

HRISHABH GAUTAM

2347224

NNDL LAB 2

Scenario:

You are tasked with implementing and visualizing various activation functions to observe how they transform inputs and affect the output. You will also train a simple neural network using these activation functions and evaluate their performance.

✓ 1. Implement and Visualize Activation Functions:

Step Function

- Visualize the activation function using matplotlib to observe how they map input values to output values.

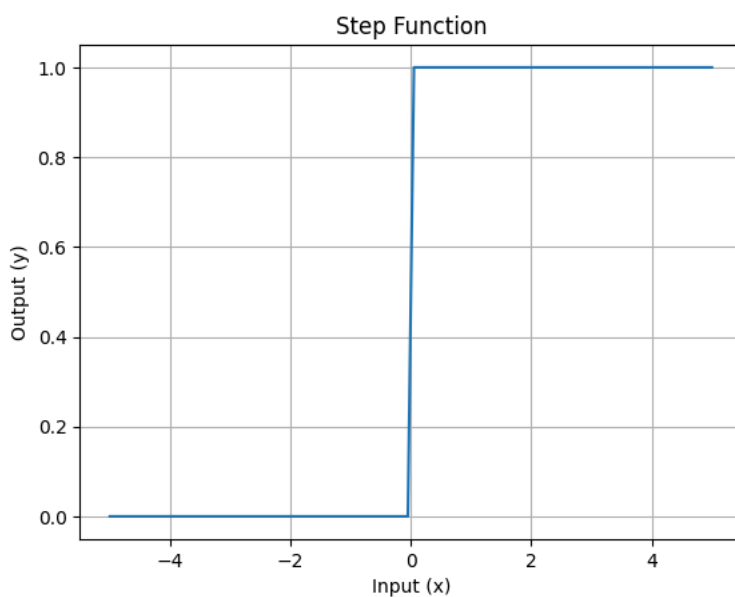
```
import numpy as np
import matplotlib.pyplot as plt

def step_function(x):
    """Step function."""
    return np.where(x >= 0, 1, 0)

# Generate input values
x = np.linspace(-5, 5, 100)

# Calculate the output values for the step function
y_step = step_function(x)

# Plot the step function
plt.plot(x, y_step)
plt.title('Step Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (y)')
plt.grid(True)
plt.show()
```



✓ Sigmoid Function (Binary and Bipolar)

- Visualize the activation function using matplotlib to observe how they map input values to output values.

```

import numpy as np
import matplotlib.pyplot as plt

def sigmoid_binary(x):
    """Binary Sigmoid function."""
    return 1 / (1 + np.exp(-x))

def sigmoid_bipolar(x):
    """Bipolar Sigmoid function."""
    return (2 / (1 + np.exp(-x))) - 1

# Generate input values
x = np.linspace(-5, 5, 100)

# Calculate the output values for the sigmoid functions
y_sigmoid_binary = sigmoid_binary(x)
y_sigmoid_bipolar = sigmoid_bipolar(x)

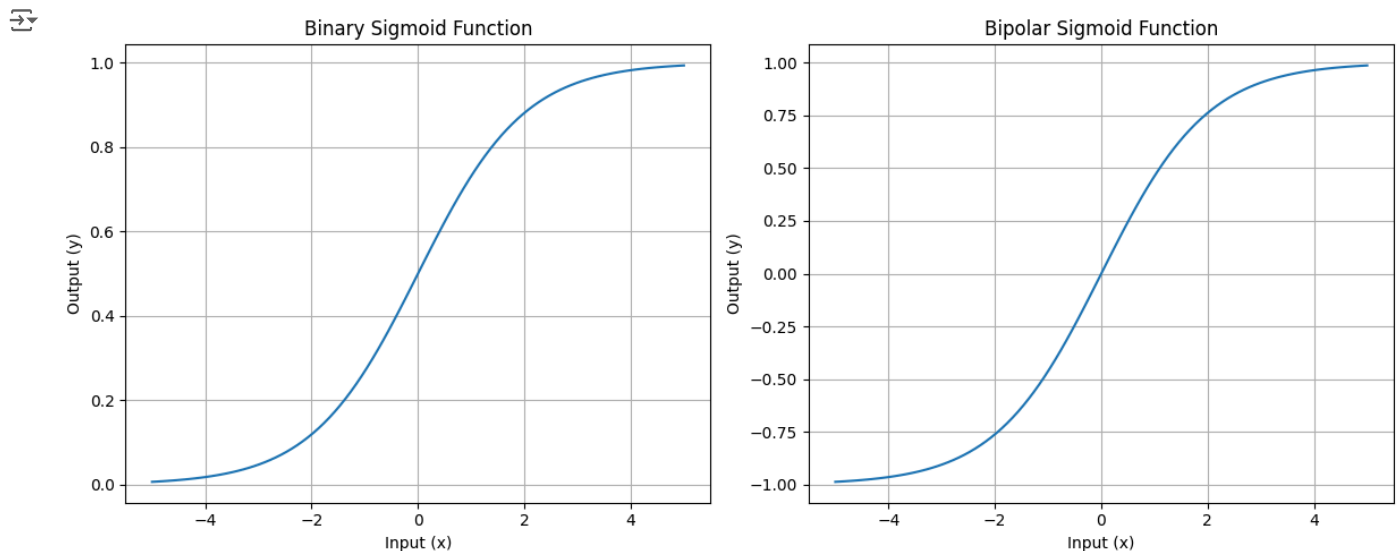
# Plot the sigmoid functions
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.plot(x, y_sigmoid_binary)
plt.title('Binary Sigmoid Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (y)')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(x, y_sigmoid_bipolar)
plt.title('Bipolar Sigmoid Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (y)')
plt.grid(True)

plt.tight_layout()
plt.show()

```



✓ Tanh Function

- Visualize the activation function using matplotlib to observe how they map input values to output values.

```

import numpy as np
import matplotlib.pyplot as plt

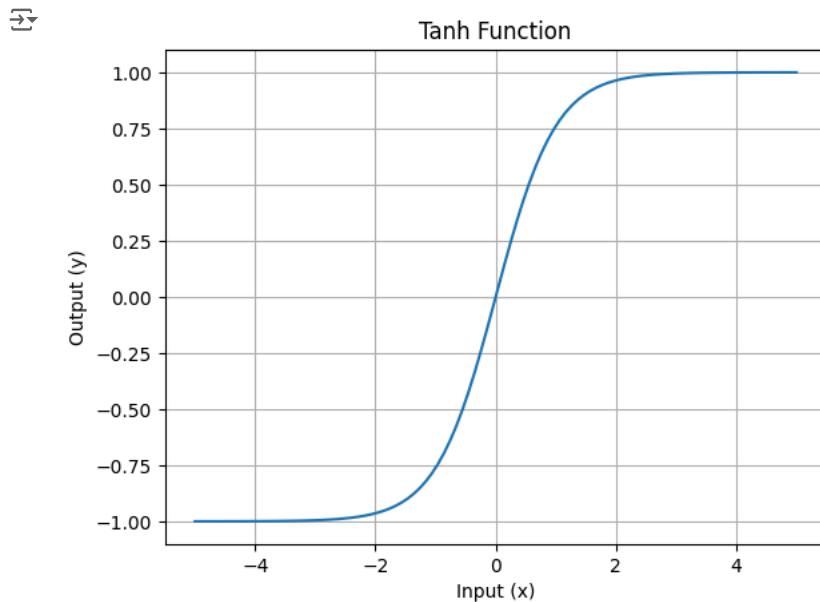
def tanh_function(x):
    """Tanh function."""
    return np.tanh(x)

```

```
# Generate input values
x = np.linspace(-5, 5, 100)

# Calculate the output values for the tanh function
y_tanh = tanh_function(x)

# Plot the tanh function
plt.plot(x, y_tanh)
plt.title('Tanh Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (y)')
plt.grid(True)
plt.show()
```



✓ ReLU Function

- Visualize the activation function using matplotlib to observe how they map input values to output values.

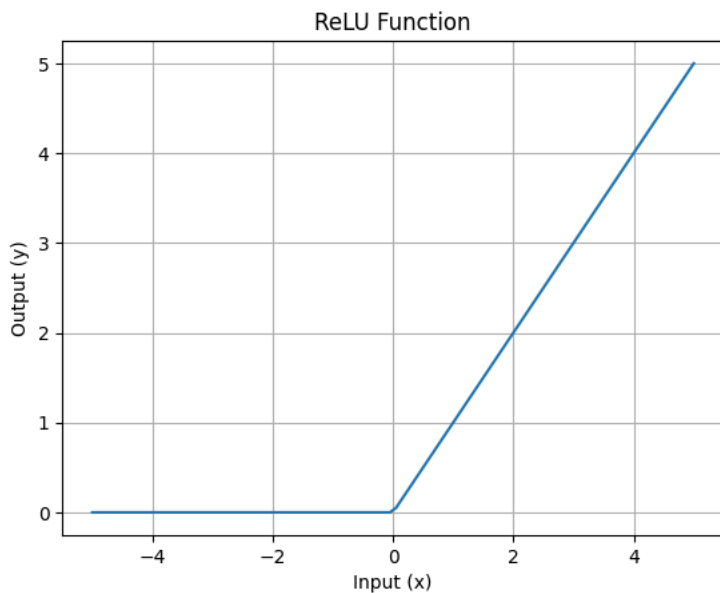
```
import numpy as np
import matplotlib.pyplot as plt

def relu_function(x):
    """ReLU function."""
    return np.maximum(0, x)
```

```
# Generate input values
x = np.linspace(-5, 5, 100)

# Calculate the output values for the ReLU function
y_relu = relu_function(x)

# Plot the ReLU function
plt.plot(x, y_relu)
plt.title('ReLU Function')
plt.xlabel('Input (x)')
plt.ylabel('Output (y)')
plt.grid(True)
plt.show()
```



✓ 2. Implement a Simple Neural Network:

- Create a simple neural network with one hidden layer using each activation function (sigmoid, tanh, and ReLU).

```
import numpy as np
```

```
class NeuralNetwork:
```

```
    def __init__(self, input_size, hidden_size, output_size, activation_function):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.activation_function = activation_function
```

```
        # Initialize weights and biases randomly
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.random.randn(hidden_size)
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.random.randn(output_size)
```

```
    def forward(self, x):
        # Calculate hidden layer output
        hidden_layer_input = np.dot(x, self.weights_input_hidden) + self.bias_hidden
        hidden_layer_output = self.activation_function(hidden_layer_input)

        # Calculate output layer output
        output_layer_input = np.dot(hidden_layer_output, self.weights_hidden_output) + self.bias_output
        output_layer_output = self.activation_function(output_layer_input)

        return output_layer_output
```

```
# Example usage:
```

```
input_size = 2
hidden_size = 3
output_size = 1
```

```
# Create neural networks with different activation functions
nn_sigmoid = NeuralNetwork(input_size, hidden_size, output_size, sigmoid_binary)
nn_tanh = NeuralNetwork(input_size, hidden_size, output_size, tanh_function)
nn_relu = NeuralNetwork(input_size, hidden_size, output_size, relu_function)
```

```
# Sample input data
input_data = np.array([0.5, 0.8])
```

```
# Perform forward pass and get outputs
output_sigmoid = nn_sigmoid.forward(input_data)
output_tanh = nn_tanh.forward(input_data)
output_relu = nn_relu.forward(input_data)
```

```
print("Output with Sigmoid:", output_sigmoid)
print("Output with Tanh:", output_tanh)
print("Output with ReLU:", output_relu)
```

```

Output with Sigmoid: [0.47477065]
Output with Tanh: [-0.86794851]
Output with ReLU: [0.77694365]

```

- ✓ Train the network on a binary classification task (e.g. AND problem) using a small dataset i.e heart disease prediction after fetching it from uci repository that has 0 and 1 as binary categories.

Compare the performance of the neural network with different activation functions by calculating their accuracy.

```

import numpy as np
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
import requests
from io import StringIO

# Activation functions
def sigmoid(x):
    return 1 / (1 + np.exp(-x))

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

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

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size, activation_function):
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.output_size = output_size
        self.activation_function = activation_function

        # Initialize weights with Xavier/Glorot initialization
        self.W1 = np.random.randn(self.input_size, self.hidden_size) * np.sqrt(2.0 / (self.input_size + self.hidden_size))
        self.b1 = np.zeros((1, self.hidden_size))
        self.W2 = np.random.randn(self.hidden_size, self.output_size) * np.sqrt(2.0 / (self.hidden_size + self.output_size))
        self.b2 = np.zeros((1, self.output_size))

    def forward(self, X):
        self.z1 = np.dot(X, self.W1) + self.b1
        self.a1 = self.activation_function(self.z1)
        self.z2 = np.dot(self.a1, self.W2) + self.b2
        self.a2 = sigmoid(self.z2)
        return self.a2

    def backward(self, X, y, output, learning_rate):
        m = X.shape[0]

        dZ2 = output - y
        dW2 = np.dot(self.a1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m

        dZ1 = np.dot(dZ2, self.W2.T) * (self.a1 > 0) # Using ReLU derivative for all activation functions
        dW1 = np.dot(X.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m

        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1

    def train(self, X, y, epochs, learning_rate, batch_size=32):
        for _ in range(epochs):
            for i in range(0, X.shape[0], batch_size):
                X_batch = X[i:i+batch_size]
                y_batch = y[i:i+batch_size]
                output = self.forward(X_batch)
                self.backward(X_batch, y_batch, output, learning_rate)

    def predict(self, X):
        return (self.forward(X) > 0.5).astype(int)

# Fetch and prepare the dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/heart-disease/processed.cleveland.data"
response = requests.get(url)
data = StringIO(response.text)

```

```

df = pd.read_csv(data, header=None)

# Preprocess the data
df = df.replace('?', np.nan).dropna()
X = df.iloc[:, :-1].astype(float)
y = (df.iloc[:, -1] > 0).astype(int) # Binary classification: 0 for no disease, 1 for disease

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# Scale the features
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

# Train and evaluate models with different activation functions
input_size = X_train_scaled.shape[1]
hidden_size = 64
output_size = 1
epochs = 5000
learning_rate = 0.001
batch_size = 32

activation_functions = [sigmoid, tanh, relu]
accuracies = []

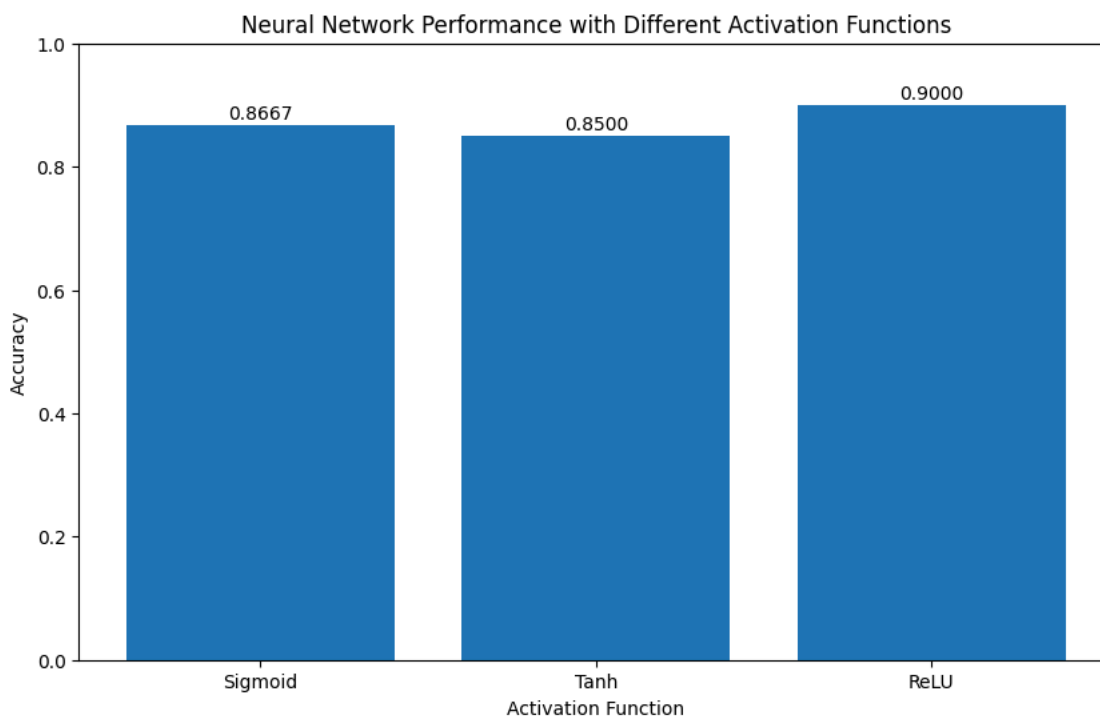
for activation in activation_functions:
    nn = NeuralNetwork(input_size, hidden_size, output_size, activation)
    nn.train(X_train_scaled, y_train.values.reshape(-1, 1), epochs, learning_rate, batch_size)

    y_pred = nn.predict(X_test_scaled)
    accuracy = accuracy_score(y_test, y_pred)
    accuracies.append(accuracy)
    print(f"Accuracy with {activation.__name__} activation: {accuracy:.4f}")

# Visualize the results
plt.figure(figsize=(10, 6))
plt.bar(['Sigmoid', 'Tanh', 'ReLU'], accuracies)
plt.title('Neural Network Performance with Different Activation Functions')
plt.xlabel('Activation Function')
plt.ylabel('Accuracy')
plt.ylim(0, 1)
for i, v in enumerate(accuracies):
    plt.text(i, v + 0.01, f'{v:.4f}', ha='center')
plt.show()

```

↗ Accuracy with sigmoid activation: 0.8667
 Accuracy with tanh activation: 0.8500
 Accuracy with relu activation: 0.9000



Interpretation:

- These results show the performance of a neural network on a classification task (likely heart disease prediction) using different activation functions in the hidden layer.
- The ReLU activation function achieved the highest accuracy at 90.00%, followed by sigmoid at 86.67%, and then tanh at 85.00%.
- All three activation functions performed well, with accuracies above 85%, indicating that the network can effectively learn the underlying patterns in the data regardless of the activation function used.
- The ReLU function's superior performance in this case might be due to its ability to mitigate the vanishing gradient problem, allowing for faster and more effective training of deep neural networks.
- The sigmoid function's strong performance (86.67%) aligns with its suitability for binary classification tasks, as it outputs values between 0 and 1.
- The tanh function, while performing slightly lower than the others, still achieved a good accuracy of 85.00%, demonstrating its effectiveness in this context.
- The relatively small gap between the best and worst results (5%) suggests that all these activation functions are viable options for this particular problem, with ReLU showing a slight edge in performance.