Date of the Session:__/__/___ **Time of the Session:___to_____**

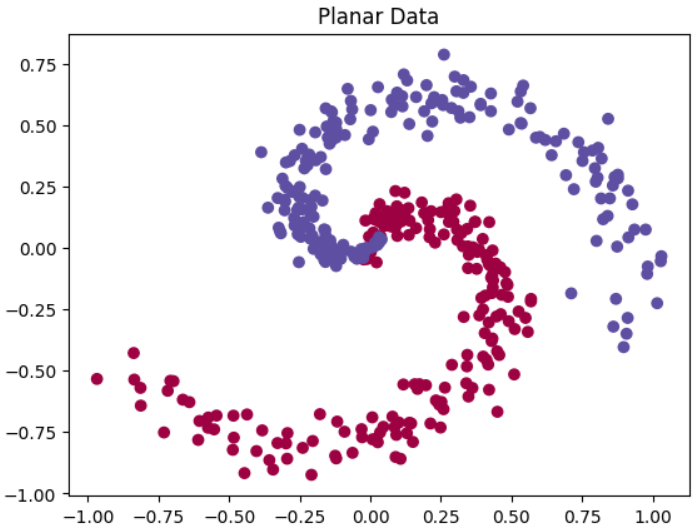
SKILLING-5:

Implement a 2-class classification neural network with two hidden layers Use units with a non-linear activation function, such as tanh. Compute the cross entropy loss, Implement forward and backward propagation using python functions Use Planar data from Kaggle.

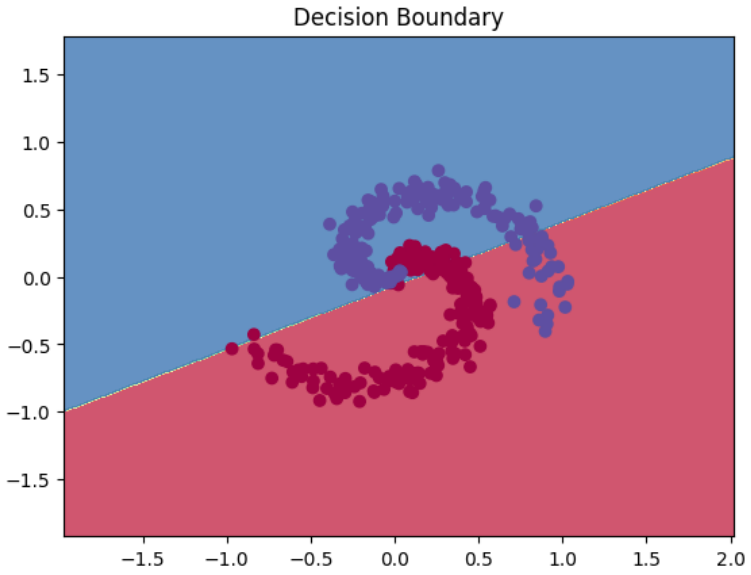
```
import torch, numpy as np, matplotlib.pyplot as plt

np.random.seed(1)
m, N, D = 400, 200, 2
X, y = np.zeros((m, D)), np.zeros((m, 1), dtype='uint8')

for j in range(2):
   ix = range(N * j, N * (j + 1))
    t, r = np.linspace(j * 4, (j + 1) * 4, N) + np.random.randn(N) * 0.2, np.linspace(0.0, 1, N) +
np.random.randn(N) * 0.05
   X[ix], y[ix] = np.c_[r * np.sin(t), r * np.cos(t)], j

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

def sigmoid(z): return 1 / (1 + np.exp(-z))
def tanh(z): return np.tanh(z)
def sigmoid_derivative(z): return sigmoid(z) * (1 - sigmoid(z))
def tanh_derivative(z): return 1 - np.tanh(z)**2

def init_params(n_x, n_h1, n_h2, n_y):
   np.random.seed(1)
   return {"W1": np.random.randn(n_h1, n_x) * 0.01, "b1": np.zeros((n_h1, 1)),
         "W2": np.random.randn(n_h2, n_h1) * 0.01, "b2": np.zeros((n_h2, 1)),
         "W3": np.random.randn(n_y, n_h2) * 0.01, "b3": np.zeros((n_y, 1))}

def forward(X, p):
   Z1, A1 = np.dot(p["W1"], X.T) + p["b1"], tanh(np.dot(p["W1"], X.T) + p["b1"])
   Z2, A2 = np.dot(p["W2"], A1) + p["b2"], tanh(np.dot(p["W2"], A1) + p["b2"])
   Z3, A3 = np.dot(p["W3"], A2) + p["b3"], sigmoid(np.dot(p["W3"], A2) + p["b3"])
```

```
    return A3, {"Z1": Z1, "A1": A1, "Z2": Z2, "A2": A2, "Z3": Z3, "A3": A3}

def loss(A3, Y): return np.sum(-np.multiply(Y, np.log(A3.T)) - np.multiply(1 - Y, np.log(1 - A3.T))) / Y.shape[0]

def backward(X, Y, p, c):
    m, dZ3 = X.shape[0], c["A3"] - Y.T
    dW3, db3 = np.dot(dZ3, c["A2"].T) / m, np.sum(dZ3, axis=1, keepdims=True) / m
    dZ2 = np.dot(p["W3"].T, dZ3) * tanh_derivative(c["Z2"])
    dW2, db2 = np.dot(dZ2, c["A1"].T) / m, np.sum(dZ2, axis=1, keepdims=True) / m
    dZ1 = np.dot(p["W2"].T, dZ2) * tanh_derivative(c["Z1"])
    dW1, db1 = np.dot(dZ1, X) / m, np.sum(dZ1, axis=1, keepdims=True) / m
    return {"dW1": dW1, "db1": db1, "dW2": dW2, "db2": db2, "dW3": dW3, "db3": db3}

def update(p, g, lr): return {k: p[k] - lr * g["d" + k] for k in p}

def train(X, Y, n_h1, n_h2, lr, iters):
    p = init_params(X.shape[1], n_h1, n_h2, 1)
    for i in range(iters):
        A3, c = forward(X, p)
        if i % 1000 == 0: print(f"Iteration {i}: Loss = {loss(A3, Y):.4f}")
        p = update(p, backward(X, Y, p, c), lr)
    return p

p = train(X, y, 5, 5, 0.01, 10000)

def predict(X, p): return (forward(X, p)[0] > 0.5).astype(int).T

def plot_boundary(p, X, y):
    xx, yy = np.meshgrid(np.arange(X[:, 0].min()-1, X[:, 0].max()+1, 0.01), np.arange(X[:, 1].min()-1, X[:, 1].max()+1, 0.01))
      plt.contourf(xx, yy, predict(np.c_[xx.ravel(), yy.ravel()], p).reshape(xx.shape), alpha=0.8, cmap=plt.cm.Spectral)
    plt.scatter(X[:, 0], X[:, 1], c=y.ravel(), cmap=plt.cm.Spectral)
    plt.show()

plot_boundary(p, X, y)
```

Output:



Iteration 0: Loss = 0.6931
Iteration 1000: Loss = 0.6931
Iteration 2000: Loss = 0.6931
Iteration 3000: Loss = 0.6931
Iteration 4000: Loss = 0.6931
Iteration 5000: Loss = 0.6931
Iteration 6000: Loss = 0.6931
Iteration 7000: Loss = 0.6931
Iteration 8000: Loss = 0.6931
Iteration 9000: Loss = 0.6931



Comment of the Evaluator (if Any)	Evaluator's Observation Marks Secured:_____out of_____ Full Name of the Evaluator: Signature of the Evaluator Date of Evaluation: