5))
plt.plot(x, y_derivative, label="Derivative of Tanh", color="blue")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Gradient (tanh'(x))")
plt.title("Derivative of Tanh Function")
plt.legend()
plt.grid()
plt.show()
```



Observations

- The derivative is highest around x=0x = 0x=0.
- `For large |x||x|, the gradient approaches zero, confirming the vanishing gradient issue.`

4.4 ReLU (Rectified Linear Unit) Activation Function

The ReLU (Rectified Linear Unit) Activation Function is the most popular activation function used in deep learning nowadays. It is easy, computationally cheap, and solves the vanishing gradient problem encountered in Sigmoid and Tanh.

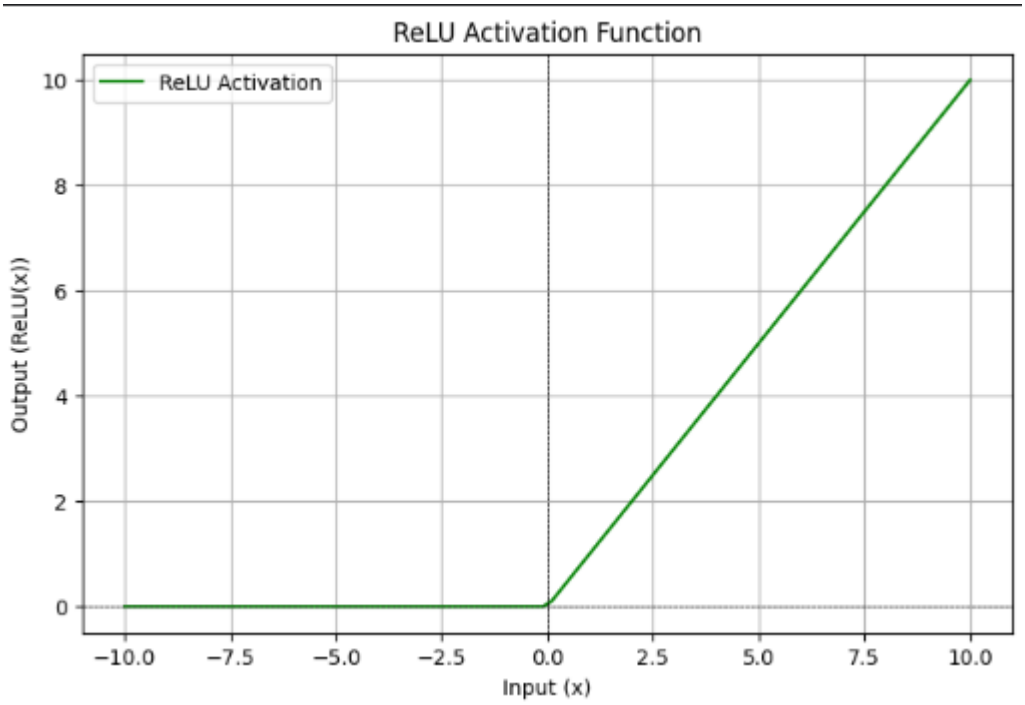
Mathematical Formula:

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

- If x is **positive**, the function outputs **x**.
- If x is **negative**, the function outputs **0**.

This means that **ReLU is non-linear**, but still easy to compute.

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

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

# Generate input values
x = np.linspace(-10, 10, 100)
y = relu(x)

# Plot the ReLU function
plt.figure(figsize=(8,5))
plt.plot(x, y, label="ReLU Activation", color="green")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Output (ReLU(x))")
plt.title("ReLU Activation Function")
plt.legend()
plt.grid()
plt.show()
```



Observations from the Plot

- For negative values of xxx, the output is 0.
- For positive values of xxx, the output is xxx.
- This makes ReLU computationally efficient and effective for deep networks.

4.5 Leaky ReLU (Leaky Rectified Linear Unit) Activation Function

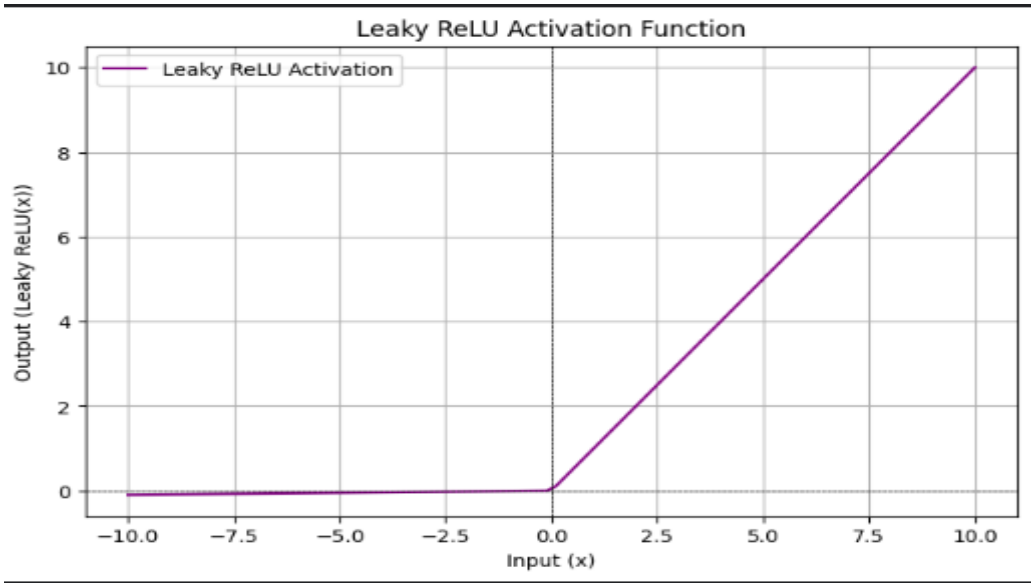
The **Leaky ReLU** Activation Function is an **improvement over ReLU** that helps **fix** the **Dying ReLU** problem by allowing a small negative slope for negative inputs instead of completely setting them to **zero**.

Mathematical Formula:

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

where α is a small constant (e.g., **0.01**), ensuring that even negative values get a **small, nonzero output** instead of being zeroed out like in standard ReLU.

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

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

# Generate input values
x = np.linspace(-10, 10, 100)
y = leaky_relu(x)

# Plot the Leaky ReLU function
plt.figure(figsize=(8,5))
plt.plot(x, y, label="Leaky ReLU Activation", color="purple")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Output (Leaky ReLU(x))")
plt.title("Leaky ReLU Activation Function")
plt.legend()
plt.grid()
plt.show()
```



Observations from the Plot

- **Leaky ReLU prevents neurons from dying**, improving model learning compared to standard ReLU.
- **Performance is often slightly better than ReLU** in deep networks with many negative activations.

4.6 Parametric ReLU (PReLU) Activation Function

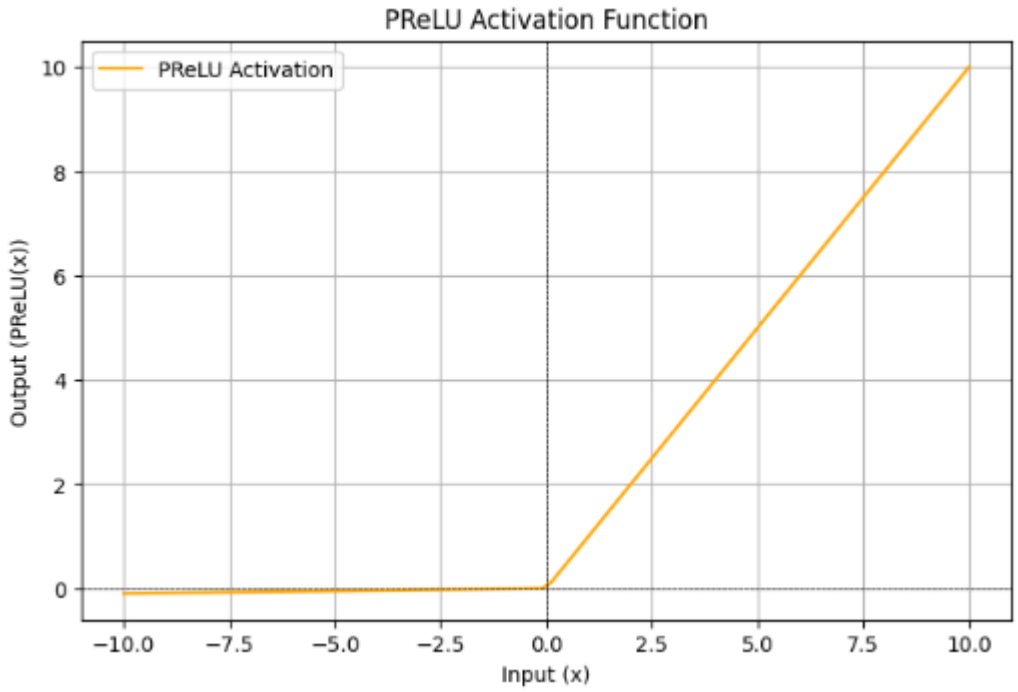
The **Parametric ReLU (PReLU) Activation Function** is an **enhanced version of Leaky ReLU**, where the **negative slope (α\alphaα) is learned during training** instead of being manually set. This makes PReLU more flexible and adaptable than both **ReLU and Leaky ReLU**.

Mathematical Formula:

$$f(x) = \begin{cases} x, & x > 0 \\ \alpha x, & x \leq 0 \end{cases}$$

where α is **learnable** instead of a fixed value like in Leaky ReLU.

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

# Define the PReLU function
def prelu(x, alpha=0.01):
    return np.where(x > 0, x, alpha * x)  # Alpha will be learned in an actual model

# Generate input values
x = np.linspace(-10, 10, 100)
y = prelu(x)

# Plot the PReLU function
plt.figure(figsize=(8,5))
plt.plot(x, y, label="PReLU Activation", color="orange")
plt.axhline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.axvline(0, color="black", linewidth=0.5, linestyle="dashed")
plt.xlabel("Input (x)")
plt.ylabel("Output (PReLU(x))")
plt.title("PReLU Activation Function")
plt.legend()
plt.grid()
plt.show()
```



Observations

- **PReLU dynamically learns the negative slope** (α\alphaα), which improves **training efficiency** compared to Leaky ReLU.
- **Performance can be better than standard ReLU**, especially for complex tasks.

Module 5: Code Demonstration

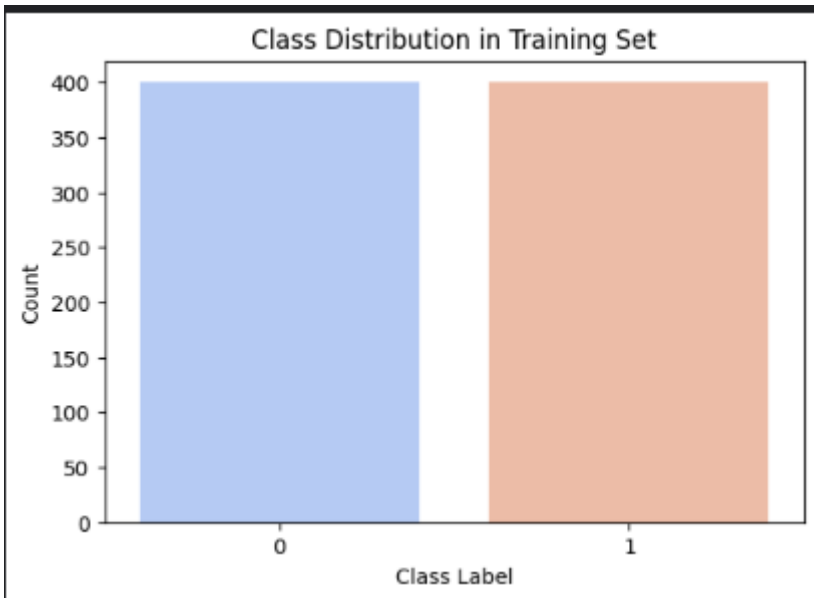
5.1 Loading Moons Dataset and Splitting into Training and Testing Data

```
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd
from sklearn.datasets import make_moons
from sklearn.model_selection import train_test_split
from collections import Counter

# Generate the dataset
X, y = make_moons(n_samples=1000, noise=0.2, random_state=42)

# Convert to DataFrame for EDA
df = pd.DataFrame(X, columns=['Feature 1', 'Feature 2'])
df['Target'] = y

# Split into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Print dataset shape
train_size, test_size = X_train.shape, X_test.shape
class_distribution = Counter(y_train)
```

5.2 Exploratory Data Analysis

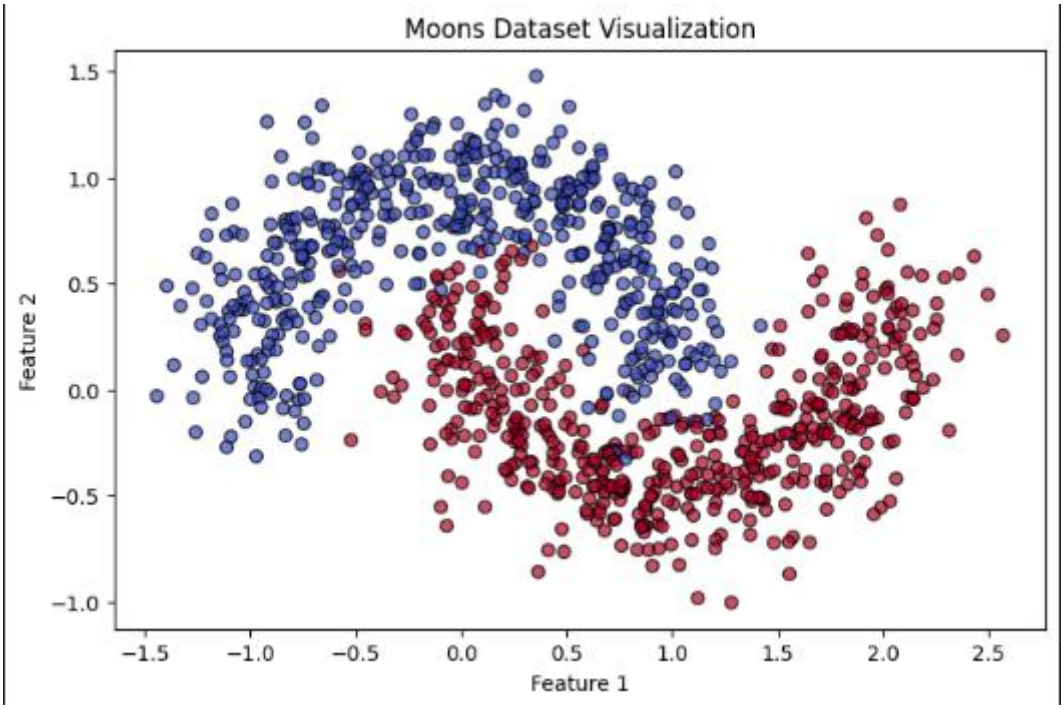
Plotting Class Distribution

```
# Plot class distribution
plt.figure(figsize=(6,4))
sns.countplot(x=y_train, palette="coolwarm")
plt.title("Class Distribution in Training Set")
plt.xlabel("Class Label")
plt.ylabel("Count")
plt.show()
```



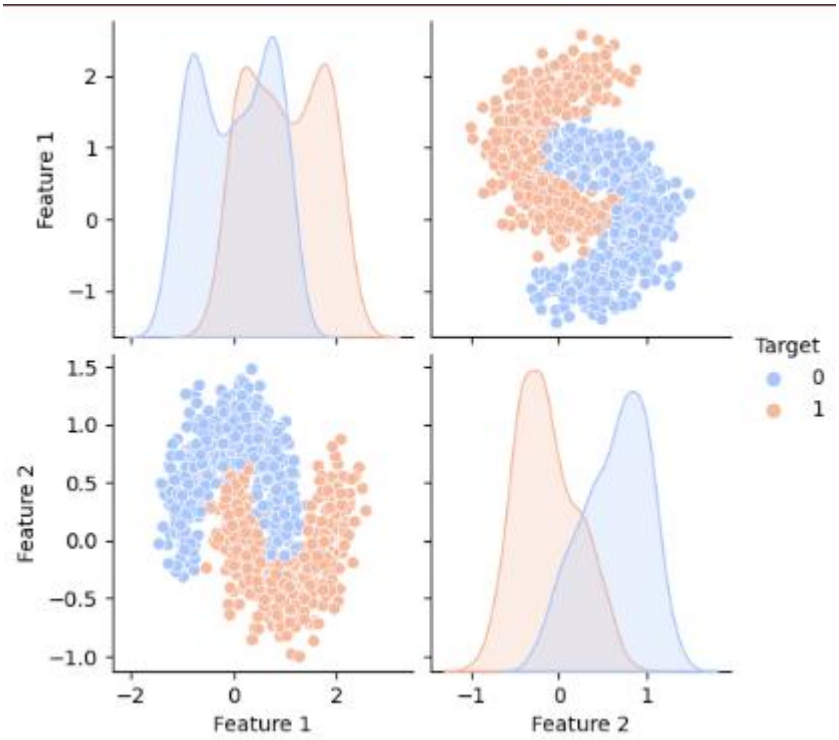
Scatter Plot of Dataset with Class Labels

```
# Scatter plot of dataset with class labels
plt.figure(figsize=(8,5))
plt.scatter(X[:,0], X[:,1], c=y, cmap="coolwarm", edgecolors="k", alpha=0.7)
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("Moons Dataset Visualization")
plt.show()
```



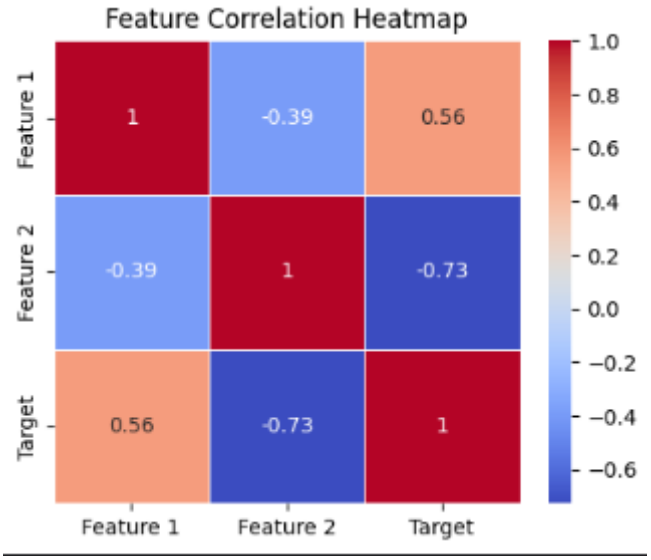
Pairplot to Observe Feature Relationships

```
# Pairplot to observe feature relationships
sns.pairplot(df, hue="Target", palette="coolwarm")
plt.show()
```



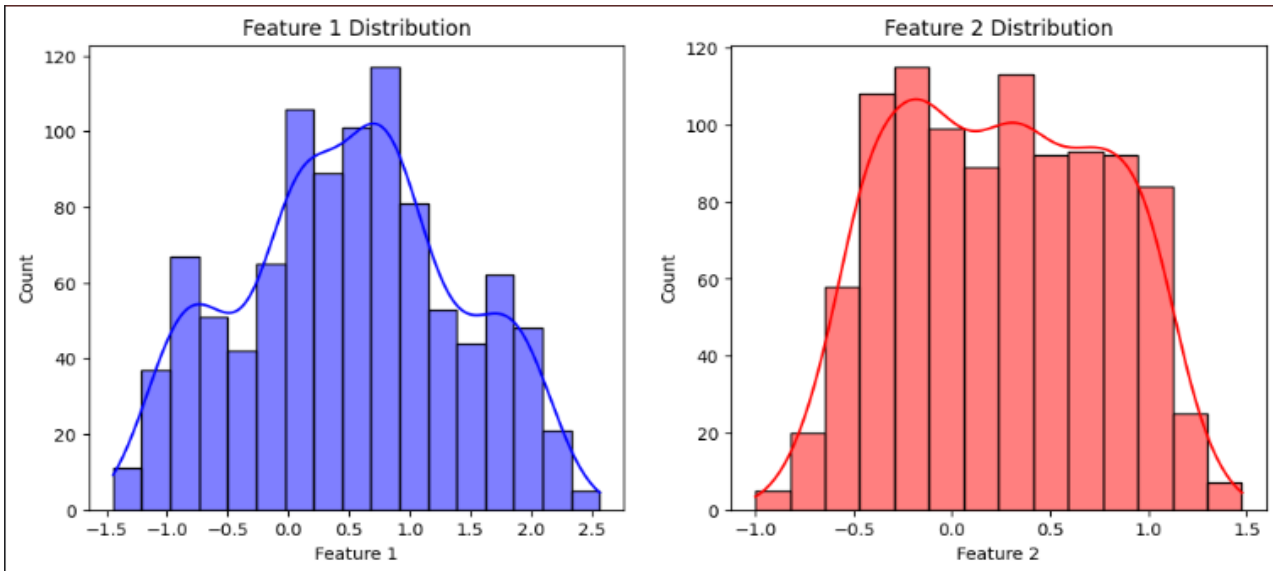
Correlation Heatmap

```
# Correlation heatmap
plt.figure(figsize=(5,4))
sns.heatmap(df.corr(), annot=True, cmap="coolwarm", linewidths=0.5)
plt.title("Feature Correlation Heatmap")
plt.show()
```



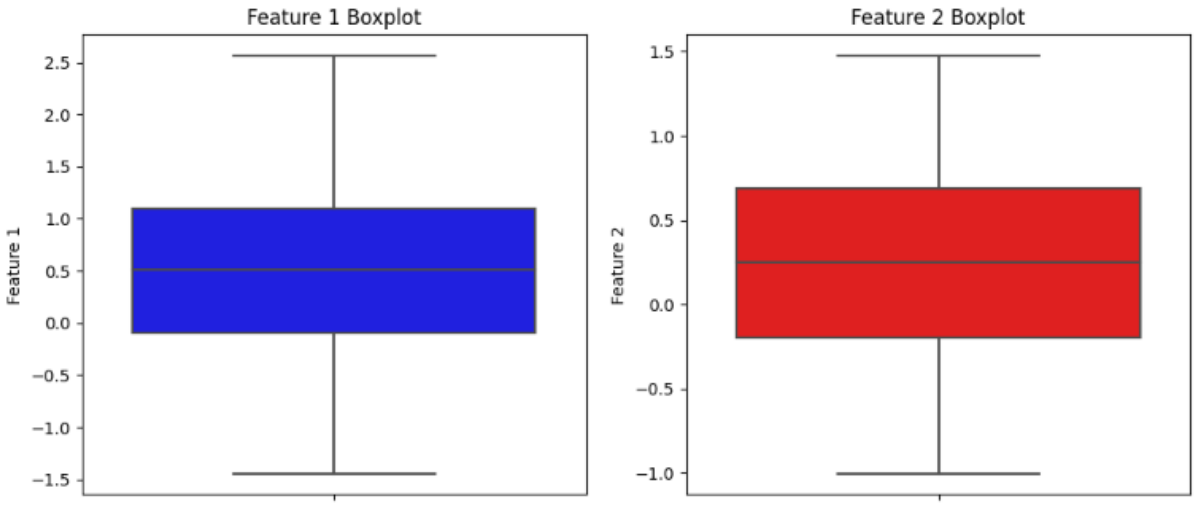
Distribution plots of Each Feature

```
# Distribution plots of each feature
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.histplot(df['Feature 1'], kde=True, ax=axes[0], color='blue')
axes[0].set_title("Feature 1 Distribution")
sns.histplot(df['Feature 2'], kde=True, ax=axes[1], color='red')
axes[1].set_title("Feature 2 Distribution")
plt.show()
```



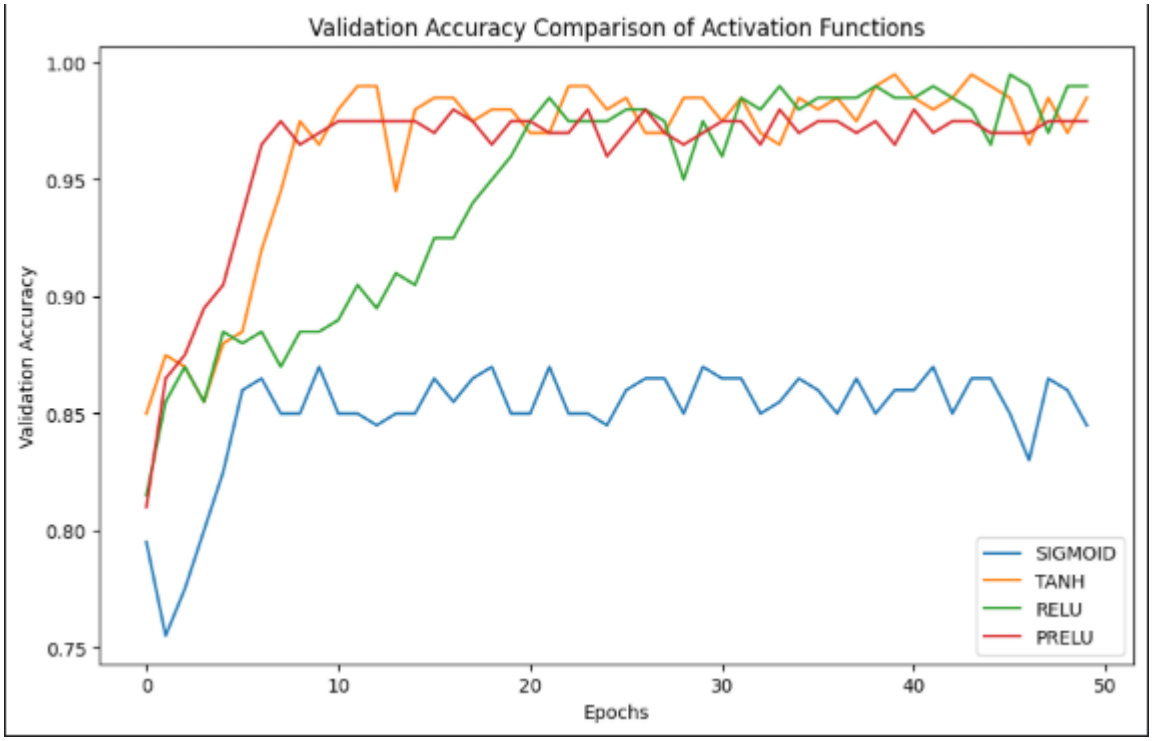
Boxplots for Feature Outliers Detection

```python
# Boxplots for feature outliers detection
fig, axes = plt.subplots(1, 2, figsize=(12, 5))
sns.boxplot(y=df['Feature 1'], ax=axes[0], color='blue')
axes[0].set_title("Feature 1 Boxplot")
sns.boxplot(y=df['Feature 2'], ax=axes[1], color='red')
axes[1].set_title("Feature 2 Boxplot")
plt.show()
```

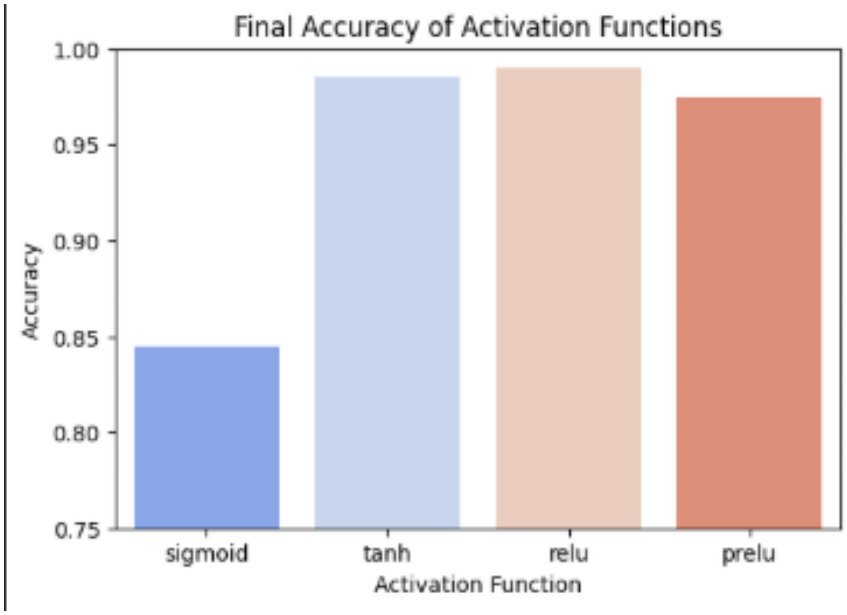


5.3 Model Building and Visualization

```python
# Function to create and train a model with a given activation function
def train_model(activation, alpha=None):
    model = Sequential()
    model.add(Dense(10, input_shape=(2,), activation=activation if activation != 'prelu' else None))
    if activation == 'prelu':
        model.add(PReLU())
    model.add(Dense(10, activation=activation if activation != 'prelu' else None))
    if activation == 'prelu':
        model.add(PReLU())
    model.add(Dense(1, activation='sigmoid'))  # Output layer for binary classification

    # Compile model
    model.compile(optimizer=Adam(learning_rate=0.01), loss='binary_crossentropy', metrics=['accuracy'])

    # Train model
    history = model.fit(X_train, y_train, epochs=50, validation_data=(X_test, y_test), verbose=0)

    # Evaluate model
    loss, accuracy = model.evaluate(X_test, y_test, verbose=0)

    return history, accuracy

# Train models with different activation functions
activations = ['sigmoid', 'tanh', 'relu', 'prelu']
results = {}

for act in activations:
    history, acc = train_model(act)
    results[act] = {'history': history, 'accuracy': acc}

# Plot training accuracy for each activation function
plt.figure(figsize=(10,6))
for act in activations:
    plt.plot(results[act]['history'].history['val_accuracy'], label=f"{act.upper()}")

plt.title("Validation Accuracy Comparison of Activation Functions")
plt.xlabel("Epochs")
plt.ylabel("Validation Accuracy")
plt.legend()
plt.show()

# Display final accuracy for each activation function
accuracy_results = pd.DataFrame.from_dict({act: [results[act]['accuracy']] for act in activations}, orient='index', columns=['Final Accuracy'])
# Print final accuracy results
print("\nFinal Accuracy for Each Activation Function:")
print(accuracy_results)

# Display results as a table
import seaborn as sns

plt.figure(figsize=(6,4))
sns.barplot(x=accuracy_results.index, y=accuracy_results['Final Accuracy'], palette="coolwarm")
plt.title("Final Accuracy of Activation Functions")
plt.xlabel("Activation Function")
plt.ylabel("Accuracy")
plt.ylim(0.75, 1.0)  # Ensuring better visualization
plt.show()
```




```
Final Accuracy for Each Activation Function:
         Final Accuracy
sigmoid           0.845
tanh              0.985
relu              0.990
prelu             0.975
```



Module 6: Decision Boundary Visualization

Now that we have trained the models using different activation functions and compared their accuracy, the next step is to analyze how each activation function affects decision boundaries.

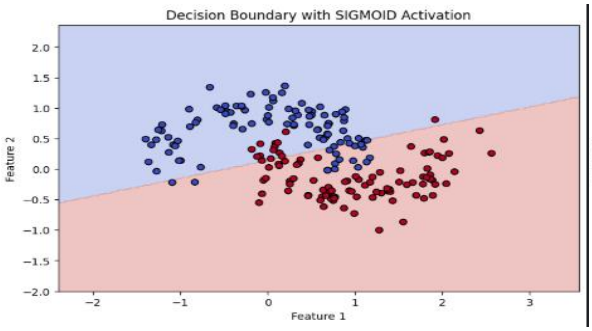
Why is this important?

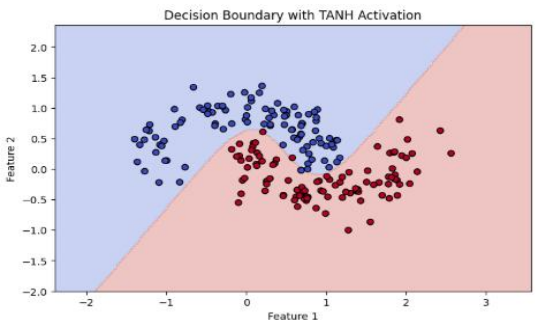
Helps us visually understand how each activation function learns patterns in the dataset. Shows whether non-linearity is captured well (e.g., Sigmoid vs. ReLU). Allows us to compare model generalization (smooth vs. sharp decision boundaries).

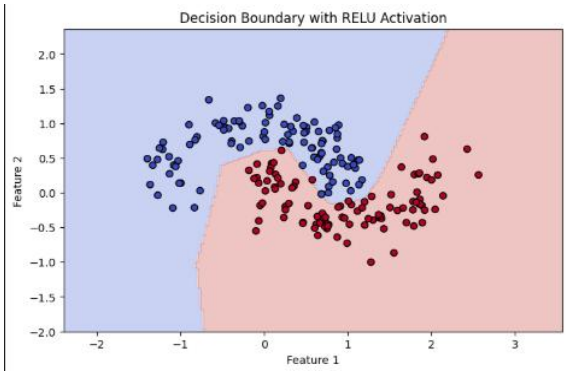
- ◇ Steps to Implement Decision Boundary Visualization

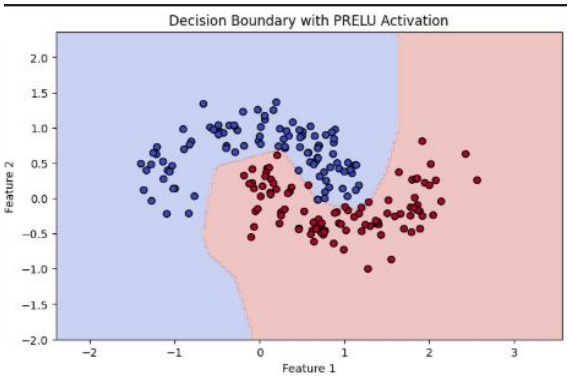
1. Generate a mesh grid of input values.
2. Predict class labels using the trained model for each point in the grid.
3. Plot the decision boundary for each activation function.
4. Overlay the actual dataset points to compare predictions.

```
# Redefining the list of activation functions
activations = ['sigmoid', 'tanh', 'relu', 'prelu']

# Function to plot decision boundaries for different activation functions
def plot_decision_boundary(model, X, y, title):
    # Create a mesh grid
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + 1
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + 1
    xx, yy = np.meshgrid(np.linspace(x_min, x_max, 200), np.linspace(y_min, y_max, 200))
    grid = np.c_[xx.ravel(), yy.ravel()]

    # Predict labels for each point in the mesh grid
    Z = model.predict(grid)
    Z = (Z > 0.5).astype(int)  # Convert probabilities to class labels
    Z = Z.reshape(xx.shape)

    # Plot the decision boundary
    plt.figure(figsize=(8,5))
    plt.contourf(xx, yy, Z, alpha=0.3, cmap="coolwarm")
    plt.scatter(X[:, 0], X[:, 1], c=y, cmap="coolwarm", edgecolors="k")
    plt.xlabel("Feature 1")
    plt.ylabel("Feature 2")
    plt.title(title)
    plt.show()

# Train and plot decision boundaries for each activation function
for act in activations:
    model = Sequential()
    model.add(Dense(10, input_shape=(2,), activation=act if act != 'prelu' else None))
    if act == 'prelu':
        model.add(PReLU())
    model.add(Dense(10, activation=act if act != 'prelu' else None))
    if act == 'prelu':
        model.add(PReLU())
    model.add(Dense(1, activation='sigmoid'))  # Output layer for binary classification

    # Compile and train the model
    model.compile(optimizer=Adam(learning_rate=0.01), loss='binary_crossentropy', metrics=['accuracy'])
    model.fit(X_train, y_train, epochs=50, verbose=0)

    # Plot decision boundary
    plot_decision_boundary(model, X_test, y_test, title=f"Decision Boundary with {act.upper()} Activation")
```









Conclusion

1. **Sigmoid**: **Not suitable for deep networks** (vanishing gradients, slow learning). Only useful in **output layers for binary classification**.
2. **Tanh**: **Better than Sigmoid**, but still suffers from vanishing gradients. Suitable for **shallow networks**.
3. **ReLU**: **Best for most deep learning models** (fast learning, no vanishing gradients). Can suffer from **Dying ReLU issue**.
4. **PReLU**: **Best overall** (learns negative slopes dynamically, stable convergence). Slightly **higher computation cost** than ReLU.