EXP NO: 6: Implement Gradient Descent and Back propagation in DNN

Aim:-

Implement gradient descent and back-propagation for a deep neural network from scratch (numpy only), train it on a non-linear dataset (XOR), and demonstrate convergence with loss / accuracy

Objectives:-

1. Build a multi-layer perceptron with ReLU hidden layers and soft max ~~input~~ output.
2. Derive and code forward pass, loss, and backward pass. → analytical gradient.
   ↳ cross entropy
3. Train with vanilla gradient descent; show loss decreasing and accuracy improving.
4. Reflect on effects of learning rate, initialization, and depth.

Algorithm:-

1. Initialize parameters:
   → Randomly assign small values to weights W and set biases b to zero (or small values).
2. Forward pass:
   - Input data goes through each layer:
     → Multiply input by weight, add

bias → $z = a_{prev} W + b$

→ Apply activation ~~layer~~ function (like ReLU or sigmoid) to get a

- At last layer, use softmax to get probabilities of each class.

3. Compute loss:

- Compare predicted output $\hat{y}$ with true output y using a loss function

4. Backward pass:

- Calculate error at output layer: $\delta = \hat{y} - y$

→ Multiply error with weight matrix

→ Apply derivative of activation function

- Find gradients of weights and biases.

5. Update parameters:

- Adjust weights and biases using learning rate η:

→ $W = W - \eta \cdot dW$

→ $b = b - \eta \cdot db$

6. Repeat:

- Do step 2 → 5 for many epochs
- Each time, loss decreases and predictions get better.

7 Stop Training:

- when loss becomes very small or accuracy is high enough.

Pseudo Code:

```
initialize w[1], b[1] with the init for
l = 1 ... L
for epoch in 1 -- E:
    a[0] = x
    for l in 1 -- L-1:
        z[l] = a[L-1] @ w[l] + b[l]
        a[l] = ReLU(z[l])
    z[L] = a[L-1] @ w[L] + b[L]
    y hat = softmax(z[L])
    loss = cross-entropy(y hat, y)

    dz[L] = (y hat - y)/N
    dw[L] = a[L-1]^T @ dz[L]
    db[L] = sum_rows(dz[L])
    da = dz[L] @ w[L]^T
    for l in L-1 down to 1:
        dz[l] = da * ReLU'(z[l])
        dw[l] = a[l-1]^T @ dz[l]
        db[l] = sum_rows(dz[l])
    if l > 1: da = dz[l] @ w[l]^T

    for l in 1 -- L:
        w[l] -= lr * dw[l]
```

$b[l] -= lr * db[l]$

Observation:-

- Convergence: with He intialization + ReLU, vanilla gradient descent learns XOR reliably; loss smoothly decrease to $\approx 1e-4$ and accur-acy reaches 100%.

- Learning rate: if lr is too small, conver-gence is slow; too large leads to oscilla-tion/ divergence. For this setup, 0.05-0.2 works well.

- Depth matters: A single linear layer can't solve XOR; adding at least one non-linear hidden layer is essential. Deeper networks can fit more complex patterns but need good initialization and possibly regulariza-tion

- stability: softmax with log-sum-exp stabilization and one-hot labels keeps training numerically stable

Conclusion:-

Gradient Descent with Backpropagation effectively trains a deep neural network. A non-linear hidden layer is essential for solving problems like XOR, and with proper initialization and learning rate, the model converges to accurate results.

[1]

```
import numpy as np
import matplotlib.pyplot as plt
np.random.seed(42)
X_base = np.array([[0,0],[0,1],[1,0],[1,1]], dtype=np.float64)
y_base = np.array([0,1,1,0])
repeats = 500
X = np.tile(X_base, (repeats, 1))
y = np.tile(y_base, repeats)
num_classes = 2
Y = np.eye(num_classes)[y]
layers = [2, 8, 8, 2]
lr = 0.1
epochs = 5000
print_every = 500
def he_init(fan_in, fan_out):
    return np.random.randn(fan_in, fan_out) * np.sqrt(2.0 / fan_in)
def relu(z):
    return np.maximum(0, z)
def drelu(z):
    return (z > 0).astype(z.dtype)
def softmax(z):
    z = z - np.max(z, axis=1, keepdims=True)
    exp = np.exp(z)
    return exp / np.sum(exp, axis=1, keepdims=True)
def cross_entropy(yhat, Y):
    eps = 1e-9
    return -np.mean(np.sum(Y * np.log(yhat + eps), axis=1))
def accuracy(yhat, y):
    return np.mean(np.argmax(yhat, axis=1) == y)
params = {}
for i in range(len(layers) - 1):
    params[f"W{i+1}"] = he_init(layers[i], layers[i+1])
    params[f"b{i+1}"] = np.zeros((1, layers[i+1]))
def forward(X, params):
    cache = {}
    a = X
```

Variables   Terminal

CO DLT LAB-6.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

[1]

```
        cache["a0"] = X
        L = len(layers) - 1
        for i in range(1, L):
            z = a @ params[f"W{i}"] + params[f"b{i}"]
            a = relu(z)
            cache[f"z{i}"] = z
            cache[f"a{i}"] = a
        zL = a @ params[f"W{L}"] + params[f"b{L}"]
        yhat = softmax(zL)
        cache[f"z{L}"] = zL
        cache["yhat"] = yhat
        return yhat, cache
    def backward(Y, cache, params):
        grads = {}
        L = len(layers) - 1
        yhat = cache["yhat"]
        N = Y.shape[0]
        dz = (yhat - Y) / N
        a_prev = cache[f"a{L-1}"]
        grads[f"dW{L}"] = a_prev.T @ dz
        grads[f"db{L}"] = np.sum(dz, axis=0, keepdims=True)
        da_prev = dz @ params[f"W{L}"].T
        for i in range(L-1, 0, -1):
            z = cache[f"z{i}"]
            dz = da_prev * drelu(z)
            a_prev = cache["a0"] if i == 1 else cache[f"a{i-1}"]
            grads[f"dW{i}"] = a_prev.T @ dz
            grads[f"db{i}"] = np.sum(dz, axis=0, keepdims=True)
            if i > 1:
                da_prev = dz @ params[f"W{i}"].T
        return grads
    losses = []
    accuracies = []
    epochs_list = []
    for epoch in range(1, epochs + 1):
        yhat, cache = forward(X, params)
        Ls = cross_entropy(yhat, Y)
        grads = backward(Y, cache, params)
```

{} Variables Terminal

```
            da_prev = dz @ params[f"W{i}"].T
        return grads
    losses = []
    accuracies = []
    epochs_list = []
    for epoch in range(1, epochs + 1):
        yhat, cache = forward(X, params)
        Ls = cross_entropy(yhat, Y)
        grads = backward(Y, cache, params)
        for k in params:
            params[k] -= lr * grads["d" + k]
        if epoch % print_every == 0:
            acc = accuracy(yhat, y)
            losses.append(Ls)
            accuracies.append(acc)
            epochs_list.append(epoch)
            print(f"Epoch {epoch:4d} | loss={Ls:.6f} | acc={acc:.4f}")
    yhat, _ = forward(X, params)
    print("\nFinal accuracy:", accuracy(yhat, y))
    print("Sample predictions (first 8):", np.argmax(yhat[:8], axis=1))
    print("Sample truths      (first 8):", y[:8])
    plt.figure(figsize=(12,5))
    plt.subplot(1,2,1)
    plt.plot(epochs_list, losses, marker='o')
    plt.title('Training Loss over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Loss')
    plt.grid(True)
    plt.subplot(1,2,2)
    plt.plot(epochs_list, accuracies, marker='o', color='green')
    plt.title('Training Accuracy over Epochs')
    plt.xlabel('Epoch')
    plt.ylabel('Accuracy')
    plt.ylim([0, 1.05])
    plt.grid(True)
    plt.show()
```

[1]
✓ 11s

```
Epoch  500 | loss=0.004159 | acc=1.0000
Epoch 1000 | loss=0.001368 | acc=1.0000
Epoch 1500 | loss=0.000765 | acc=1.0000
Epoch 2000 | loss=0.000516 | acc=1.0000
Epoch 2500 | loss=0.000384 | acc=1.0000
Epoch 3000 | loss=0.000303 | acc=1.0000
Epoch 3500 | loss=0.000248 | acc=1.0000
Epoch 4000 | loss=0.000209 | acc=1.0000
Epoch 4500 | loss=0.000180 | acc=1.0000
Epoch 5000 | loss=0.000158 | acc=1.0000

Final accuracy: 1.0
Sample predictions (first 8): [0 1 1 0 0 1 1 0]
Sample truths      (first 8): [0 1 1 0 0 1 1 0]
```