In [26]:

```
import numpy as np

class NeuralNetwork:
    def __init__(self, input_size, hidden_size, output_size):
        self.weights_input_hidden = np.random.rand(input_size, hidden_size)
        self.bias_hidden = np.zeros((1, hidden_size))
        self.weights_hidden_output = np.random.rand(hidden_size, output_size)
        self.bias_output = np.zeros((1, output_size))
        self.predicted_output = None  # Initialize predicted_output attribute

    def sigmoid(self, x):
        return 1 / (1 + np.exp(-x))

    def sigmoid_derivative(self, x):
        return x * (1 - x)

    def forward(self, inputs):
        self.hidden_layer_activation = np.dot(inputs, self.weights_input_hidden) + self.bias_hidden
        self.hidden_layer_output = self.sigmoid(self.hidden_layer_activation)
        self.output_layer_activation = np.dot(self.hidden_layer_output, self.weights_hidden_output) + self.bia
        self.predicted_output = self.sigmoid(self.output_layer_activation)
        return self.predicted_output

    def backward(self, inputs, targets, learning_rate):
        error = targets - self.predicted_output
        output_delta = error * self.sigmoid_derivative(self.predicted_output)
        hidden_layer_error = output_delta.dot(self.weights_hidden_output.T)
        hidden_layer_delta = hidden_layer_error * self.sigmoid_derivative(self.hidden_layer_output)
        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 += inputs.T.dot(hidden_layer_delta) * learning_rate
        self.bias_hidden += np.sum(hidden_layer_delta, axis=0, keepdims=True) * learning_rate

    def train(self, inputs, targets, epochs, learning_rate):
        for epoch in range(epochs):
            for i in range(len(inputs)):
                input_data = np.array([inputs[i]])
                target_data = np.array([targets[i]])
                self.forward(input_data)  # Call forward to calculate predicted_output
            if epoch % 100 == 0:
                loss = np.mean(np.square(targets - self.predicted_output))
                print(f"Epoch {epoch}, Loss: {loss}")
```

```
# Sample input and target data
input_size = 2
hidden_size = 4
output_size = 1
epochs = 1000
learning_rate = 0.1
X = np.array([[0.1, 0.4],
              [0.4, 0.7],
              [0.6, 0.9],
              [0.8, 0.5]])
y = np.array([[0.3],
              [0.6],
              [0.8],
              [0.4]])

# Create and train the neural network
nn = NeuralNetwork(input_size, hidden_size, output_size)
nn.train(X, y, epochs, learning_rate)

# Test with new data
new_data = np.array([[0.8, 0.2]])
predicted_output = nn.forward(new_data)
print("Predicted Output:", predicted_output)
```

```
Epoch 0, Loss: 0.07177177283206063
Epoch 100, Loss: 0.07177177283206063
Epoch 200, Loss: 0.07177177283206063
Epoch 300, Loss: 0.07177177283206063
Epoch 400, Loss: 0.07177177283206063
Epoch 500, Loss: 0.07177177283206063
Epoch 600, Loss: 0.07177177283206063
Epoch 700, Loss: 0.07177177283206063
Epoch 800, Loss: 0.07177177283206063
Epoch 900, Loss: 0.07177177283206063
Predicted Output: [[0.70681539]]
```

In [ ]: