Planar data classification with one hidden layer v5-Copy1

September 25, 2018

1 Planar data classification with one hidden layer

let us build your first neural network, which will have a hidden layer. You will see a big difference between this model and the model using logistic regression.

You will see how to: - Implement a 2-class classification neural network with a single hidden layer - Use units with a non-linear activation function, such as tanh - Compute the cross entropy loss - Implement forward and backward propagation

1.1 1 - Packages

Let's first import all the packages that you will need during this assignment. - numpy is the fundamental package for scientific computing with Python. - sklearn provides simple and efficient tools for data mining and data analysis. - matplotlib is a library for plotting graphs in Python. - testCases provides some test examples to assess the correctness of your functions - planar_utils provide various useful functions used in this assignment

```
In [1]: # Package imports
        import numpy as np
        import matplotlib.pyplot as plt
        from testCases_v2 import *
        import sklearn
        import sklearn.datasets
        import sklearn.linear_model
        from planar_utils import plot_decision_boundary, sigmoid, load_planar_datas

        %matplotlib inline

        np.random.seed(1) # set a seed so that the results are consistent
```

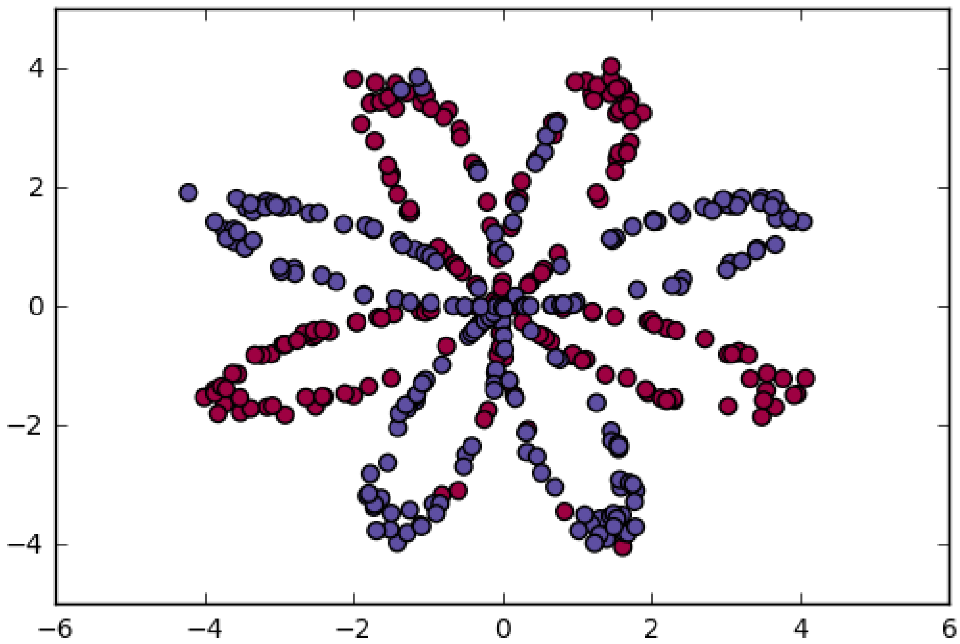
1.2 2 - Dataset

First, let's get the dataset we will work on. The following code will load a "flower" 2-class dataset into variables `X` and `Y`.

```
In [2]: X, Y = load_planar_dataset()
```

Visualize the dataset using matplotlib. The data looks like a "flower" with some red (label y=0) and some blue (y=1) points. Your goal is to build a model to fit this data.

```
In [3]: # Visualize the data:
        plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);
```



we have: - a numpy-array (matrix) X that contains your features (x1, x2) - a numpy-array (vector) Y that contains your labels (red:0, blue:1).

Lets first get a better sense of what our data is like.

Let us see how many training examples do you have? In addition, what is the `shape` of the variables `X` and `Y`?

```
In [4]: shape_X = X.shape
        shape_Y = Y.shape
        m = Y.shape[1] # training set size

        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, 400)
The shape of Y is: (1, 400)
I have m = 400 training examples!
```

1.3 3 - Simple Logistic Regression

Before building a full neural network, lets first see how logistic regression performs on this problem. You can use sklearn's built-in functions to do that. Run the code below to train a logistic regression classifier on the dataset.

```
In [5]: # Train the logistic regression classifier
        clf = sklearn.linear_model.LogisticRegressionCV();
        clf.fit(X.T, Y.T);
```

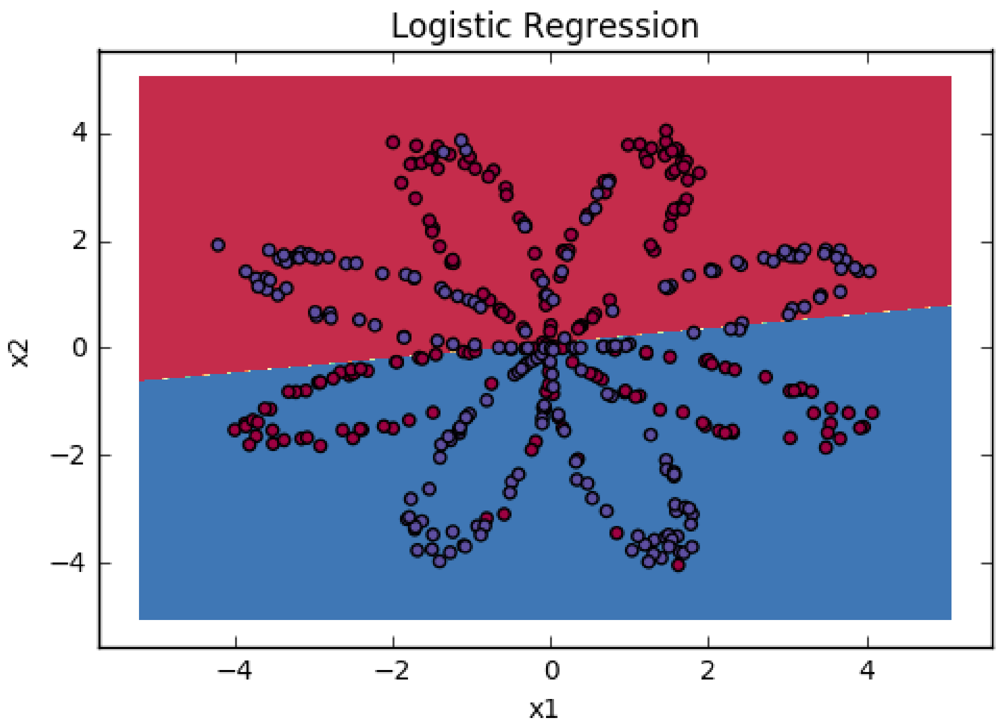
```
/opt/conda/lib/python3.5/site-packages/sklearn/utils/validation.py:515: DataConvers
  y = column_or_1d(y, warn=True)
```

we can now plot the decision boundary of these models with the help of the code below.

```
In [6]: # Plot the decision boundary for logistic regression
        plot_decision_boundary(lambda x: clf.predict(x), X, Y)
        plt.title("Logistic Regression")

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

```
Accuracy of logistic regression: 47 % (percentage of correctly labelled datapoints)
```



Expected Output:
Accuracy
47%

Interpretation: The dataset is not linearly separable, so logistic regression doesn't perform well. Hopefully a neural network will do better. Let's try this now!

1.4 4 - Neural Network model

Logistic regression did not work well on the "flower dataset". You are going to train a Neural Network with a single hidden layer.

Here is our model:

Mathematically:

For one example $x^{(i)}$:

$$z^{[1](i)} = W^{[1]}x^{(i)} + b^{[1]} \tag{1}$$

$$a^{[1](i)} = \tanh(z^{[1](i)}) \tag{2}$$

$$z^{[2](i)} = W^{[2]}a^{[1](i)} + b^{[2]} \tag{3}$$

$$\hat{y}^{(i)} = a^{[2](i)} = \sigma(z^{[2](i)}) \tag{4}$$

$$y^{(i)}_{prediction} = \begin{cases} 1 & \text{if } a^{[2](i)} > 0.5 \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

Given the predictions on all the examples, you can also compute the cost J as follows:

$$J = -\frac{1}{m}\sum_{i=0}^{m}\left(y^{(i)}\log\left(a^{[2](i)}\right) + (1-y^{(i)})\log\left(1-a^{[2](i)}\right)\right) \tag{6}$$

Reminder: The general methodology to build a Neural Network is to: 1. Define the neural network structure (# of input units, # of hidden units, etc). 2. Initialize the model's parameters 3. Loop: - Implement forward propagation - Compute loss - Implement backward propagation to get the gradients - Update parameters (gradient descent)

You often build helper functions to compute steps 1-3 and then merge them into one function we call `nn_model()`. Once you've built `nn_model()` and learnt the right parameters, you can make predictions on new data.

1.4.1 4.1 - Defining the neural network structure

Define three variables: - n_x: the size of the input layer - n_h: the size of the hidden layer (set this to 4) - n_y: the size of the output layer

```
In [7]: def layer_sizes(X, Y):
            """
            Arguments:
            X -- input dataset of shape (input size, number of examples)
            Y -- labels of shape (output size, number of examples)

            Returns:
            n_x -- the size of the input layer
            n_h -- the size of the hidden layer
            n_y -- the size of the output layer
            """

            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 [8]: X_assess, Y_assess = layer_sizes_test_case()
        (n_x, n_h, n_y) = layer_sizes(X_assess, Y_assess)
        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 = 5
The size of the hidden layer is: n_h = 4
The size of the output layer is: n_y = 2
```

Expected Output (these are not the sizes you will use for your network, they are just used to assess the function you've just coded).

n_x
5

<tr>
<td>**n_h**</td>
<td> 4 </td>

<tr>
<td>**n_y**</td>
<td> 2 </td>

1.4.2 4.2 - Initialize the model's parameters

Implement the function `initialize_parameters()`.

- We will initialize the weights matrices with random values.
- We will initialize the bias vectors as zeros.

```
In [9]: def initialize_parameters(n_x, n_h, n_y):
            """
            Argument:
            n_x -- size of the input layer
            n_h -- size of the hidden layer
            n_y -- size of the output layer

            Returns:
            params -- python dictionary containing your parameters:
                            W1 -- weight matrix of shape (n_h, n_x)
                            b1 -- bias vector of shape (n_h, 1)
                            W2 -- weight matrix of shape (n_y, n_h)
                            b2 -- bias vector of shape (n_y, 1)
            """

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

            W1 = np.random.randn(n_h,n_x)*0.01
```

```
    b1 = np.zeros((n_h,1))
    W2 = np.random.randn(n_y,n_h)
    b2 = np.zeros((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 [10]: 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 = [[-1.05795222 -0.90900761  0.55145404  2.29220801]]
b2 = [[ 0.]]
```

1.4.3 4.3 - The Loop

Now we will Implement `forward_propagation()`.

- We can use the function `sigmoid()`. It is built-in (imported) in the notebook.
- We can use the function `np.tanh()`. It is part of the numpy library.
- The steps we have to implement are:
  1. Retrieve each parameter from the dictionary "parameters" (which is the output of `initialize_parameters()`) by using `parameters[".."]`.
  2. Implement Forward Propagation. Compute $Z^{[1]}, A^{[1]}, Z^{[2]}$ and $A^{[2]}$ (the vector of all your predictions on all the examples in the training set).

- Values needed in the backpropagation are stored in "`cache`". The `cache` will be given as an input to the backpropagation function.

```
In [11]: def forward_propagation(X, parameters):
             """
             Argument:
             X -- input data of size (n_x, m)
             parameters -- python dictionary containing your parameters (output of

             Returns:
             A2 -- The sigmoid output of the second activation
             cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
             """
             # Retrieving each parameter from the dictionary "parameters"
             W1 = parameters['W1']
             b1 = parameters['b1']
             W2 = parameters['W2']
             b2 = parameters['b2']

             # Implementing 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 [12]: X_assess, parameters = forward_propagation_test_case()
         A2, cache = forward_propagation(X_assess, parameters)

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

```
0.262818640198 0.091999045227 -1.30766601287 0.212877681719
```

Now that we have computed $A^{[2]}$ (in the Python variable "`A2`"), which contains $a^{[2](i)}$ for every example, you can compute the cost function as follows:

$$J = -\frac{1}{m} \sum_{i=0}^{m} (y^{(i)} \log\left(a^{[2](i)}\right) + (1-y^{(i)}) \log\left(1-a^{[2](i)}\right)) \tag{13}$$

Exercise: Implement `compute_cost()` to compute the value of the cost J.

Instructions: - There are many ways to implement the cross-entropy loss. To help you, we give you how we would have implemented $-\sum_{i=0}^{m} y^{(i)} \log(a^{[2](i)})$:

```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)                # no need to use a for loop!
```

(we can use either `np.multiply()` and then `np.sum()` or directly `np.dot()`).

```
In [13]: def compute_cost(A2, Y, parameters):
             """
             Computes the cross-entropy cost given in equation (13)

             Arguments:
             A2 -- The sigmoid output of the second activation, of shape (1, number
             Y -- "true" labels vector of shape (1, number of examples)
             parameters -- python dictionary containing your parameters W1, b1, W2

             Returns:
             cost -- cross-entropy cost given equation (13)
             """

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

             # Compute the cross-entropy cost
             logprobs = np.multiply(np.log(A2),Y)+np.multiply(np.log(1-A2),(1-Y))
             cost = - np.sum(logprobs)/m

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

             return cost

In [14]: A2, Y_assess, parameters = compute_cost_test_case()

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

cost = 0.693058761039
```

Using the cache computed during forward propagation, you can now implement backward propagation.

Question: Implement the function `backward_propagation()`.

- Tips:
  - 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 [15]:
```
def backward_propagation(parameters, cache, X, Y):
    """
    Implement the backward propagation using the instructions above.

    Arguments:
    parameters -- python dictionary containing our parameters
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
    X -- input data of shape (2, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

    Returns:
    grads -- python dictionary containing your gradients with respect to d
    """
    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 [16]:
```
parameters, cache, X_assess, Y_assess = backward_propagation_test_case()

grads = backward_propagation(parameters, cache, X_assess, Y_assess)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dW2 = "+ str(grads["dW2"]))
print ("db2 = "+ str(grads["db2"]))
```

```
dW1 = [[ 0.00301023 -0.00747267]
 [ 0.00257968 -0.00641288]
```

```
 [-0.00156892  0.003893  ]
 [-0.00652037  0.01618243]]
db1 = [[ 0.00176201]
 [ 0.00150995]
 [-0.00091736]
 [-0.00381422]]
dW2 = [[ 0.00078841  0.01765429 -0.00084166 -0.01022527]]
db2 = [[-0.16655712]]
```

In [17]:
```
def update_parameters(parameters, grads, learning_rate = 1.2):
    """
    Updates parameters using the gradient descent update rule given above

    Arguments:
    parameters -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients

    Returns:
    parameters -- python dictionary containing your updated parameters
    """
    # 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 [18]:
```
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 = [[-0.00643025  0.01936718]
 [-0.02410458  0.03978052]
 [-0.01653973 -0.02096177]
 [ 0.01046864 -0.05990141]]
b1 = [[ -1.02420756e-06]
 [  1.27373948e-05]
 [  8.32996807e-07]
 [ -3.20136836e-06]]
W2 = [[-0.01041081 -0.04463285  0.01758031  0.04747113]]
b2 = [[ 0.00010457]]
```

1.4.4 4.4 - Integrating parts 4.1, 4.2 and 4.3 in nn_model()

```
In [19]: def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):
             """
             Arguments:
             X -- dataset of shape (2, number of examples)
             Y -- labels of shape (1, number of examples)
             n_h -- size of the hidden layer
             num_iterations -- Number of iterations in gradient descent loop
             print_cost -- if True, print the cost every 1000 iterations

             Returns:
             parameters -- parameters learnt by the model. They can then be used to
             """

             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
             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, cach
        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: "gr
        grads = backward_propagation(parameters, cache, X, Y)

        # Gradient descent parameter update. Inputs: "parameters, grads".
        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 [20]:
```
X_assess, Y_assess = nn_model_test_case()
parameters = nn_model(X_assess, Y_assess, 4, num_iterations=10000, print_c
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
Cost after iteration 0: 0.653216
Cost after iteration 1000: 0.000214
Cost after iteration 2000: 0.000107
Cost after iteration 3000: 0.000071
Cost after iteration 4000: 0.000053
Cost after iteration 5000: 0.000042
Cost after iteration 6000: 0.000035
Cost after iteration 7000: 0.000030
Cost after iteration 8000: 0.000026
Cost after iteration 9000: 0.000023
W1 = [[-0.67217213  1.33737914]
 [-0.65598488  1.27247072]
 [ 0.55737732 -1.09668721]
 [ 0.90635375 -2.08077263]]
b1 = [[ 0.15253561]
 [ 0.16843239]
 [-0.1741792 ]
 [ 0.19390201]]
W2 = [[-2.64738303 -2.48746615  1.94802517  4.02291264]]
b2 = [[ 0.06638778]]
```

1.4.5 4.5 Predictions

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

```
In [21]: def predict(parameters, X):
             """
             Using the learned parameters, predicts a class for each example in X

             Arguments:
             parameters -- python dictionary containing your parameters
             X -- input data of size (n_x, m)

             Returns
             predictions -- vector of predictions of our model (red: 0 / blue: 1)
             """

             # Computes probabilities using forward propagation, and classifies to
             A2, cache = forward_propagation(X, parameters)
             predictions = np.round(A2)

             return predictions
```

```
In [22]: parameters, X_assess = predict_test_case()

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

```
predictions mean = 0.666666666667
```

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

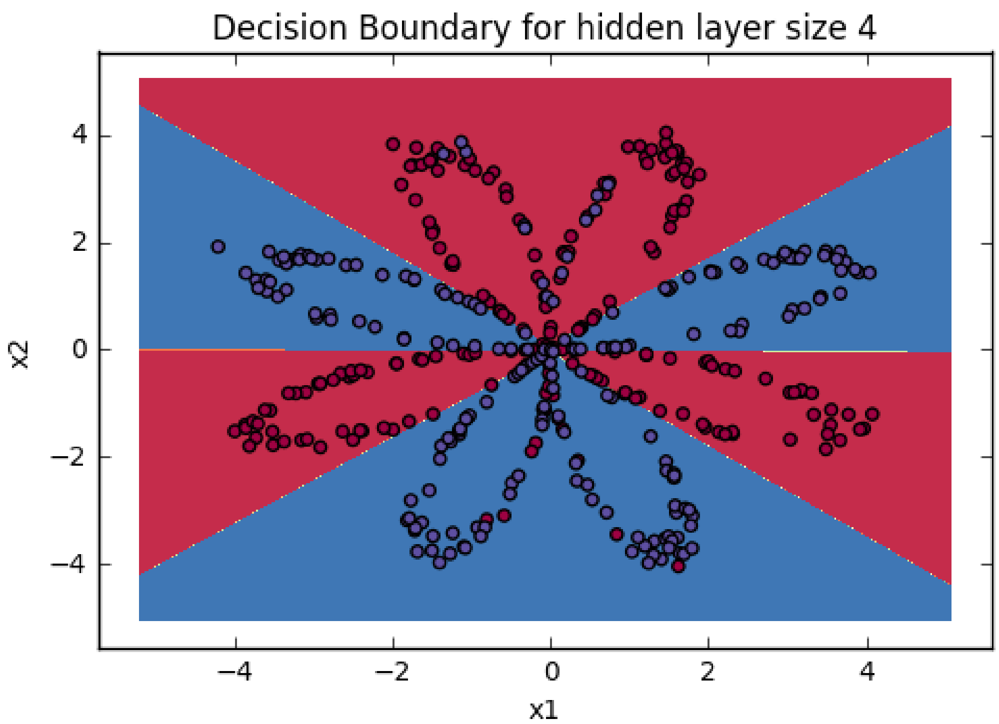
```
In [23]: # Build a model with a n_h-dimensional hidden layer
         parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000, print_cost=Tr

         # 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: 0.684638
Cost after iteration 1000: 0.280383
Cost after iteration 2000: 0.268529
Cost after iteration 3000: 0.261708
Cost after iteration 4000: 0.257236
Cost after iteration 5000: 0.254139
Cost after iteration 6000: 0.251894
Cost after iteration 7000: 0.250204
```

```
Cost after iteration 8000: 0.248892
Cost after iteration 9000: 0.247846
```

```
Out[23]: <matplotlib.text.Text at 0x7f7c1113df98>
```




```
In [24]: # Print accuracy
         predictions = predict(parameters, X)
         print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-prec
```

```
Accuracy: 91%
```

Accuracy is really high compared to Logistic Regression. The model has learnt the leaf patterns of the flower! Neural networks are able to learn even highly non-linear decision boundaries, unlike logistic regression.

Now, let's try out several hidden layer sizes.

1.4.6 4.6 - Tuning hidden layer size

We will observe different behaviors of the model for various hidden layer sizes.

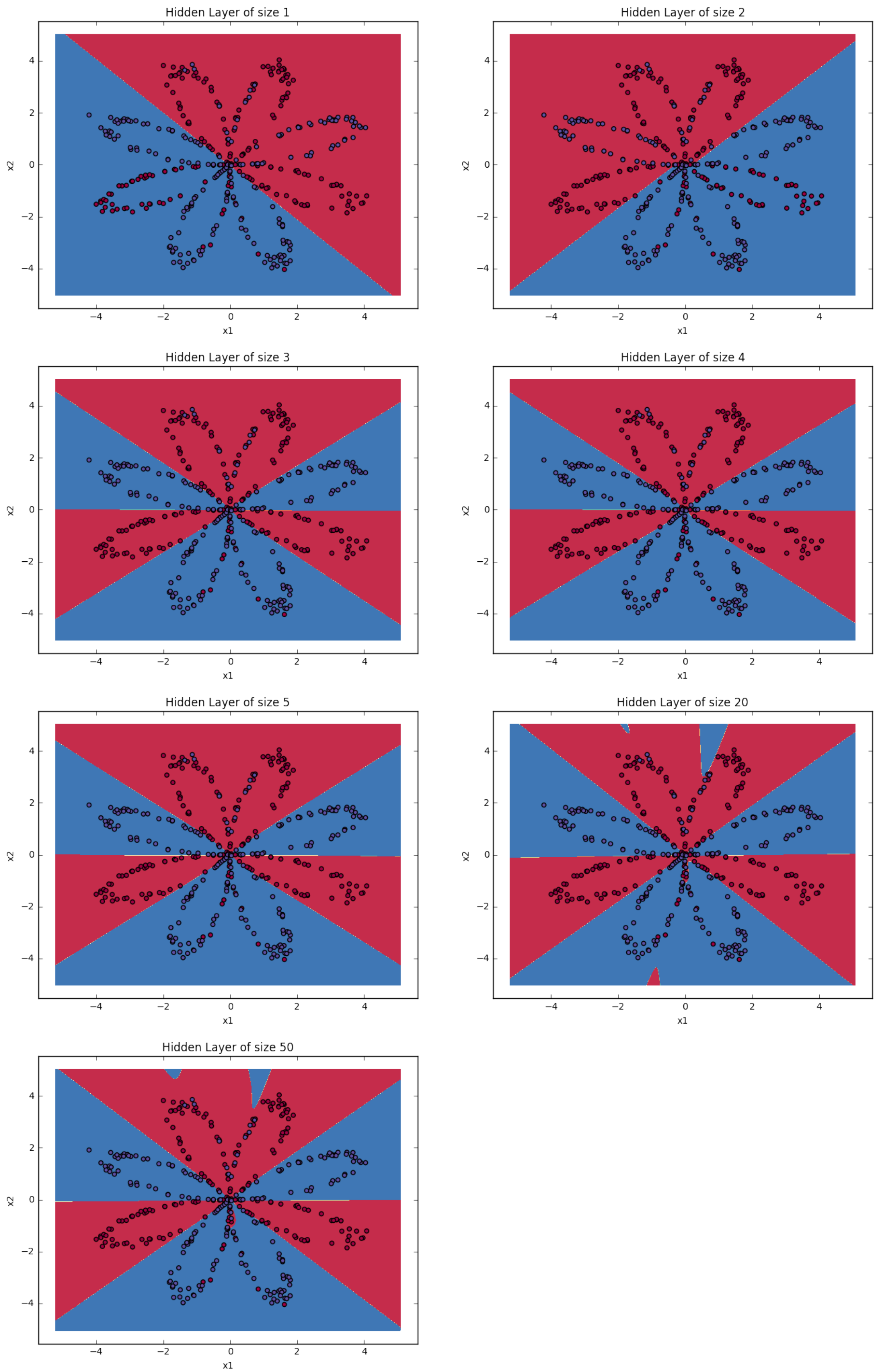
```
In [25]: plt.figure(figsize=(16, 32))
         hidden_layer_sizes = [1, 2, 3, 4, 5, 20, 50]
```

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

```
Accuracy for 1 hidden units: 66.25 %
Accuracy for 2 hidden units: 67.25 %
Accuracy for 3 hidden units: 90.75 %
Accuracy for 4 hidden units: 90.75 %
Accuracy for 5 hidden units: 91.0 %
Accuracy for 20 hidden units: 90.5 %
```

```
/opt/conda/lib/python3.5/site-packages/ipykernel/__main__.py:17: RuntimeWarning: di
/opt/conda/lib/python3.5/site-packages/ipykernel/__main__.py:17: RuntimeWarning: in
```

```
Accuracy for 50 hidden units: 90.75 %
```


Hidden Layer of size 1
Hidden Layer of size 2
Hidden Layer of size 3
Hidden Layer of size 4
Hidden Layer of size 5
Hidden Layer of size 20
Hidden Layer of size 50
x1
x2
−4
−2
0
2
4

Interpretation: - 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.

We saw to: - Build a complete neural network with a hidden layer - Make a good use of a non-linear unit - Implemented forward propagation and backpropagation, and trained a neural network - See the impact of varying the hidden layer size, including overfitting.