```
#!/usr/local/bin/python3.6

### Kevin Sheng
### ECE471 Selected Topics in Machine Learning - Assignment 2

# Initially, I tried ReLU and sigmoidal activations on a neural net
with 1
# hidden layer, and they had a hard time converging. Adding sin(x1)
and sin(x2)
# as inputs or using sin as the activation function made it converge
almost
# instantly, but that seemed to be cheating since we know the form of
the data.
# The L2 regularization also made it difficult to converge, so I
turned it off
# when experimenting. The sigmoid and tanh activations worked over a
very large
# numer of epochs.

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

from tqdm import tqdm

class Data():
    def __init__(self):
        num_samp = 250
        num_classes = 2
        sigma = 0.1
        np.random.seed(31415)

        self.x = np.zeros((num_samp * num_classes, 2))
        self.y = np.zeros(num_samp * num_classes, dtype='uint8')
        for c in range(num_classes):
            i = range(num_samp * c, num_samp * (c+1))
            r = np.linspace(1, 15, num_samp)
            theta = (np.linspace(c*3, (c+4)*3, num_samp)
                + np.random.randn(num_samp) * sigma)
            self.x[i] = np.c_[r*np.sin(theta), r*np.cos(theta)]
            self.y[i] = c

class Model():
    def __init__(self, data, training_epochs=10, learning_rate=.01,
n_hidden=64, hidden_layers=3):
        self.training_epochs = training_epochs
        self.learning_rate = learning_rate
        self.data = data
        n_input = 2
        n_output = 1
```

```
        self.x = tf.placeholder(tf.float32, [1, n_input])
        self.y = tf.placeholder(tf.float32, [1, n_output])

        weights = ([tf.Variable(tf.random_normal([n_input,
n_hidden]))]
                   + [tf.Variable(tf.random_normal([n_hidden,
n_hidden]))]*(hidden_layers-1)
                   + [tf.Variable(tf.random_normal([n_hidden,
n_output]))])

        biases = ([tf.Variable(tf.zeros([n_hidden]))]
                  +
[tf.Variable(tf.zeros([n_hidden]))]*(hidden_layers-1)
                   + [tf.Variable(tf.zeros([n_output]))])

        layer_hidden = []
        for h in range(hidden_layers):
            if h == 0:
                fc = tf.add(tf.matmul(self.x, weights[h]), biases[h])
            else:
                fc = tf.add(tf.matmul(layer_hidden[h-1], weights[h]),
biases[h])
            layer_hidden.append(tf.sin(fc))
        #  layer_hidden2 = tf.add(tf.matmul(layer_hidden,
weights['hidden2']), biases['hidden2'])
        #  layer_hidden2 = tf.tanh(layer_hidden2)
        layer_output = tf.add(tf.matmul(layer_hidden[-1],
weights[-1]), biases[-1])
        self.layer_output = tf.sigmoid(layer_output)

        loss =
tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(logits=layer_ou
tput, labels=self.y))
        l2 = tf.reduce_sum([tf.reduce_sum(tf.pow(var,2)) for var in
weights + biases])
        self.loss = loss + .01 * l2

    def train(self):
        optim =
tf.train.AdamOptimizer(learning_rate=self.learning_rate).minimize(self
.loss)
        init = tf.global_variables_initializer()

        self.sess = tf.Session()
        self.sess.run(init)

        with tqdm(range(self.training_epochs)) as t:
            for epoch in t:
                cost = 0
```

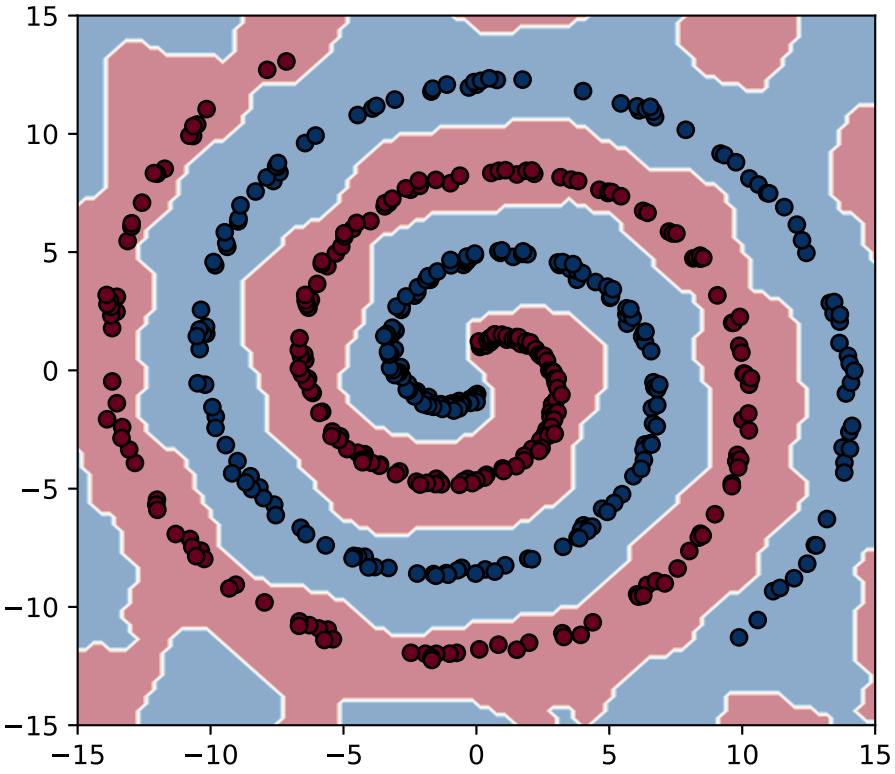
```
                for i in range(self.data.x.shape[0]):
                    c, _ = self.sess.run([self.loss, optim],
                            feed_dict={self.x:
self.data.x[i].reshape(1, 2), self.y: self.data.y[i].reshape(1,1)})
                    cost += c
                t.set_postfix(loss=cost/self.data.x.shape[0])

if __name__ == '__main__':
    data = Data()
    model = Model(data)
    model.train()

    n = 100
    x = np.linspace(-15, 15, n)
    y = np.linspace(-15, 15, n)
    xx, yy, = np.meshgrid(x, y)

    decision = np.zeros((n, n))
    for i in range(n):
        for j in range(n):
            xy = np.reshape([x[j], y[i]], (1,2))
            p = model.sess.run(model.layer_output, feed_dict={model.x:
xy})
            if p >= .5:
                decision[i, j] = 1
    cm = plt.cm.RdBu

    plt.contourf(xx, yy, decision, cmap=cm, alpha=.5)
    plt.scatter(data.x[:, 0], data.x[:, 1], c=data.y,
                cmap=cm, alpha=1, edgecolors='black')
    plt.show()
```


15
10
5
0
−5
−10
−15
−15
−10
−5
0
5
10
15