

Vishwakarma Institute of Technology, Pune

Name: Tanishq Thusé

PRN- 12310237

Artificial Neural Networks Lab

Rollno. 52

Experiment Number: 04

Title : Write a program to implement the training process of an Artificial Neural Network, covering forward propagation and backpropagation, fundamental steps in neural network training.

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

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

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

# Neural network class
class NeuralNetwork:
    def __init__(self, layers, alpha=0.1):
        # Initialize weights and biases
        self.W = []
        self.b = []
        self.alpha = alpha # learning rate

        for i in range(len(layers) - 1):
            # Initialize weights with random values from a normal distribution
            self.W.append(np.random.randn(layers[i], layers[i+1]))
            # Initialize biases with zeros
            self.b.append(np.zeros((1, layers[i+1]))) ♦
    self.losses = [] # To store loss values
```

```

# Forward propagation
def forward(self, X):
    activations = [X]
    # Propagate through layers
    for i in range(len(self.W)):
        # Calculate net input
        net = np.dot(activations[-1], self.W[i]) + self.b[i]
        # Apply activation function
        output = sigmoid(net)
        activations.append(output)
    return activations

# Backpropagation
def backward(self, y, activations):
    errors = [y - activations[-1]] # Calculate error at output layer
    deltas = [errors[-1] * sigmoid_derivative(activations[-1])] # Calculate delta at output layer

    # Propagate errors backward through layers
    for i in range(len(self.W) - 1, 0, -1):
        # Calculate error at current layer
        error = np.dot(deltas[-1], self.W[i].T)
        errors.append(error)
        # Calculate delta at current layer
        delta = error * sigmoid_derivative(activations[i])
        deltas.append(delta)

    deltas.reverse() # Reverse deltas to match layer order
    return deltas

# Update weights and biases
def update(self, activations, deltas):
    # Update weights and biases for each layer
    for i in range(len(self.W)):
        self.W[i] += self.alpha * np.dot(activations[i].T, deltas[i])
        self.b[i] += self.alpha * np.sum(deltas[i], axis=0, keepdims=True)

# Training function with mini-batches
def fit(self, X, y, epochs=10000, batch_size=4): # Added batch_size parameter
    num_samples = X.shape[0]

    for epoch in range(epochs):
        # Shuffle data for each epoch

```

```

    .. Some code here to load data

permutation = np.random.permutation(num_samples)
X_shuffled = X[permutation]
y_shuffled = y[permutation]

epoch_loss = 0

# Iterate over mini-batches
for i in range(0, num_samples, batch_size):
    X_batch = X_shuffled[i:i + batch_size]
    y_batch = y_shuffled[i:i + batch_size]

    # Forward propagation
    activations = self.forward(X_batch)
    # Backpropagation
    deltas = self.backward(y_batch, activations)
    # Update weights and biases
    self.update(activations, deltas)

    # Calculate loss for the batch and accumulate
    loss = np.mean(np.square(y_batch - activations[-1]))
    epoch_loss += loss * len(X_batch) # Accumulate weighted loss

    # Calculate average epoch loss and store
    epoch_loss /= num_samples
    self.losses.append(epoch_loss)

    # Print loss every 1000 epochs
    if epoch % 1000 == 0:
        print(f"Epoch {epoch}, Loss: {self.losses[-1]}")

# Example usage (XOR problem)
if __name__ == "__main__":
    # Input data (XOR problem)
    X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])
    # Output data (XOR problem)
    y = np.array([[0], [1], [1], [0]])

    # Define network architecture (input layer: 2, hidden layer: 4, output layer: 1)
    layers = [2, 4, 1]
    # Create neural network instance

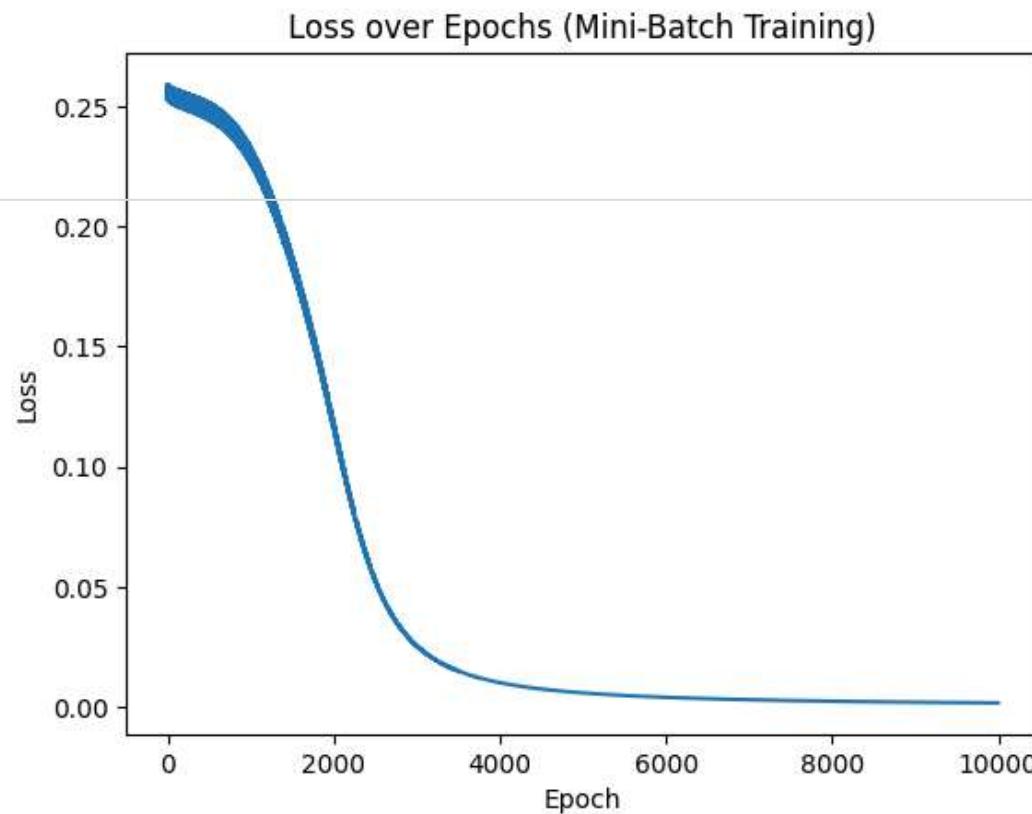
```

```
nn = NeuralNetwork(layers)
# Train the network using mini-batches
nn.fit(X, y, epochs=10000, batch_size=2) # Example with batch size 2

# Plot the loss
plt.plot(nn.losses)
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Loss over Epochs (Mini-Batch Training)")
plt.show()

# Test the trained network
print("\nTesting the trained network:")
for i in range(len(X)):
    prediction = nn.forward(X[i])[
        -1
    ] # Get prediction from the output layer
    print(f"Input: {X[i]}, Prediction: {prediction}, Actual: {y[i]}")
```

```
Epoch 0, Loss: 0.2527284932265027
Epoch 1000, Loss: 0.2258749967746393
Epoch 2000, Loss: 0.1159836512382848
Epoch 3000, Loss: 0.025261322114042077
Epoch 4000, Loss: 0.010082869693528163
Epoch 5000, Loss: 0.005841280370137009
Epoch 6000, Loss: 0.003990943081954449
Epoch 7000, Loss: 0.002991078600902452
Epoch 8000, Loss: 0.0023717429110026176
Epoch 9000, Loss: 0.001957596565113911
```



```
Testing the trained network:
Input: [0 0], Prediction: [[0.02668913]], Actual: [0]
Input: [0 1], Prediction: [[0.95712542]], Actual: [1]
Input: [1 0], Prediction: [[0.9927554611]], Actual: [1]
```