**PYTHON CODES**

**# Code for spiral dataset**

import numpy as np

import matplotlib.pyplot as plt

N = 100 # number of points per class

D = 2 # dimensionality

K = 3 # number of classes

X = np.zeros((N\*K,D)) # data matrix (each row = single example)

y = np.zeros(N\*K, dtype='uint8') # class labels

for j in range(K):

  ix = range(N\*j,N\*(j+1))

  r = np.linspace(0.0,1,N) # radius

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

  X[ix] = np.c\_[r\*np.sin(t), r\*np.cos(t)]

  y[ix] = j

# lets visualize the data:

plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)

plt.show()

A diagram of a spiraling spiral

Description automatically generated with medium confidence

**Task 1: Score Function**

def score\_function(X, W, b):

    """

    Computes the score for each class using a linear classifier.

    Parameters:

    X : ndarray

        Input data of shape (N, D) where N is the number of data points, and D is the dimensionality.

    W : ndarray

        Weights matrix of shape (D, K) where K is the number of classes.

    b : ndarray

        Biases vector of shape (K,).

    Returns:

    scores : ndarray

        Score matrix of shape (N, K) where each element represents the score of a class for a data point.

    """

    return np.dot(X, W) + b

**Task 2: Loss Functions**

**# Code for hinge loss:**

def svm\_loss(X, y, W, b, reg):

    """

    Computes the hinge loss (SVM).

    Parameters:

    X : ndarray

        Input data of shape (N, D).

    y : ndarray

        Labels of shape (N,).

    W : ndarray

        Weights matrix of shape (D, K).

    b : ndarray

        Biases vector of shape (K,).

    reg : float

        Regularization strength.

    Returns:

    loss : float

        Hinge loss value.

    """

    scores = score\_function(X, W, b)

    correct\_class\_scores = scores[np.arange(X.shape[0]), y]

    margins = np.maximum(0, scores - np.reshape(correct\_class\_scores, (-1, 1)) + 1)

    margins[np.arange(X.shape[0]), y] = 0

    loss = np.sum(margins) / X.shape[0]

    loss += reg \* np.sum(W \* W)  # L2 regularization

    return loss

**# Code for cross-entropy loss:**

def softmax\_loss(X, y, W, b, reg):

    """

    Computes the softmax loss.

    Parameters:

    X : ndarray

        Input data of shape (N, D).

    y : ndarray

        Labels of shape (N,).

    W : ndarray

        Weights matrix of shape (D, K).

    b : ndarray

        Biases vector of shape (K,).

    reg : float

        Regularization strength.

    Returns:

    loss : float

        Cross-entropy loss value.

    """

    scores = score\_function(X, W, b)

    exp\_scores = np.exp(scores - np.max(scores, axis=1, keepdims=True))

    probs = exp\_scores / np.sum(exp\_scores, axis=1, keepdims=True)

    correct\_log\_probs = -np.log(probs[np.arange(X.shape[0]), y])

    loss = np.sum(correct\_log\_probs) / X.shape[0]

    loss += reg \* np.sum(W \* W)  # L2 regularization

    return loss

**Task 3: Regularization**

# Both loss functions already include L2 regularization

**Task 4: Gradient Descent**

def gradient\_descent(X, y, W, b, learning\_rate, reg, num\_iters):

    """

    Performs gradient descent to optimize W and b.

    Parameters:

    X : ndarray

        Input data of shape (N, D).

    y : ndarray

        Labels of shape (N,).

    W : ndarray

        Weights matrix of shape (D, K).

    b : ndarray

        Biases vector of shape (K,).

    learning\_rate : float

        Learning rate for optimization.

    reg : float

        Regularization strength.

    num\_iters : int

        Number of iterations for gradient descent.

    Returns:

    W, b : Updated weights and biases.

    """

    for i in range(num\_iters):

        # Compute loss and gradients

        loss = softmax\_loss(X, y, W, b, reg)

        # Compute gradients (omitted here for brevity)

        # Update parameters

        W -= learning\_rate \* grad\_W

        b -= learning\_rate \* grad\_b

        if i % 100 == 0:

            print(f"Iteration {i}/{num\_iters}: Loss = {loss}")

    return W, b