

```
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 = 0.3
M = 128          # Base dimension for w
numEpochs = 200
learnRate = 0.1
momentumVal = 0.6
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 ) )
bRanges = np.array( [ 1, 1, 1, 1, 1 ] )

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

    def __init__( self ):

        # Trainable Tensorflow variables
        self.w0 = tf.Variable( tf.random.uniform( [ wInputs[0][0], wOutputs[0][0] ], -r[0], r[0] ) )
        self.w1 = tf.Variable( tf.random.uniform( [ wInputs[1][0], wOutputs[1][0] ], -r[1], r[1] ) )
        self.w2 = tf.Variable( tf.random.uniform( [ wInputs[2][0], wOutputs[2][0] ], -r[2], r[2] ) )
        self.w3 = tf.Variable( tf.random.uniform( [ wInputs[3][0], wOutputs[3][0] ], -r[3], r[3] ) )

        self.b0 = tf.Variable( tf.random.normal( [ wOutputs[0][0] ], -bRanges[0], bRanges[0] ) )
        self.b1 = tf.Variable( tf.random.normal( [ wOutputs[1][0] ], -bRanges[1], bRanges[1] ) )
        self.b2 = tf.Variable( tf.random.normal( [ wOutputs[2][0] ], -bRanges[2], bRanges[2] ) )
        self.b3 = tf.Variable( tf.random.normal( [ wOutputs[3][0] ], -bRanges[3], bRanges[3] ) )

        self.weights = [ self.w0, self.w1, self.w2, self.w3 ]
        self.biases = [ self.b0, self.b1, self.b2, self.b3 ]

```

```

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

    tempX = x

    for i in range( len( self.weights ) ):

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

        # Applies eLU to every layer except for the last, which applies sigmoid
        if i != len( self.weights ) - 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 = momentumVal )

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

    print( "Starting Loss (Unregularized):", bce( self.estY( trainInput ), trainOutput ).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 ), train
Output ).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 0.6 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 \mathbb{R}^2 and \mathbb{R}^2 , with the final layer mapping from \mathbb{R}^2 to \mathbb{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 \mathbb{R}^2 to \mathbb{R}^{64} to \mathbb{R}^{32} to \mathbb{R}^4 to \mathbb{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 \mathbb{R}^2 to \mathbb{R}^{128} to \mathbb{R}^{32} to \mathbb{R}^8 to \mathbb{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.

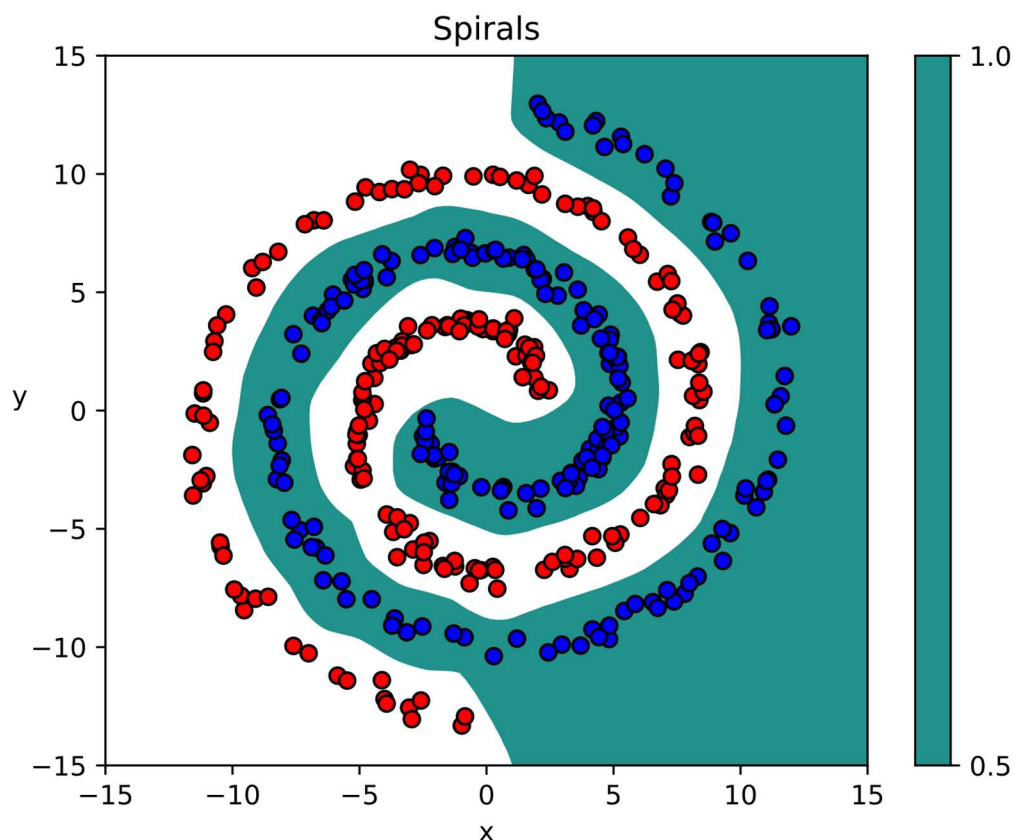


Figure 1: Example plot for the boundary of the logistic classification of two spirals using a multilayer perceptron