Output :-

Epoch 0, Loss: 0.2558

Epoch 2000, Loss: 0.2454

Epoch 4000, Loss: 0.1532

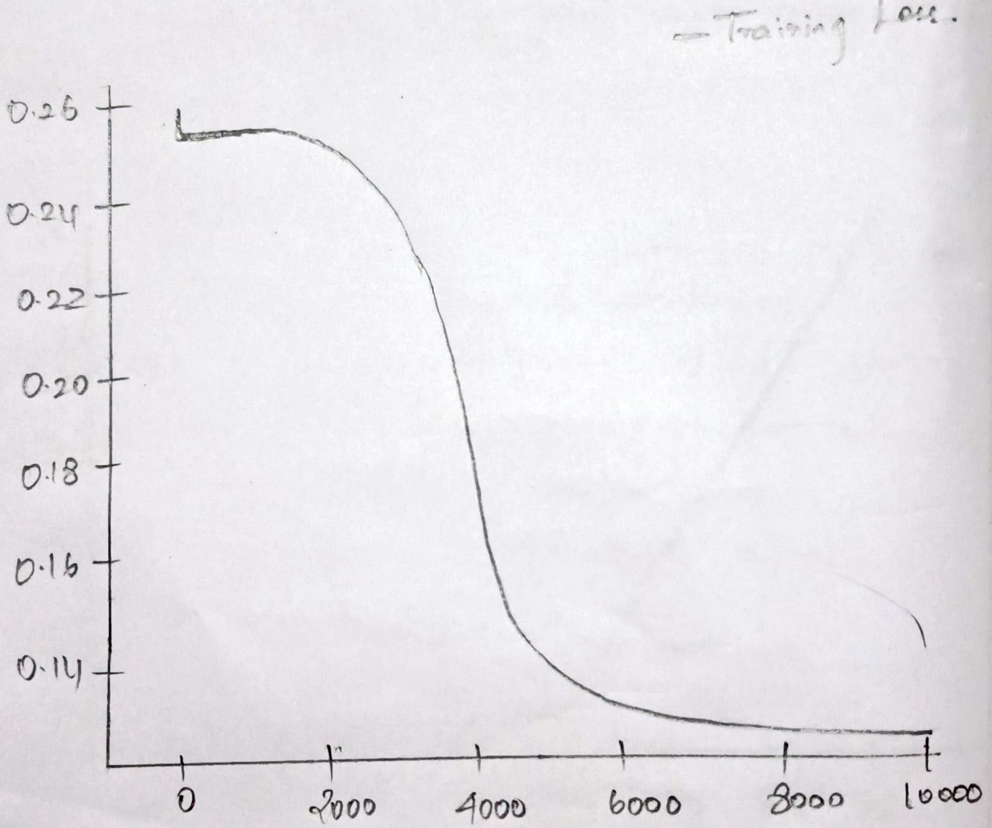
Epoch 6000, Loss: 0.1336

Epoch 8000, Loss: 0.1267

Final predictions :-

[[0.05300868]
 [0.49554213]
 [0.95091319]
 [0.50319888]]

Loss Curve - Gradient Descent and Back propagation



09.09.25

Ex:- 6 Implement gradient decent and backpropogation in deep neural network.

Aim:- To implement gradient descent and Backpropogation in deep neural network.

Objectives:-

* To understand gradient descent in an optimized method.

* To implement Backpropogation in deep neural network to update weights.

* To implements a simple neural network for classification task.

* To observe how loss decrease with iteration.

Observation:-

* Loss decreases as number of iterations increase

* weights and bias object to minimize error.

* Backpropogation ensures errors are efficiently distributed layer by layer.

* Learning rate (n) greatly influences con speed.

Pseudocode:-

Begin
  Initialize weight and bias randomly
  For epoch in range (max_epochs)
  for each input Sample
    * Forward pass
      Compute $z = w^x x + 6$
      apply activation to get A
      Compute output prediction
  * Compute loss
    loss = Costly (y_tree, y_pred)
  * Backward pase
    Compute gradient dw, db using Chain

Rule update parameters:

$w = w - a * dw$

$b = b - a * db$

END for

Print after epoch

END FOR

END.

Results:-

Therefore implementation of gradiant descent and backpropogation in neural network.

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

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

def sigmoid_derivative(x):
    return x * (1 - x)  # derivative assumes input = sigmoid(x)

# Training data (XOR problem)
X = np.array([[0,0],[0,1],[1,0],[1,1]])
y = np.array([[0],[1],[1],[0]])

# Set random seed for reproducibility
np.random.seed(42)

# Network architecture
input_size = 2
hidden_size = 2
output_size = 1

# Initialize weights and biases
W1 = np.random.randn(input_size, hidden_size)
b1 = np.zeros((1, hidden_size))
W2 = np.random.randn(hidden_size, output_size)
b2 = np.zeros((1, output_size))

# Hyperparameters
lr = 0.1
epochs = 10000
losses = []

# Training loop
for epoch in range(epochs):
    # ---- Forward pass ----
    z1 = np.dot(X, W1) + b1
    a1 = sigmoid(z1)
    z2 = np.dot(a1, W2) + b2
    a2 = sigmoid(z2)

    # ---- Loss (Mean Squared Error) ----
    loss = np.mean((y - a2) ** 2)
    losses.append(loss)

    # ---- Backpropagation ----
    d_a2 = (a2 - y)
    d_z2 = d_a2 * sigmoid_derivative(a2)
    dW2 = np.dot(a1.T, d_z2)
    db2 = np.sum(d_z2, axis=0, keepdims=True)

    d_a1 = np.dot(d_z2, W2.T)
    d_z1 = d_a1 * sigmoid_derivative(a1)
    dW1 = np.dot(X.T, d_z1)
    db1 = np.sum(d_z1, axis=0, keepdims=True)

    # ---- Update weights ----
    W2 -= lr * dW2
    b2 -= lr * db2
    W1 -= lr * dW1
    b1 -= lr * db1

    # Print loss occasionally
    if epoch % 2000 == 0:
        print(f"Epoch {epoch}, Loss: {loss:.4f}")

# ---- Final predictions ----
print("\nFinal Predictions:")
print(a2)

# ---- Plot the loss curve ----
plt.figure(figsize=(8,5))
plt.plot(losses, label="Training Loss", color="blue")
plt.xlabel("Epochs")
plt.ylabel("Loss")
plt.title("Loss Curve - Gradient Descent & Backpropagation")
plt.legend()
plt.grid(True)
plt.show()
```



```
Epoch 0, Loss: 0.2558
Epoch 2000, Loss: 0.2454
Epoch 4000, Loss: 0.1532
Epoch 6000, Loss: 0.1336
Epoch 8000, Loss: 0.1297

Final Predictions:
[[0.05300868]
 [0.49554213]
 [0.95091319]
 [0.50319888]]
```