**Updates to Assignment**[**¶**](https://ukoudwsv.labs.coursera.org/notebooks/Week%203/Planar%20data%20classification%20with%20one%20hidden%20layer/Planar_data_classification_with_onehidden_layer_v6c.ipynb#Updates-to-Assignment)

**If you were working on the older version:**

* Please click on the "Coursera" icon in the top right to open up the folder directory.
* Navigate to the folder: Week 3/ Planar data classification with one hidden layer. You can see your prior work in version 6b: "Planar data classification with one hidden layer v6b.ipynb"

**List of bug fixes and enhancements**

* Clarifies that the classifier will learn to classify regions as either red or blue.
* compute\_cost function fixes np.squeeze by casting it as a float.
* compute\_cost instructions clarify the purpose of np.squeeze.
* compute\_cost clarifies that "parameters" parameter is not needed, but is kept in the function definition until the auto-grader is also updated.
* nn\_model removes extraction of parameter values, as the entire parameter dictionary is passed to the invoked functions.

**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 a hidden layer. You will see a big difference between this model and the one you implemented using logistic regression.

**You will learn 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 - Packages**

Let's 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 [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\_dataset, load\_extra\_datasets

​

**%**matplotlib inline

​

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

**2 - Dataset**

First, let's get the dataset you 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 other words, we want the classifier to define regions as either red or blue.

In [3]:

*# Visualize the data:*

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

Chart

Description automatically generated

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).

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

**Exercise**: 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)

In [4]:

*### START CODE HERE ### (≈ 3 lines of code)*

shape\_X = X.shape

shape\_Y = Y.shape

m = shape\_X[1] *# training set size*

*### END CODE 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))

The shape of X is: (2, 400)

The shape of Y is: (1, 400)

I have m = 400 training examples!

**Expected Output**:

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

**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);

You can now plot the decision boundary of these models. Run 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\_predictions) **+** np.dot(1**-**Y,1**-**LR\_predictions))**/**float(Y.size)**\***100) **+**

'% ' **+** "(percentage of correctly labelled datapoints)")

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

A close up of a logo

Description automatically generated

**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 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)x(i):

z[1](i)=W[1]x(i)+b[1](1)(1)z[1](i)=W[1]x(i)+b[1]

a[1](i)=tanh(z[1](i))(2)(2)a[1](i)=tanh⁡(z[1](i))

z[2](i)=W[2]a[1](i)+b[2](3)(3)z[2](i)=W[2]a[1](i)+b[2]

ŷ (i)=a[2](i)=σ(z[2](i))(4)(4)y^(i)=a[2](i)=σ(z[2](i))

y(i)prediction={10if a[2](i)>0.5otherwise (5)(5)yprediction(i)={1if a[2](i)>0.50otherwise

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

J=−1m∑i=0m(y(i)log(a[2](i))+(1−y(i))log(1−a[2](i)))(6)(6)J=−1m∑i=0m(y(i)log⁡(a[2](i))+(1−y(i))log⁡(1−a[2](i)))

**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.

**4.1 - Defining the neural network structure**

**Exercise**: 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

**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 [15]:

*# 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

"""

*### START CODE HERE ### (≈ 3 lines of code)*

n\_x = X.shape[0] *# size of input layer*

n\_h = 4

n\_y = Y.shape[0] *# size of output layer*

*### END CODE HERE ###*

**return** (n\_x, n\_h, n\_y)

In [16]:

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 |
| **n\_h** | 4 |
| **n\_y** | 2 |

**4.2 - Initialize the model's parameters**

**Exercise**: 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 [17]:

*# 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)

"""

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

*### START CODE HERE ### (≈ 4 lines of code)*

W1 = np.random.randn(n\_h, n\_x) **\*** 0.01

b1 = np.zeros((n\_h, 1))

W2 = np.random.randn(n\_y, n\_h) **\*** 0.01

b2 = np.zeros((n\_y, 1))

*### END CODE HERE ###*

**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 [18]:

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.]]

**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.]] |

**4.3 - The Loop**

**Question**: Implement forward\_propagation().

**Instructions**:

* Look above at the mathematical representation of your classifier.
* You can use the function sigmoid(). It is built-in (imported) in the notebook.
* You can use the function np.tanh(). It is part of the numpy library.
* The steps you 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]Z[1],A[1],Z[2] and A[2]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 [19]:

*# 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"*

*### START CODE HERE ### (≈ 4 lines of code)*

W1 = parameters['W1']

b1 = parameters['b1']

W2 = parameters['W2']

b2 = parameters['b2']

*### END CODE HERE ###*

*# Implement Forward Propagation to calculate A2 (probabilities)*

*### START CODE HERE ### (≈ 4 lines of code)*

Z1 = np.dot(W1, X) **+** b1

A1 = np.tanh(Z1)

Z2 = np.dot(W2, A1) **+** b2

A2 = sigmoid(Z2)

*### END CODE HERE ###*

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

cache = {"Z1": Z1,

"A1": A1,

"Z2": Z2,

"A2": A2}

**return** A2, cache

In [20]:

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 ours.*

print(np.mean(cache['Z1']) ,np.mean(cache['A1']),np.mean(cache['Z2']),np.mean(cache['A2']))

0.262818640198 0.091999045227 -1.30766601287 0.212877681719

**Expected Output**:

|  |
| --- |
| 0.262818640198 0.091999045227 -1.30766601287 0.212877681719 |

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

J=−1m∑i=1m(y(i)log(a[2](i))+(1−y(i))log(1−a[2](i)))(13)(13)J=−1m∑i=1m(y(i)log⁡(a[2](i))+(1−y(i))log⁡(1−a[2](i)))

**Exercise**: Implement compute\_cost() to compute the value of the cost JJ.

**Instructions**:

* There are many ways to implement the cross-entropy loss. To help you, we give you how we would have implemented −∑i=0my(i)log(a[2](i))−∑i=0my(i)log⁡(a[2](i)):
* logprobs = np.multiply(np.log(A2),Y)
* cost = **-** np.sum(logprobs) *# no need to use a for loop!*

(you can use either np.multiply() and then np.sum() or directly np.dot()).  
Note that 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. We can use np.squeeze() to remove redundant dimensions (in the case of single float, this will be reduced to a zero-dimension array). We can cast the array as a type float using float().

In [21]:

*# GRADED FUNCTION: compute\_cost*

​

**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 of examples)

Y -- "true" labels vector of shape (1, number of examples)

parameters -- python dictionary containing your parameters W1, b1, W2 and b2

[Note that the parameters argument is not used in this function,

but the auto-grader currently expects this parameter.

Future version of this notebook will fix both the notebook

and the auto-grader so that `parameters` is not needed.

For now, please include `parameters` in the function signature,

and also when invoking this function.]

Returns:

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

"""

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

​

*# Compute the cross-entropy cost*

*### START CODE HERE ### (≈ 2 lines of code)*

logprobs = np.multiply(np.log(A2), Y) **+** np.multiply(np.log(1 **-** A2), 1 **-** Y)

cost = **-** (1 **/** m) **\*** np.sum(logprobs)

*### END CODE HERE ###*

cost = float(np.squeeze(cost)) *# makes sure cost is the dimension we expect.*

*# E.g., turns [[17]] into 17*

**assert**(isinstance(cost, float))

**return** cost

In [23]:

A2, Y\_assess, parameters = compute\_cost\_test\_case()

​

print("cost = " **+** str(compute\_cost(A2, Y\_assess, parameters)))

cost = 0.6930587610394646

**Expected Output**:

|  |  |
| --- | --- |
| **cost** | 0.693058761... |

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

**Question**: 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.

* Tips:
  + To compute dZ1 you'll need to compute g[1]′(Z[1])g[1]′(Z[1]). Since g[1](.)g[1](.) is the tanh activation function, if a=g[1](z)a=g[1](z) then g[1]′(z)=1−a2g[1]′(z)=1−a2. So you can compute g[1]′(Z[1])g[1]′(Z[1]) using (1 - np.power(A1, 2)).

In [24]:

*# 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".*

*### START CODE HERE ### (≈ 2 lines of code)*

W1 = parameters["W1"]

W2 = parameters["W2"]

*### END CODE HERE ###*

*# Retrieve also A1 and A2 from dictionary "cache".*

*### START CODE HERE ### (≈ 2 lines of code)*

A1 = cache["A1"]

A2 = cache["A2"]

*### END CODE HERE ###*

*# Backward propagation: calculate dW1, db1, dW2, db2.*

*### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on slide above)*

dZ2 = A2 **-** Y

dW2 = (1 **/** m) **\*** (np.dot(dZ2, A1.T))

db2 = (1 **/** m) **\*** (np.sum(dZ2, axis = 1, keepdims = **True**))

dZ1 = 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**))

*### END CODE HERE ###*

grads = {"dW1": dW1,

"db1": db1,

"dW2": dW2,

"db2": db2}

**return** grads

In [25]:

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]]

**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]] |

**Question**: 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**: θ=θ−α∂J∂θθ=θ−α∂J∂θ where αα is the learning rate and θθ represents a parameter.

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

In [26]:

*# 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 each parameter from the dictionary "parameters"*

*### START CODE HERE ### (≈ 4 lines of code)*

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

*### END CODE HERE ###*

*# Retrieve each gradient from the dictionary "grads"*

*### START CODE HERE ### (≈ 4 lines of code)*

dW1 = grads["dW1"]

db1 = grads["db1"]

dW2 = grads["dW2"]

db2 = grads["db2"]

*## END CODE HERE ###*

*# Update rule for each parameter*

*### START CODE HERE ### (≈ 4 lines of code)*

W1 = W1 **-** learning\_rate **\*** dW1

b1 = b1 **-** learning\_rate **\*** db1

W2 = W2 **-** learning\_rate **\*** dW2

b2 = b2 **-** learning\_rate **\*** db2

*### END CODE HERE ###*

parameters = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

**return** parameters

In [27]:

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]]

**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]] |

**4.4 - Integrate parts 4.1, 4.2 and 4.3 in nn\_model()**

**Question**: Build your neural network model in nn\_model().

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

In [28]:

*# 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*

*### START CODE HERE ### (≈ 1 line of code)*

parameters = initialize\_parameters(n\_x, n\_h, n\_y)

*### END CODE HERE ###*

*# Loop (gradient descent)*

​

**for** i **in** range(0, num\_iterations):

*### START CODE HERE ### (≈ 4 lines of code)*

*# 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)

*### END CODE 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 [29]:

X\_assess, Y\_assess = nn\_model\_test\_case()

parameters = nn\_model(X\_assess, Y\_assess, 4, num\_iterations=10000, print\_cost=**True**)

print("W1 = " **+** str(parameters["W1"]))

print("b1 = " **+** str(parameters["b1"]))

print("W2 = " **+** str(parameters["W2"]))

print("b2 = " **+** str(parameters["b2"]))

Cost after iteration 0: 0.692739

Cost after iteration 1000: 0.000218

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 = [[-0.65848169 1.21866811]

[-0.76204273 1.39377573]

[ 0.5792005 -1.10397703]

[ 0.76773391 -1.41477129]]

b1 = [[ 0.287592 ]

[ 0.3511264 ]

[-0.2431246 ]

[-0.35772805]]

W2 = [[-2.45566237 -3.27042274 2.00784958 3.36773273]]

b2 = [[ 0.20459656]]

**Expected Output**:

|  |  |
| --- | --- |
| **cost after iteration 0** | 0.692739 |
| ⋮⋮ | ⋮⋮ |
| **W1** | [[-0.65848169 1.21866811] [-0.76204273 1.39377573] [ 0.5792005 -1.10397703] [ 0.76773391 -1.41477129]] |
| **b1** | [[ 0.287592 ] [ 0.3511264 ] [-0.2431246 ] [-0.35772805]] |
| **W2** | [[-2.45566237 -3.27042274 2.00784958 3.36773273]] |
| **b2** | [[ 0.20459656]] |

**4.5 Predictions**

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

**Reminder**: predictions = yprediction=𝟙{activation > 0.5}={10if activation>0.5otherwiseyprediction=1{activation > 0.5}={1if activation>0.50otherwise

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 [32]:

*# 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.*

*### START CODE HERE ### (≈ 2 lines of code)*

A2, cache = forward\_propagation(X, parameters)

predictions = (A2 **>** 0.5)

*### END CODE HERE ###*

**return** predictions

In [33]:

parameters, X\_assess = predict\_test\_case()

​

predictions = predict(parameters, X\_assess)

print("predictions mean = " **+** str(np.mean(predictions)))

predictions mean = 0.666666666667

**Expected Output**:

|  |  |
| --- | --- |
| **predictions mean** | 0.666666666667 |

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 nhnh hidden units.

In [34]:

*# 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))

Cost after iteration 0: 0.693048

Cost after iteration 1000: 0.288083

Cost after iteration 2000: 0.254385

Cost after iteration 3000: 0.233864

Cost after iteration 4000: 0.226792

Cost after iteration 5000: 0.222644

Cost after iteration 6000: 0.219731

Cost after iteration 7000: 0.217504

Cost after iteration 8000: 0.219471

Cost after iteration 9000: 0.218612

Out[34]:

<matplotlib.text.Text at 0x7f75745af2e8>

Chart

Description automatically generated

**Expected Output**:

|  |  |
| --- | --- |
| **Cost after iteration 9000** | 0.218607 |

In [35]:

*# 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) **+** '%')

Accuracy: 90%

**Expected Output**:

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

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.

**4.6 - 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 [36]:

*# This may take about 2 minutes to run*

​

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))**/**float(Y.size)**\***100)

print ("Accuracy for {} hidden units: {} %".format(n\_h, accuracy))

Accuracy for 1 hidden units: 67.5 %

Accuracy for 2 hidden units: 67.25 %

Accuracy for 3 hidden units: 90.75 %

Accuracy for 4 hidden units: 90.5 %

Accuracy for 5 hidden units: 91.25 %

Accuracy for 20 hidden units: 90.0 %

Accuracy for 50 hidden units: 90.25 %

A picture containing building, window, airplane, door

Description automatically generated

**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.
* You will also learn later about regularization, which lets you use very large models (such as n\_h = 50) without much overfitting.

**Optional questions**:

**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 5 below!)

**You've learnt 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.

Nice work!

**5) Performance on other datasets**

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

In [40]:

*# 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 = "gaussian\_quantiles"

*### 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);

Chart, scatter chart

Description automatically generated

Congrats on finishing this Programming Assignment!

Reference:

* <http://scs.ryerson.ca/~aharley/neural-networks/>
* <http://cs231n.github.io/neural-networks-case-study/>