Derek Lee Deep Learning Fall 2020

Professor Curro Assignment #2

```
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt

physical_devices = tf.config.list_physical_devices('GPU')
tf.config.experimental.set_memory_growth(physical_devices[0], True)

# Constants
N = 200
sigNoise = 0.2
M = 128         # Base dimension for w
numEpochs = 200
learnRate = 0.1
momentumVal = 0.7
alpha = 0.001   # Penalty term for L2 Regularization

# Parameters for Spirals
a = [ 2, -2 ]
b = [ 1, -1 ]
t = tf.random.uniform( [2, N, 1], 0.25, 3.5*np.pi )
eps = tf.random.normal( [4, N, 1], 0, sigNoise )

# Spirals
x = ( a[0] + b[0] * t[0] ) * tf.cos( t[0] ) + eps[0]
y = ( a[0] + b[0] * t[0] ) * tf.sin( t[0] ) + eps[1]

x2 = ( a[1] + b[1] * t[1] ) * tf.cos( t[1] ) + eps[2]
y2 = ( a[1] + b[1] * t[1] ) * tf.sin( t[1] ) + eps[3]

# Creating training data
    # Class labels
classZero = tf.constant( 0, shape = [N,1], dtype = tf.float32 )
classOne = tf.constant( 1, shape = [N,1], dtype = tf.float32 )
    # Combine x and y into an input
xTrain = tf.concat( [ x, y ], 1 )
x2Train = tf.concat( [ x2, y2 ], 1 )
trainInput = tf.concat( [ xTrain, x2Train ], 0 )
trainOutput = tf.concat( [ classZero, classOne ], 0 )
```

```
    # Randomize
train = tf.concat( [ trainInput, trainOutput ], 1 )
train = tf.random.shuffle( train )
trainInput = train[ :, :-1 ]
trainOutput = train[ :, -1, np.newaxis ]


# Glorot initialization inspired by:
# https://stats.stackexchange.com/questions/47590/what-are-good-initial-weights-
in-a-neural-network
wInputs = np.array( [ 2, M, M//4, M//16 ] )
wOutputs = np.array( [ wInputs[1], wInputs[2], wInputs[3],  1 ] )
wInputs = wInputs[ :, np.newaxis ]
wOutputs = wOutputs[ :, np.newaxis ]
r = 2 * np.sqrt( 6 / ( wInputs + wOutputs ) )
numLayers = len( wInputs )
bVariance = np.ones( [ numLayers, 1 ] )
bMean = -1 * np.ones( [ numLayers, 1 ] )



# Module containing Logistic Classification model
class logClassMod( tf.Module ):

    def __init__( self ):

        # Trainable Tensorflow variables
        self.weights = []
        for i in range( numLayers ):
            self.weights.append( tf.Variable( tf.random.uniform( [ wInputs[i][0],
 wOutputs[i][0] ], -r[i], r[i] ) ) )

        self.biases = []
        for i in range( numLayers ):
            self.biases.append( tf.Variable( tf.random.normal( [ wOutputs[i][0] ]
, bMean[i], bVariance[i] ) ) )


    # Calculates yHat given x
    # Uses multilayer perceptron
    @tf.function
    def estY( self, x ):

        tempX = x
```

```
        for i in range( numLayers ):

            tempX = tempX @ self.weights[i] + self.biases[i]

            # Applies eLU to every layer except for the last, which applies sigmo
id
            if i != numLayers - 1:
                tempX = self.elu( tempX )
            else:
                return self.sigmoid( tempX )


    @tf.function
    def elu( self, x ):

        eluAlpha = 0.5
        return tf.where( x >= 0, x, eluAlpha * ( tf.exp(x) - 1 ) )  # eLU


    # Sigmoid function to get activation levels
    @tf.function
    def sigmoid( self, x ):

        return 1 / ( 1 + tf.exp( -x ) )


    def train( self ):

        # Stochastic Gradient Descent
        opt = tf.keras.optimizers.SGD( learning_rate = learnRate, momentum = mome
ntumVal )

        # Binary Cross Entropy
        bce = tf.keras.losses.BinaryCrossentropy()

        print( "Starting Loss (Unregularized):", bce( self.estY( trainInput ), tr
ainOutput ).numpy() )

        # Iterate through epochs
        for _ in range( numEpochs ):

            with tf.GradientTape() as tape:

                yHat = self.estY( trainInput )
                loss = bce( trainOutput, yHat )
```

```
                # L2 Regularization
                for i in self.weights:
                    regularizer = tf.nn.l2_loss( i )
                    loss += alpha * regularizer

                print(loss)

            grads = tape.gradient( loss, self.variables )
            opt.apply_gradients( zip( grads, self.variables ) )

        print( "Final Loss (Unregularized):", bce( self.estY( trainInput ), trainOutput ).numpy() )


    def plotSpirals( self ):

        # Create true input data
        xTrue, yTrue = np.meshgrid( tf.cast( tf.linspace( -15, 15, 100 ), tf.float32 ), tf.cast( tf.linspace( -15, 15, 100 ), tf.float32 ) )
        xTrueCol = xTrue.flatten()
        xTrueCol = xTrueCol[ :, np.newaxis ]
        yTrueCol = yTrue.flatten()
        yTrueCol = yTrueCol[ :, np.newaxis ]
        trueInput = tf.concat( [ xTrueCol, yTrueCol ], 1 )

        # Calculate output
        trueOutput = self.estY( trueInput )

        # Plot boundary
        plt.figure()
        conPlot = plt.contourf( xTrue, yTrue, np.reshape( trueOutput, xTrue.shape ), levels = [ 0.5, 1 ] )
        plt.colorbar( conPlot )

        # Plot spirals
        plt.plot( x, y, 'ro', markeredgecolor = 'black' )
        plt.plot( x2, y2, 'bo', markeredgecolor = 'black' )
        plt.xlabel( 'x' )
        plt.ylabel( 'y', rotation = 0 )
        plt.title( "Spirals" )
        plt.show()
```

```
def main():

    # Ensure input data is valid
    assert not np.any( np.isnan( trainInput ) )
    assert not np.any( np.isnan( trainOutput ) )

    model = logClassMod()

    # Training
    model.train()

    # Get predictions
    yHat = model.estY( trainInput )
    yPred = tf.where( yHat >= 0.5, float(1), float(0) )
    z = tf.math.equal( trainOutput, yPred )
    print( "Number of 1's: ", tf.reduce_sum( yPred ).numpy() )
    print( "% Correct: ", ( tf.reduce_sum(tf.cast(z, tf.float32))/len(z) ).numpy(
)*100 )

    # Plot boundary between classes
    model.plotSpirals()


main()
```

My f was relatively consistent throughout the assignment. I initially used a ReLU as my nonlinear function, but quickly moved to an eLU instead. I initially found a relatively high alpha parameter (0.5) for my eLU gave me a large improvement, but with the final version of my program, changing the alpha parameter did not have much of an effect. I use a sigmoid as my activation function. I played around with my learning rate, but it did not have a significant effect on my model (except when it was so high that it caused nan values). On the final version of the program, I tried a momentum of 0.99, which gave me nan values. I found 0.7 was the sweet spot, which gave me a good improvement while also consistently not giving me nan values.

I started off with 3 layers of weights, the first 2 mapping between R^2 and R^2, with the final layer mapping from R^2 to R^1. Initially, I did not have much success. This did not do much better than random guessing (and in some cases, did worse). Next, I tried an additional layer, along with mapping to different dimensions with every layer. I mapped from R^2 to R^{64} to R^{32} to R^4 to R^1. Additionally, I researched how to initialize my weights and learned about Glorot initialization. This was better, but still not very successful. I achieved around a 70% accuracy. Finally, I tried changing the initializations of my biases. I had my biases initialized to 0, but I changed it to a normal distribution, with mean -1 and standard deviation 1. I also changed the dimensions of my weights. I mapped from R^2 to R^{128} to R^{32} to R^8 to R^1. This gave me the greatest success. Although changing the biases gave me a significant improvement in my accuracy, changing the dimensions of my weights gave me consistency. I consistently achieve perfect accuracy.

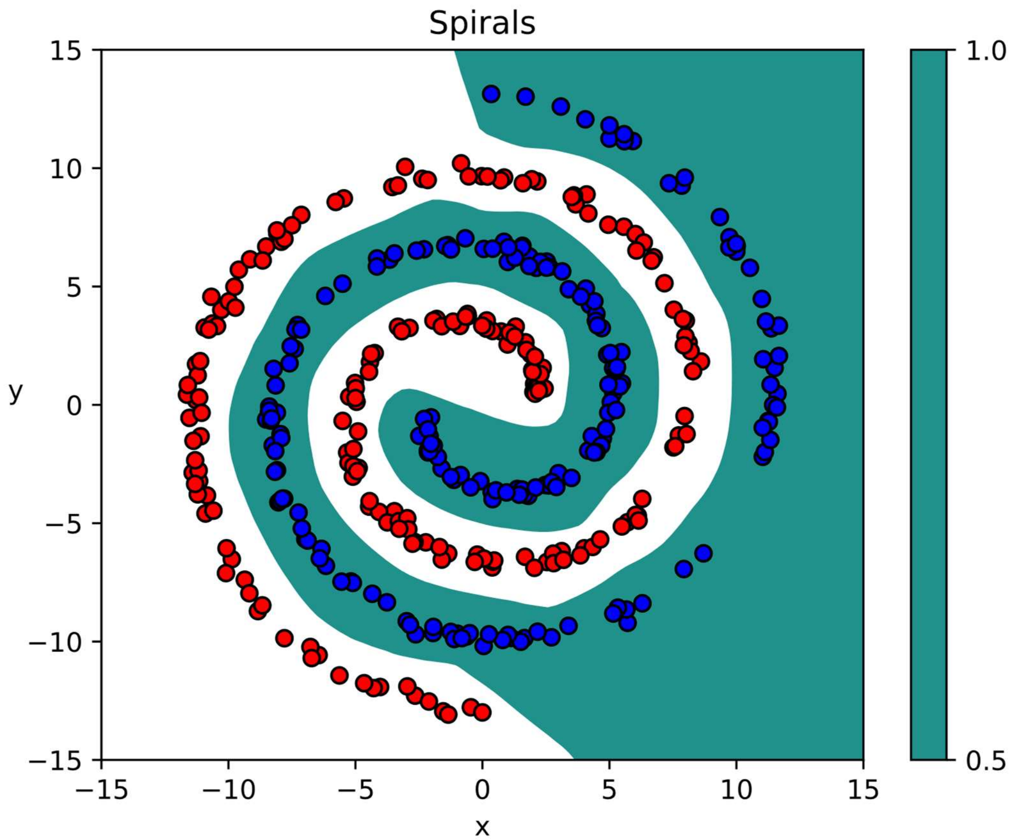

Figure 1: Example plot for the boundary of the logistic classification of two spirals using a multilayer perceptron