# Planar data classification with one hidden layer

Welcome to your week 3 programming assignment! It's time to build your first neural network, which will have one hidden layer. Now, you'll notice a big difference between this model and the one you implemented previously using logistic regression.

By the end of this assignment, you'll be able 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

## Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any *extra* `print` statement(s) in the assignment.
2. You have not added any *extra* code cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating *extra* variables.

If you do any of the following, you will get something like, `Grader Error: Grader feedback not found` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [instructions (https://www.coursera.org/learn/neural-networks-deep-learning/supplement/xLEC1/optional-downloading-your-notebook-downloading-your-workspace-and-refreshing)].

## Table of Contents



# 1 - Packages

First import all the packages that you will need during this assignment.

- numpy (https://www.numpy.org/) is the fundamental package for scientific computing with Python.
- sklearn (http://scikit-learn.org/stable/) provides simple and efficient tools for data mining and data analysis.
- matplotlib (http://matplotlib.org) 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 [ ]:

```
### v3.0
```

In [34]:

```
# Package imports
import numpy as np
import copy
import matplotlib.pyplot as plt
from testCases_v2 import *
from public_tests import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
from planar_utils import plot_decision_boundary, sigmoid, load_planar_dataset, load_extra_datasets

%matplotlib inline

%load_ext autoreload
%autoreload 2
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

## 2 - Load the Dataset

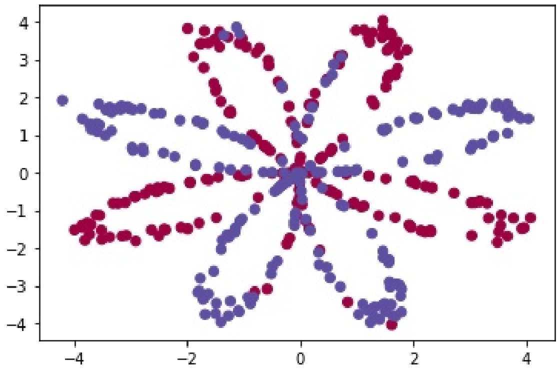
In [35]:

```
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 other words, we want the classifier to define regions as either red or blue.

In [36]:

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




You 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).
```

First, get a better sense of what your data is like.

### Exercise 1

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

**Hint**: How do you get the shape of a numpy array? [(help) (https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html)](https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.shape.html)

In [1]:

```
# (≈ 3 lines of code)
# shape_X = ...
# shape_Y = ...
# training set size
# m = ...
# YOUR CODE STARTS HERE


# YOUR CODE ENDS HERE

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))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-77bbc0834cb8> in <module>
      9 # YOUR CODE ENDS HERE
     10 
---> 11 print ('The shape of X is: ' + str(shape_X))
     12 print ('The shape of Y is: ' + str(shape_Y))
     13 print ('I have m = %d training examples!' % (m))

NameError: name 'shape_X' is not defined
```

**Expected Output**:

| | |
|---|---|
| shape of X | (2, 400) |
| shape of Y | (1, 400) |
| m | 400 |

## 3 - Simple Logistic Regression

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

In [39]:

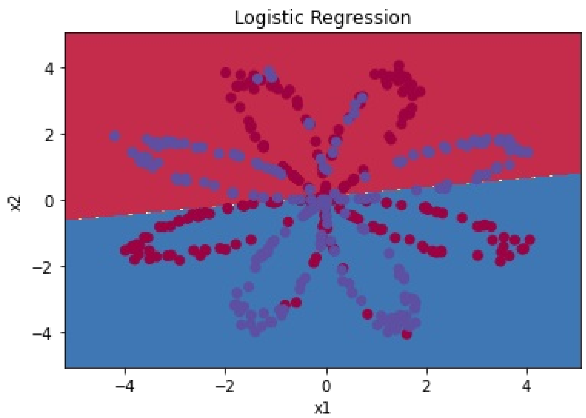
```
# Train the logistic regression classifier
clf = sklearn.linear_model.LogisticRegressionCV();
clf.fit(X.T, Y.T);
```

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

In [40]:

```
# 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_predictions) + np.dot(1-Y,1-LR_predictions))/f
loat(Y.size)*100) +
       '% ' + "(percentage of correctly labelled datapoints)")
```

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



**Expected Output**:

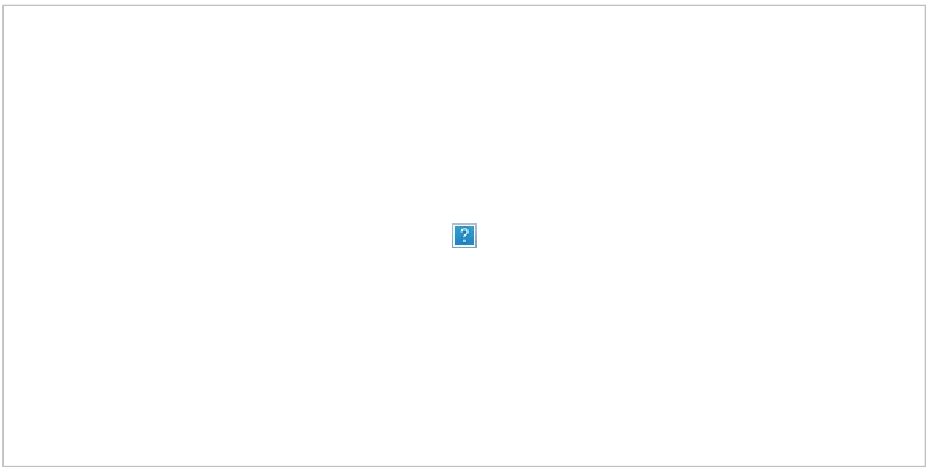
| | |
|---|---|
| Accuracy | 47% |

**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!

## 4 - Neural Network model

Logistic regression didn't work well on the flower dataset. Next, you're going to train a Neural Network with a single hidden layer and see how that handles the same problem.

**The model**:



**Mathematically**:

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

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

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

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

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

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

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

$$J = - \frac{1}{m} \sum\limits_{i = 0}^{m} \large\left(\small y^{(i)}\log\left(a^{[2] (i)}\right) + (1-y^{(i)})\log\left(1- a^{[2] (i)}\right) \large \right) \small$$

**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)
```

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

### 4.1 - Defining the neural network structure

### Exercise 2 - layer_sizes

Define three variables:

- n_x: the size of the input layer
- n_h: the size of the hidden layer (**set this to 4, as `n_h = 4`, but only for this Exercise 2**)
- n_y: the size of the output layer

**Hint**: Use shapes of X and Y to find n_x and n_y. Also, hard code the hidden layer size to be 4.

In [17]:

```
# GRADED FUNCTION: layer_sizes

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
    """
    #(≈ 3 lines of code)
    # n_x = ...
    # n_h = ...
    # n_y = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE
    return (n_x, n_h, n_y)
```

In [3]:

```
t_X, t_Y = layer_sizes_test_case()
(n_x, n_h, n_y) = layer_sizes(t_X, t_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))

layer_sizes_test(layer_sizes)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-a625d2d64083> in <module>
----> 1 t_X, t_Y = layer_sizes_test_case()
      2 (n_x, n_h, n_y) = layer_sizes(t_X, t_Y)
      3 print("The size of the input layer is: n_x = " + str(n_x))
      4 print("The size of the hidden layer is: n_h = " + str(n_h))
      5 print("The size of the output layer is: n_y = " + str(n_y))

NameError: name 'layer_sizes_test_case' is not defined
```

***Expected output***

```
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
All tests passed!
```

### 4.2 - Initialize the model's parameters

### Exercise 3 - initialize_parameters

Implement the function `initialize_parameters()`.

**Instructions**:

- Make sure your parameters' sizes are right. Refer to the neural network figure above if needed.
- You will initialize the weights matrices with random values.
    - Use: `np.random.randn(a,b) * 0.01` to randomly initialize a matrix of shape (a,b).
- You will initialize the bias vectors as zeros.
    - Use: `np.zeros((a,b))` to initialize a matrix of shape (a,b) with zeros.

In [18]:

```
# GRADED FUNCTION: initialize_parameters

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)
    """
    #(≈ 4 lines of code)
    # W1 = ...
    # b1 = ...
    # W2 = ...
    # b2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

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

    return parameters
```

In [19]:

```
np.random.seed(2)
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"]))

initialize_parameters_test(initialize_parameters)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-19-0eb4c3a6d62e> in <module>
      1 np.random.seed(2)
      2 n_x, n_h, n_y = initialize_parameters_test_case()
----> 3 parameters = initialize_parameters(n_x, n_h, n_y)
      4 
      5 print("W1 = " + str(parameters["W1"]))

<ipython-input-18-76ef0a3c0977> in initialize_parameters(n_x, n_h, n_y)
     25     # YOUR CODE ENDS HERE
     26 
---> 27     parameters = {"W1": W1,
     28                   "b1": b1,
     29                   "W2": W2,

NameError: name 'W1' is not defined
```

**Expected output**

```
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.]]
All tests passed!
```

### 4.3 - The Loop

### Exercise 4 - forward_propagation

Implement `forward_propagation()` using the following equations:

$$Z^{[1]} = W^{[1]}X + b^{[1]}$$

$$A^{[1]} = \tanh(Z^{[1]})$$

$$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$$

$$\hat{Y} = A^{[2]} = \sigma(Z^{[2]})$$

**Instructions**:

- Check the mathematical representation of your classifier in the figure above.
- Use the function `sigmoid()`. It's built into (imported) this notebook.
- Use the function `np.tanh()`. It's part of the numpy library.
- Implement using these steps:
  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 [20]:

```
# GRADED FUNCTION:forward_propagation

def forward_propagation(X, parameters):
    """
    Argument:
    X -- input data of size (n_x, m)
    parameters -- python dictionary containing your parameters (output of initialization function)

    Returns:
    A2 -- The sigmoid output of the second activation
    cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"
    """
    # Retrieve each parameter from the dictionary "parameters"
    #(≈ 4 lines of code)
    # W1 = ...
    # b1 = ...
    # W2 = ...
    # b2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

    # Implement Forward Propagation to calculate A2 (probabilities)
    # (≈ 4 lines of code)
    # Z1 = ...
    # A1 = ...
    # Z2 = ...
    # A2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

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

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

    return A2, cache
```

In [21]:

```
t_X, parameters = forward_propagation_test_case()
A2, cache = forward_propagation(t_X, parameters)
print("A2 = " + str(A2))

forward_propagation_test(forward_propagation)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-21-e41f1de7d764> in <module>
      1 t_X, parameters = forward_propagation_test_case()
----> 2 A2, cache = forward_propagation(t_X, parameters)
      3 print("A2 = " + str(A2))
      4 
      5 forward_propagation_test(forward_propagation)

<ipython-input-20-ee74ee6ac54e> in forward_propagation(X, parameters)
     33     # YOUR CODE ENDS HERE
     34 
---> 35     assert(A2.shape == (1, X.shape[1]))
     36 
     37     cache = {"Z1": Z1,

NameError: name 'A2' is not defined
```

***Expected output***

```
A2 = [[0.21292656 0.21274673 0.21295976]]
All tests passed!
```

## 4.4 - Compute the Cost

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

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

### Exercise 5 - compute_cost

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

**Instructions**:

- There are many ways to implement the cross-entropy loss. This is one way to implement one part of the equation without for loops: $-\sum\limits_{i=1}^{m} y^{(i)}\log(a^{[2](i)})$:

```
logprobs = np.multiply(np.log(A2),Y)
cost = - np.sum(logprobs)
```

- Use that to build the whole expression of the cost function.

**Notes**:

- You can use either `np.multiply()` and then `np.sum()` or directly `np.dot()` ).
- If you use `np.multiply` followed by `np.sum` the end result will be a type `float` , whereas if you use `np.dot` , the result will be a 2D numpy array.
- You can use `np.squeeze()` to remove redundant dimensions (in the case of single float, this will be reduced to a zero-dimension array).
- You can also cast the array as a type `float` using `float()` .

In [22]:

```
# GRADED FUNCTION: compute_cost

def compute_cost(A2, Y):
    """
    Computes the cross-entropy cost given in equation (13)

    Arguments:
    A2 -- The sigmoid output of the second activation, of shape (1, number of examples)
    Y -- "true" labels vector of shape (1, number of examples)

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

    """

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

    # Compute the cross-entropy cost
    # (≈ 2 lines of code)
    # logprobs = ...
    # cost = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

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

    return cost
```

In [23]:

```
A2, t_Y = compute_cost_test_case()
cost = compute_cost(A2, t_Y)
print("cost = " + str(compute_cost(A2, t_Y)))

compute_cost_test(compute_cost)
```

```
---------------------------------------------------------------------------
UnboundLocalError                         Traceback (most recent call last)
<ipython-input-23-9523b300d7a7> in <module>
      1 A2, t_Y = compute_cost_test_case()
----> 2 cost = compute_cost(A2, t_Y)
      3 print("cost = " + str(compute_cost(A2, t_Y)))
      4 
      5 compute_cost_test(compute_cost)

<ipython-input-22-feec0108924b> in compute_cost(A2, Y)
     25     # YOUR CODE ENDS HERE
     26 
---> 27     cost = float(np.squeeze(cost))  # makes sure cost is the dimension we expect.
     28                                     # E.g., turns [[17]] into 17
     29 

UnboundLocalError: local variable 'cost' referenced before assignment
```

***Expected output***

```
cost = 0.6930587610394646
All tests passed!
```

### 4.5 - Implement Backpropagation

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

### Exercise 6 - backward_propagation

Implement the function `backward_propagation()` .

**Instructions**: Backpropagation is usually the hardest (most mathematical) part in deep learning. To help you, here again is the slide from the lecture on backpropagation. You'll want to use the six equations on the right of this slide, since you are building a vectorized implementation.

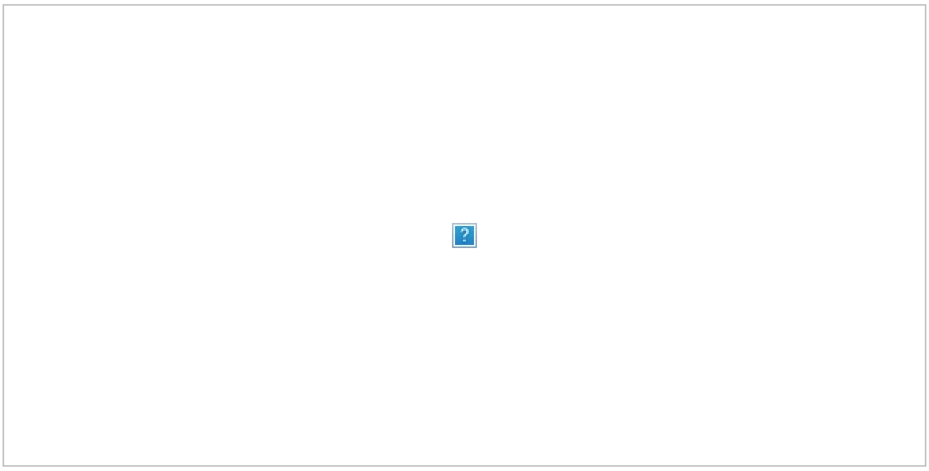


**Figure 1**: Backpropagation. Use the six equations on the right.

- 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 [ ]:

```python
# GRADED FUNCTION: backward_propagation

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 different parameters
    """
    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary "parameters".
    #(≈ 2 lines of code)
    # W1 = ...
    # W2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

    # Retrieve also A1 and A2 from dictionary "cache".
    #(≈ 2 lines of code)
    # A1 = ...
    # A2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

    # Backward propagation: calculate dW1, db1, dW2, db2.
    #(≈ 6 lines of code, corresponding to 6 equations on slide above)
    # dZ2 = ...
    # dW2 = ...
    # db2 = ...
    # dZ1 = ...
    # dW1 = ...
    # db1 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

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

    return grads
```

In [24]:

```python
parameters, cache, t_X, t_Y = backward_propagation_test_case()

grads = backward_propagation(parameters, cache, t_X, t_Y)
print ("dW1 = "+ str(grads["dW1"]))
print ("db1 = "+ str(grads["db1"]))
print ("dW2 = "+ str(grads["dW2"]))
print ("db2 = "+ str(grads["db2"]))

backward_propagation_test(backward_propagation)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-24-a06d396e2b09> in <module>
      1 parameters, cache, t_X, t_Y = backward_propagation_test_case()
      2 
----> 3 grads = backward_propagation(parameters, cache, t_X, t_Y)
      4 print ("dW1 = "+ str(grads["dW1"]))
      5 print ("db1 = "+ str(grads["db1"]))

NameError: name 'backward_propagation' is not defined
```

*Expected output*

```
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]]
All tests passed!
```

### 4.6 - Update Parameters

### Exercise 7 - update_parameters

Implement the update rule. Use gradient descent. You have to use (dW1, db1, dW2, db2) in order to update (W1, b1, W2, b2).

**General gradient descent rule**: $\theta = \theta - \alpha \frac{\partial J}{\partial \theta}$ where $\alpha$ is the learning rate and $\theta$ represents a parameter.

**Figure 2**: The gradient descent algorithm with a good learning rate (converging) and a bad learning rate (diverging). Images courtesy of Adam Harley.

**Hint**

- Use `copy.deepcopy(...)` when copying lists or dictionaries that are passed as parameters to functions. It avoids input parameters being modified within the function. In some scenarios, this could be inefficient, but it is required for grading purposes.

In [ ]:

```
# GRADED FUNCTION: update_parameters

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 a copy of each parameter from the dictionary "parameters". Use copy.deepcopy(...) for W1 and W2
    #(≈ 4 lines of code)
    # W1 = ...
    # b1 = ...
    # W2 = ...
    # b2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

    # Retrieve each gradient from the dictionary "grads"
    #(≈ 4 lines of code)
    # dW1 = ...
    # db1 = ...
    # dW2 = ...
    # db2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

    # Update rule for each parameter
    #(≈ 4 lines of code)
    # W1 = ...
    # b1 = ...
    # W2 = ...
    # b2 = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

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

    return parameters
```

In [25]:

```
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"]))

update_parameters_test(update_parameters)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-25-c37f65e5c916> in <module>
      1 parameters, grads = update_parameters_test_case()
----> 2 parameters = update_parameters(parameters, grads)
      3 
      4 print("W1 = " + str(parameters["W1"]))
      5 print("b1 = " + str(parameters["b1"]))

NameError: name 'update_parameters' is not defined
```

***Expected output***

```
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]]
All tests passed!
```

## 4.7 - Integration

Integrate your functions in `nn_model()`

## Exercise 8 - nn_model

Build your neural network model in `nn_model()` .

**Instructions**: The neural network model has to use the previous functions in the right order.

In [26]:

```
# GRADED FUNCTION: nn_model

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 predict.
    """

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

    # Initialize parameters
    #(≈ 1 line of code)
    # parameters = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        #(≈ 4 lines of code)
        # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache".
        # A2, cache = ...

        # Cost function. Inputs: "A2, Y". Outputs: "cost".
        # cost = ...

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

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

        # YOUR CODE STARTS HERE


        # YOUR CODE ENDS HERE

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

    return parameters
```

In [27]:

```
nn_model_test(nn_model)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-27-1243cded3c7d> in <module>
----> 1 nn_model_test(nn_model)

~/work/release/W3A1/public_tests.py in nn_model_test(target)
    261
    262     t_X, t_Y = nn_model_test_case()
--> 263     parameters = target(t_X, t_Y, n_h, num_iterations=10000, print_cost=True)
    264
    265     print("W1 = " + str(parameters["W1"]))

<ipython-input-26-1fabbb913996> in nn_model(X, Y, n_h, num_iterations, print_cost)
     50         # Print the cost every 1000 iterations
     51         if print_cost and i % 1000 == 0:
---> 52             print ("Cost after iteration %i: %f" %(i, cost))
     53
     54     return parameters

NameError: name 'cost' is not defined
```

***Expected output***

```
Cost after iteration 0: 0.693497
Cost after iteration 1000: 0.000180
Cost after iteration 2000: 0.000088
...
Cost after iteration 8000: 0.000022
Cost after iteration 9000: 0.000019
W1 = [[-0.48394893 -0.91443482]
 [-0.69227123 -1.28596651]
 [ 0.63806018  1.18124948]
 [ 0.73594679  1.35718308]
 [-0.62621054 -1.16022636]]
b1 = [[ 0.00285386]
 [ 0.01642488]
 [-0.01114986]
 [-0.01656405]
 [ 0.01042274]]
W2 = [[-1.45325879 -2.65768451  2.26286606  2.97274518 -2.18860534]]
b2 = [[0.00758617]]
All tests passed!
```

# 5 - Test the Model

## 5.1 - Predict

## Exercise 9 - predict

Predict with your model by building `predict()` . Use forward propagation to predict results.

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

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 [28]:

```
# GRADED FUNCTION: predict

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 0/1 using 0.5 as the threshold.
    #(≈ 2 lines of code)
    # A2, cache = ...
    # predictions = ...
    # YOUR CODE STARTS HERE


    # YOUR CODE ENDS HERE

    return predictions
```

In [29]:

```
parameters, t_X = predict_test_case()

predictions = predict(parameters, t_X)
print("Predictions: " + str(predictions))

predict_test(predict)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-29-eb08347ef392> in <module>
      1 parameters, t_X = predict_test_case()
      2 
----> 3 predictions = predict(parameters, t_X)
      4 print("Predictions: " + str(predictions))
      5 

<ipython-input-28-98939f4c2e75> in predict(parameters, X)
     22     # YOUR CODE ENDS HERE
     23 
---> 24     return predictions

NameError: name 'predictions' is not defined
```

***Expected output***

```
Predictions: [[ True False  True]]
All tests passed!
```

### 5.2 - Test the Model on the Planar Dataset

It's 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 [30]:

```
# Build a model with a n_h-dimensional hidden layer
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))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-30-11bad20c9992> in <module>
      1 # Build a model with a n_h-dimensional hidden layer
----> 2 parameters = nn_model(X, Y, n_h = 4, num_iterations = 10000, print_cost=True)
      3 
      4 # Plot the decision boundary
      5 plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)

<ipython-input-26-1fabbb913996> in nn_model(X, Y, n_h, num_iterations, print_cost)
     50         # Print the cost every 1000 iterations
     51         if print_cost and i % 1000 == 0:
---> 52             print ("Cost after iteration %i: %f" %(i, cost))
     53 
     54     return parameters

NameError: name 'cost' is not defined
```

In [31]:

```
# 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) * 100
) + '%')
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-31-04f44a68b329> in <module>
      1 # Print accuracy
----> 2 predictions = predict(parameters, X)
      3 print ('Accuracy: %d' % float((np.dot(Y, predictions.T) + np.dot(1 - Y, 1 - predictions.T))
/ float(Y.size) * 100) + '%')

<ipython-input-28-98939f4c2e75> in predict(parameters, X)
     22     # YOUR CODE ENDS HERE
     23 
---> 24     return predictions

NameError: name 'predictions' is not defined
```

**Expected Output**:

| | |
|---|---|
| **Accuracy** | 90% |

Accuracy is really high compared to Logistic Regression. The model has learned the patterns of the flower's petals! Unlike logistic regression, neural networks are able to learn even highly non-linear decision boundaries.

### Congrats on finishing this Programming Assignment!

Here's a quick recap of all you just accomplished:

- Built a complete 2-class classification neural network with a hidden layer
- Made good use of a non-linear unit
- Computed the cross entropy loss
- Implemented forward and backward propagation
- Seen the impact of varying the hidden layer size, including overfitting.

You've created a neural network that can learn patterns! Excellent work. Below, there are some optional exercises to try out some other hidden layer sizes, and other datasets.

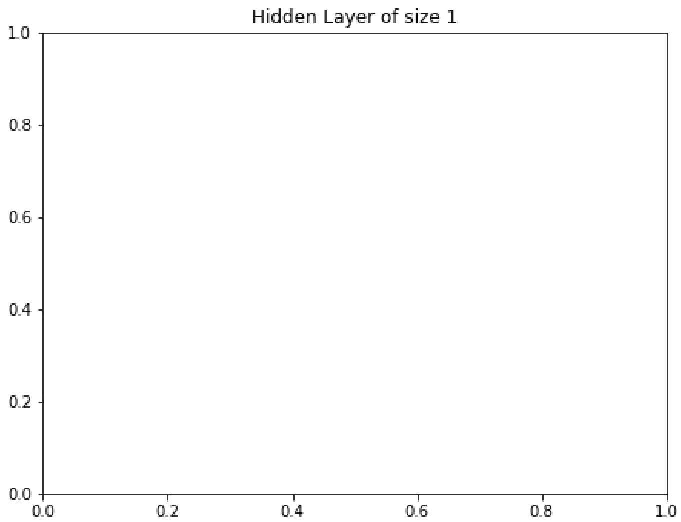
## 6 - Tuning hidden layer size (optional/ungraded exercise)

Run the following code(it may take 1-2 minutes). Then, observe different behaviors of the model for various hidden layer sizes.

In [32]:

```
# This may take about 2 minutes to run

plt.figure(figsize=(16, 32))
hidden_layer_sizes = [1, 2, 3, 4, 5]

# you can try with different hidden layer sizes
# but make sure before you submit the assignment it is set as "hidden_layer_sizes = [1, 2, 3, 4, 5]"
# 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)) / float(Y.size)*100)
    print ("Accuracy for {} hidden units: {} %".format(n_h, accuracy))
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-32-47bd2bcd1976> in <module>
     12     plt.title('Hidden Layer of size %d' % n_h)
     13     parameters = nn_model(X, Y, n_h, num_iterations = 5000)
---> 14     plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
     15     predictions = predict(parameters, X)
     16     accuracy = float((np.dot(Y,predictions.T) + np.dot(1 - Y, 1 - predictions.T)) / float(Y.
size)*100)

~/work/release/W3A1/planar_utils.py in plot_decision_boundary(model, X, y)
     13     xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max, h))
     14     # Predict the function value for the whole grid
---> 15     Z = model(np.c_[xx.ravel(), yy.ravel()])
     16     Z = Z.reshape(xx.shape)
     17     # Plot the contour and training examples

<ipython-input-32-47bd2bcd1976> in <lambda>(x)
     12     plt.title('Hidden Layer of size %d' % n_h)
     13     parameters = nn_model(X, Y, n_h, num_iterations = 5000)
---> 14     plot_decision_boundary(lambda x: predict(parameters, x.T), X, Y)
     15     predictions = predict(parameters, X)
     16     accuracy = float((np.dot(Y,predictions.T) + np.dot(1 - Y, 1 - predictions.T)) / float(Y.
size)*100)

<ipython-input-28-98939f4c2e75> in predict(parameters, X)
     22     # YOUR CODE ENDS HERE
     23 
---> 24     return predictions

NameError: name 'predictions' is not defined
```




**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 noticeable overfitting.
- Later, you'll become familiar with regularization, which lets you use very large models (such as n_h = 50) without much overfitting.

**Note**: Remember to submit the assignment by clicking the blue "Submit Assignment" button at the upper-right.

**Some optional/ungraded questions that you can explore if you wish**:

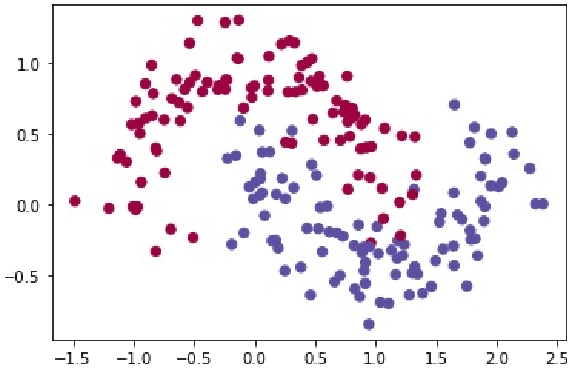
- What happens when you change the tanh activation for a sigmoid activation or a ReLU activation?
- Play with the learning_rate. What happens?
- What if we change the dataset? (See part 7 below!)

## 7- Performance on other datasets

If you want, you can rerun the whole notebook (minus the dataset part) for each of the following datasets.

In [33]:

```
# Datasets
noisy_circles, noisy_moons, blobs, gaussian_quantiles, no_structure = load_extra_datasets()

datasets = {"noisy_circles": noisy_circles,
            "noisy_moons": noisy_moons,
            "blobs": blobs,
            "gaussian_quantiles": gaussian_quantiles}

### START CODE HERE ### (choose your dataset)
dataset = "noisy_moons"
### END CODE HERE ###

X, Y = datasets[dataset]
X, Y = X.T, Y.reshape(1, Y.shape[0])

# make blobs binary
if dataset == "blobs":
    Y = Y%2

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




**References**:


- http://cs231n.github.io/neural-networks-case-study/ (http://cs231n.github.io/neural-networks-case-study/)


In [ ]:

```
```