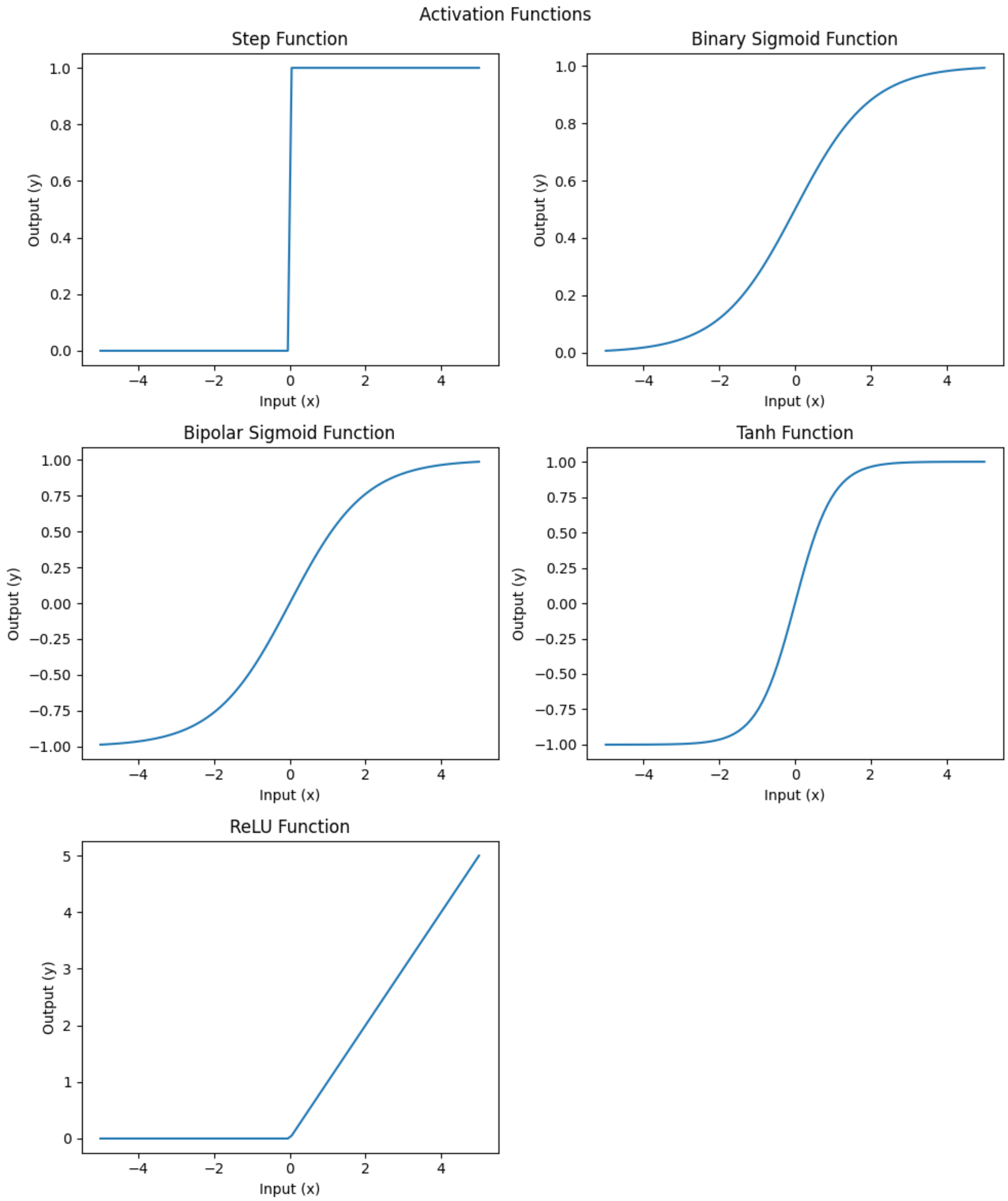
anshika-103-lab2

September 24, 2024

[1]:
```python
# 1. Implement and Visualize Activation Functions:
# o Implement the following activation functions in Python:
#   Step Function
#   Sigmoid Function (Binary and Bipolar)
#   Tanh Function
#   ReLU Function
# o Visualize each activation function using matplotlib/seaborn/bokeh to
# observe how they map input values to output values.

import numpy as np
import matplotlib.pyplot as plt


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


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


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


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


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

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

# Calculate outputs for each activation function
y_step = step_function(x)
y_sigmoid_binary = sigmoid_binary(x)
y_sigmoid_bipolar = sigmoid_bipolar(x)
y_tanh = tanh_function(x)
y_relu = relu_function(x)

# Create subplots
fig, axs = plt.subplots(3, 2, figsize=(10, 12))
axs = axs.ravel()

# Plot each activation function
axs[0].plot(x, y_step)
axs[0].set_title('Step Function')

axs[1].plot(x, y_sigmoid_binary)
axs[1].set_title('Binary Sigmoid Function')

axs[2].plot(x, y_sigmoid_bipolar)
axs[2].set_title('Bipolar Sigmoid Function')

axs[3].plot(x, y_tanh)
axs[3].set_title('Tanh Function')

axs[4].plot(x, y_relu)
axs[4].set_title('ReLU Function')

# Remove the last subplot since we only have 5 activation functions
fig.delaxes(axs[5])


# Set common labels and title
for ax in axs[:5]:
  ax.set_xlabel('Input (x)')
  ax.set_ylabel('Output (y)')

fig.suptitle('Activation Functions')
plt.tight_layout()
plt.show()
```



[ ]:
```
# The code provided already visualizes how the activation functions map input␣
↪values to output values.
# The plots show the relationship between the input (x) and the output (y) for␣
↪each function.
#
# For example, the step function shows that any input below 0 results in an␣
↪output of 0, and any input at or above 0 results in an output of 1.
```

```
# The sigmoid function shows a smooth, S-shaped curve, mapping inputs to a range of 0 to 1 for the binary sigmoid and -1 to 1 for the bipolar sigmoid.
# The tanh function also shows an S-shaped curve but maps inputs to a range of -1 to 1.
# The ReLU function shows a linear relationship for positive inputs, with an output of 0 for negative inputs.
#
# By examining these plots, you can understand how each activation function behaves and how it affects the output of a neural network based on its input.

# If you want to further explore the relationship, you can try:
# - Changing the range of input values (x)
# - Zooming in on specific parts of the plots
# - Adding more data points for smoother curves
# - Comparing the plots of different activation functions side-by-side
```

[ ]:

```
# Interpretation of Activation Functions

# 1. Step Function:
# - Output is binary (0 or 1).
# - It's a threshold-based function.
# - Useful for simple binary classification tasks.
# - Not differentiable, which can be a problem for gradient-based optimization in neural networks.


# 2. Binary Sigmoid Function:
# - Output is between 0 and 1.
# - Smooth, S-shaped curve.
# - Useful for binary classification problems where you want a probability-like output.
# - Differentiable, allowing for gradient-based training.


# 3. Bipolar Sigmoid Function:
# - Output is between -1 and 1.
# - Smooth, S-shaped curve similar to the binary sigmoid but centered around 0.
# - Useful when you want outputs that can represent both positive and negative values.
# - Differentiable.


# 4. Tanh Function (Hyperbolic Tangent):
# - Output is between -1 and 1.
# - Smooth, S-shaped curve.
```

```
# - Often preferred over the sigmoid function in hidden layers because it has␣
↪zero-centered output.
# - Differentiable.


# 5. ReLU Function (Rectified Linear Unit):
# - Output is 0 for negative inputs and the input itself for positive inputs.
# - Non-linear function.
# - Very popular in deep learning due to its efficiency and ability to avoid␣
↪vanishing gradient problems.
# - Differentiable almost everywhere (except at 0), but the derivative at 0 is␣
↪undefined.

# In general, the choice of activation function depends on the specific task,␣
↪network architecture, and desired properties of the output.
# For example, Sigmoid is often used in the output layer for binary␣
↪classification, while ReLU is preferred in hidden layers of deep neural␣
↪networks.
```

[4]:

```
import numpy as np

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

        # Initialize weights and biases randomly
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

    def forward(self, X):
        # Calculate hidden layer output
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.
↪bias_hidden
        self.hidden_layer_output = self.activation(self.hidden_layer_input)

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

        return self.output_layer_output
```

```python
# Example usage:

# Input data
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

# Expected output (XOR problem)
y = np.array([[0], [1], [1], [0]])

# Create neural networks with different activation functions
def sigmoid(x):
  """Sigmoid activation function."""
  return 1 / (1 + np.exp(-x))

def tanh(x): # define the tanh function here so it is in scope
  """Tanh activation function."""
  return np.tanh(x)

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

nn_sigmoid = NeuralNetwork(2, 4, 1, sigmoid)
nn_tanh = NeuralNetwork(2, 4, 1, tanh)
nn_relu = NeuralNetwork(2, 4, 1, relu)


# Perform forward pass for each network
output_sigmoid = nn_sigmoid.forward(X)
output_tanh = nn_tanh.forward(X)
output_relu = nn_relu.forward(X)

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

```
Output with Sigmoid activation: [[0.44137492]
 [0.53486142]
 [0.35556009]
 [0.45686501]]
Output with Tanh activation: [[ 0.        ]
 [ 0.33106024]
 [-0.52886077]
 [ 0.12091578]]
Output with ReLU activation: [[0.        ]
```

```
[0.        ]
[1.96100262]
[0.        ]]
```

[ ]:
```
# Interpretation of the Code and Functions

# 1. Activation Functions:

# - `sigmoid(x)`: This function calculates the sigmoid activation. It squashes␣
↪the input into a range between 0 and 1. It's often used in output layers for␣
↪binary classification problems as it can be interpreted as a probability.

# - `relu(x)`: This function calculates the Rectified Linear Unit (ReLU)␣
↪activation. It outputs the input directly if it's positive and 0 if it's␣
↪negative. It's widely used in deep learning because of its efficiency and␣
↪ability to avoid vanishing gradient problems.

# - `tanh(x)`: This function calculates the hyperbolic tangent activation. It␣
↪outputs a value between -1 and 1. It's often preferred over the sigmoid␣
↪function in hidden layers because it has zero-centered output.

# - `leaky_relu(x)`: This function calculates the Leaky ReLU activation. It's a␣
↪variant of ReLU that allows a small, non-zero gradient for negative inputs,␣
↪which helps mitigate the "dying ReLU" problem.

# 2. Plotting Activation Functions:

# The code generates plots for different activation functions. It uses␣
↪Matplotlib to visualize the relationship between input (x) and output (y)␣
↪for each function. This helps you understand how each activation function␣
↪transforms its input.

# 3. `NeuralNetwork` Class:

# - `__init__(self, input_size, hidden_size, output_size, activation)`: The␣
↪constructor initializes the neural network with the specified sizes for the␣
↪input, hidden, and output layers. It also takes the activation function to␣
↪be used in the network. It initializes the weights and biases of the network␣
↪randomly.

# - `forward(self, X)`: This method performs the forward pass of the network.␣
↪It calculates the output of the hidden layer using the input data, weights,␣
↪and biases. Then it applies the activation function to the hidden layer␣
↪output. Next, it calculates the output of the output layer using the hidden␣
↪layer output, weights, and biases, and applies the activation function again.

# 4. Example Usage:
```

```
# The example creates a neural network for the XOR problem (a classic problem␣
↪in machine learning).
# - It defines the input data `X` and the expected output `y`.
# - Then it creates three neural networks, each using a different activation␣
↪function (Sigmoid, Tanh, and ReLU).
# - It performs a forward pass for each network and prints the output.
# - The output of the neural networks may not be accurate because they are not␣
↪trained.

# 5. Interpretation of Each Function and Code:
# - The code creates a neural network class with the capacity of performing a␣
↪forward pass and has the capability of changing the activation function and␣
↪visualizing it with various activation functions.
# - The functions defined are activation functions and their implementations␣
↪are based on Numpy to handle matrix and vector operations.
# - The code demonstrates how to implement different activation functions and␣
↪how to create and run a neural network with them.
```

[5]:
```
#  Train the network on a binary classification task (e.g., XOR problem) using a
# small dataset.

import numpy as np

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

def sigmoid_derivative(x):
  """Derivative of the sigmoid function."""
  return x * (1 - x)

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

        # Initialize weights and biases randomly
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

    def forward(self, X):
        # Calculate hidden layer output
```

```
        self.hidden_layer_input = np.dot(X, self.weights_input_hidden) + self.
↪bias_hidden
        self.hidden_layer_output = sigmoid(self.hidden_layer_input)

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

        return self.output_layer_output

    def backward(self, X, y, learning_rate):
        # Calculate output layer error
        output_error = y - self.output_layer_output
        output_delta = output_error * sigmoid_derivative(self.
↪output_layer_output)

        # Calculate hidden layer error
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * sigmoid_derivative(self.
↪hidden_layer_output)

        # Update weights and biases
        self.weights_hidden_output += self.hidden_layer_output.T.
↪dot(output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) *␣
↪learning_rate
        self.weights_input_hidden += X.T.dot(hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) *␣
↪learning_rate

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

            # Backward pass and weight update
            self.backward(X, y, learning_rate)

            # Print loss every 100 epochs
            if epoch % 100 == 0:
                loss = np.mean(np.square(y - output))
                print(f"Epoch {epoch}, Loss: {loss}")


# Example usage (XOR problem):
```

```python
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
y = np.array([[0], [1], [1], [0]])

nn = NeuralNetwork(2, 4, 1)
nn.train(X, y, epochs=10000, learning_rate=0.1)

# Test the trained network
output = nn.forward(X)
print("Final Output:", output)
```

```
Epoch 0, Loss: 0.2663637025597461
Epoch 100, Loss: 0.2569107783759882
Epoch 200, Loss: 0.25069449650673803
Epoch 300, Loss: 0.2463820704867954
Epoch 400, Loss: 0.24274021314313365
Epoch 500, Loss: 0.23883997882026517
Epoch 600, Loss: 0.23396760652048873
Epoch 700, Loss: 0.22749811973452183
Epoch 800, Loss: 0.21884099836355075
Epoch 900, Loss: 0.2074572186970398
Epoch 1000, Loss: 0.1929407225248284
Epoch 1100, Loss: 0.17519575043077368
Epoch 1200, Loss: 0.15471017331302772
Epoch 1300, Loss: 0.1327499734926194
Epoch 1400, Loss: 0.1111429631865925
Epoch 1500, Loss: 0.09159609151599615
Epoch 1600, Loss: 0.07507848861667338
Epoch 1700, Loss: 0.06174208043885886
Epoch 1800, Loss: 0.05123117021518532
Epoch 1900, Loss: 0.04301421990763283
Epoch 2000, Loss: 0.036576628881190294
Epoch 2100, Loss: 0.031492394927720074
Epoch 2200, Loss: 0.027432779847138326
Epoch 2300, Loss: 0.024151879285893168
Epoch 2400, Loss: 0.021467767711019655
Epoch 2500, Loss: 0.019245878384854104
Epoch 2600, Loss: 0.01738610750322379
Epoch 2700, Loss: 0.01581333096375854
Epoch 2800, Loss: 0.014470587502301152
Epoch 2900, Loss: 0.013314214843218528
Epoch 3000, Loss: 0.012310376745610356
Epoch 3100, Loss: 0.011432570087592341
Epoch 3200, Loss: 0.01065982147053112
Epoch 3300, Loss: 0.009975370714111162
Epoch 3400, Loss: 0.009365700415803301
Epoch 3500, Loss: 0.008819813494719652
Epoch 3600, Loss: 0.008328690043644765
```

```
Epoch 3700, Loss: 0.007884875052315509
Epoch 3800, Loss: 0.007482162555441527
Epoch 3900, Loss: 0.007115351491511053
Epoch 4000, Loss: 0.0067800553804865955
Epoch 4100, Loss: 0.006472552750074063
Epoch 4200, Loss: 0.006189668677314745
Epoch 4300, Loss: 0.005928680283704465
Epoch 4400, Loss: 0.0056872408145467995
Epoch 4500, Loss: 0.00546331824437556
Epoch 4600, Loss: 0.0052551453172605895
Epoch 4700, Loss: 0.0050611786497111524
Epoch 4800, Loss: 0.0048800650625108796
Epoch 4900, Loss: 0.00471061371439037
Epoch 5000, Loss: 0.00455177291955349
Epoch 5100, Loss: 0.004402610767710875
Epoch 5200, Loss: 0.004262298847636398
Epoch 5300, Loss: 0.004130098516695034
Epoch 5400, Loss: 0.0040053492691505006
Epoch 5500, Loss: 0.003887458842682285
Epoch 5600, Loss: 0.0037758947709127045
Epoch 5700, Loss: 0.003670177144003461
Epoch 5800, Loss: 0.0035698723826632257
Epoch 5900, Loss: 0.003474587865607172
Epoch 6000, Loss: 0.0033839672784604902
Epoch 6100, Loss: 0.00329768657471722
Epoch 6200, Loss: 0.0032154504577511104
Epoch 6300, Loss: 0.00313698930788338
Epoch 6400, Loss: 0.0030620564908132985
Epoch 6500, Loss: 0.0029904259938397047
Epoch 6600, Loss: 0.0029218903446626967
Epoch 6700, Loss: 0.0028562587744865816
Epoch 6800, Loss: 0.0027933555929120606
Epoch 6900, Loss: 0.002733018746919921
Epoch 7000, Loss: 0.00267509854028095
Epoch 7100, Loss: 0.002619456493115002
Epoch 7200, Loss: 0.0025659643241774926
Epoch 7300, Loss: 0.002514503040865759
Epoch 7400, Loss: 0.0024649621239839483
Epoch 7500, Loss: 0.002417238796044846
Epoch 7600, Loss: 0.0023712373633701647
Epoch 7700, Loss: 0.0023268686235179345
Epoch 7800, Loss: 0.0022840493306516994
Epoch 7900, Loss: 0.002242701712398922
Epoch 8000, Loss: 0.0022027530325489995
Epoch 8100, Loss: 0.0021641351946341085
Epoch 8200, Loss: 0.0021267843820355417
Epoch 8300, Loss: 0.002090640730777795
Epoch 8400, Loss: 0.002055648031623226
```

```
Epoch 8500, Loss: 0.0020217534584738056
Epoch 8600, Loss: 0.0019889073204280164
Epoch 8700, Loss: 0.001957062835140729
Epoch 8800, Loss: 0.0019261759213958921
Epoch 8900, Loss: 0.001896205009031254
Epoch 9000, Loss: 0.001867110864556847
Epoch 9100, Loss: 0.0018388564309859677
Epoch 9200, Loss: 0.0018114066805547048
Epoch 9300, Loss: 0.0017847284791439586
Epoch 9400, Loss: 0.001758790461340514
Epoch 9500, Loss: 0.0017335629151821442
Epoch 9600, Loss: 0.0017090176757277795
Epoch 9700, Loss: 0.0016851280266795238
Epoch 9800, Loss: 0.0016618686093590124
Epoch 9900, Loss: 0.0016392153384089119
Final Output: [[0.02713143]
 [0.95847843]
 [0.96043197]
 [0.04942466]]
```

[ ]:

```
# Interpretation of the Code

# 1. Activation Functions:
#   - `sigmoid(x)`: The sigmoid function squashes the input to a range between
↪0 and 1.
#     It's used in both the hidden and output layers for this neural network.
#   - `sigmoid_derivative(x)`: This function calculates the derivative of the
↪sigmoid
#     function, which is needed for backpropagation to update the weights.


# 2. `NeuralNetwork` Class:
#   - `__init__(self, input_size, hidden_size, output_size)`:
#     - Initializes the neural network with specified input, hidden, and output
↪layer
#       sizes.
#     - Randomly initializes weights and biases for the connections between the
↪input
#       and hidden layers, and the hidden and output layers.


# 3. `forward(self, X)`:
#   - Performs the forward pass through the network.
#   - Calculates the hidden layer output by performing a weighted sum of the
↪input data
#     and applying the sigmoid activation function.
#   - Calculates the output layer output using a similar process with the hidden
```

```
#     layer's output.


# 4. `backward(self, X, y, learning_rate)`:
#   - Performs the backpropagation algorithm.
#   - Calculates the error in the output layer by comparing the network's␣
↪output to
#     the expected output.
#   - Uses the chain rule to calculate the error for the hidden layer.
#   - Updates the weights and biases of the network based on the calculated␣
↪errors
#     using gradient descent.


# 5. `train(self, X, y, epochs, learning_rate)`:
#   - Trains the neural network for a specified number of `epochs`.
#   - In each epoch:
#     - Performs a forward pass to get the network's output.
#     - Performs a backward pass to calculate the errors and update the weights␣
↪and
#       biases.
#     - Prints the loss (mean squared error) every 100 epochs to track the␣
↪progress
#       of the training.


# 6. Example Usage (XOR Problem):
#   - Creates a neural network with 2 input nodes, 4 hidden nodes, and 1 output
#     node.
#   - Defines the input data `X` and the expected output `y` for the XOR␣
↪problem.
#   - Trains the network using the `train()` method.
#   - Tests the trained network by making a forward pass with the input data and
#     prints the final output.

# 7. XOR Problem:
#   - The XOR problem is a classic example in machine learning that shows how a
#     single-layer perceptron cannot solve it.
#   - This code demonstrates how a multi-layer perceptron (with a hidden layer)␣
↪can
#     learn to solve the XOR problem by adjusting its weights and biases during
#     training.
```

[6]:
```
#  Compare the performance of the neural network with different activation
# functions.
```

```python
import numpy as np

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

def sigmoid_derivative(x):
  """Derivative of the sigmoid function."""
  return x * (1 - x)

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

def tanh_derivative(x):
  """Derivative of the tanh function."""
  return 1 - np.square(x)

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

def relu_derivative(x):
  """Derivative of the ReLU function."""
  return np.where(x > 0, 1, 0)


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

        # Initialize weights and biases randomly
        self.weights_input_hidden = np.random.randn(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.randn(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))

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

```
        self.hidden_layer_output = self.activation_function(self.
↪hidden_layer_input)

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

        return self.output_layer_output

    def backward(self, X, y, learning_rate):
        # Calculate output layer error
        output_error = y - self.output_layer_output
        output_delta = output_error * self.derivative_function(self.
↪output_layer_output)

        # Calculate hidden layer error
        hidden_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_delta = hidden_error * self.derivative_function(self.
↪hidden_layer_output)

        # Update weights and biases
        self.weights_hidden_output += self.hidden_layer_output.T.
↪dot(output_delta) * learning_rate
        self.bias_output += np.sum(output_delta, axis=0, keepdims=True) *␣
↪learning_rate
        self.weights_input_hidden += X.T.dot(hidden_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_delta, axis=0, keepdims=True) *␣
↪learning_rate

    def train(self, X, y, epochs, learning_rate):
        for epoch in range(epochs):
            # Forward pass
            output = self.forward(X)

            # Backward pass and weight update
            self.backward(X, y, learning_rate)

            # Print loss every 100 epochs
            if epoch % 100 == 0:
                loss = np.mean(np.square(y - output))
                print(f"Epoch {epoch}, Loss: {loss}")

# Example usage (XOR problem):
X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
```

```python
y = np.array([[0], [1], [1], [0]])

epochs = 10000
learning_rate = 0.1

# Create and train neural networks with different activation functions
nn_sigmoid = NeuralNetwork(2, 4, 1, sigmoid, sigmoid_derivative)
nn_tanh = NeuralNetwork(2, 4, 1, tanh, tanh_derivative)
nn_relu = NeuralNetwork(2, 4, 1, relu, relu_derivative)

print("Training Sigmoid Network:")
nn_sigmoid.train(X, y, epochs, learning_rate)
print("\nTraining Tanh Network:")
nn_tanh.train(X, y, epochs, learning_rate)
print("\nTraining ReLU Network:")
nn_relu.train(X, y, epochs, learning_rate)

# Compare performance (e.g., final loss or accuracy on a test set)
output_sigmoid = nn_sigmoid.forward(X)
output_tanh = nn_tanh.forward(X)
output_relu = nn_relu.forward(X)

loss_sigmoid = np.mean(np.square(y - output_sigmoid))
loss_tanh = np.mean(np.square(y - output_tanh))
loss_relu = np.mean(np.square(y - output_relu))

print("\nFinal Loss (Sigmoid):", loss_sigmoid)
print("Final Loss (Tanh):", loss_tanh)
print("Final Loss (ReLU):", loss_relu)

# You can further analyze the performance by plotting the loss curves during␣
↪training,
# testing on a separate dataset, or using other metrics like accuracy or␣
↪precision/recall.
```

```
Training Sigmoid Network:
Epoch 0, Loss: 0.28680219120829487
Epoch 100, Loss: 0.24434548025137937
Epoch 200, Loss: 0.24222244591090847
Epoch 300, Loss: 0.2396226874877884
Epoch 400, Loss: 0.23648656525699746
Epoch 500, Loss: 0.23275441783201245
Epoch 600, Loss: 0.2283715303765847
Epoch 700, Loss: 0.22330533845236294
Epoch 800, Loss: 0.21757019526859517
Epoch 900, Loss: 0.21124465337659018
Epoch 1000, Loss: 0.20446362677406682
```

```
Epoch 1100, Loss: 0.19738156901619378
Epoch 1200, Loss: 0.1901243210742351
Epoch 1300, Loss: 0.1827546583906875
Epoch 1400, Loss: 0.1752641079005644
Epoch 1500, Loss: 0.16758806483047667
Epoch 1600, Loss: 0.1596356731010155
Epoch 1700, Loss: 0.1513272245140445
Epoch 1800, Loss: 0.14263180679072893
Epoch 1900, Loss: 0.13359453473224892
Epoch 2000, Loss: 0.12434148145955923
Epoch 2100, Loss: 0.11505782284181879
Epoch 2200, Loss: 0.10594823041664422
Epoch 2300, Loss: 0.09719696205514752
Epoch 2400, Loss: 0.0889416899581299
Epoch 2500, Loss: 0.08126498979130578
Epoch 2600, Loss: 0.07419938063576781
Epoch 2700, Loss: 0.06773927340408607
Epoch 2800, Loss: 0.061854416276923896
Epoch 2900, Loss: 0.05650170457125924
Epoch 3000, Loss: 0.05163405716580774
Epoch 3100, Loss: 0.047206148474377096
Epoch 3200, Loss: 0.043177344669191445
Epoch 3300, Loss: 0.03951248585438994
Epoch 3400, Loss: 0.03618129479912978
Epoch 3500, Loss: 0.033157174752839594
Epoch 3600, Loss: 0.030415984972019854
Epoch 3700, Loss: 0.02793512372458322
Epoch 3800, Loss: 0.025693008866450685
Epoch 3900, Loss: 0.023668893739559462
Epoch 4000, Loss: 0.021842898091553216
Epoch 4100, Loss: 0.02019613901299034
Epoch 4200, Loss: 0.01871087939501299
Epoch 4300, Loss: 0.017370646867589584
Epoch 4400, Loss: 0.01616030374602396
Epoch 4500, Loss: 0.015066066016348524
Epoch 4600, Loss: 0.014075478538428075
Epoch 4700, Loss: 0.013177357124359012
Epoch 4800, Loss: 0.012361708327750076
Epoch 4900, Loss: 0.011619636343454214
Epoch 5000, Loss: 0.010943244413552579
Epoch 5100, Loss: 0.010325536135577865
Epoch 5200, Loss: 0.009760320337406404
Epoch 5300, Loss: 0.009242121805272917
Epoch 5400, Loss: 0.008766099119296905
Epoch 5500, Loss: 0.00832797011630059
Epoch 5600, Loss: 0.00792394500226026
Epoch 5700, Loss: 0.007550666818748123
Epoch 5800, Loss: 0.007205158780039354
```

```
Epoch 5900, Loss: 0.006884777901078734
Epoch 6000, Loss: 0.006587174301539189
Epoch 6100, Loss: 0.0063102555760028295
Epoch 6200, Loss: 0.0060521556493766514
Epoch 6300, Loss: 0.005811207579305917
Epoch 6400, Loss: 0.005585919816362521
Epoch 6500, Loss: 0.0053749554834763725
Epoch 6600, Loss: 0.005177114285526297
Epoch 6700, Loss: 0.004991316706502136
Epoch 6800, Loss: 0.004816590194304625
Epoch 6900, Loss: 0.004652057071703138
Epoch 7000, Loss: 0.0044969239462015415
Epoch 7100, Loss: 0.004350472421750018
Epoch 7200, Loss: 0.004212050941677084
Epoch 7300, Loss: 0.004081067615247873
Epoch 7400, Loss: 0.003956983900240247
Epoch 7500, Loss: 0.0038393090312258734
Epoch 7600, Loss: 0.003727595098178354
Epoch 7700, Loss: 0.003621432692908016
Epoch 7800, Loss: 0.0035204470519207506
Epoch 7900, Loss: 0.003424294633852025
Epoch 8000, Loss: 0.0033326600778557517
Epoch 8100, Loss: 0.003245253496413061
Epoch 8200, Loss: 0.003161808062129748
Epoch 8300, Loss: 0.0030820778533536238
Epoch 8400, Loss: 0.003005835927981499
Epoch 8500, Loss: 0.002932872598744112
Epoch 8600, Loss: 0.0028629938866443316
Epoch 8700, Loss: 0.0027960201321530903
Epoch 8800, Loss: 0.00273178474630492
Epoch 8900, Loss: 0.0026701330860350397
Epoch 9000, Loss: 0.0026109214400093115
Epoch 9100, Loss: 0.0025540161128589727
Epoch 9200, Loss: 0.002499292597176489
Epoch 9300, Loss: 0.0024466348238880225
Epoch 9400, Loss: 0.002395934482716761
Epoch 9500, Loss: 0.0023470904054112914
Epoch 9600, Loss: 0.0023000080052533894
Epoch 9700, Loss: 0.002254598767094789
Epoch 9800, Loss: 0.002210779782819225
Epoch 9900, Loss: 0.002168473327691836

Training Tanh Network:
Epoch 0, Loss: 0.7703717132274408
Epoch 100, Loss: 0.02182939815555863
Epoch 200, Loss: 0.006046786905529024
Epoch 300, Loss: 0.003079670915165034
Epoch 400, Loss: 0.001980427887115295
```

```
Epoch 500, Loss: 0.0014311127053568276
Epoch 600, Loss: 0.0011080397455115732
Epoch 700, Loss: 0.0008976687439815908
Epoch 800, Loss: 0.0007508336539707764
Epoch 900, Loss: 0.0006430558683107277
Epoch 1000, Loss: 0.0005608735515158275
Epoch 1100, Loss: 0.0004963103084005746
Epoch 1200, Loss: 0.0004443576542115013
Epoch 1300, Loss: 0.00040172118509884056
Epoch 1400, Loss: 0.00036614996861966854
Epoch 1500, Loss: 0.00033605648105073933
Epoch 1600, Loss: 0.00031029061421909124
Epoch 1700, Loss: 0.0002879997440131249
Epoch 1800, Loss: 0.0002685390366765728
Epoch 1900, Loss: 0.0002514122103961651
Epoch 2000, Loss: 0.00023623137505137878
Epoch 2100, Loss: 0.0002226891715020721
Epoch 2200, Loss: 0.00021053904418547392
Epoch 2300, Loss: 0.00019958101495789538
Epoch 2400, Loss: 0.0001896512540097479
Epoch 2500, Loss: 0.000180614319867357
Epoch 2600, Loss: 0.00017235730686012573
Epoch 2700, Loss: 0.0001647853764289108
Epoch 2800, Loss: 0.00015781830629962154
Epoch 2900, Loss: 0.0001513877978449672
Epoch 3000, Loss: 0.0001454353548106315
Epoch 3100, Loss: 0.00013991059726776642
Epoch 3200, Loss: 0.00013476991040859372
Epoch 3300, Loss: 0.00012997535334995794
Epoch 3400, Loss: 0.0001254937715833419
Epoch 3500, Loss: 0.00012129607021723888
Epoch 3600, Loss: 0.00011735661513668916
Epoch 3700, Loss: 0.00011365273664889588
Epoch 3800, Loss: 0.00011016431578764874
Epoch 3900, Loss: 0.0001068734377040367
Epoch 4000, Loss: 0.00010376409982736406
Epoch 4100, Loss: 0.00010082196499168145
Epoch 4200, Loss: 9.803415167408794e-05
Epoch 4300, Loss: 9.538905501658008e-05
Epoch 4400, Loss: 9.287619350402931e-05
Epoch 4500, Loss: 9.04860771217195e-05
Epoch 4600, Loss: 8.821009357336005e-05
Epoch 4700, Loss: 8.604040974704132e-05
Epoch 4800, Loss: 8.396988610502468e-05
Epoch 4900, Loss: 8.199200206849792e-05
Epoch 5000, Loss: 8.010079078969506e-05
Epoch 5100, Loss: 7.829078196629381e-05
Epoch 5200, Loss: 7.655695156826694e-05
```

```
Epoch 5300, Loss: 7.489467752476683e-05
Epoch 5400, Loss: 7.329970056531868e-05
Epoch 5500, Loss: 7.176808953142051e-05
Epoch 5600, Loss: 7.029621057617606e-05
Epoch 5700, Loss: 6.888069975447973e-05
Epoch 5800, Loss: 6.751843857755314e-05
Epoch 5900, Loss: 6.620653216561763e-05
Epoch 6000, Loss: 6.494228968317044e-05
Epoch 6100, Loss: 6.372320678428892e-05
Epoch 6200, Loss: 6.254694983183893e-05
Epoch 6300, Loss: 6.141134168560503e-05
Epoch 6400, Loss: 6.0314348880882895e-05
Epoch 6500, Loss: 5.925407004188705e-05
Epoch 6600, Loss: 5.82287253938439e-05
Epoch 6700, Loss: 5.7236647254501026e-05
Epoch 6800, Loss: 5.627627140031889e-05
Epoch 6900, Loss: 5.534612921517941e-05
Epoch 7000, Loss: 5.444484054036444e-05
Epoch 7100, Loss: 5.3571107154016216e-05
Epoch 7200, Loss: 5.2723706816574396e-05
Epoch 7300, Loss: 5.190148782586257e-05
Epoch 7400, Loss: 5.1103364031808456e-05
Epoch 7500, Loss: 5.032831026629989e-05
Epoch 7600, Loss: 4.9575358148521527e-05
Epoch 7700, Loss: 4.884359223038367e-05
Epoch 7800, Loss: 4.8132146450392536e-05
Epoch 7900, Loss: 4.744020086764631e-05
Epoch 8000, Loss: 4.676697865054201e-05
Epoch 8100, Loss: 4.61117432974013e-05
Epoch 8200, Loss: 4.5473796068490326e-05
Epoch 8300, Loss: 4.4852473610979336e-05
Epoch 8400, Loss: 4.424714576018275e-05
Epoch 8500, Loss: 4.365721350205242e-05
Epoch 8600, Loss: 4.308210708333823e-05
Epoch 8700, Loss: 4.252128425710709e-05
Epoch 8800, Loss: 4.197422865248961e-05
Epoch 8900, Loss: 4.144044825853554e-05
Epoch 9000, Loss: 4.091947401299976e-05
Epoch 9100, Loss: 4.041085848770065e-05
Epoch 9200, Loss: 3.99141746628661e-05
Epoch 9300, Loss: 3.9429014783521566e-05
Epoch 9400, Loss: 3.895498929161791e-05
Epoch 9500, Loss: 3.8491725828117005e-05
Epoch 9600, Loss: 3.803886829977213e-05
Epoch 9700, Loss: 3.7596076005769214e-05
Epoch 9800, Loss: 3.7163022819814096e-05
Epoch 9900, Loss: 3.6739396423612404e-05
```

```
Training ReLU Network:
Epoch 0, Loss: 0.25584173011328093
Epoch 100, Loss: 0.25000000000000006
Epoch 200, Loss: 0.25
Epoch 300, Loss: 0.25
Epoch 400, Loss: 0.25
Epoch 500, Loss: 0.25
Epoch 600, Loss: 0.25
Epoch 700, Loss: 0.25
Epoch 800, Loss: 0.25
Epoch 900, Loss: 0.25
Epoch 1000, Loss: 0.25
Epoch 1100, Loss: 0.25
Epoch 1200, Loss: 0.25
Epoch 1300, Loss: 0.25
Epoch 1400, Loss: 0.25
Epoch 1500, Loss: 0.25
Epoch 1600, Loss: 0.25
Epoch 1700, Loss: 0.25
Epoch 1800, Loss: 0.25
Epoch 1900, Loss: 0.25
Epoch 2000, Loss: 0.25
Epoch 2100, Loss: 0.25
Epoch 2200, Loss: 0.25
Epoch 2300, Loss: 0.25
Epoch 2400, Loss: 0.25
Epoch 2500, Loss: 0.25
Epoch 2600, Loss: 0.25
Epoch 2700, Loss: 0.25
Epoch 2800, Loss: 0.25
Epoch 2900, Loss: 0.25
Epoch 3000, Loss: 0.25
Epoch 3100, Loss: 0.25
Epoch 3200, Loss: 0.25
Epoch 3300, Loss: 0.25
Epoch 3400, Loss: 0.25
Epoch 3500, Loss: 0.25
Epoch 3600, Loss: 0.25
Epoch 3700, Loss: 0.25
Epoch 3800, Loss: 0.25
Epoch 3900, Loss: 0.25
Epoch 4000, Loss: 0.25
Epoch 4100, Loss: 0.25
Epoch 4200, Loss: 0.25
Epoch 4300, Loss: 0.25
Epoch 4400, Loss: 0.25
Epoch 4500, Loss: 0.25
Epoch 4600, Loss: 0.25
```

```
Epoch 4700, Loss: 0.25
Epoch 4800, Loss: 0.25
Epoch 4900, Loss: 0.25
Epoch 5000, Loss: 0.25
Epoch 5100, Loss: 0.25
Epoch 5200, Loss: 0.25
Epoch 5300, Loss: 0.25
Epoch 5400, Loss: 0.25
Epoch 5500, Loss: 0.25
Epoch 5600, Loss: 0.25
Epoch 5700, Loss: 0.25
Epoch 5800, Loss: 0.25
Epoch 5900, Loss: 0.25
Epoch 6000, Loss: 0.25
Epoch 6100, Loss: 0.25
Epoch 6200, Loss: 0.25
Epoch 6300, Loss: 0.25
Epoch 6400, Loss: 0.25
Epoch 6500, Loss: 0.25
Epoch 6600, Loss: 0.25
Epoch 6700, Loss: 0.25
Epoch 6800, Loss: 0.25
Epoch 6900, Loss: 0.25
Epoch 7000, Loss: 0.25
Epoch 7100, Loss: 0.25
Epoch 7200, Loss: 0.25
Epoch 7300, Loss: 0.25
Epoch 7400, Loss: 0.25
Epoch 7500, Loss: 0.25
Epoch 7600, Loss: 0.25
Epoch 7700, Loss: 0.25
Epoch 7800, Loss: 0.25
Epoch 7900, Loss: 0.25
Epoch 8000, Loss: 0.25
Epoch 8100, Loss: 0.25
Epoch 8200, Loss: 0.25
Epoch 8300, Loss: 0.25
Epoch 8400, Loss: 0.25
Epoch 8500, Loss: 0.25
Epoch 8600, Loss: 0.25
Epoch 8700, Loss: 0.25
Epoch 8800, Loss: 0.25
Epoch 8900, Loss: 0.25
Epoch 9000, Loss: 0.25
Epoch 9100, Loss: 0.25
Epoch 9200, Loss: 0.25
Epoch 9300, Loss: 0.25
Epoch 9400, Loss: 0.25
```

```
Epoch 9500, Loss: 0.25
Epoch 9600, Loss: 0.25
Epoch 9700, Loss: 0.25
Epoch 9800, Loss: 0.25
Epoch 9900, Loss: 0.25

Final Loss (Sigmoid): 0.0021276064735579305
Final Loss (Tanh): 3.632489758802801e-05
Final Loss (ReLU): 0.25
```