Reference

Coursera deep learning series by Andrew NG.

Planar data classification with one hidden layer

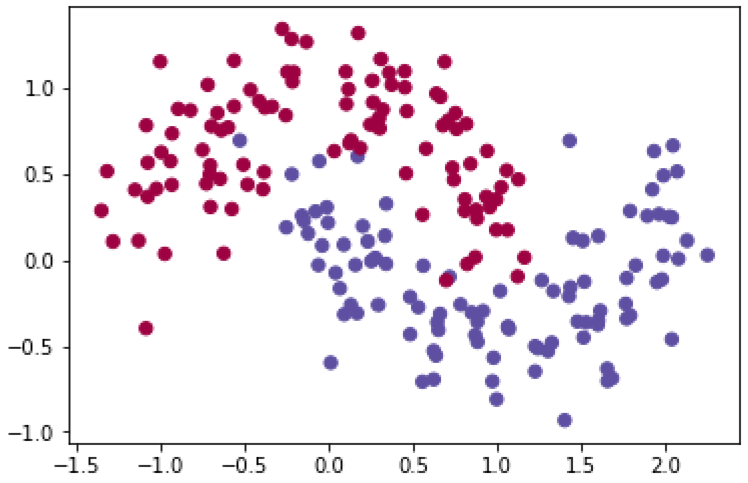
Data generation

In [2]:
```
# Package imports
import numpy as np
import matplotlib.pyplot as plt
import sklearn
import sklearn.datasets
import sklearn.linear_model
from sklearn.datasets import make_moons

%matplotlib inline
```

In [3]:
```
# Generate a dataset and plot it
np.random.seed(0)
m = 200
X, Y = sklearn.datasets.make_moons(m, noise=0.20)

#To comply with the equations.
X = X.T
Y = Y.reshape(1,m)
print(X.shape, Y.shape)

plt.scatter(X[0,:], X[1,:], s=40, c=np.squeeze(Y), cmap=plt.cm.Spectral) #plt.cm.Spectral
```

```
(2, 200) (1, 200)
```

Out[3]:
```
<matplotlib.collections.PathCollection at 0x1f02e10c278>
```



In [4]:
```
shape_X = X.shape
shape_Y = Y.shape
m = Y.shape[1]

print ('The shape of X is: ' + str(shape_X))
print ('The shape of Y is: ' + str(shape_Y))
print ('I have m = %d training examples!' % (m))
```

```
The shape of X is: (2, 200)
The shape of Y is: (1, 200)
I have m = 200 training examples!
```

Simple Logistic Regression

In [5]:
```
clf = sklearn.linear_model.LogisticRegressionCV()
clf.fit(X.T, np.squeeze(Y))
#Note the format for X in sklearn is (m,n), where m for number of samples and n for data dimension.
#Also I squeenze the unnecessady dimension for Y to eliminate warnings.
```

```
C:\Users\ljyan\Anaconda3\lib\site-packages\sklearn\model_selection\_split.py:2053: FutureWarning: You should sp
ecify a value for 'cv' instead of relying on the default value. The default value will change from 3 to 5 in ve
rsion 0.22.
  warnings.warn(CV_WARNING, FutureWarning)
```

Out[5]:
```
LogisticRegressionCV(Cs=10, class_weight=None, cv='warn', dual=False,
           fit_intercept=True, intercept_scaling=1.0, max_iter=100,
           multi_class='warn', n_jobs=None, penalty='l2',
           random_state=None, refit=True, scoring=None, solver='lbfgs',
           tol=0.0001, verbose=0)
```

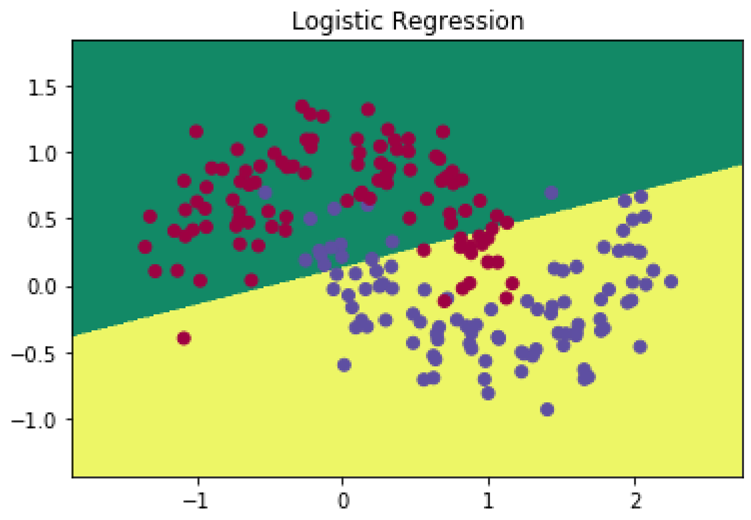
You can now plot the decision boundary of these models. Run the code below.

In [6]:
```python
def plot_decision_boundary(pred_func, X, Y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0,:].min() - .5, X[0,:].max() + .5
    y_min, y_max = X[1,:].min() - .5, X[1,:].max() + .5
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))

#      #The code below is for testing
#      print(xx.shape)
#      print(yy.shape)
#      print(xx.ravel().shape)
#      print(yy.ravel().shape)
#     Z_test = np.c_[xx.ravel(), yy.ravel()]
#      print(Z_test.shape)
#      print(Z_test)
#      #The bode above is for testing

    # Predict the function value for the whole gid
    Z = pred_func(np.c_[xx.ravel(), yy.ravel()])
#      print(Z.shape)
#      print(Z)
    Z = Z.reshape(xx.shape)
#      print(Z.shape)

    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap = 'summer') #cmap=plt.cm.Spectral
    plt.scatter(X[0,:], X[1,:], c=np.squeeze(Y), cmap=plt.cm.Spectral)
```

https://stackoverflow.com/questions/10894323/what-does-the-c-underscore-expression-c-do-exactly (https://stackoverflow.com/questions/10894323/what-does-the-c-underscore-expression-c-do-exactly) About np.c_ function, better explanation than the official document

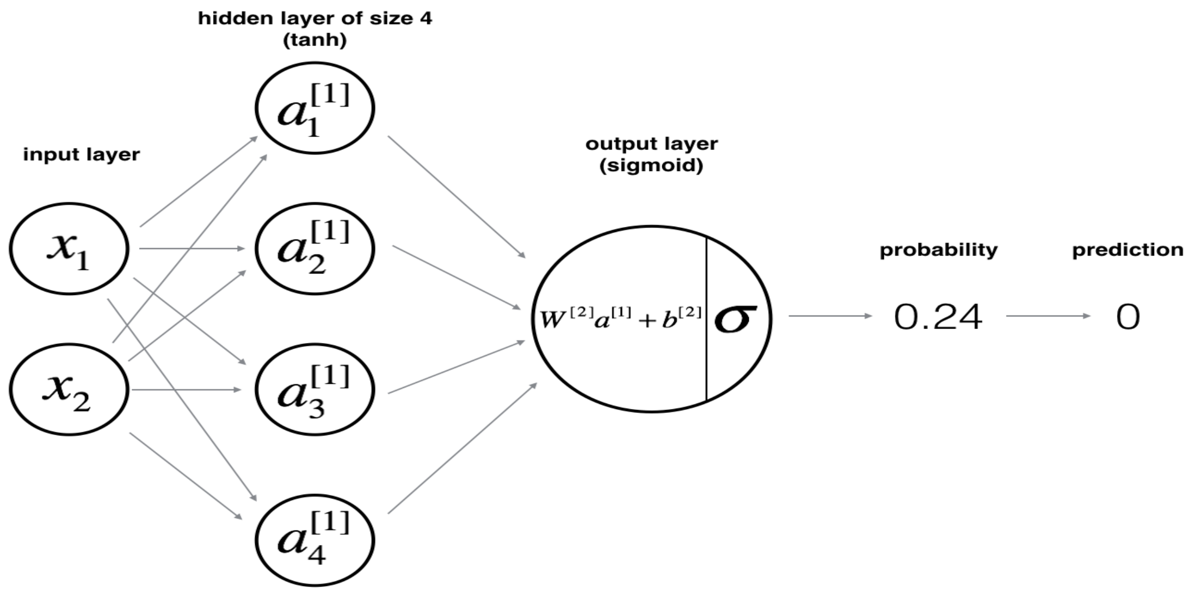
In [7]:
```
plot_decision_boundary(lambda x: clf.predict(x), X, Y)
plt.title("Logistic Regression")

LR_predictions = clf.predict(X.T)
print ('Accuracy of logistic regression: %d ' % float((np.dot(Y, LR_predictions) + np.dot(1 - Y,1 - LR_predictio
       '% ' + "(percentage of correctly labelled datapoints)")
```

```
Accuracy of logistic regression: 85 % (percentage of correctly labelled datapoints)
```



Neural Network model with a single hidden layer

Model and derivations



In [8]:
```
def layer_sizes(X, Y):

    n_x = X.shape[0] # size of input layer
    n_h = 4
    n_y = Y.shape[0] # size of output layer

    return (n_x, n_h, n_y)
```

In [9]:
```
#X_assess, Y_assess = layer_sizes_test_case()
(n_x, n_h, n_y) = layer_sizes(X, Y)
print("The size of the input layer is: n_x = " + str(n_x))
print("The size of the hidden layer is: n_h = " + str(n_h))
print("The size of the output layer is: n_y = " + str(n_y))
```

```
The size of the input layer is: n_x = 2
The size of the hidden layer is: n_h = 4
The size of the output layer is: n_y = 1
```

In [10]:
```
def initialize_parameters(n_x, n_h, n_y):

    np.random.seed(2) # we set up a seed so that your output matches ours although the initialization is random.

    W1 = np.random.randn(n_h, n_x) * 0.01
    b1 = np.zeros(shape=(n_h, 1))
    W2 = np.random.randn(n_y, n_h) * 0.01
    b2 = np.zeros(shape=(n_y, 1))

    assert (W1.shape == (n_h, n_x))
    assert (b1.shape == (n_h, 1))
    assert (W2.shape == (n_y, n_h))
    assert (b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

In [11]:
```
#n_x, n_h, n_y = initialize_parameters_test_case()

parameters = initialize_parameters(n_x, n_h, n_y)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-0.00416758 -0.00056267]
 [-0.02136196  0.01640271]
 [-0.01793436 -0.00841747]
 [ 0.00502881 -0.01245288]]
b1 = [[0.]
 [0.]
 [0.]
 [0.]]
W2 = [[-0.01057952 -0.00909008  0.00551454  0.02292208]]
b2 = [[0.]]
```

In [12]:
```
def sigmoid(z):

    s = 1 / (1 + np.exp(-z))

    return s
```

In [13]:
```
def forward_propagation(X, parameters):
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    # Implement Forward Propagation to calculate A2 (probabilities)
    Z1 = np.dot(W1, X) + b1
    A1 = np.tanh(Z1)
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)

    assert(A2.shape == (1, X.shape[1]))

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache
```

In [14]:
```
A2, cache = forward_propagation(X, parameters)

# Note: we use the mean here just to make sure that your output matches ours.
print(np.mean(cache['Z1']), np.mean(cache['A1']), np.mean(cache['Z2']), np.mean(cache['A2']))
```

```
-0.004994672287995899 -0.004993046451487895 7.548898600385004e-07 0.5000001887224258
```

In [15]:
```
def compute_cost(A2, Y, parameters):

    m = Y.shape[1] # number of example

    W1 = parameters['W1']
    W2 = parameters['W2']

    #np.multiply implement element-wise multiplication by default
    logprobs = np.multiply(np.log(A2), Y) + np.multiply((1 - Y), np.log(1 - A2))
    cost = - np.sum(logprobs) / m

    cost = np.squeeze(cost)     # makes sure cost is the dimension we expect.
                                # E.g., turns [[17]] into 17
    assert(isinstance(cost, float))

    return cost
```

In [16]:
```
#A2, Y_assess, parameters = compute_cost_test_case()

print("cost = " + str(compute_cost(A2, Y, parameters)))
```

```
cost = 0.692991677492394
```

To compute dZ1 you'll need to compute $g^{[1]'}(Z^{[1]})$. Since $g^{[1]}(.)$ is the tanh activation function, if $a = g^{[1]}(z)$ then $g^{[1]'}(z) = 1 - a^2$. So you can compute $g^{[1]'}(Z^{[1]})$ using `(1 - np.power(A1, 2))` .

In [17]:
```
def backward_propagation(parameters, cache, X, Y):
    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary "parameters".
    W1 = parameters['W1']
    W2 = parameters['W2']

    # Retrieve also A1 and A2 from dictionary "cache".
    A1 = cache['A1']
    A2 = cache['A2']

    # Backward propagation: calculate dW1, db1, dW2, db2.
    dZ2= A2 - Y
    dW2 = (1 / m) * np.dot(dZ2, A1.T)
    db2 = (1 / m) * np.sum(dZ2, axis=1, keepdims=True)
    dZ1 = np.multiply(np.dot(W2.T, dZ2), 1 - np.power(A1, 2))
    dW1 = (1 / m) * np.dot(dZ1, X.T)
    db1 = (1 / m) * np.sum(dZ1, axis=1, keepdims=True)

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
```

In [18]:
```
grads = backward_propagation(parameters, cache, X, Y)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dW2 = "+ str(grads["dW2"]))
print ("db2 = "+ str(grads["db2"]))
```

```
dW1 = [[ 0.00267914 -0.00202724]
 [ 0.0022986  -0.00174083]
 [-0.00139552  0.00105656]
 [-0.00580419  0.00439167]]
db1 = [[-4.61136122e-08]
 [-9.87203086e-07]
 [ 3.56507374e-07]
 [-1.12014058e-07]]
dW2 = [[ 0.0009476   0.00854968  0.00292786 -0.00365959]]
db2 = [[1.88722426e-07]]
```

In [19]:
```python
def update_parameters(parameters, grads, learning_rate=1.2):
    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    # Retrieve each gradient from the dictionary "grads"
    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']

    # Update rule for each parameter
    W1 = W1 - learning_rate * dW1
    b1 = b1 - learning_rate * db1
    W2 = W2 - learning_rate * dW2
    b2 = b2 - learning_rate * db2

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters
```

In [20]:
```
#parameters, grads = update_parameters_test_case()
parameters = update_parameters(parameters, grads)

print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-0.00738255  0.00187002]
 [-0.02412028  0.0184917 ]
 [-0.01625973 -0.00968534]
 [ 0.01199384 -0.01772289]]
b1 = [[ 5.53363347e-08]
 [ 1.18464370e-06]
 [-4.27808849e-07]
 [ 1.34416870e-07]]
W2 = [[-0.01171665 -0.0193497   0.00200111  0.02731359]]
b2 = [[-2.26466911e-07]]
```

Integrating the neural network model

```
In [21]: def nn_model(X, Y, n_h, num_iterations=10000, print_cost=False):

             np.random.seed(3)
             n_x = layer_sizes(X, Y)[0]
             n_y = layer_sizes(X, Y)[2]

             # Initialize parameters, then retrieve W1, b1, W2, b2. Inputs: "n_x, n_h, n_y". Outputs = "W1, b1, W2, b2, pa
             parameters = initialize_parameters(n_x, n_h, n_y)
             W1 = parameters['W1']
             b1 = parameters['b1']
             W2 = parameters['W2']
             b2 = parameters['b2']
             # Loop (gradient descent)

             for i in range(0, num_iterations):

                 # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
                 A2, cache = forward_propagation(X, parameters)

                 # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
                 cost = compute_cost(A2, Y, parameters)

                 # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grads".
                 grads = backward_propagation(parameters, cache, X, Y)

                 # Gradient descent parameter update. Inputs: "parameters, grads". Outputs: "parameters".
                 parameters = update_parameters(parameters, grads)

                 # Print the cost every 1000 iterations
                 if print_cost and i % 1000 == 0:
                     print ("Cost after iteration %i: %f" % (i, cost))

             return parameters
```

In [22]:
```
#X_assess, Y_assess = nn_model_test_case()

parameters = nn_model(X, Y, 4, num_iterations=10000, print_cost=False)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[-4.87116597  2.3674425 ]
 [-4.88904261  5.41157043]
 [-0.87387549 -6.82482148]
 [-3.59447394 -4.05446954]]
b1 = [[ 5.57653094]
 [-3.34443259]
 [ 2.25181883]
 [ 2.85656944]]
W2 = [[-13.12065561 -10.82635037  -7.37662091  13.03556626]]
b2 = [[1.55750878]]
```

Predictions

Question: Use your model to predict by building predict(). Use forward propagation to predict results.

Reminder: predictions = $y_{prediction} = \mathbb{1}\{\text{activation} > 0.5\} = \begin{cases} 1 & \text{if } activation > 0.5 \\ 0 & \text{otherwise} \end{cases}$

As an example, if you would like to set the entries of a matrix X to 0 and 1 based on a threshold you would do: `X_new = (X > threshold)`

In [23]:
```
def predict(parameters, X):

    # Computes probabilities using forward propagation, and classifies to 0/1 using 0.5 as the threshold.
    A2, cache = forward_propagation(X, parameters)
    predictions = np.round(A2)

    return predictions
```

In [24]:
```
#parameters, X_assess = predict_test_case()

predictions = predict(parameters, X)
print("predictions mean = " + str(np.mean(predictions)))
```

```
predictions mean = 0.5
```

It is time to run the model and see how it performs on a planar dataset. Run the following code to test your model with a single hidden layer of n_h hidden units.

In [25]:
```
# Build a model with a n_h-dimensional hidden layer
#parameters = nn_model(X, Y, n_h = 4, num_iterations=10000, print_cost=True)
parameters = nn_model(X, Y, n_h = 4, num_iterations=10000, print_cost=True)

# Plot the decision boundary
# plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
# plt.title("Decision Boundary for hidden layer size " + str(4))
```

```
Cost after iteration 0: 0.692992
Cost after iteration 1000: 0.130578
Cost after iteration 2000: 0.066981
Cost after iteration 3000: 0.060318
Cost after iteration 4000: 0.052363
Cost after iteration 5000: 0.047602
Cost after iteration 6000: 0.044696
Cost after iteration 7000: 0.042731
Cost after iteration 8000: 0.041307
Cost after iteration 9000: 0.040225
```

In [26]:
```
# Print accuracy
predictions = predict(parameters, X)
print ('Accuracy: %d' % float((np.dot(Y, predictions.T) + np.dot(1 - Y, 1 - predictions.T)) / float(Y.size) * 10
```

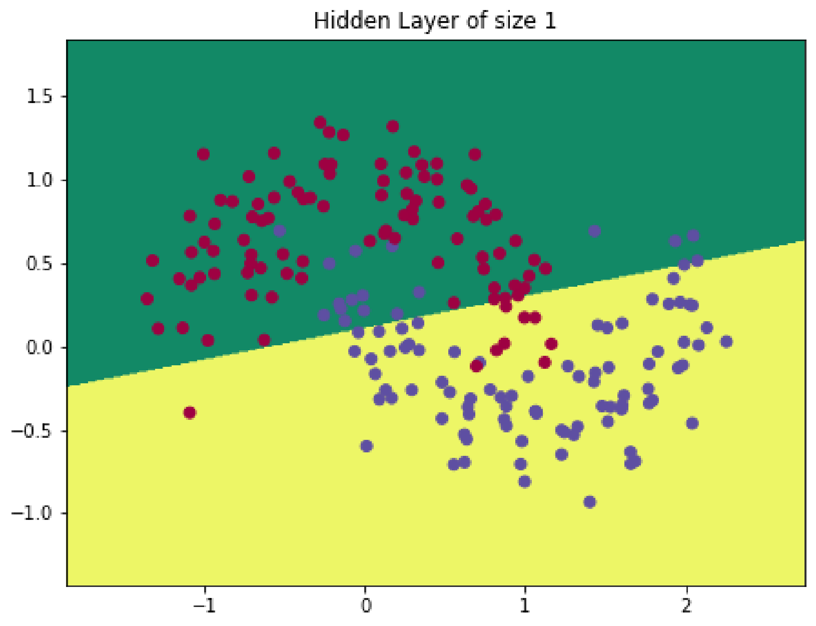
```
Accuracy: 99%
```

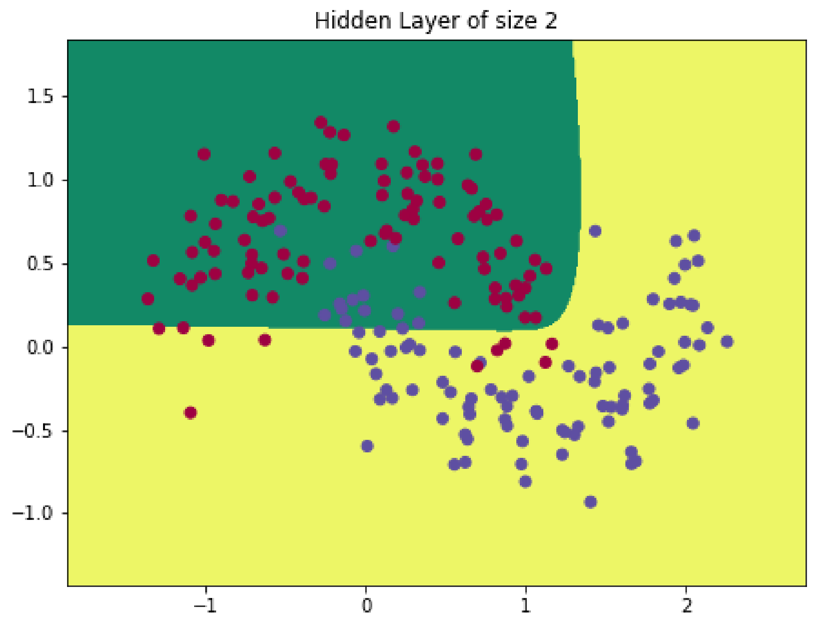
Tuning hidden layer size (optional/ungraded exercise)

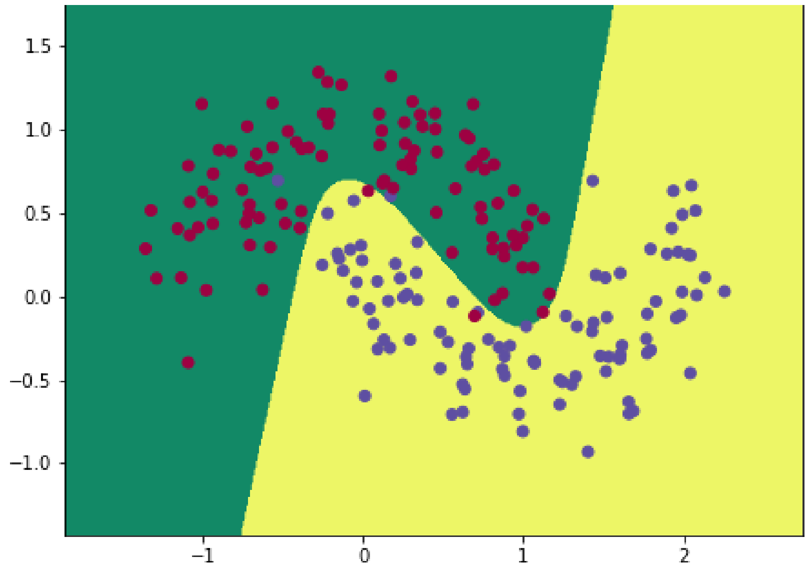
Run the following code. It may take 1-2 minutes. You will observe different behaviors of the model for various hidden layer sizes.

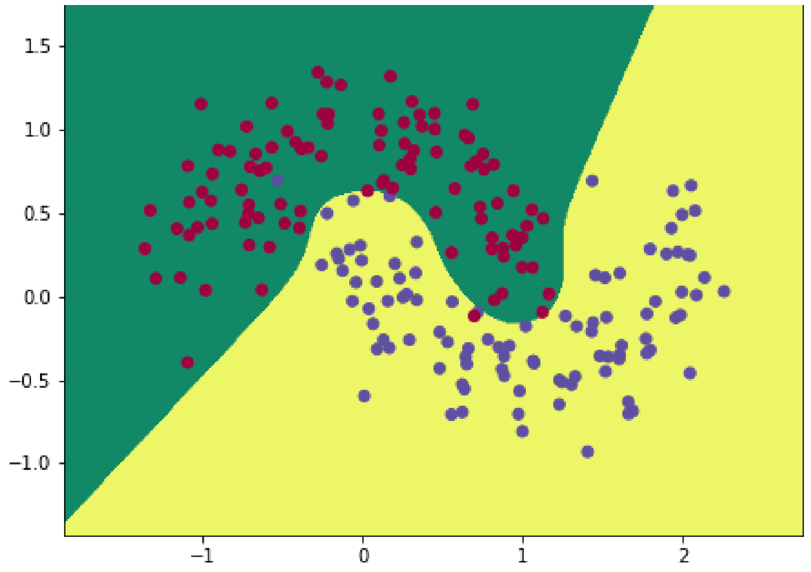
In [27]:
```
plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
for i, n_h in enumerate(hidden_layer_sizes):
    plt.subplot(5, 2, i + 1)
    plt.title('Hidden Layer of size %d' % n_h)
    parameters = nn_model(X, Y, n_h, num_iterations=5000)

    plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
    #plot_decision_boundary(lambda x: predict(nn_model, x))

    predictions = predict(parameters, X)
    accuracy = float((np.dot(Y, predictions.T) + np.dot(1 - Y, 1 - predictions.T)) / float(Y.size) * 100)
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

```
Accuracy for 1 hidden units: 86.5 %
Accuracy for 2 hidden units: 87.5 %
Accuracy for 3 hidden units: 97.5 %
Accuracy for 4 hidden units: 99.0 %
Accuracy for 5 hidden units: 98.0 %
Accuracy for 20 hidden units: 97.0 %
Accuracy for 50 hidden units: 98.0 %
```

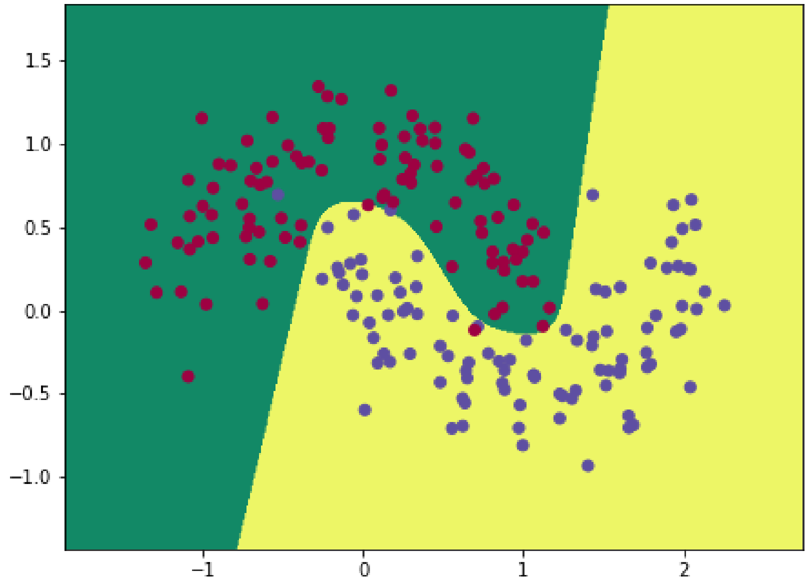




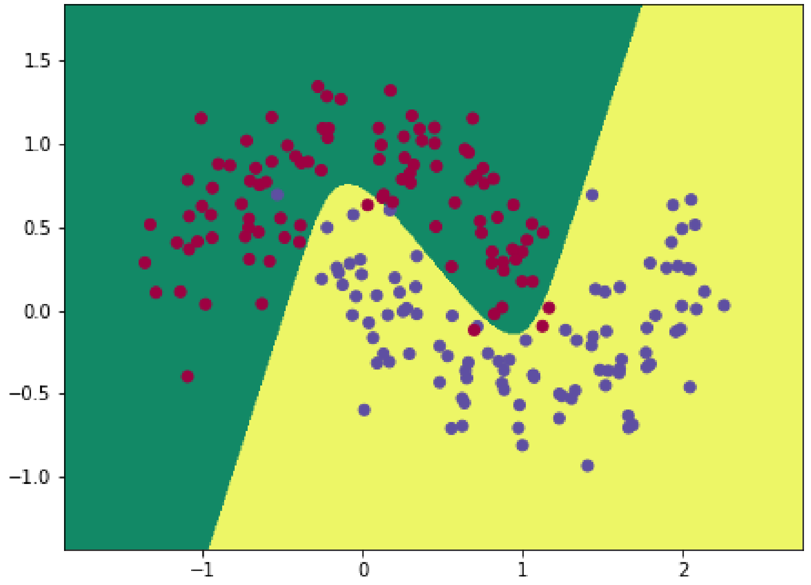




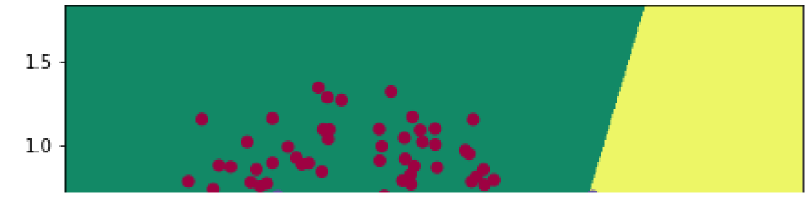
Hidden Layer of size 5

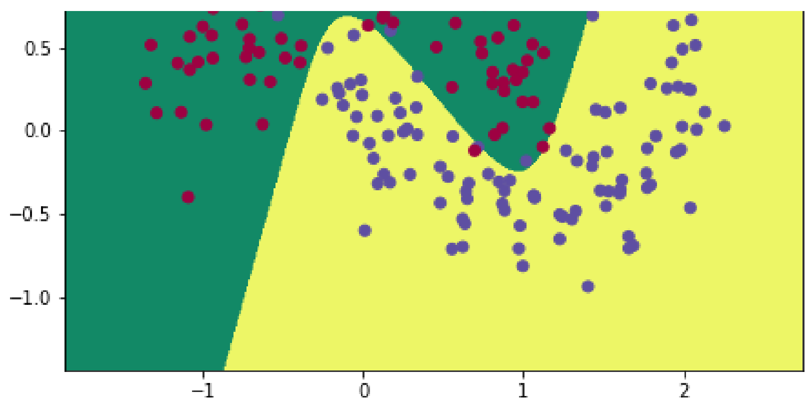


Hidden Layer of size 20



Hidden Layer of size 50





- The larger models (with more hidden units) are able to fit the training set better, until eventually the largest models overfit the data.
- The best hidden layer size seems to be around n_h = 5. Indeed, a value around here seems to fits the data well without also incurring noticable overfitting.