import numpy as np

class Perceptron:

    def \_\_init\_\_(self, weights, desired\_output, bias, eta):

        self.weights = np.array(weights)

        self.bias = bias

        self.desired\_output = desired\_output

        self.eta = eta

        self.activity = 0.0

        self.activation = 0.0

        self.delta = 0.0

    # Calculate activity

    def calc\_activity(self, input):

        input = np.array(input)

        self.activity = self.bias + np.dot(self.weights, input)

    # Calculate activation with sigmoid function

    def calc\_activation(self):

        self.activation = 1 / (1 + np.exp(-self.activity))

    # Calculate delta/error

    def set\_delta(self, error):

        activation\_derivative = self.activation \* (1 - self.activation)

        self.delta = error \* activation\_derivative

    # Update weights of each perceptron

    def update\_weights(self, inputs):

        self.weights -= self.eta \* self.delta \* np.array(inputs)

        self.bias -= self.eta \* self.delta

class Network:

    def \_\_init\_\_(self, input\_size=2, hidden\_size=2, output\_size=1, eta=1.0):

        self.eta = eta

        # Hidden layer perceptrons

        self.hidden\_layer = [Perceptron(weights=[0.3] \* input\_size, desired\_output=0, bias=0, eta=eta)

                             for \_ in range(hidden\_size)]

        # Output layer perceptrons

        self.output\_layer = [Perceptron(weights=[0.8] \* hidden\_size, desired\_output=0.7, bias=0, eta=eta)]

    # Feed-Forward

    def forward(self, inputs):

        # Hidden layer feed forward

        hidden\_activations = []

        for perceptron in self.hidden\_layer:

            perceptron.calc\_activity(inputs)

            perceptron.calc\_activation()

            hidden\_activations.append(perceptron.activation)

        # Output layer feed forward

        for perceptron in self.output\_layer:

            perceptron.calc\_activity(hidden\_activations)

            perceptron.calc\_activation()

        return hidden\_activations, self.output\_layer[0].activation

    def ffbp(self, inputs, desired\_output):

        # Feed Forward

        hidden\_activations, output\_activation = self.forward(inputs)

        # Get output layer error and delta

        output\_error = output\_activation - desired\_output

        self.output\_layer[0].set\_delta(output\_error)

        # Back propagate to hidden layer

        for i, hidden\_neuron in enumerate(self.hidden\_layer):

            hidden\_error = self.output\_layer[0].weights[i] \* self.output\_layer[0].delta

            hidden\_neuron.set\_delta(hidden\_error)

        # Update weights for output layer

        self.output\_layer[0].update\_weights(hidden\_activations)

        # Update weights for hidden layer

        for i, hidden\_neuron in enumerate(self.hidden\_layer):

            hidden\_neuron.update\_weights(inputs)

    # Train Network for certain number of epochs

    def train(self, inputs, desired\_output, epochs):

        for \_ in range(epochs):

            for i in range(len(inputs)):

                self.ffbp(inputs[i], desired\_output[i])

    # Plug in an input to the trained network

    def predict(self, inputs):

        \_, output\_activation = self.forward(inputs)

        return output\_activation

    # Get error

    def getBigE(self, inputs, desired\_output):

        output\_activation = self.predict(inputs)

        small\_e = desired\_output - output\_activation

        bigE = 0.50 \* small\_e \*\* 2

        return bigE

    # Get weights

    def getWeights(self):

        weights\_info = {}

        # Get weights and biases from hidden layer perceptrons

        for i, perceptron in enumerate(self.hidden\_layer):

            weights\_info[f"Hidden Layer Perceptron {i + 1}"] = {

                "weights": perceptron.weights,

                "bias": perceptron.bias

            }

        # Collect weights and biases from output layer perceptron

        for i, perceptron in enumerate(self.output\_layer):

            weights\_info["Output Layer Perceptron"] = {

                "weights": perceptron.weights,

                "bias": perceptron.bias

            }

        return weights\_info

# TESTING THE FFBP MULTI-Layer NEURAL NETWORK

# Single input set [1, 2] with a desired output of 0.7

inputs = np.array([[1, 2]])

desired\_output = np.array([0.7])

# Initialize the neural network

neural\_network = Network(input\_size=2, hidden\_size=2, output\_size=1, eta=1.0)

# Train the network

neural\_network.train(inputs, desired\_output, epochs=500)

# Test the network

print("Final output after training:")

for i in range(len(inputs)):

    print(f"Input: {inputs[i]}, Predicted Output: {neural\_network.predict(inputs[i])}")

Final output after training:

Input: [1 2], Predicted Output: 0.7

# Method 1. Two input pairs training, looped 15 times.

inputs = np.array([[1, 1], [-1, -1]])  # Two inputs, each is a 1D array

desired\_output = np.array([0.9, 0.05])  # Target outputs

# Initialize the neural network

neural\_network = Network(input\_size=2, hidden\_size=2, output\_size=1, eta=1.0)

# Training loop

for \_ in range(15):

    neural\_network.train([inputs[0]], [desired\_output[0]], epochs=1)

    neural\_network.train([inputs[1]], [desired\_output[1]], epochs=1)

# Retrieve and print updated weights

weights\_info = neural\_network.getWeights()

print(f"Updated weights after training: {weights\_info}")

# Print final output after training

print("Final output after training:")

print("Output for (1,1)", neural\_network.predict(inputs[0]))  # This prints without concatenation issues

print("Big E:", neural\_network.getBigE(inputs[0], desired\_output[0]))

print("Output for (-1,-1), ", neural\_network.predict(inputs[1]))

print("Big E:", neural\_network.getBigE(inputs[1], desired\_output[1]))

Updated weights after training: {'Hidden Layer Perceptron 1': {'weights': array([0.69205177, 0.69205177]), 'bias': np.float64(-0.12850040822123737)}, 'Hidden Layer Perceptron 2': {'weights': array([0.69205177, 0.69205177]), 'bias': np.float64(-0.12850040822123737)}, 'Output Layer Perceptron': {'weights': array([0.95185797, 0.95185797]), 'bias': np.float64(-0.8257813486188029)}}

Final output after training:

Output for (1,1) 0.6583208713508027

Big E: 0.029204400612317622

Output for (-1,-1), 0.3817659623378531

Big E: 0.055034326882980884

# Method 2. Looped training of input 1 15 times and subsequent looped training of input 2 15 times.

inputs = np.array([[1, 1], [-1, -1]])  # Two inputs, each is a 1D array

desired\_output = np.array([0.9, 0.05])  # Target outputs

# Initialize the neural network

neural\_network = Network(input\_size=2, hidden\_size=2, output\_size=1, eta=1.0)

# Training loop

neural\_network.train([inputs[0]], [desired\_output[0]], epochs=15)

neural\_network.train([inputs[1]], [desired\_output[1]], epochs=15)

# Retrieve and print updated weights

weights\_info = neural\_network.getWeights()

print(f"Updated weights after training: {weights\_info}")

# Print final output after training

print("Final output after training:")

print("Output for (1,1)", neural\_network.predict(inputs[0]))

print("Big E:", neural\_network.getBigE(inputs[0], desired\_output[0]))

print("Output for (-1,-1), ", neural\_network.predict(inputs[1]))

print("Big E:", neural\_network.getBigE(inputs[1], desired\_output[1]))

Updated weights after training: {'Hidden Layer Perceptron 1': {'weights': array([0.57602653, 0.57602653]), 'bias': np.float64(-0.17015279256338045)}, 'Hidden Layer Perceptron 2': {'weights': array([0.57602653, 0.57602653]), 'bias': np.float64(-0.17015279256338045)}, 'Output Layer Perceptron': {'weights': array([0.58941714, 0.58941714]), 'bias': np.float64(-1.1735567184480915)}}

Final output after training:

Output for (1,1) 0.4216576338099176

Big E: 0.11440570964616345

Output for (-1,-1), 0.2838448109486975

Big E: 0.027341697803816043