NetID: sa857

Minimal neural network implementation

This is a "bare bones" implementation of a 2-layer neural network for classification, using rectified linear units as activation functions. The code is from Andrej Karpathy; please see [this page (http://cs231n.github.io/neural-networks-case-study/)](http://cs231n.github.io/neural-networks-case-study/) for an annotated description of the code.

Your task in this part of the assigment is to extend this to a 3-layer network, and to experiment with some different settings of the parameters.

Problem 1 (a): Gradients (5 points)

Calculate the gradients as described in the `assn7.pdf` document.

Working:

$$h_1 = relu(w_1 x + b_1)$$

$$h_2 = relu(w_2 h_1 + b_2)$$

$$f = w_3 h_2 + b_3$$

$$L = \frac{1}{2}(y - f(x))^2$$

$$\frac{\partial f}{\partial w_3} = h_2^T \qquad \frac{\partial f}{\partial h_2} = w_3^T$$

$$\frac{\partial h_2}{\partial b_2} = 0 + 1 = 1$$

$$\frac{\partial h_2}{\partial w_2} = h_1 \qquad \frac{\partial h_2}{\partial h_1} = w_2$$

$$\frac{\partial h_1}{\partial b_1} = 0 + 1 = 1$$

$$\frac{\partial h_1}{\partial w_1} = x$$

LAYER 3

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial b_3} = (f-y) \cdot 1$$

$$\frac{\partial L}{\partial w_3} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial w_3} = (f-y) \cdot h_2^T$$

$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial h_2} = (f-y) \cdot w_3^T$$

LAYER 2

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial h_2} \cdot \frac{\partial h_2}{\partial b_2} = (f-y) \cdot w_3^T \cdot 1$$

$$\frac{\partial L}{\partial w_2} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial h_2} \cdot \frac{\partial h_2}{\partial w_2} = (f-y) \cdot w_3^T \cdot h_1^T$$

$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} = (f-y) \cdot w_3^T \cdot w_2^T$$

LAYER 1

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial b_1}$$

$$= (f-y) \cdot w_3^T \cdot w_2^T \cdot 1$$

$$\frac{\partial L}{\partial w_1} = \frac{\partial L}{\partial f} \cdot \frac{\partial f}{\partial h_2} \cdot \frac{\partial h_2}{\partial h_1} \cdot \frac{\partial h_1}{\partial w_1}$$

$$= (f-y) \cdot w_3^T \cdot w_2^T \cdot x^T$$

$\frac{\partial L}{\partial x}$ == not required

Final answers in LaTeX (fully expanded except dL/df):

Layer 3

$$\frac{\partial L}{\partial b_3} = \frac{\partial L}{\partial f}$$
$$\frac{\partial L}{\partial W_3} = \frac{\partial L}{\partial f} h_2^T$$
$$\frac{\partial L}{\partial h_2} = \frac{\partial L}{\partial f} W_3^T$$

Layer 2

$$\frac{\partial L}{\partial b_2} = \frac{\partial L}{\partial f} W_3^T$$
$$\frac{\partial L}{\partial W_2} = \frac{\partial L}{\partial f} W_3^T h_1^T$$
$$\frac{\partial L}{\partial h_1} = \frac{\partial L}{\partial f} W_3^T W_2^T$$

Layer 1

$$\frac{\partial L}{\partial b_1} = \frac{\partial L}{\partial f} W_3^T W_2^T$$
$$\frac{\partial L}{\partial W_1} = \frac{\partial L}{\partial f} W_3^T W_2^T X^T$$

In [33]:
```
import numpy as np
import matplotlib.pyplot as plt

%matplotlib inline
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
plt.rcParams['axes.facecolor'] = 'lightgray'
```

In [34]:
```
np.random.seed(0)
N = 100 # number of points per class
D = 2 # dimensionality
K = 3 # number of classes to classify into
X = np.zeros((N*K,D))
y = np.zeros(N*K, dtype='uint8')
```

In [35]:
```
print(X.shape)
print(y.shape)
```

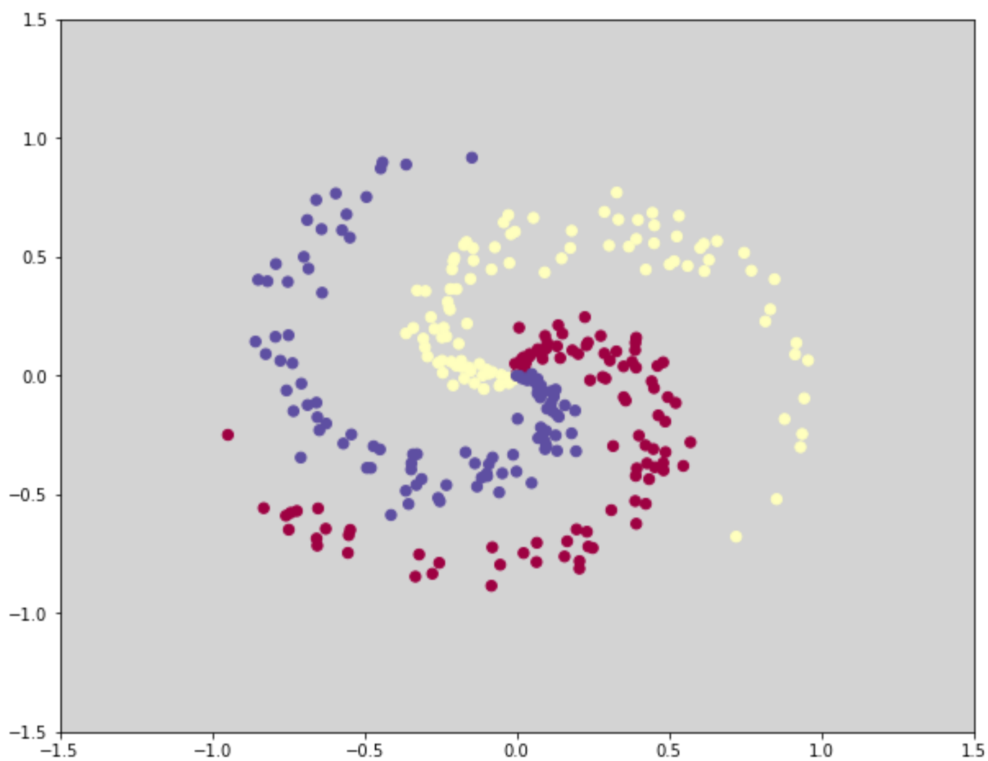
```
(300, 2)
(300,)
```

In [36]:
```
# generate random spiral data

# for each class/color
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)*0.3 # theta
    X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
    y[ix] = j

# plot the points
fig = plt.figure()
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.xlim([-1.5,1.5])
plt.ylim([-1.5,1.5])
```

Out[36]:
```
(-1.5, 1.5)
```



In [37]:
```
def train_2_layer_network(H1=100):
    # initialize parameters randomly
    # H1 = 100 # size of hidden layer
    W1 = np.random.randn(D,H1)
    b1 = np.zeros((1,H1))
    W2 = np.random.randn(H1,K)
    b2 = np.zeros((1,K))

    # some hyperparameters
    step_size = 1e-1

    # gradient descent loop
    num_examples = X.shape[0]
    for i in range(20000):

        # evaluate class scores, [N x K]
        hidden_layer = np.maximum(0, np.dot(X, W1) + b1) # note, ReLU activation
        scores = np.dot(hidden_layer, W2) + b2

        # compute the class probabilities
        exp_scores = np.exp(scores)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

        # compute the loss: minus log prob
        correct_logprobs = -np.log(probs[range(num_examples),y])
        loss = np.sum(correct_logprobs)/num_examples

        # log for 20 iterations
        if i % 1000 == 0:
            print("iteration %d: loss %f" % (i, loss))

        # compute the gradient on scores
        dscores = np.array(probs)
        dscores[range(num_examples),y] -= 1
        dscores /= num_examples

        # backpropate the gradient to the parameters
        # first backprop into parameters W2 and b2
        dW2 = np.dot(hidden_layer.T, dscores)
        db2 = np.sum(dscores, axis=0, keepdims=True)
        # next backprop into hidden layer
        dhidden = np.dot(dscores, W2.T)
        # backprop the ReLU non-linearity
        dhidden[hidden_layer <= 0] = 0
        # finally into W,b
        dW1 = np.dot(X.T, dhidden)
        db1 = np.sum(dhidden, axis=0, keepdims=True)

        # perform a parameter update
        W1 += -step_size * dW1
        b1 += -step_size * db1
        W2 += -step_size * dW2
        b2 += -step_size * db2

    return W1, b1, W2, b2


W1, b1, W2, b2 = train_2_layer_network(100)
```

```
iteration 0: loss 5.234923
iteration 1000: loss 0.135617
iteration 2000: loss 0.102535
iteration 3000: loss 0.085465
iteration 4000: loss 0.074823
iteration 5000: loss 0.067282
iteration 6000: loss 0.061611
iteration 7000: loss 0.057220
iteration 8000: loss 0.053762
iteration 9000: loss 0.050898
iteration 10000: loss 0.048406
iteration 11000: loss 0.046287
iteration 12000: loss 0.044470
iteration 13000: loss 0.042857
iteration 14000: loss 0.041421
iteration 15000: loss 0.040143
iteration 16000: loss 0.038996
iteration 17000: loss 0.037952
iteration 18000: loss 0.036997
iteration 19000: loss 0.036121
```

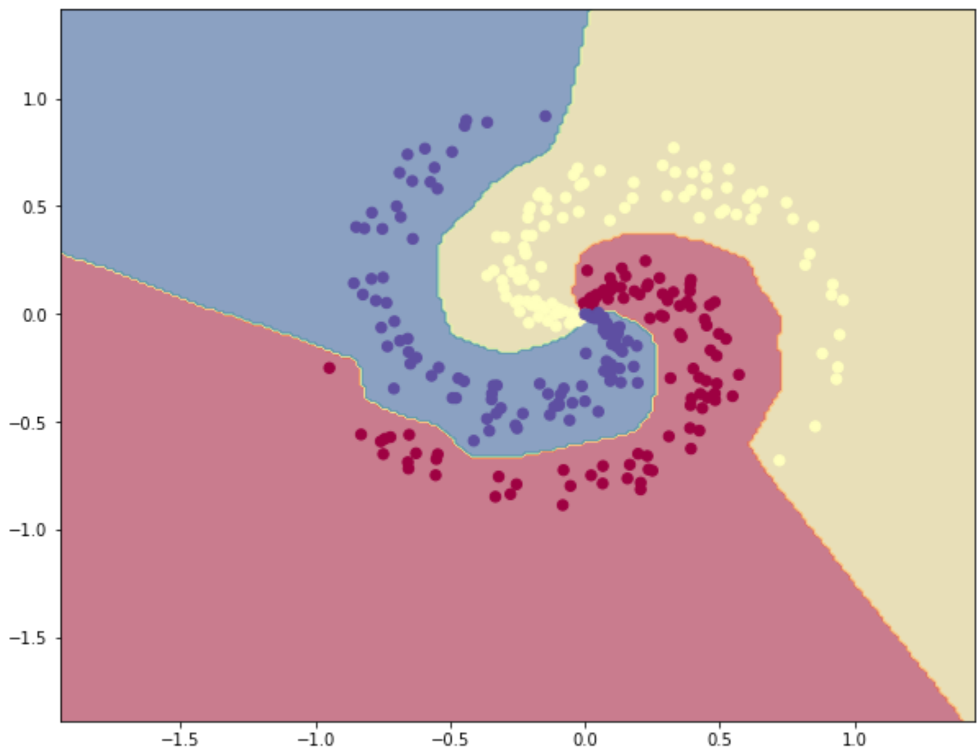
In [38]:

```
# evaluate training set accuracy
hidden_layer = np.maximum(0, np.dot(X, W1) + b1)
scores = np.dot(hidden_layer, W2) + b2
predicted_class = np.argmax(scores, axis=1)
print('training accuracy: %.2f' % (np.mean(predicted_class == y)))
```

```
training accuracy: 0.99
```

In [39]:
```
# plot the resulting classifier
h = 0.015
x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + .5
y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + .5
xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                     np.arange(y_min, y_max, h))
Z = np.dot(np.maximum(0, np.dot(np.c_[xx.ravel(), yy.ravel()], W1) + b1), W2) + b2
Z = np.argmax(Z, axis=1)
Z = Z.reshape(xx.shape)
fig = plt.figure()
plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.5)
plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
plt.xlim(xx.min(), xx.max())
plt.ylim(yy.min(), yy.max())
```

Out[39]:
```
(-1.8850693285424291, 1.4149306714575494)
```



Problem 1 (b): Extend the code from two layers to three layers (15 points)

Run the code provided in the notebook minimal neural network.ipynb and inspect it to be sure you understand how it works. (We did this in class!) Then, after working out the derivatives in part (a) above, extend the code by writing a function that implements a 3-layer version. Your function declaration should look like this:

```
def train_3_layer_network(H1=100, H2=100)
```

where H1 is the number of hidden units in the first layer, and H2 is the number of hidden units in the second layer. Then train a 3-layer network and display the classification results in your notebook, as is done for the 2-layer network in the starter code.

In [28]:
```
def train_3_layer_network(H1=100, H2=100):

    # X :: (300, 2)
    W1 = np.random.randn(D,H1) # (2, 100)
    b1 = np.zeros((1,H1)) # (1, 100)
    # z1 == (X * W1) + b :: (300, 100)
    # a1 == relu(z1) :: (300, 100)

    # assume H2 == 150
    # a1 :: (300, 100)
    W2 = np.random.randn(H1, H2) # (100, 150)
    b2 = np.zeros((1, H2)) # (1, 150)
    # z2 == (a1 * W2) + b2 :: (300, 150)
    # a2 == relu(z2) :: (300, 150)

    # a2 :: (300, 150)
    W3 = np.random.randn(H2,K) # (150, 3)
    b3 = np.zeros((1,K)) # (1, 3)
    # z3 "scores" == (a2 * W3) + b3 :: (300, 3)
    # a3 == softmax(z3) :: (300, 3)

    # some hyperparameters
    step_size = 1e-1

    # gradient descent loop
    num_examples = X.shape[0] ### 300
    for i in range(20000):

        # evaluate class scores, [N x K] :: (300, 3) i.e. which class could it be
for each example?
        hidden_layer = np.maximum(0, np.dot(X, W1) + b1) # note, ReLU activation
        hidden_layer_two = np.maximum(0, np.dot(hidden_layer, W2) + b2)
        scores = np.dot(hidden_layer_two, W3) + b3

        # compute the class probabilities i.e. softmax
        exp_scores = np.exp(scores)
        probs = exp_scores / np.sum(exp_scores, axis=1, keepdims=True) # [N x K]

        # compute the loss: minus log prob i.e. the loss function
        correct_logprobs = -np.log(probs[range(num_examples),y])
        loss = np.sum(correct_logprobs)/num_examples

        # log for 20 iterations
        if i % 1000 == 0:
            print("iteration %d: loss %f" % (i, loss))

        # compute the gradient on scores
        dscores = np.array(probs)
        dscores[range(num_examples),y] -= 1
        dscores /= num_examples


        # backpropate the gradient to the parameters
        dW3 = np.dot(hidden_layer_two.T, dscores)
        db3 = np.sum(dscores, axis=0, keepdims=True)
        dhidden_two = np.dot(dscores, W3.T)
        dhidden_two[hidden_layer_two <= 0] = 0 # backprop the ReLU non-linearity

        dW2 = np.dot(hidden_layer.T, dhidden_two)
        db2 = np.sum(dhidden_two, axis=0, keepdims=True)
        dhidden = np.dot(dhidden_two, W2.T)
        dhidden[hidden_layer <= 0] = 0 # backprop the ReLU non-linearity
```

In [40]:
```
def plotClassifier(_W1, _b1, _W2, _b2, _W3, _b3):
    h = 0.015
    x_min, x_max = X[:, 0].min() - 1, X[:, 0].max() + .5
    y_min, y_max = X[:, 1].min() - 1, X[:, 1].max() + .5
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h),
                         np.arange(y_min, y_max, h))

#     ravel_term = np.c_[xx.ravel(), yy.ravel()]
#     first = np.dot(ravel_term, _W1) + _b1
#     second = np.maximum(0, first)
#     third = np.dot(second, _W2) + _b2
#     fourth = np.maximum(0, third)
#     Z = np.dot(third, _W3) + _b3

    Z = np.dot(np.maximum(0, np.dot(np.maximum(0, np.dot(np.c_[xx.ravel(), yy.rave
l()], _W1) + _b1), _W2) + _b2), _W3) + _b3


    Z = np.argmax(Z, axis=1)
    Z = Z.reshape(xx.shape)
    fig = plt.figure()
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral, alpha=0.5)
    plt.scatter(X[:, 0], X[:, 1], c=y, s=40, cmap=plt.cm.Spectral)
    plt.xlim(xx.min(), xx.max())
    plt.ylim(yy.min(), yy.max())
```

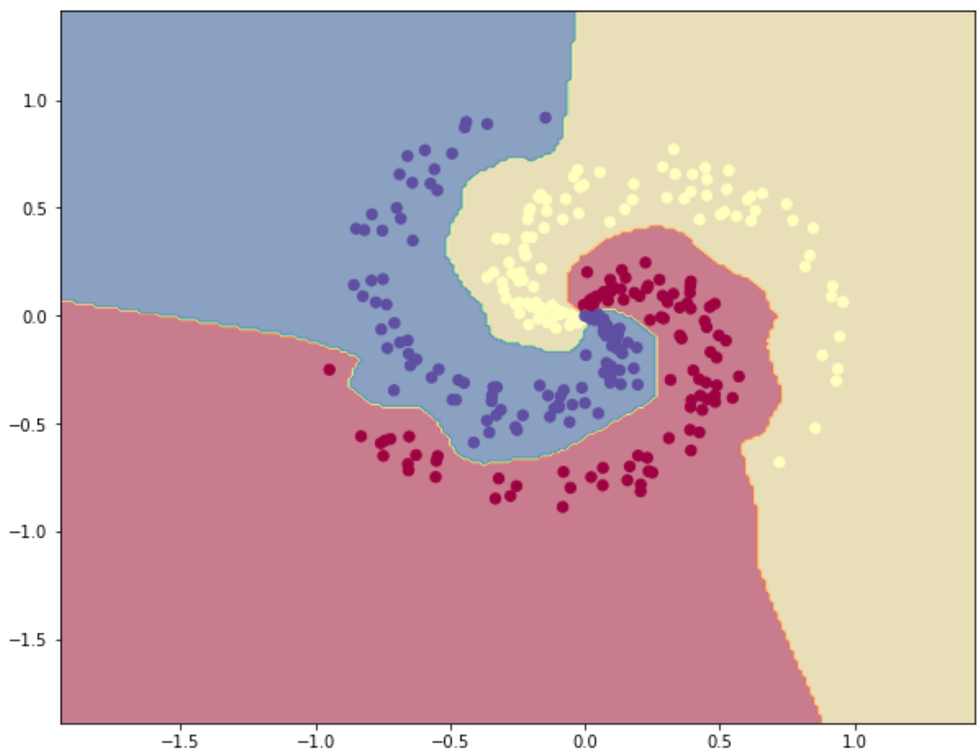
In [41]:
```
def evaluateAccuracy(_W1, _b1, _W2, _b2, _W3, _b3):
    _hidden_layer = np.maximum(0, np.dot(X, _W1) + _b1)
    _hidden_layer_two = np.maximum(0, np.dot(_hidden_layer, _W2) + _b2)
    _scores = np.dot(_hidden_layer_two, _W3) + _b3
    _predicted_class = np.argmax(_scores, axis=1)
    print('training accuracy: %.2f' % (np.mean(_predicted_class == y)))
```

In [31]:
```
three_layer_test = train_3_layer_network(100, 100)
```

```
iteration 0: loss 18.962838
iteration 1000: loss 0.020524
iteration 2000: loss 0.018109
iteration 3000: loss 0.016697
iteration 4000: loss 0.015892
iteration 5000: loss 0.015310
iteration 6000: loss 0.014854
iteration 7000: loss 0.014486
iteration 8000: loss 0.014179
iteration 9000: loss 0.013835
iteration 10000: loss 0.013618
iteration 11000: loss 0.013438
iteration 12000: loss 0.013279
iteration 13000: loss 0.013137
iteration 14000: loss 0.013016
iteration 15000: loss 0.012904
iteration 16000: loss 0.012789
iteration 17000: loss 0.012648
iteration 18000: loss 0.012564
iteration 19000: loss 0.012490
```

In [43]:
```
evaluateAccuracy(*three_layer_test)
plotClassifier(*three_layer_test)
```

```
training accuracy: 0.99
```



- This seems to be overfitting the data due to the contorted nature of the decision boundary. Perhaps the size of one or both of the hidden layers should be reduced.
- The bias is low and the variance is high.

Problem 1 (c): Experiment with different parameter settings (10 points)

Now experiment with different network configurations and training parameters. For example, you can train models with different numbers of hidden nodes H1 and H2. Train at least three and no more than five networks. For each network, display the decision boundaries on the training data, and include a Markdown cell that describes its behavior relative to the other networks you train. Specifically, comment on how the different settings of the parameters change the bias and variance of the fitted model.

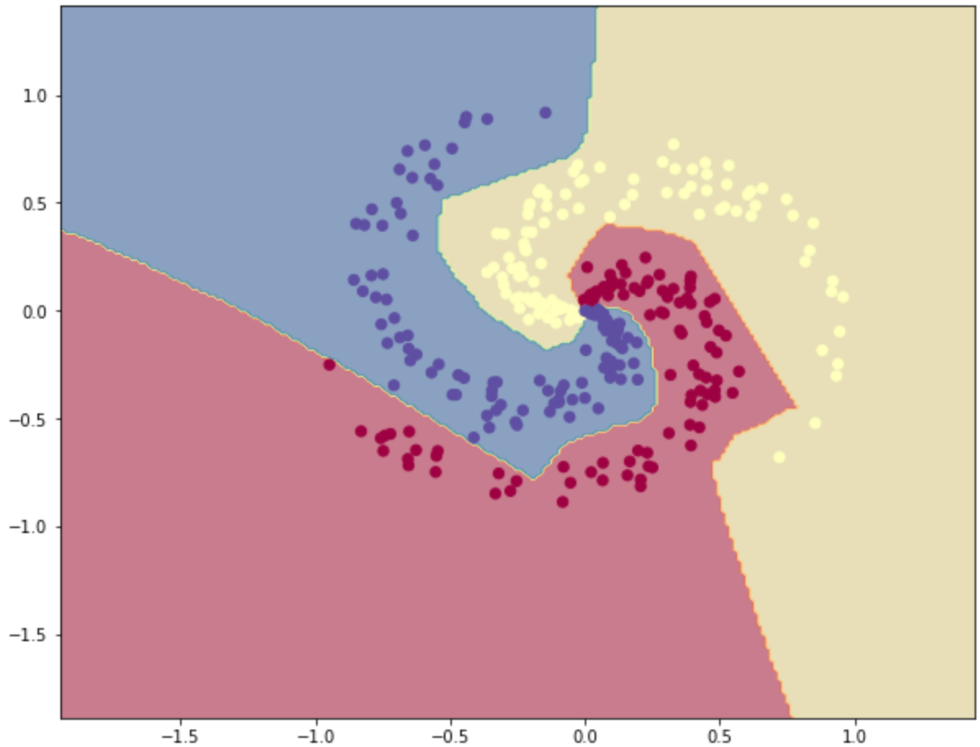
In [44]:
```
# two layer vs three layer?
# fewer nodes in first layer, many in second,
# fewer nodes in both
# even fewer in both


# Network 1
experiment_one = train_3_layer_network(5, 100)
```

```
iteration 0: loss 2.682754
iteration 1000: loss 0.142557
iteration 2000: loss 0.083928
iteration 3000: loss 0.067824
iteration 4000: loss 0.059139
iteration 5000: loss 0.049983
iteration 6000: loss 0.042809
iteration 7000: loss 0.037702
iteration 8000: loss 0.033456
iteration 9000: loss 0.030236
iteration 10000: loss 0.027612
iteration 11000: loss 0.025666
iteration 12000: loss 0.024171
iteration 13000: loss 0.022964
iteration 14000: loss 0.021950
iteration 15000: loss 0.021088
iteration 16000: loss 0.020370
iteration 17000: loss 0.019752
iteration 18000: loss 0.019208
iteration 19000: loss 0.018690
```

In [45]:
```
evaluateAccuracy(*experiment_one)
plotClassifier(*experiment_one)
```

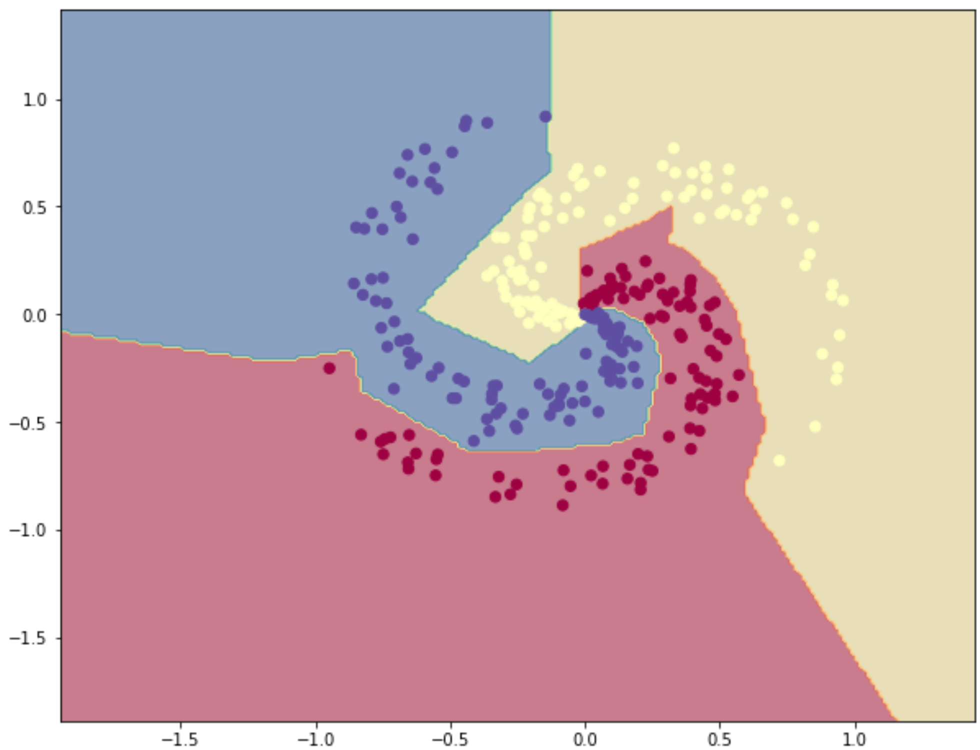
```
training accuracy: 0.99
```



- Compared to the two-layer network, this three layer (5, 100) network seems to be underfitting, as shown by the straight lines in the decision boundary. The variance is low. The bias is also somewhat low because no point seems to be misclassified.
- Compared to the (100, 100) three layer network, it is also underfitting, probably because the first hidden layer has only size 5.

In [46]:
```
# Network 2
experiment_two = train_3_layer_network(5, 10)
evaluateAccuracy(*experiment_two)
plotClassifier(*experiment_two)
```

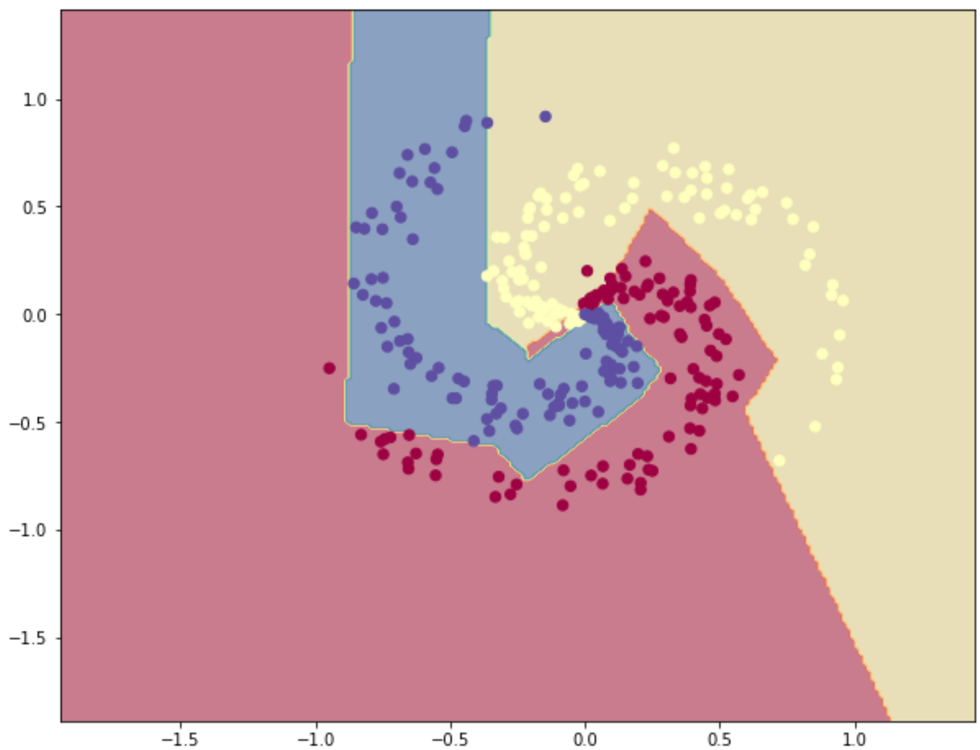
```
iteration 0: loss 1.406998
iteration 1000: loss 0.186421
iteration 2000: loss 0.099437
iteration 3000: loss 0.068371
iteration 4000: loss 0.050886
iteration 5000: loss 0.040704
iteration 6000: loss 0.035230
iteration 7000: loss 0.031719
iteration 8000: loss 0.029032
iteration 9000: loss 0.027066
iteration 10000: loss 0.025212
iteration 11000: loss 0.023771
iteration 12000: loss 0.022694
iteration 13000: loss 0.021773
iteration 14000: loss 0.020943
iteration 15000: loss 0.020302
iteration 16000: loss 0.019788
iteration 17000: loss 0.019164
iteration 18000: loss 0.018746
iteration 19000: loss 0.018329
training accuracy: 0.99
```



- In this network, both hidden layers have a small size (5, 10), which is quite low compared to the others.
- Because of this, it seems to be underfitting, as shown by the jagged/straight decision boundary.
- However, no point seems to be misclassified, so even though variance is low the bias is also mid to low.

In [49]:
```
# Network 3
experiment_three = train_3_layer_network(3, 5)
evaluateAccuracy(*experiment_three)
plotClassifier(*experiment_three)
```

```
iteration 0: loss 1.351175
iteration 1000: loss 0.381542
iteration 2000: loss 0.291126
iteration 3000: loss 0.228146
iteration 4000: loss 0.211076
iteration 5000: loss 0.203899
iteration 6000: loss 0.199923
iteration 7000: loss 0.196938
iteration 8000: loss 0.194510
iteration 9000: loss 0.192650
iteration 10000: loss 0.189899
iteration 11000: loss 0.187884
iteration 12000: loss 0.186355
iteration 13000: loss 0.185143
iteration 14000: loss 0.184235
iteration 15000: loss 0.183386
iteration 16000: loss 0.182749
iteration 17000: loss 0.182151
iteration 18000: loss 0.181376
iteration 19000: loss 0.180762
training accuracy: 0.94
```



- Here I reduced the layer sizes even further, which leads to dramatic underfit, as shown by the straight blocked regions created by the decision boundary.
- Bias is high and variance is low.

2 layer vs 3 layer:

- if the hidden layer sizes in the 3 layer network are sufficiently small (as shown above), the 2 layer network can easily outperform it
- although in general, a 3 layer network should have greater model complexity/variance than a 2 layer network, and fit the data better