Assignment 1

Aim: Write a program Logistic Regression with Neural Network mindset

Problem Statement:

You are given a dataset (”data.h5”) containing:

- a training set of m_train images labeled as cat (y=1) or non-cat (y=0)
- a test set of m_test images labeled as cat or non-cat
- each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).Thus, each image is square (height = num_px) and(width = num_px).

Code:

```
import numpy as np

import matplotlib.pyplot as plt

from sklearn.metrics import accuracy_score

# Sigmoid activation function

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

# Initialize parameters

def initialize_parameters(dim):

    w = np.zeros((dim, 1))

    b = 0

    return w, b

# Forward and backward propagation

def propagate(w, b, X, Y):

    m = X.shape[1]

    # Forward propagation

    A = sigmoid(np.dot(w.T, X) + b)  # predictions

    cost = -(1/m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A))  # loss

    # Backward propagation

    dw = (1/m) * np.dot(X, (A - Y).T)
```

```python
    db = (1/m) * np.sum(A - Y)
    grads = {"dw": dw, "db": db}
    return grads, cost
# Optimize parameters
def optimize(w, b, X, Y, learning_rate, iterations):
    costs = []
    for i in range(iterations):
        grads, cost = propagate(w, b, X, Y)
        # Update rule
        w -= learning_rate * grads["dw"]
        b -= learning_rate * grads["db"]
        # Record cost
        if i % 100 == 0:
            costs.append(cost)
            print(f"Cost after iteration {i}: {cost:.6f}")
    return w, b, costs
# Predict
def predict(w, b, X):
    A = sigmoid(np.dot(w.T, X) + b)
    return (A > 0.5).astype(int)
# Model
def model(X_train, Y_train, X_test, Y_test, learning_rate=0.01, iterations=1500):
    w, b = initialize_parameters(X_train.shape[0])
    w, b, costs = optimize(w, b, X_train, Y_train, learning_rate, iterations)
    # Predictions
    Y_pred_train = predict(w, b, X_train)
    Y_pred_test = predict(w, b, X_test)
    # Accuracy
    train_acc = accuracy_score(Y_train[0], Y_pred_train[0]) * 100
```

```
    test_acc = accuracy_score(Y_test[0], Y_pred_test[0]) * 100
    print(f"\nTrain Accuracy: {train_acc:.2f}%")
    print(f"Test Accuracy: {test_acc:.2f}%")
    # Plot learning curve
    plt.plot(range(0, iterations, 100), costs)
    plt.xlabel("Iterations")
    plt.ylabel("Cost")
    plt.title("Learning Curve")
    plt.grid(True)
    plt.show()
    return {"w": w, "b": b, "costs": costs,
            "train_accuracy": train_acc,
            "test_accuracy": test_acc}
# Example: Generating synthetic dataset
np.random.seed(1)
num_features = 100
m_train, m_test = 1000, 200
# Training data
class1_train = np.random.normal(0.7, 0.3, (num_features, m_train//2))
class2_train = np.random.normal(0.3, 0.3, (num_features, m_train//2))
X_train = np.hstack((class1_train, class2_train))
Y_train = np.hstack((np.ones(m_train//2), np.zeros(m_train//2))).reshape(1, -1)
# Test data
class1_test = np.random.normal(0.7, 0.3, (num_features, m_test//2))
class2_test = np.random.normal(0.3, 0.3, (num_features, m_test//2))
X_test = np.hstack((class1_test, class2_test))
Y_test = np.hstack((np.ones(m_test//2), np.zeros(m_test//2))).reshape(1, -1)
# Train the model
result = model(X_train, Y_train, X_test, Y_test, learning_rate=0.05, iterations=1500)
```

Output:

Cost after iteration 0: 0.693147

Cost after iteration 100: 0.494629

Cost after iteration 200: 0.400875

Cost after iteration 300: 0.333155

Cost after iteration 400: 0.282969

Cost after iteration 500: 0.244780

Cost after iteration 600: 0.214999

Cost after iteration 700: 0.191261

Cost after iteration 800: 0.171975

Cost after iteration 900: 0.156044

Cost after iteration 1000: 0.142692

Cost after iteration 1100: 0.131360

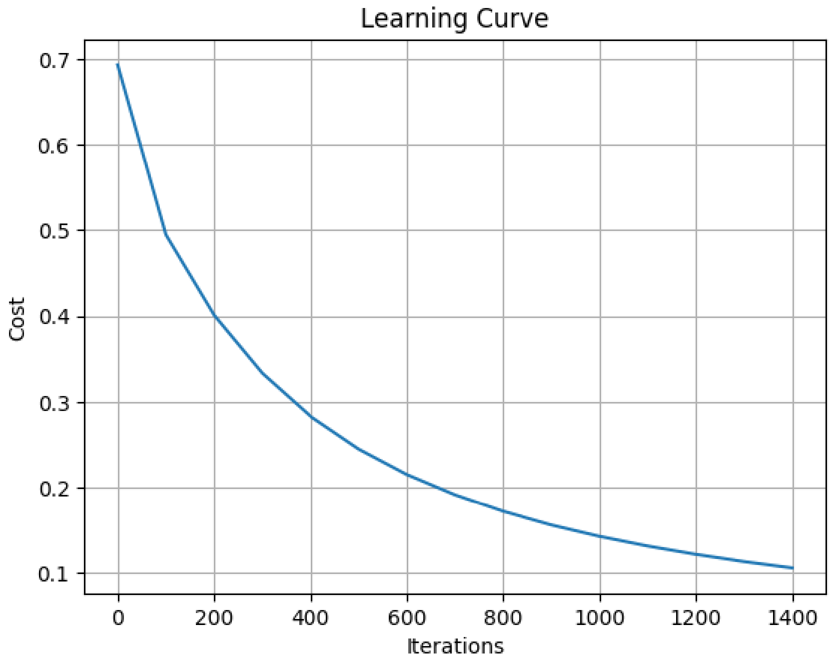
Cost after iteration 1200: 0.121633

Cost after iteration 1300: 0.113203

Cost after iteration 1400: 0.105832

Train Accuracy: 100.00%

Test Accuracy: 100.00%



Assignment 2

Aim: Implement Planner data classification with one hidden layer

Problem Statement:

- Develop an intuition of back-propagation and see it work on data
- Recognize that the more hidden layers you have the more complex structure you couldcapture.
- Build all the helper functions to implement a full model with one hidden layer.
- Implement a 2-class classification neural network with a single hidden layer
- Use units with a non-linear activation function, such as tanh
- Compute the cross entropy loss
- Implement forward and backward propagation

Code:

```
import numpy as np

import matplotlib.pyplot as plt

import sklearn

import sklearn.datasets

# Generate dataset

np.random.seed(1)

X, Y = sklearn.datasets.make_circles(n_samples=300, noise=0.05)

X = X.T

Y = Y.reshape(1, Y.shape[0])

# Plot dataset

plt.figure(figsize=(8, 6))

plt.scatter(X[0, :], X[1, :], c=Y.squeeze(), cmap=plt.cm.Spectral)

plt.xlabel('Feature 1')

plt.ylabel('Feature 2')

plt.title('Planar Dataset')

plt.show()

# Define neural net architecture

def layer_sizes(X, Y, n_h):
```

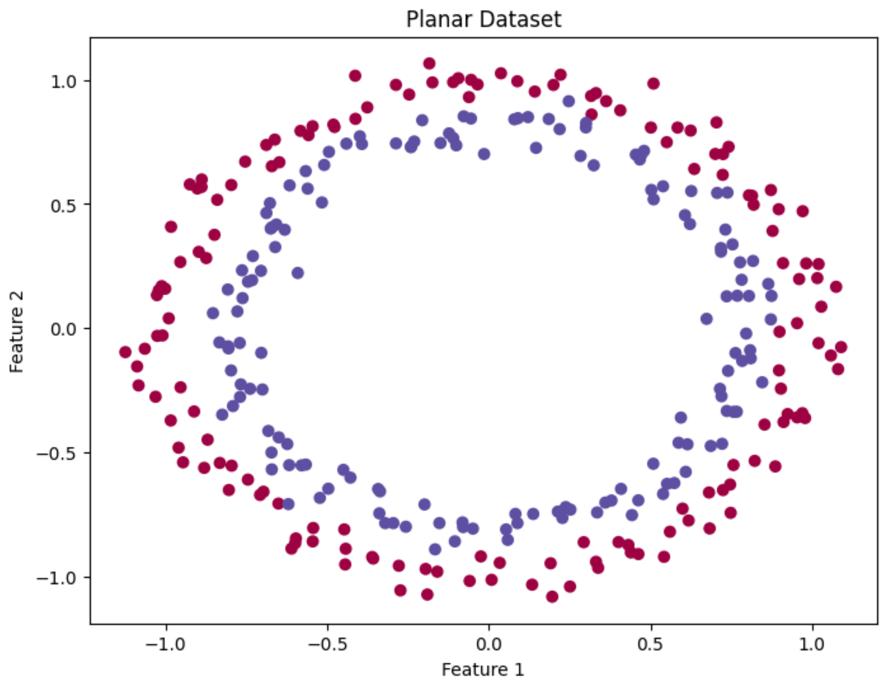
```
    n_x = X.shape[0]
    n_y = Y.shape[0]
    return (n_x, n_h, n_y)
def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(2)
    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros((n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros((n_y, 1))
    return {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
def forward_propagation(X, parameters):
    W1, b1 = parameters["W1"], parameters["b1"]
    W2, b2 = parameters["W2"], parameters["b2"]
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = 1 / (1 + np.exp(-Z2))
    return A2, {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2}
def compute_cost(A2, Y):
    m = Y.shape[1]
    logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), 1 - Y)
    cost = -np.sum(logprobs) / m
    return cost
def backward_propagation(parameters, cache, X, Y):
    m = X.shape[1]
    W2 = parameters["W2"]
    A1, A2 = cache["A1"], cache["A2"]
    dZ2 = A2 - Y
    dW2 = 1 / m * np.dot(dZ2, A1.T)
```

```python
    db2 = 1 / m * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.dot(W2.T, dZ2) * (1 - np.power(A1, 2))
    dW1 = 1 / m * np.dot(dZ1, X.T)
    db1 = 1 / m * np.sum(dZ1, axis=1, keepdims=True)
    return {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2}
def update_parameters(parameters, grads, learning_rate):
    parameters["W1"] -= learning_rate * grads["dW1"]
    parameters["b1"] -= learning_rate * grads["db1"]
    parameters["W2"] -= learning_rate * grads["dW2"]
    parameters["b2"] -= learning_rate * grads["db2"]
    return parameters
def neural_network_model(X, Y, n_h, num_iterations=10000, learning_rate=1.2,
print_cost=False):
    np.random.seed(3)
    n_x, n_y = layer_sizes(X, Y, n_h)[0], layer_sizes(X, Y, n_h)[2]
    parameters = initialize_parameters(n_x, n_h, n_y)
    costs = []
    for i in range(num_iterations):
        A2, cache = forward_propagation(X, parameters)
        cost = compute_cost(A2, Y)
        grads = backward_propagation(parameters, cache, X, Y)
        parameters = update_parameters(parameters, grads, learning_rate)
        if print_cost and i % 1000 == 0:
            print(f"Cost after iteration {i}: {cost:.6f}")
            costs.append(cost)
    return parameters, costs
# Predict function
def predict(parameters, X):
    A2, _ = forward_propagation(X, parameters)
```

```
    return (A2 > 0.5)

# Plot decision boundary

def plot_decision_boundary(model, X, Y):

    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1

    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1

    h = 0.01

    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),

                         np.arange(y_min, y_max, h))

    grid_points = np.c_[xx.ravel(), yy.ravel()].T

    Z = model(grid_points)

    Z = Z.reshape(xx.shape)

    plt.figure(figsize=(8, 6))

    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)

    plt.scatter(X[0, :], X[1, :], c=Y.squeeze(), cmap=plt.cm.Spectral)

    plt.xlabel('Feature 1')

    plt.ylabel('Feature 2')

    plt.title('Neural Network Decision Boundary')

# Train the model

print("\nTraining Neural Network...")

parameters, costs = neural_network_model(X, Y, n_h=4, num_iterations=10000,
print_cost=True)

# Cost plot

plt.figure(figsize=(8, 6))

plt.plot(np.squeeze(costs))

plt.ylabel('Cost')

plt.xlabel('Iterations (every 1000 steps)')

plt.title("Learning rate = 1.2")

plt.grid()

plt.show()
```

```
# Predict and accuracy

predictions = predict(parameters, X)

accuracy = float((np.dot(Y, predictions.T) + np.dot(1-Y, 1-predictions.T))/float(Y.size)*100)

print('\nModel Accuracy:', f"{accuracy:.2f}%")

# Plot decision boundary

plot_decision_boundary(lambda x: predict(parameters, x), X, Y)

plt.show()

# Summary

print("\nNeural Network Architecture:")

print(f"- Input Layer: {X.shape[0]} units")

print(f"- Hidden Layer: 4 units (tanh activation)")

print(f"- Output Layer: 1 unit (sigmoid activation)")

print("\nTraining Parameters:")

print(f"- Learning Rate: 1.2")

print(f"- Iterations: 10000")

print(f"- Final Accuracy: {accuracy:.2f}%")
```

Output:



Training Neural Network...

Cost after iteration 0: 0.693147

Cost after iteration 1000: 0.693110

Cost after iteration 2000: 0.693096

Cost after iteration 3000: 0.661667

Cost after iteration 4000: 0.477707

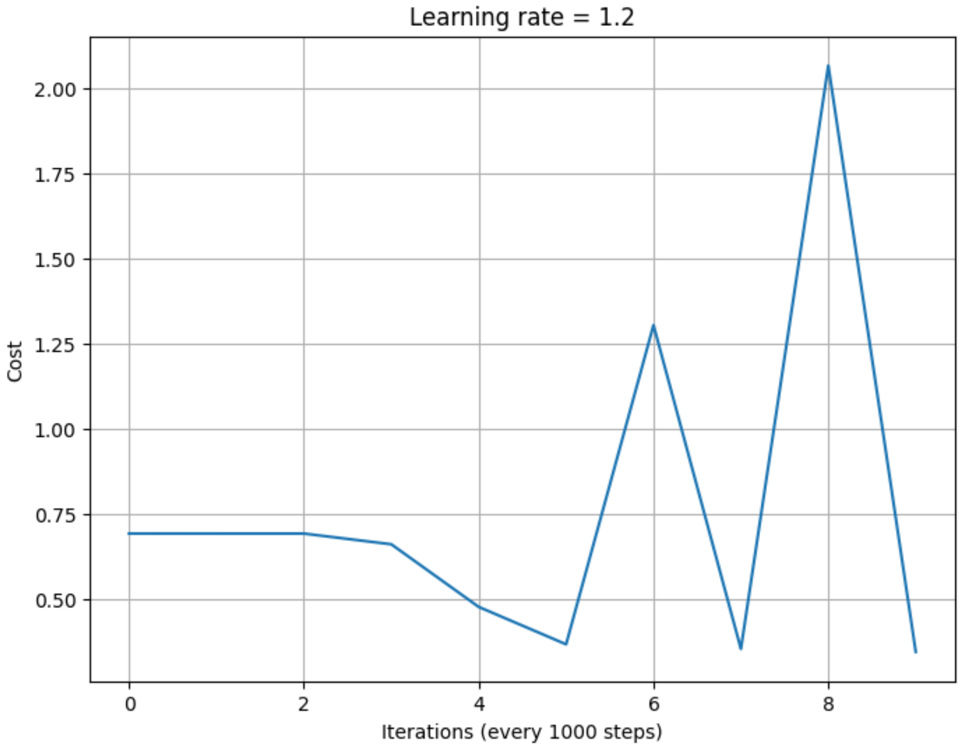
Cost after iteration 5000: 0.367807

Cost after iteration 6000: 1.305162

Cost after iteration 7000: 0.354300

Cost after iteration 8000: 2.067404

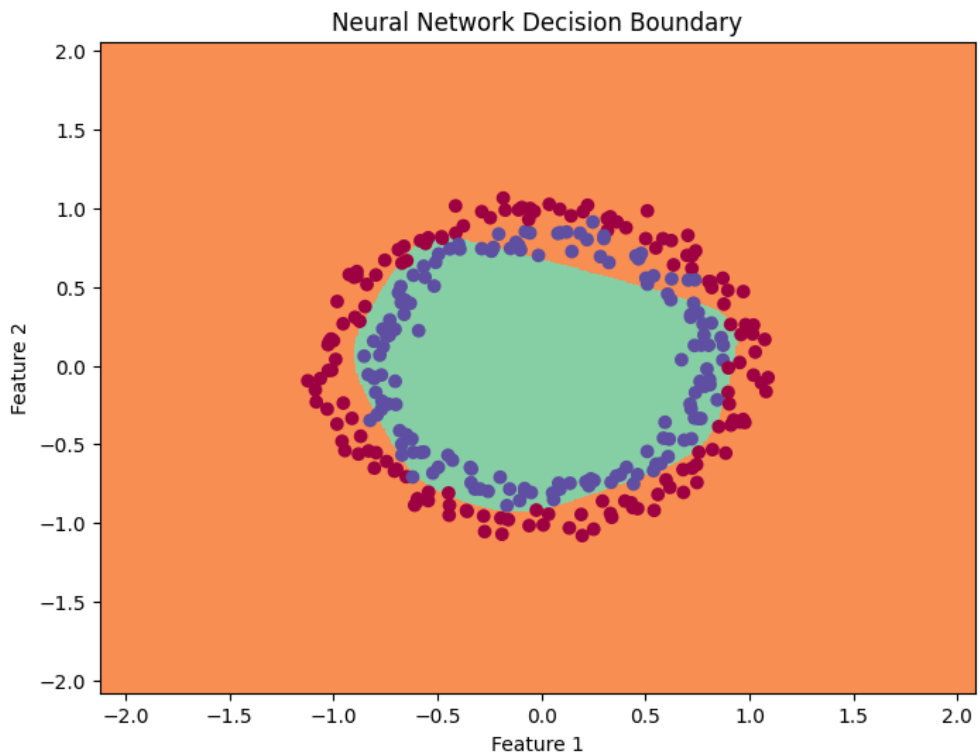
Cost after iteration 9000: 0.345819



<ipython-input-14-3e2994ebf70a>:123: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is deprecated, and will error in future. Ensure you extract a single element from your array before performing this operation. (Deprecated NumPy 1.25.)

```
  accuracy = float((np.dot(Y, predictions.T) + np.dot(1-Y, 1-predictions.T))/float(Y.size)*100)
```

Model Accuracy: 86.33%



Neural Network Architecture:

- Input Layer: 2 units

- Hidden Layer: 4 units (tanh activation)

- Output Layer: 1 unit (sigmoid activation)

Training Parameters:

- Learning Rate: 1.2

- Iterations: 10000

- Final Accuracy: 86.33%

Assignment 3

Aim: Implement Neural Network with one hidden layer

Problem Statement:

- Develop an intuition of forward-propagation and see it work on data
- Recognize that the one hidden layers you have the more complex structure you couldcapture.
- Build all the helper functions to implement a full model with one hidden layer.
- Use units with a non-linear activation function, such as tanh

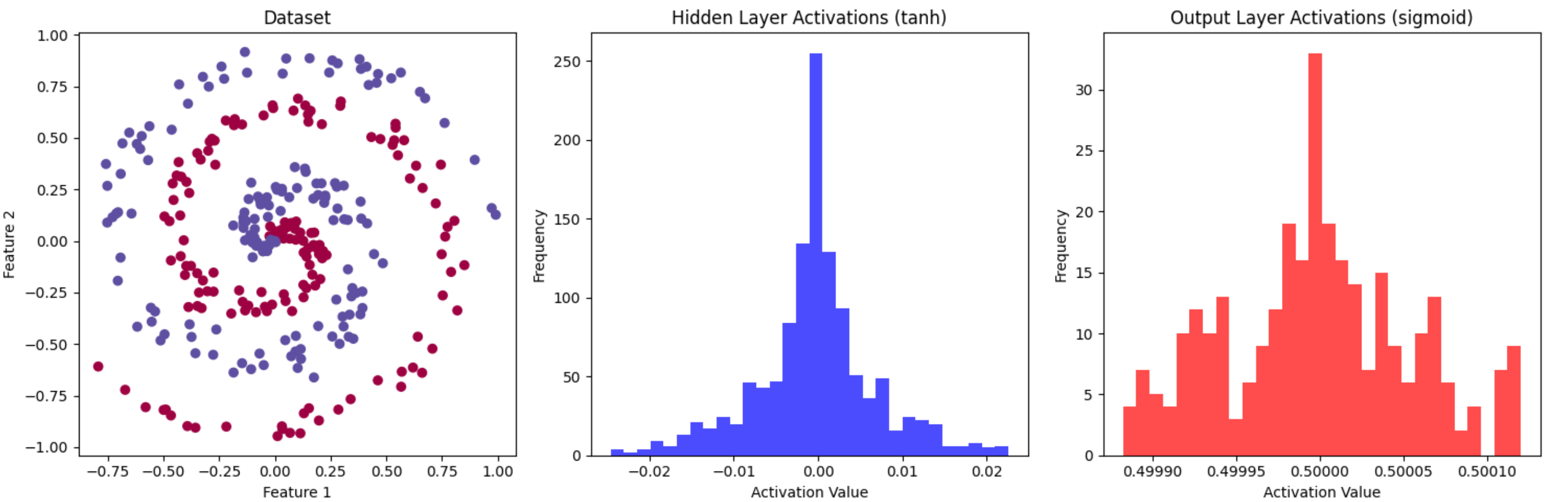
Code:

```
import numpy as np

import matplotlib.pyplot as plt

# Generate spiral dataset

def generate_data(points=300):

    np.random.seed(1)

    N = points // 2

    D = 2

    X = np.zeros((N*2, D))

    Y = np.zeros((N*2, 1))

    for j in range(2):

        ix = range(N*j, N*(j+1))

        r = np.linspace(0.0, 1, N)

        t = np.linspace(j*4, (j+1)*4, N) + np.random.randn(N)*0.2

        X[ix] = np.c_[r*np.sin(t*2.5), r*np.cos(t*2.5)]

        Y[ix] = j

    return X.T, Y.T

X, Y = generate_data()

class NeuralNetwork:

    def __init__(self, input_size, hidden_size, output_size):

        self.parameters = self.initialize_parameters(input_size, hidden_size, output_size)
```

```
        self.cache = {}
    def initialize_parameters(self, n_x, n_h, n_y):
        np.random.seed(2)
        W1 = np.random.randn(n_h, n_x) * 0.01
        b1 = np.zeros((n_h, 1))
        W2 = np.random.randn(n_y, n_h) * 0.01
        b2 = np.zeros((n_y, 1))
        return {"W1": W1, "b1": b1, "W2": W2, "b2": b2}
    def forward_propagation(self, X):
        W1, b1 = self.parameters["W1"], self.parameters["b1"]
        W2, b2 = self.parameters["W2"], self.parameters["b2"]
        Z1 = np.dot(W1, X) + b1
        A1 = np.tanh(Z1)
        Z2 = np.dot(W2, A1) + b2
        A2 = 1 / (1 + np.exp(-Z2))
        self.cache = {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2}
        return A2
nn = NeuralNetwork(input_size=2, hidden_size=4, output_size=1)
output = nn.forward_propagation(X)
# Plotting
plt.figure(figsize=(15, 5))
plt.subplot(1, 3, 1)
plt.scatter(X[0, :], X[1, :], c=Y.squeeze(), cmap=plt.cm.Spectral)
plt.title("Dataset")
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.subplot(1, 3, 2)
plt.hist(nn.cache["A1"].flatten(), bins=30, color="blue", alpha=0.7)
plt.title("Hidden Layer Activations (tanh)")
```

```
plt.xlabel("Activation Value")
plt.ylabel("Frequency")
plt.subplot(1, 3, 3)
plt.hist(output.flatten(), bins=30, color="red", alpha=0.7)
plt.title("Output Layer Activations (sigmoid)")
plt.xlabel("Activation Value")
plt.ylabel("Frequency")
plt.tight_layout()
plt.show()
```

Output:



Assignment 4

Aim: To build deep neural network step by step.

Problem Statement:

- Develop an intuition of the over all structure of a neural network.
- Write functions (e.g. forward propagation, backward propagation, logistic loss, etc...) that would help you decompose your code and ease the process of building a neural network.
- Initialize/update parameters according to your desired structure

Code:

```
import numpy as np

import matplotlib.pyplot as plt

class DeepNeuralNetwork:

  def __init__(self, layer_dims):

    """

    Initialize parameters with He initialization

    """

    self.parameters = {}

    self.L = len(layer_dims)

    for l in range(1, self.L):

      # He initialization for better ReLU performance

      self.parameters["W" + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1]) *
np.sqrt(2. / layer_dims[l-1])

      self.parameters["b" + str(l)] = np.zeros((layer_dims[l], 1))

  def relu(self, Z):

    return np.maximum(0., Z)

  def relu_derivative(self, Z):

    return (Z > 0.).astype(float)

  def sigmoid(self, Z):
```

```
        Z = np.clip(Z, -500, 500)  # Prevent overflow
        return 1. / (1. + np.exp(-Z))
    def sigmoid_derivative(self, Z):
        s = self.sigmoid(Z)
        return s * (1. - s)
    def forward_propagation(self, X):
        caches = []
        A = X
        for l in range(1, self.L - 1):
            Z = np.dot(self.parameters["W" + str(l)], A) + self.parameters["b" + str(l)]
            A = self.relu(Z)
            caches.append((Z, A))
        ZL = np.dot(self.parameters["W" + str(self.L - 1)], A) + self.parameters["b" + str(self.L - 1)]
        AL = self.sigmoid(ZL)
        caches.append((ZL, AL))
        return AL, caches
    def compute_cost(self, AL, Y):
        m = Y.shape[1]
        epsilon = 1e-15
        AL = np.clip(AL, epsilon, 1 - epsilon)  # Prevent log(0)
        cost = -1./m * np.sum(Y * np.log(AL) + (1 - Y) * np.log(1 - AL))
        return cost
    def backward_propagation(self, X, Y, caches):
        grads = {}
        m = X.shape[1]
        L = self.L
        AL = caches[-1][1]
        dAL = -(np.divide(Y, AL + 1e-15) - np.divide(1 - Y, 1 - AL + 1e-15))
```

```
        ZL = caches[-1][0]
        dZL = dAL * self.sigmoid_derivative(ZL)
        AL_prev = caches[-2][1] if L > 2 else X
        grads["dW" + str(L-1)] = 1./m * np.dot(dZL, AL_prev.T)
        grads["db" + str(L-1)] = 1./m * np.sum(dZL, axis=1, keepdims=True)
        dA_prev = np.dot(self.parameters["W" + str(L-1)].T, dZL)
        for l in reversed(range(L-2)):
            Z = caches[l][0]
            A_prev = caches[l-1][1] if l > 0 else X
            dZ = dA_prev * self.relu_derivative(Z)
            grads["dW" + str(l+1)] = 1./m * np.dot(dZ, A_prev.T)
            grads["db" + str(l+1)] = 1./m * np.sum(dZ, axis=1, keepdims=True)
            dA_prev = np.dot(self.parameters["W" + str(l+1)].T, dZ)
        return grads
    def train(self, X, Y, learning_rate=0.01, num_iterations=3000, print_cost=False):
        costs = []
        for i in range(num_iterations):
            AL, caches = self.forward_propagation(X)
            cost = self.compute_cost(AL, Y)
            grads = self.backward_propagation(X, Y, caches)
            # Gradient descent with momentum
            if not hasattr(self, "velocity"):
                self.velocity = {}
                for l in range(1, self.L):
                    self.velocity["dW" + str(l)] = np.zeros_like(self.parameters["W" + str(l)])
                    self.velocity["db" + str(l)] = np.zeros_like(self.parameters["b" + str(l)])
            beta = 0.9  # momentum parameter
            for l in range(1, self.L):
```

```
                self.velocity["dW" + str(l)] = beta * self.velocity["dW" + str(l)] + (1 - beta) * grads["dW" + str(l)]
                self.velocity["db" + str(l)] = beta * self.velocity["db" + str(l)] + (1 - beta) * grads["db" + str(l)]
                self.parameters["W" + str(l)] -= learning_rate * self.velocity["dW" + str(l)]
                self.parameters["b" + str(l)] -= learning_rate * self.velocity["db" + str(l)]
            if print_cost and i % 100 == 0:
                print(f"Cost after iteration {i}: {cost}")
            if i % 100 == 0:
                costs.append(cost)
        return costs
    def predict(self, X):
        AL, _ = self.forward_propagation(X)
        predictions = (AL > 0.5).astype(int)
        return predictions


# Test the neural network with XOR problem
np.random.seed(1)
X = np.random.randn(2, 400)
Y = np.logical_xor(X[0] > 0, X[1] > 0).astype(int).reshape(1, 400)
# Create neural network with improved architecture
layer_dims = [2, 10, 5, 1]  # 2 input features, 2 hidden layers, 1 output
dnn = DeepNeuralNetwork(layer_dims)
print("Training the neural network...")
costs = dnn.train(X, Y, learning_rate=0.003, num_iterations=3000, print_cost=True)
# Plot learning curve
plt.figure(figsize=(10, 6))
plt.plot(np.arange(0, 3000, 100), costs)
plt.ylabel("Cost")
```

```
plt.xlabel("Iterations (hundreds)")
plt.title("Learning curve")
plt.grid(True)
plt.show()
# Calculate and display accuracy
predictions = dnn.predict(X)
accuracy = np.mean(predictions == Y)
print(f"\nAccuracy: {accuracy * 100:.2f}%")
# Visualize the decision boundary
h = 0.01
x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
Z = dnn.predict(np.c_[xx.ravel(), yy.ravel()].T)
Z = Z.reshape(xx.shape)
plt.figure(figsize=(10, 8))
plt.contourf(xx, yy, Z, cmap=plt.cm.RdYlBu, alpha=0.3)
plt.scatter(X[0, :], X[1, :], c=Y.reshape(-1), cmap=plt.cm.RdYlBu)
plt.xlabel("Feature 1")
plt.ylabel("Feature 2")
plt.title("Decision Boundary of Neural Network (XOR Problem)")
plt.grid(True)
plt.show()
```

Output:

Training the neural network...

Cost after iteration 0: 1.1746293671335157

Cost after iteration 100: 0.8636249632138138

Cost after iteration 200: 0.7481185469341665

Cost after iteration 300: 0.6850855243102608

Cost after iteration 400: 0.6383433854495254

Cost after iteration 500: 0.5997478147381539

Cost after iteration 600: 0.5665574394386088

Cost after iteration 700: 0.5367641133983891

Cost after iteration 800: 0.5095246188264464

Cost after iteration 900: 0.4844771302999445

Cost after iteration 1000: 0.4613732883918645

Cost after iteration 1100: 0.4400716591157378

Cost after iteration 1200: 0.4207175140394736

Cost after iteration 1300: 0.4028319051746484

Cost after iteration 1400: 0.38626796468613556

Cost after iteration 1500: 0.37083365014119746

Cost after iteration 1600: 0.35649909691151566

Cost after iteration 1700: 0.3431103480704594

Cost after iteration 1800: 0.3305889717049587

Cost after iteration 1900: 0.3188737239973459

Cost after iteration 2000: 0.307824374259448

Cost after iteration 2100: 0.29739287363133154

Cost after iteration 2200: 0.28758035552341077

Cost after iteration 2300: 0.27833031399997016

Cost after iteration 2400: 0.26959255843589147

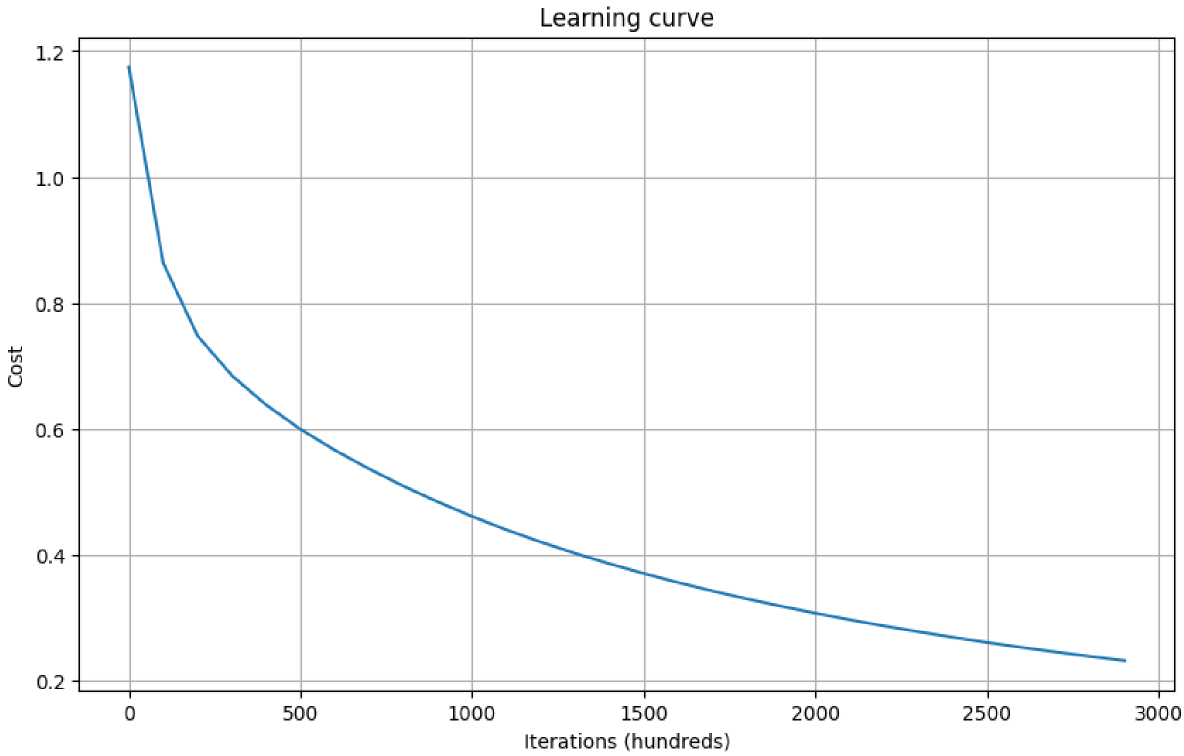
Cost after iteration 2500: 0.26136595481598635

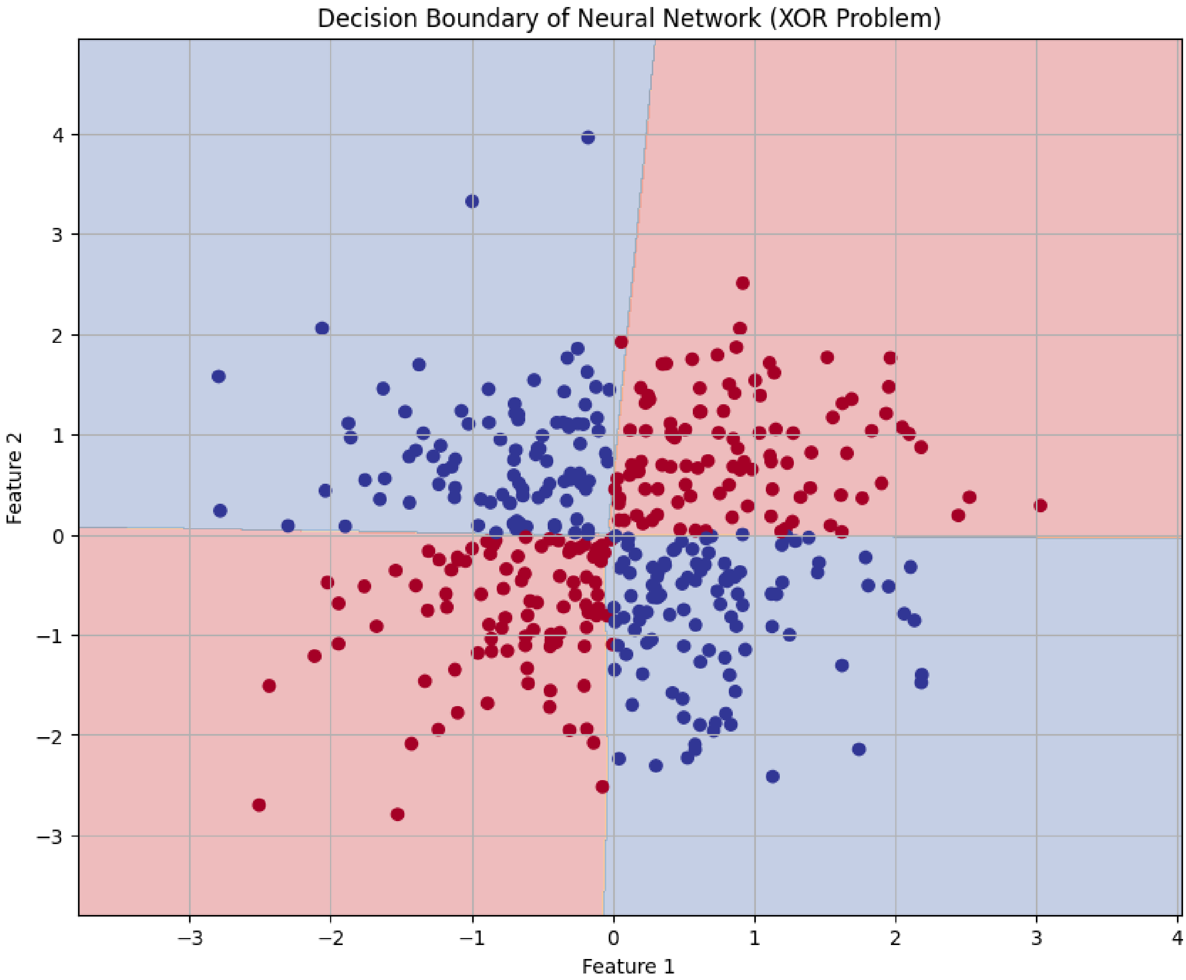
Cost after iteration 2600: 0.25359196915033005

Cost after iteration 2700: 0.2462377011710446

Cost after iteration 2800: 0.2392989154223038

Cost after iteration 2900: 0.2327424279615854


Learning curve
Cost
Iterations (hundreds)
1.2
1.0
0.8
0.6
0.4
0.2
0
500
1000
1500
2000
2500
3000


Decision Boundary of Neural Network (XOR Problem)
Feature 2
Feature 1
4
3
2
1
0
−1
−2
−3
−3
−2
−1
0
1
2
3
4

Assignment 5

Aim: Implement the concept of regularization, gradient checking and optimization in convolutional model: step by step

Problem Statement:

- Develop an intuition of the overall structure of a neural network.
- Write functions (e.g. forward propagation, backward propagation, logistic loss, etc...) that would help you decompose your code and ease the process of building a neural network.
- Initialize/update parameters according to your desired structure

Code:

```
import numpy as np

import matplotlib.pyplot as plt

class ConvolutionalNeuralNetwork:

  def __init__(self, input_shape, filter_size=3, num_filters=4):

    self.filters = np.random.randn(num_filters, filter_size, filter_size) * np.sqrt(2. /
(filter_size * filter_size))

    self.biases = np.zeros(num_filters)

    self.lambda_reg = 0.01  # L2 regularization parameter

    self.learning_rate = 0.01

  def relu(self, Z):

    return np.maximum(0, Z)

  def relu_derivative(self, Z):

    return (Z > 0).astype(float)

  def forward_prop(self, X):

    self.X = X

    m, h, w = X.shape

    f_h, f_w = self.filters.shape[1], self.filters.shape[2]

    h_out = h - f_h + 1
```

```
        w_out = w - f_w + 1
        self.Z = np.zeros((m, self.filters.shape[0], h_out, w_out))
        for i in range(m):
            for f in range(self.filters.shape[0]):
                for hi in range(h_out):
                    for wi in range(w_out):
                        self.Z[i, f, hi, wi] = np.sum(
                            X[i, hi:hi + f_h, wi:wi + f_w] * self.filters[f]
                        ) + self.biases[f]
        return self.relu(self.Z)
    def backward_prop(self, dA):
        m = self.X.shape[0]
        dW = np.zeros_like(self.filters)
        db = np.zeros_like(self.biases)
        dZ = dA * self.relu_derivative(self.Z)
        for i in range(m):
            for f in range(self.filters.shape[0]):
                for h in range(dZ.shape[2]):
                    for w in range(dZ.shape[3]):
                        dW[f] += self.X[i, h:h + self.filters.shape[1], w:w + self.filters.shape[2]] * dZ[i, f,
h, w]
                        db[f] += dZ[i, f, h, w]
        # Add L2 regularization
        dW += self.lambda_reg * self.filters
        return {"dW": dW / m, "db": db / m}
    def optimize(self, X, num_iterations=30):
        costs = []
        v_W = np.zeros_like(self.filters)  # Momentum
```

```
        v_b = np.zeros_like(self.biases)
        beta = 0.9
        for i in range(num_iterations):
            output = self.forward_prop(X)
            dA = np.random.randn(*output.shape) * 0.1  # Dummy gradient for testing
            grads = self.backward_prop(dA)
            # Momentum updates
            v_W = beta * v_W + (1 - beta) * grads["dW"]
            v_b = beta * v_b + (1 - beta) * grads["db"]
            self.filters -= self.learning_rate * v_W
            self.biases -= self.learning_rate * v_b
            cost = np.mean(output ** 2) + (self.lambda_reg / 2) * np.sum(self.filters ** 2)
            costs.append(cost)
            if i % 10 == 0:
                print(f"Cost after iteration {i}: {cost:.6f}")
        return costs
# Test the implementation
np.random.seed(42)
X = np.random.randn(5, 14, 14)  # 5 samples of 14x14 images
cnn = ConvolutionalNeuralNetwork(input_shape=X.shape)
print("Training CNN...")
costs = cnn.optimize(X)
# Plot learning curve
plt.figure(figsize=(10, 6))
plt.plot(costs)
plt.title("Learning Curve")
plt.xlabel("Iteration")
plt.ylabel("Cost")
```

```
plt.grid(True)
plt.show()
```

Output:

Training CNN...

Cost after iteration 0: 0.917353

Cost after iteration 10: 0.919443

Cost after iteration 20: 0.922964