EXPERIMENT - 6

27/8/25

Aim: To implement gradient descent and backpropogation for training a simple feedforward neural network on the XOR problem

Description:

* Gradient descent is an optimization algorithm used to minimize the loss function by updating weights in the opposite direction of the gradient.
* Backpropogation is the process of calculating the gradient of the loss function with respect to each weight using the chain rule, so we can apply gradient descent efficiently.

The steps are,

1. Forward pass - compute output from inputs
2. Compute loss - difference between predicted and actual values.
3. Backpropogation pass - compute gradients of loss w.r.t weights
4. update weights - use gradient descent rule,

$$w = w - \eta \cdot \frac{dL}{dw}$$

where η is the learning rate.

Procedure:

1. Initialize weights randomly
2. Forward pass
   - → compute the hidden layer
   - → Compute the final output
3. Compute loss: mean squared error between predicted output and actual output

OUTPUT:

Epoch 0, loss : 0.2558

Epoch 1000, loss: 0.2494

Epoch 2000, loss: 0.2454

Epoch 3000, loss : 0.2047

Epoch 4000, loss : 0.1532

Epoch 5000, loss : 0.1387

Epoch 6000, loss : 0.1336

Epoch 7000, loss : 0.1312

Epoch 8000, loss : 0.1297

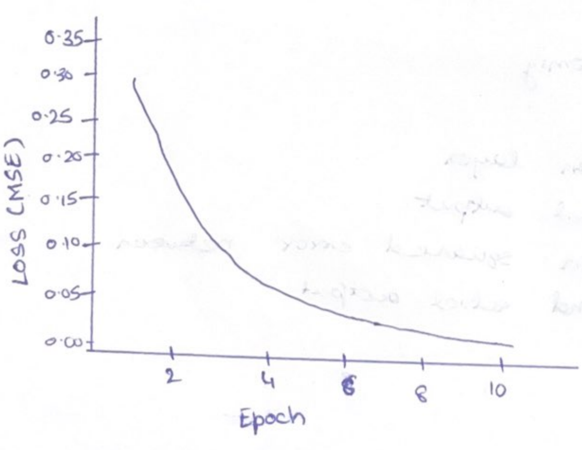
Epoch 9000, loss : 0.1288

Final predictions:

[[0.005300868]
 [0.49554213]
 [0.95091319]
 [0.50319888]]

Loss curve

LOSS Curve



4. Backpropogation
5. weight update
update weights and biases using gradient descent

$$w = w - \eta \cdot \frac{dL}{dw}$$

6. Repeat steps 2-5 for several epochs until the loss converges.
7. Print final predictions after training

Result :

The code has been successfully executed and shows the network learned the XOR logic output near 0 for [0, 0] and [1,1] and near 1 for [0,1] and [1,0]

EXPERIMENT-7 23-9-25

Aim: To build a convolutional neural network model that can classify images of cats and dogs using a labelled dataset.

Description:

A CNN is a deep learning algorithm designed to recogni patterns in visual data. ~~CNN are inspired~~ The core idea is to use a series of convolutions over input i to capture features such as edges, textures, and hi structures.

architecture -

- Convolution layer - apply filters to images to ex
- Activation (ReLU layer) - Adds non-linearity to th each convolution.
- Pooling layer - Reduces the spatial dimensio helping reduce computational and overfitting.
- F.C layer - After feature extraction, flat through FC layer to predi
- Softmax func. - Used in output laye final layers values i in classification.

Pseudocode:

Output:

Epoch	Accuracy	Loss	val_accuracy
1/10	[illegible]	0.9051	0.6909
2/10	[illegible]	0.6159	0.7636
3/10	[illegible]	0.3842	0.7729
4/10	0.7967	0.4149	0.7445
5/10	0.8501	0.3423	0.7455
6/10	0.8345	0.3607	0.8364
7/10	0.7897	0.4351	0.8091
8/10	0.8389	0.3291	0.7727
9/10	0.8389	0.3139	0.8182
10/10	0.8546	0.3006	0.7909

Workflow of CNN

```python
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from sklearn.datasets import load_iris
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import OneHotEncoder, StandardScaler
```

```python
iris = load_iris()
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = pd.DataFrame(iris.target, columns=["species"])

print("First 5 rows of dataset:")
print(X.head())

print("\nTarget distribution:")
print(y.value_counts())

# Visualize features
pd.plotting.scatter_matrix(X, figsize=(10, 8), diagonal='kde')
plt.suptitle("Iris Dataset Feature Distributions", fontsize=14)
plt.show()
```

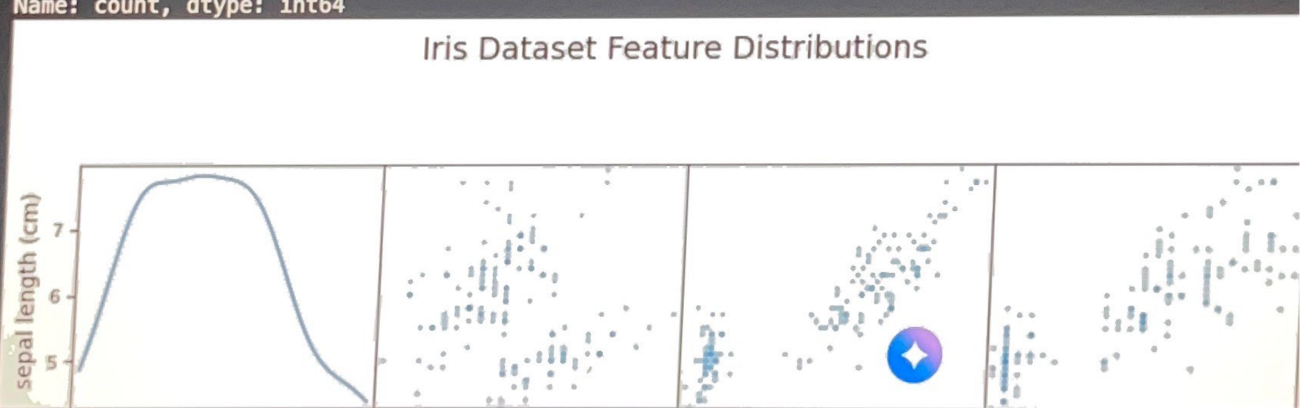
```
First 5 rows of dataset:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

Target distribution:
species
0          50
1          50
2          50
Name: count, dtype: int64
```

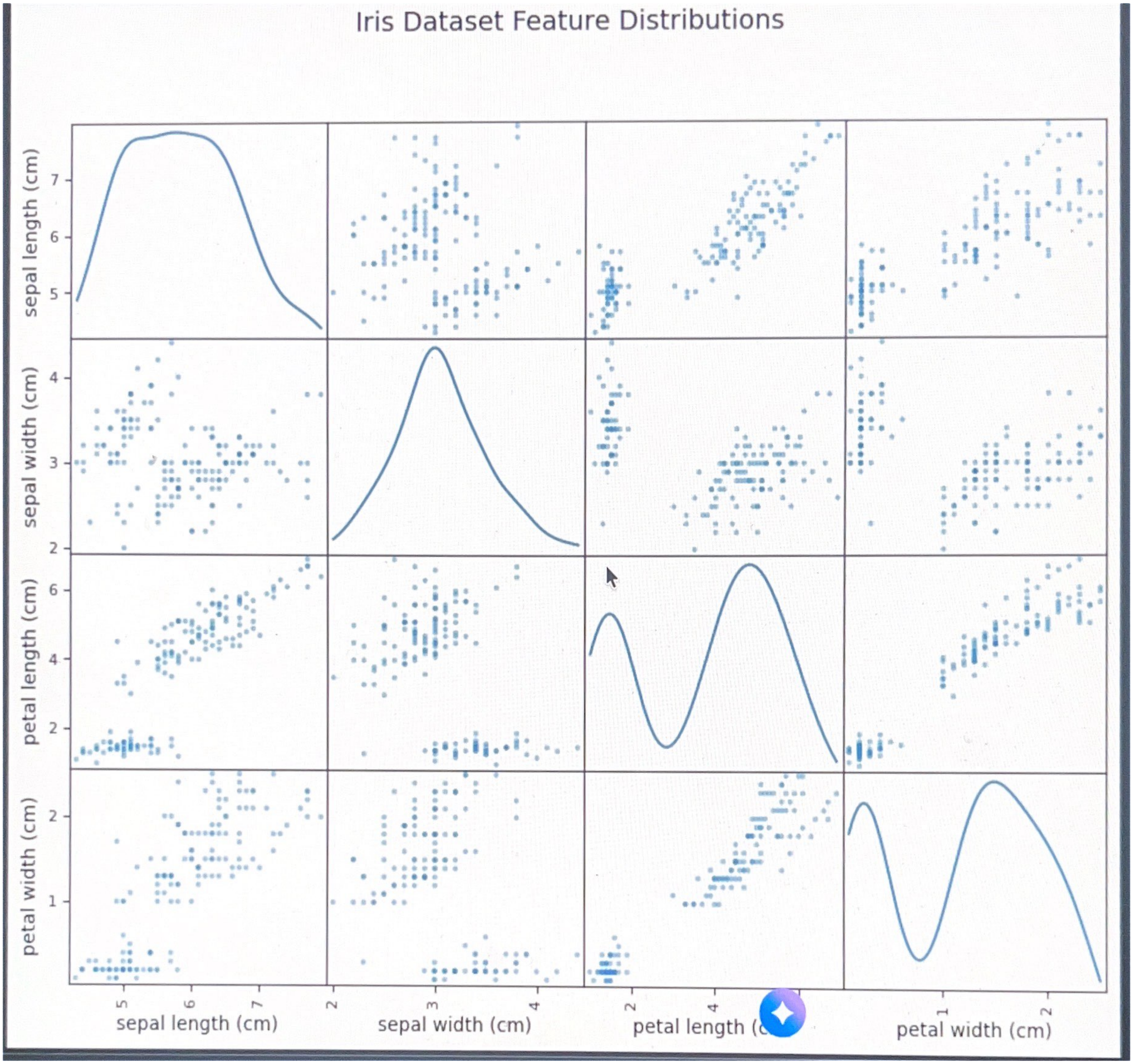
File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

```python
X = pd.DataFrame(iris.data, columns=iris.feature_names)
y = pd.DataFrame(iris.target, columns=["species"])

print("First 5 rows of dataset:")
print(X.head())

print("\nTarget distribution:")
print(y.value_counts())

# Visualize features
pd.plotting.scatter_matrix(X, figsize=(10, 8), diagonal='kde')
plt.suptitle("Iris Dataset Feature Distributions", fontsize=14)
plt.show()
```

```
First 5 rows of dataset:
   sepal length (cm)  sepal width (cm)  petal length (cm)  petal width (cm)
0                5.1               3.5                1.4               0.2
1                4.9               3.0                1.4               0.2
2                4.7               3.2                1.3               0.2
3                4.6               3.1                1.5               0.2
4                5.0               3.6                1.4               0.2

Target distribution:
species
0          50
1          50
2          50
Name: count, dtype: int64
```




Iris Dataset Feature Distributions
sepal length (cm)
sepal width (cm)
petal length (cm)
petal width (cm)
5
6
7
2
3
4
2
4
6
1
2

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▷ Run all

```
scaler = StandardScaler()
X_scaled = scaler.fit_transform(X)

# One-hot encode labels
encoder = OneHotEncoder(sparse_output=False)
y_encoded = encoder.fit_transform(y)

# Train-test split
X_train, X_test, y_train, y_test = train_test_split(
    X_scaled, y_encoded, test_size=0.2, random_state=42, stratify=y
)
```

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def sigmoid_derivative(a):
    return a * (1 - a)

def softmax(z):
    exp_z = np.exp(z - np.max(z, axis=1, keepdims=True))  # stability
    return exp_z / np.sum(exp_z, axis=1, keepdims=True)

def cross_entropy(y_true, y_pred):
    m = y_true.shape[0]
    eps = 1e-9
    return -np.sum(y_true * np.log(y_pred + eps)) / m

def accuracy(y_true, y_pred):
    return np.mean(np.argmax(y_true, axis=1) == np.argmax(y_pred, axis=1))
```

```
class SimpleDNN:
    def __init__(self, input_size, hidden_size, output_size, lr=0.01):
        np.random.seed(42)
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))
```

Variables Terminal

```python
lass SimpleDNN:
    def __init__(self, input_size, hidden_size, output_size, lr=0.01):
        np.random.seed(42)
        self.W1 = np.random.randn(input_size, hidden_size) * 0.01
        self.b1 = np.zeros((1, hidden_size))
        self.W2 = np.random.randn(hidden_size, output_size) * 0.01
        self.b2 = np.zeros((1, output_size))
        self.lr = lr

    def forward(self, X):
        self.Z1 = np.dot(X, self.W1) + self.b1
        self.A1 = sigmoid(self.Z1)
        self.Z2 = np.dot(self.A1, self.W2) + self.b2
        self.A2 = softmax(self.Z2)
        return self.A2

    def backward(self, X, y, output):
        m = X.shape[0]

        # Output layer gradients
        dZ2 = output - y
        dW2 = np.dot(self.A1.T, dZ2) / m
        db2 = np.sum(dZ2, axis=0, keepdims=True) / m

        # Hidden layer gradients
        dA1 = np.dot(dZ2, self.W2.T)
        dZ1 = dA1 * sigmoid_derivative(self.A1)
        dW1 = np.dot(X.T, dZ1) / m
        db1 = np.sum(dZ1, axis=0, keepdims=True) / m

        # Gradient descent update
        self.W1 -= self.lr * dW1
        self.b1 -= self.lr * db1
        self.W2 -= self.lr * dW2
        self.b2 -= self.lr * db2

    def train(self, X, y, epochs=1000):
        history = {"loss": [], "accuracy": []}
        for i in range(epochs):
```

Week-6.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text ▷ Run all

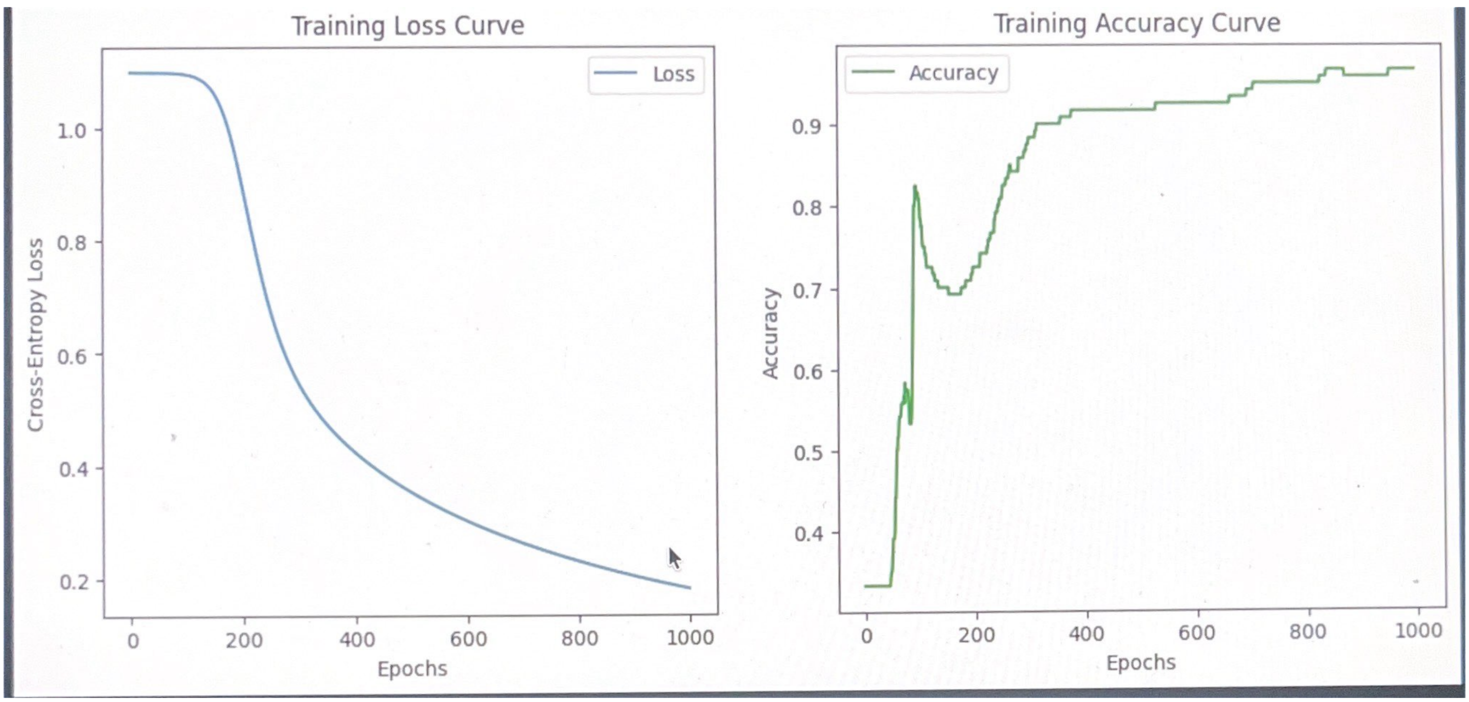
```python
        self.W1 -= self
        self.b1 -= self.lr * db1
        self.W2 -= self.lr * dW2
        self.b2 -= self.lr * db2

    def train(self, X, y, epochs=1000):
        history = {"loss": [], "accuracy": []}
        for i in range(epochs):
            output = self.forward(X)
            loss = cross_entropy(y, output)
            acc = accuracy(y, output)
            self.backward(X, y, output)

            history["loss"].append(loss)
            history["accuracy"].append(acc)

            if i % 100 == 0:
                print(f"Epoch {i}: Loss={loss:.4f}, Accuracy={acc:.4f}")

        return history

    def predict(self, X):
        probs = self.forward(X)
        return np.argmax(probs, axis=1)
```

```python
dnn = SimpleDNN(input_size=4, hidden_size=8, output_size=3, lr=0.1)
history = dnn.train(X_train, y_train, epochs=1000)
```

```
Epoch 0: Loss=1.0987, Accuracy=0.3333
Epoch 100: Loss=1.0948, Accuracy=0.8000
Epoch 200: Loss=0.8991, Accuracy=0.7250
Epoch 300: Loss=0.5464, Accuracy=0.8833
Epoch 400: Loss=0.4240, Accuracy=0.9167
Epoch 500: Loss=0.3531, Accuracy=0.9167
Epoch 600: Loss=0.3018, Accuracy=0.9250
Epoch 700: Loss=0.2621, Accuracy=0.9417
Epoch 800: Loss=0.2300, Accuracy=0.9500
Epoch 900: Loss=0.2032, Accuracy=0.9583
```

```python
y_pred = dnn.forward(X_test)
```

Variables Terminal


Training Loss Curve
Loss
Cross-Entropy Loss
1.0
0.8
0.6
0.4
0.2
0
200
400
600
800
1000
Epochs
Training Accuracy Curve
Accuracy
Accuracy
0.9
0.8
0.7
0.6
0.5
0.4
0
200
400
600
800
1000
Epochs