#### Choice of learning rate

**Reminder**: In order for Gradient Descent to work you must choose the learning rate wisely. The learning rate 𝛼α determines how rapidly we update the parameters. If the learning rate is too large we may "overshoot" the optimal value. Similarly, if it is too small we will need too many iterations to converge to the best values. That's why it is crucial to use a well-tuned learning rate.

Let's compare the learning curve of our model with several choices of learning rates. Run the cell below. This should take about 1 minute. Feel free also to try different values than the three we have initialized the learning\_rates variable to contain, and see what happens.

In [ ]:

learning\_rates **=** [0.01, 0.001, 0.0001]

models **=** {}

​

**for** lr **in** learning\_rates:

print ("Training a model with learning rate: " **+** str(lr))

models[str(lr)] **=** model(train\_set\_x, train\_set\_y, test\_set\_x, test\_set\_y, num\_iterations**=**1500, learning\_rate**=**lr, print\_cost**=False**)

print ('\n' **+** "-------------------------------------------------------" **+** '\n')

​

**for** lr **in** learning\_rates:

plt.plot(np.squeeze(models[str(lr)]["costs"]), label**=**str(models[str(lr)]["learning\_rate"]))

​

plt.ylabel('cost')

plt.xlabel('iterations (hundreds)')

​

legend **=** plt.legend(loc**=**'upper center', shadow**=True**)

frame **=** legend.get\_frame()

frame.set\_facecolor('0.90')

plt.show()

**Interpretation**:

* Different learning rates give different costs and thus different predictions results.
* If the learning rate is too large (0.01), the cost may oscillate up and down. It may even diverge (though in this example, using 0.01 still eventually ends up at a good value for the cost).
* A lower cost doesn't mean a better model. You have to check if there is possibly overfitting. It happens when the training accuracy is a lot higher than the test accuracy.
* In deep learning, we usually recommend that you:
  + Choose the learning rate that better minimizes the cost function.
  + If your model overfits, use other techniques to reduce overfitting. (We'll talk about this in later videos.)

## 7 - Test with your own image (optional/ungraded exercise)

Congratulations on finishing this assignment. You can use your own image and see the output of your model. To do that:

1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.

2. Add your image to this Jupyter Notebook's directory, in the "images" folder

3. Change your image's name in the following code

4. Run the code and check if the algorithm is right (1 = cat, 0 = non-cat)!

In [ ]:

*# change this to the name of your image file*

my\_image **=** "my\_image.jpg"

​

*# We preprocess the image to fit your algorithm.*

fname **=** "images/" **+** my\_image

image **=** np.array(Image.open(fname).resize((num\_px, num\_px)))

plt.imshow(image)

image **=** image **/** 255.

image **=** image.reshape((1, num\_px **\*** num\_px **\*** 3)).T

my\_predicted\_image **=** predict(logistic\_regression\_model["w"], logistic\_regression\_model["b"], image)

​

print("y = " **+** str(np.squeeze(my\_predicted\_image)) **+** ", your algorithm predicts a \"" **+** classes[int(np.squeeze(my\_predicted\_image)),].decode("utf-8") **+** "\" picture.")

# WEEK 4

# Building your Deep Neural Network: Step by Step

**Notation**:

* Superscript [𝑙][l] denotes a quantity associated with the 𝑙𝑡ℎlth layer.
  + Example: 𝑎[𝐿]a[L] is the 𝐿𝑡ℎLth layer activation. 𝑊[𝐿]W[L] and 𝑏[𝐿]b[L] are the 𝐿𝑡ℎLth layer parameters.
* Superscript (𝑖)(i) denotes a quantity associated with the 𝑖𝑡ℎith example.
  + Example: 𝑥(𝑖)x(i) is the 𝑖𝑡ℎith training example.
* Lowerscript 𝑖i denotes the 𝑖𝑡ℎith entry of a vector.
  + Example: 𝑎[𝑙]𝑖ai[l] denotes the 𝑖𝑡ℎith entry of the 𝑙𝑡ℎlth layer's activations).

## 1 - Packages

First, import all the packages you'll need during this assignment.

* [numpy](https://sxdjmbls.labs.coursera.org/notebooks/release/W4A1/www.numpy.org) is the main package for scientific computing with Python.
* [matplotlib](http://matplotlib.org/) is a library to plot graphs in Python.
* dnn\_utils provides some necessary functions for this notebook.
* testCases provides some test cases to assess the correctness of your functions
* np.random.seed(1) is used to keep all the random function calls consistent. It helps grade your work. Please don't change the seed!

In [1]:

**import** numpy **as** np

**import** h5py

**import** matplotlib.pyplot **as** plt

**from** testCases **import** **\***

**from** dnn\_utils **import** sigmoid, sigmoid\_backward, relu, relu\_backward

**from** public\_tests **import** **\***

​

**%**matplotlib inline

plt.rcParams['figure.figsize'] **=** (5.0, 4.0) *# set default size of plots*

plt.rcParams['image.interpolation'] **=** 'nearest'

plt.rcParams['image.cmap'] **=** 'gray'

​

**%**load\_ext autoreload

**%**autoreload 2

​

np.random.seed(1)

## 2 - Outline

To build your neural network, you'll be implementing several "helper functions." These helper functions will be used in the next assignment to build a two-layer neural network and an L-layer neural network.

Each small helper function will have detailed instructions to walk you through the necessary steps. Here's an outline of the steps in this assignment:

* Initialize the parameters for a two-layer network and for an 𝐿L-layer neural network
* Implement the forward propagation module (shown in purple in the figure below)
  + Complete the LINEAR part of a layer's forward propagation step (resulting in 𝑍[𝑙]Z[l]).
  + The ACTIVATION function is provided for you (relu/sigmoid)
  + Combine the previous two steps into a new [LINEAR->ACTIVATION] forward function.
  + Stack the [LINEAR->RELU] forward function L-1 time (for layers 1 through L-1) and add a [LINEAR->SIGMOID] at the end (for the final layer 𝐿L). This gives you a new L\_model\_forward function.
* Compute the loss
* Implement the backward propagation module (denoted in red in the figure below)
  + Complete the LINEAR part of a layer's backward propagation step
  + The gradient of the ACTIVATE function is provided for you(relu\_backward/sigmoid\_backward)
  + Combine the previous two steps into a new [LINEAR->ACTIVATION] backward function
  + Stack [LINEAR->RELU] backward L-1 times and add [LINEAR->SIGMOID] backward in a new L\_model\_backward function
* Finally, update the parameters

**Figure 1**

**Note**:

For every forward function, there is a corresponding backward function. This is why at every step of your forward module you will be storing some values in a cache. These cached values are useful for computing gradients.

In the backpropagation module, you can then use the cache to calculate the gradients. Don't worry, this assignment will show you exactly how to carry out each of these steps!

## 3 - Initialization

You will write two helper functions to initialize the parameters for your model. The first function will be used to initialize parameters for a two layer model. The second one generalizes this initialization process to 𝐿L layers.

### 3.1 - 2-layer Neural Network

### Exercise 1 - initialize\_parameters

Create and initialize the parameters of the 2-layer neural network.

**Instructions**:

* The model's structure is: LINEAR -> RELU -> LINEAR -> SIGMOID.
* Use this random initialization for the weight matrices: np.random.randn(shape)\*0.01 with the correct shape
* Use zero initialization for the biases: np.zeros(shape)

In [2]:

*# 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:

parameters -- 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(1)

*#(≈ 4 lines of code)*

*# W1 = ...*

*# b1 = ...*

*# W2 = ...*

*# b2 = ...*

*# YOUR CODE STARTS HERE*

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))

*# YOUR CODE ENDS HERE*

parameters **=** {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

**return** parameters

In [3]:

parameters **=** initialize\_parameters(3,2,1)

​

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)

W1 = [[ 0.01624345 -0.00611756 -0.00528172]

[-0.01072969 0.00865408 -0.02301539]]

b1 = [[0.]

[0.]]

W2 = [[ 0.01744812 -0.00761207]]

b2 = [[0.]]

All tests passed.

**Expected output**

W1 = [[ 0.01624345 -0.00611756 -0.00528172]

[-0.01072969 0.00865408 -0.02301539]]

b1 = [[0.]

[0.]]

W2 = [[ 0.01744812 -0.00761207]]

b2 = [[0.]]

### 3.2 - L-layer Neural Network

The initialization for a deeper L-layer neural network is more complicated because there are many more weight matrices and bias vectors. When completing the initialize\_parameters\_deep function, you should make sure that your dimensions match between each layer. Recall that 𝑛[𝑙]n[l] is the number of units in layer 𝑙l. For example, if the size of your input 𝑋X is (12288,209)(12288,209) (with 𝑚=209m=209 examples) then:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Shape of W** | **Shape of b** | **Activation** | **Shape of Activation** |
|  |  |  |  |  |
| **Layer 1** | (𝑛[1],12288)(n[1],12288) | (𝑛[1],1)(n[1],1) | 𝑍[1]=𝑊[1]𝑋+𝑏[1]Z[1]=W[1]X+b[1] | (𝑛[1],209)(n[1],209) |
|  |  |  |  |  |
| **Layer 2** | (𝑛[2],𝑛[1])(n[2],n[1]) | (𝑛[2],1)(n[2],1) | 𝑍[2]=𝑊[2]𝐴[1]+𝑏[2]Z[2]=W[2]A[1]+b[2] | (𝑛[2],209)(n[2],209) |
|  |  |  |  |  |
| ⋮⋮ | ⋮⋮ | ⋮⋮ | ⋮⋮ | ⋮⋮ |
|  |  |  |  |  |
| **Layer L-1** | (𝑛[𝐿−1],𝑛[𝐿−2])(n[L−1],n[L−2]) | (𝑛[𝐿−1],1)(n[L−1],1) | 𝑍[𝐿−1]=𝑊[𝐿−1]𝐴[𝐿−2]+𝑏[𝐿−1]Z[L−1]=W[L−1]A[L−2]+b[L−1] | (𝑛[𝐿−1],209)(n[L−1],209) |
|  |  |  |  |  |
| **Layer L** | (𝑛[𝐿],𝑛[𝐿−1])(n[L],n[L−1]) | (𝑛[𝐿],1)(n[L],1) | 𝑍[𝐿]=𝑊[𝐿]𝐴[𝐿−1]+𝑏[𝐿]Z[L]=W[L]A[L−1]+b[L] | (𝑛[𝐿],209)(n[L],209) |
|  |  |  |  |  |

Remember that when you compute 𝑊𝑋+𝑏WX+b in python, it carries out broadcasting. For example, if:

𝑊=𝑤00𝑤10𝑤20𝑤01𝑤11𝑤21𝑤02𝑤12𝑤22𝑋=𝑥00𝑥10𝑥20𝑥01𝑥11𝑥21𝑥02𝑥12𝑥22𝑏=𝑏0𝑏1𝑏2(2)(2)W=[w00w01w02w10w11w12w20w21w22]X=[x00x01x02x10x11x12x20x21x22]b=[b0b1b2]

Then 𝑊𝑋+𝑏WX+b will be:

𝑊𝑋+𝑏=(𝑤00𝑥00+𝑤01𝑥10+𝑤02𝑥20)+𝑏0(𝑤10𝑥00+𝑤11𝑥10+𝑤12𝑥20)+𝑏1(𝑤20𝑥00+𝑤21𝑥10+𝑤22𝑥20)+𝑏2(𝑤00𝑥01+𝑤01𝑥11+𝑤02𝑥21)+𝑏0(𝑤10𝑥01+𝑤11𝑥11+𝑤12𝑥21)+𝑏1(𝑤20𝑥01+𝑤21𝑥11+𝑤22𝑥21)+𝑏2⋯⋯⋯(3)(3)WX+b=[(w00x00+w01x10+w02x20)+b0(w00x01+w01x11+w02x21)+b0⋯(w10x00+w11x10+w12x20)+b1(w10x01+w11x11+w12x21)+b1⋯(w20x00+w21x10+w22x20)+b2(w20x01+w21x11+w22x21)+b2⋯]

### Exercise 2 - initialize\_parameters\_deep

Implement initialization for an L-layer Neural Network.

**Instructions**:

* The model's structure is \*[LINEAR -> RELU] ×× (L-1) -> LINEAR -> SIGMOID\*. I.e., it has 𝐿−1L−1 layers using a ReLU activation function followed by an output layer with a sigmoid activation function.
* Use random initialization for the weight matrices. Use np.random.randn(shape) \* 0.01.
* Use zeros initialization for the biases. Use np.zeros(shape).
* You'll store 𝑛[𝑙]n[l], the number of units in different layers, in a variable layer\_dims. For example, the layer\_dims for last week's Planar Data classification model would have been [2,4,1]: There were two inputs, one hidden layer with 4 hidden units, and an output layer with 1 output unit. This means W1's shape was (4,2), b1 was (4,1), W2 was (1,4) and b2 was (1,1). Now you will generalize this to 𝐿L layers!
* Here is the implementation for 𝐿=1L=1 (one layer neural network). It should inspire you to implement the general case (L-layer neural network).
* **if** L **==** 1:
* parameters["W" **+** str(L)] **=** np.random.randn(layer\_dims[1], layer\_dims[0]) **\*** 0.01

parameters["b" **+** str(L)] **=** np.zeros((layer\_dims[1], 1))

In [4]:

*# GRADED FUNCTION: initialize\_parameters\_deep*

​

**def** initialize\_parameters\_deep(layer\_dims):

"""

Arguments:

layer\_dims -- python array (list) containing the dimensions of each layer in our network

Returns:

parameters -- python dictionary containing your parameters "W1", "b1", ..., "WL", "bL":

Wl -- weight matrix of shape (layer\_dims[l], layer\_dims[l-1])

bl -- bias vector of shape (layer\_dims[l], 1)

"""

np.random.seed(3)

parameters **=** {}

L **=** len(layer\_dims) *# number of layers in the network*

​

**for** l **in** range(1, L):

*#(≈ 2 lines of code)*

*# parameters['W' + str(l)] = ...*

*# parameters['b' + str(l)] = ...*

*# YOUR CODE STARTS HERE*

parameters['W' **+** str(l)] **=** np.random.randn(layer\_dims[l], layer\_dims[l**-**1]) **\*** 0.01

parameters['b' **+**str(l)] **=** np.zeros((layer\_dims[l], 1))

*# YOUR CODE ENDS HERE*

**assert**(parameters['W' **+** str(l)].shape **==** (layer\_dims[l], layer\_dims[l **-** 1]))

**assert**(parameters['b' **+** str(l)].shape **==** (layer\_dims[l], 1))

​

**return** parameters

In [5]:

parameters **=** initialize\_parameters\_deep([5,4,3])

​

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

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

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

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

​

initialize\_parameters\_deep\_test(initialize\_parameters\_deep)

W1 = [[ 0.01788628 0.0043651 0.00096497 -0.01863493 -0.00277388]

[-0.00354759 -0.00082741 -0.00627001 -0.00043818 -0.00477218]

[-0.01313865 0.00884622 0.00881318 0.01709573 0.00050034]

[-0.00404677 -0.0054536 -0.01546477 0.00982367 -0.01101068]]

b1 = [[0.]

[0.]

[0.]

[0.]]

W2 = [[-0.01185047 -0.0020565 0.01486148 0.00236716]

[-0.01023785 -0.00712993 0.00625245 -0.00160513]

[-0.00768836 -0.00230031 0.00745056 0.01976111]]

b2 = [[0.]

[0.]

[0.]]

All tests passed.

**Expected output**

W1 = [[ 0.01788628 0.0043651 0.00096497 -0.01863493 -0.00277388]

[-0.00354759 -0.00082741 -0.00627001 -0.00043818 -0.00477218]

[-0.01313865 0.00884622 0.00881318 0.01709573 0.00050034]

[-0.00404677 -0.0054536 -0.01546477 0.00982367 -0.01101068]]

b1 = [[0.]

[0.]

[0.]

[0.]]

W2 = [[-0.01185047 -0.0020565 0.01486148 0.00236716]

[-0.01023785 -0.00712993 0.00625245 -0.00160513]

[-0.00768836 -0.00230031 0.00745056 0.01976111]]

b2 = [[0.]

[0.]

[0.]]

## 4 - Forward Propagation Module

### 4.1 - Linear Forward

Now that you have initialized your parameters, you can do the forward propagation module. Start by implementing some basic functions that you can use again later when implementing the model. Now, you'll complete three functions in this order:

* LINEAR
* LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
* [LINEAR -> RELU] ×× (L-1) -> LINEAR -> SIGMOID (whole model)

The linear forward module (vectorized over all the examples) computes the following equations:

𝑍[𝑙]=𝑊[𝑙]𝐴[𝑙−1]+𝑏[𝑙](4)(4)Z[l]=W[l]A[l−1]+b[l]

where 𝐴[0]=𝑋A[0]=X.

### Exercise 3 - linear\_forward

Build the linear part of forward propagation.

**Reminder**: The mathematical representation of this unit is 𝑍[𝑙]=𝑊[𝑙]𝐴[𝑙−1]+𝑏[𝑙]Z[l]=W[l]A[l−1]+b[l]. You may also find np.dot() useful. If your dimensions don't match, printing W.shape may help.

In [6]:

*# GRADED FUNCTION: linear\_forward*

​

**def** linear\_forward(A, W, b):

"""

Implement the linear part of a layer's forward propagation.

​

Arguments:

A -- activations from previous layer (or input data): (size of previous layer, number of examples)

W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)

b -- bias vector, numpy array of shape (size of the current layer, 1)

​

Returns:

Z -- the input of the activation function, also called pre-activation parameter

cache -- a python tuple containing "A", "W" and "b" ; stored for computing the backward pass efficiently

"""

*#(≈ 1 line of code)*

*# Z = ...*

*# YOUR CODE STARTS HERE*

Z **=** np.dot(W,A) **+** b

*# YOUR CODE ENDS HERE*

cache **=** (A, W, b)

**return** Z, cache

In [7]:

t\_A, t\_W, t\_b **=** linear\_forward\_test\_case()

t\_Z, t\_linear\_cache **=** linear\_forward(t\_A, t\_W, t\_b)

print("Z = " **+** str(t\_Z))

​

linear\_forward\_test(linear\_forward)

Z = [[ 3.26295337 -1.23429987]]

All tests passed.

**Expected output**

Z = [[ 3.26295337 -1.23429987]]

### 4.2 - Linear-Activation Forward

In this notebook, you will use two activation functions:

* **Sigmoid**: 𝜎(𝑍)=𝜎(𝑊𝐴+𝑏)=11+𝑒−(𝑊𝐴+𝑏)σ(Z)=σ(WA+b)=11+e−(WA+b). You've been provided with the sigmoid function which returns **two** items: the activation value "a" and a "cache" that contains "Z" (it's what we will feed in to the corresponding backward function). To use it you could just call:

A, activation\_cache **=** sigmoid(Z)

* **ReLU**: The mathematical formula for ReLu is 𝐴=𝑅𝐸𝐿𝑈(𝑍)=𝑚𝑎𝑥(0,𝑍)A=RELU(Z)=max(0,Z). You've been provided with the relu function. This function returns **two** items: the activation value "A" and a "cache" that contains "Z" (it's what you'll feed in to the corresponding backward function). To use it you could just call:

A, activation\_cache **=** relu(Z)

For added convenience, you're going to group two functions (Linear and Activation) into one function (LINEAR->ACTIVATION). Hence, you'll implement a function that does the LINEAR forward step, followed by an ACTIVATION forward step.

### Exercise 4 - linear\_activation\_forward

Implement the forward propagation of the LINEAR->ACTIVATION layer. Mathematical relation is: 𝐴[𝑙]=𝑔(𝑍[𝑙])=𝑔(𝑊[𝑙]𝐴[𝑙−1]+𝑏[𝑙])A[l]=g(Z[l])=g(W[l]A[l−1]+b[l]) where the activation "g" can be sigmoid() or relu(). Use linear\_forward() and the correct activation function.

In [8]:

*# GRADED FUNCTION: linear\_activation\_forward*

​

**def** linear\_activation\_forward(A\_prev, W, b, activation):

"""

Implement the forward propagation for the LINEAR->ACTIVATION layer

​

Arguments:

A\_prev -- activations from previous layer (or input data): (size of previous layer, number of examples)

W -- weights matrix: numpy array of shape (size of current layer, size of previous layer)

b -- bias vector, numpy array of shape (size of the current layer, 1)

activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

​

Returns:

A -- the output of the activation function, also called the post-activation value

cache -- a python tuple containing "linear\_cache" and "activation\_cache";

stored for computing the backward pass efficiently

"""

**if** activation **==** "sigmoid":

*#(≈ 2 lines of code)*

*# Z, linear\_cache = ...*

*# A, activation\_cache = ...*

*# YOUR CODE STARTS HERE*

Z, linear\_cache **=** linear\_forward(A\_prev, W, b)

A, activation\_cache **=** sigmoid(Z)

*# YOUR CODE ENDS HERE*

**elif** activation **==** "relu":

*#(≈ 2 lines of code)*

*# Z, linear\_cache = ...*

*# A, activation\_cache = ...*

*# YOUR CODE STARTS HERE*

Z, linear\_cache **=** linear\_forward(A\_prev, W, b)

A, activation\_cache **=** relu(Z)

*# YOUR CODE ENDS HERE*

cache **=** (linear\_cache, activation\_cache)

​

**return** A, cache

In [9]:

t\_A\_prev, t\_W, t\_b **=** linear\_activation\_forward\_test\_case()

​

t\_A, t\_linear\_activation\_cache **=** linear\_activation\_forward(t\_A\_prev, t\_W, t\_b, activation **=** "sigmoid")

print("With sigmoid: A = " **+** str(t\_A))

​

t\_A, t\_linear\_activation\_cache **=** linear\_activation\_forward(t\_A\_prev, t\_W, t\_b, activation **=** "relu")

print("With ReLU: A = " **+** str(t\_A))

​

linear\_activation\_forward\_test(linear\_activation\_forward)

With sigmoid: A = [[0.96890023 0.11013289]]

With ReLU: A = [[3.43896131 0. ]]

All tests passed.

**Expected output**

With sigmoid: A = [[0.96890023 0.11013289]]

With ReLU: A = [[3.43896131 0. ]]

**Note**: In deep learning, the "[LINEAR->ACTIVATION]" computation is counted as a single layer in the neural network, not two layers.

### 4.3 - L-Layer Model

For even more convenience when implementing the 𝐿L-layer Neural Net, you will need a function that replicates the previous one (linear\_activation\_forward with RELU) 𝐿−1L−1 times, then follows that with one linear\_activation\_forward with SIGMOID.

**Figure 2** : \*[LINEAR -> RELU] ×× (L-1) -> LINEAR -> SIGMOID\* model

### Exercise 5 - L\_model\_forward

Implement the forward propagation of the above model.

**Instructions**: In the code below, the variable AL will denote 𝐴[𝐿]=𝜎(𝑍[𝐿])=𝜎(𝑊[𝐿]𝐴[𝐿−1]+𝑏[𝐿])A[L]=σ(Z[L])=σ(W[L]A[L−1]+b[L]). (This is sometimes also called Yhat, i.e., this is 𝑌̂ Y^.)

**Hints**:

* Use the functions you've previously written
* Use a for loop to replicate [LINEAR->RELU] (L-1) times
* Don't forget to keep track of the caches in the "caches" list. To add a new value c to a list, you can use list.append(c).

In [10]:

*# GRADED FUNCTION: L\_model\_forward*

​

**def** L\_model\_forward(X, parameters):

"""

Implement forward propagation for the [LINEAR->RELU]\*(L-1)->LINEAR->SIGMOID computation

Arguments:

X -- data, numpy array of shape (input size, number of examples)

parameters -- output of initialize\_parameters\_deep()

Returns:

AL -- activation value from the output (last) layer

caches -- list of caches containing:

every cache of linear\_activation\_forward() (there are L of them, indexed from 0 to L-1)

"""

​

caches **=** []

A **=** X

L **=** len(parameters) **//** 2 *# number of layers in the neural network*

*# Implement [LINEAR -> RELU]\*(L-1). Add "cache" to the "caches" list.*

*# The for loop starts at 1 because layer 0 is the input*

**for** l **in** range(1, L):

A\_prev **=** A

*#(≈ 2 lines of code)*

*# A, cache = ...*

*# caches ...*

*# YOUR CODE STARTS HERE*

A, cache **=** linear\_activation\_forward(A\_prev, parameters['W' **+** str(l)], parameters['b' **+** str(l)], activation **=** 'relu')

caches.append(cache)

*# YOUR CODE ENDS HERE*

*# Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.*

*#(≈ 2 lines of code)*

*# AL, cache = ...*

*# caches ...*

*# YOUR CODE STARTS HERE*

AL, cache **=** linear\_activation\_forward(A, parameters['W' **+** str(L)], parameters['b'**+** str(L)], activation **=** 'sigmoid')

caches.append(cache)

*# YOUR CODE ENDS HERE*

**return** AL, caches

In [11]:

t\_X, t\_parameters **=** L\_model\_forward\_test\_case\_2hidden()

t\_AL, t\_caches **=** L\_model\_forward(t\_X, t\_parameters)

​

print("AL = " **+** str(t\_AL))

​

L\_model\_forward\_test(L\_model\_forward)

AL = [[0.03921668 0.70498921 0.19734387 0.04728177]]

All tests passed.

**Expected output**

AL = [[0.03921668 0.70498921 0.19734387 0.04728177]]

**Awesome!** You've implemented a full forward propagation that takes the input X and outputs a row vector 𝐴[𝐿]A[L] containing your predictions. It also records all intermediate values in "caches". Using 𝐴[𝐿]A[L], you can compute the cost of your predictions.

## 5 - Cost Function

Now you can implement forward and backward propagation! You need to compute the cost, in order to check whether your model is actually learning.

### Exercise 6 - compute\_cost

Compute the cross-entropy cost 𝐽J, using the following formula:

−1𝑚∑𝑖=1𝑚(𝑦(𝑖)log(𝑎[𝐿](𝑖))+(1−𝑦(𝑖))log(1−𝑎[𝐿](𝑖)))(7)(7)−1m∑i=1m(y(i)log⁡(a[L](i))+(1−y(i))log⁡(1−a[L](i)))

In [12]:

*# GRADED FUNCTION: compute\_cost*

​

**def** compute\_cost(AL, Y):

"""

Implement the cost function defined by equation (7).

​

Arguments:

AL -- probability vector corresponding to your label predictions, shape (1, number of examples)

Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number of examples)

​

Returns:

cost -- cross-entropy cost

"""

m **=** Y.shape[1]

​

*# Compute loss from aL and y.*

*# (≈ 1 lines of code)*

*# cost = ...*

*# YOUR CODE STARTS HERE*

abc **=**np.multiply(np.log(AL),Y) **+** np.multiply(np.log(1**-**AL), (1**-**Y))

cost **=** **-**np.sum(abc)**/**m

*# YOUR CODE ENDS HERE*

cost **=** np.squeeze([cost]) *# To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).*

​

**return** cost

In [13]:

t\_Y, t\_AL **=** compute\_cost\_test\_case()

t\_cost **=** compute\_cost(t\_AL, t\_Y)

​

print("Cost: " **+** str(t\_cost))

​

compute\_cost\_test(compute\_cost)

Cost: 0.2797765635793423

All tests passed.

**Expected Output**:

|  |  |
| --- | --- |
| **cost** | 0.2797765635793422 |

## 6 - Backward Propagation Module

Just as you did for the forward propagation, you'll implement helper functions for backpropagation. Remember that backpropagation is used to calculate the gradient of the loss function with respect to the parameters.

**Reminder**:

**Figure 3**: Forward and Backward propagation for LINEAR->RELU->LINEAR->SIGMOID  
*The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.*

Now, similarly to forward propagation, you're going to build the backward propagation in three steps:

1. LINEAR backward
2. LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation
3. [LINEAR -> RELU] ×× (L-1) -> LINEAR -> SIGMOID backward (whole model)

For the next exercise, you will need to remember that:

* b is a matrix(np.ndarray) with 1 column and n rows, i.e: b = [[1.0], [2.0]] (remember that b is a constant)
* np.sum performs a sum over the elements of a ndarray
* axis=1 or axis=0 specify if the sum is carried out by rows or by columns respectively
* keepdims specifies if the original dimensions of the matrix must be kept.
* Look at the following example to clarify:

In [14]:

A **=** np.array([[1, 2], [3, 4]])

​

print('axis=1 and keepdims=True')

print(np.sum(A, axis**=**1, keepdims**=True**))

print('axis=1 and keepdims=False')

print(np.sum(A, axis**=**1, keepdims**=False**))

print('axis=0 and keepdims=True')

print(np.sum(A, axis**=**0, keepdims**=True**))

print('axis=0 and keepdims=False')

print(np.sum(A, axis**=**0, keepdims**=False**))

axis=1 and keepdims=True

[[3]

[7]]

axis=1 and keepdims=False

[3 7]

axis=0 and keepdims=True

[[4 6]]

axis=0 and keepdims=False

[4 6]

### 6.1 - Linear Backward

For layer 𝑙l, the linear part is: 𝑍[𝑙]=𝑊[𝑙]𝐴[𝑙−1]+𝑏[𝑙]Z[l]=W[l]A[l−1]+b[l] (followed by an activation).

Suppose you have already calculated the derivative 𝑑𝑍[𝑙]=∂∂𝑍[𝑙]dZ[l]=∂L∂Z[l]. You want to get (𝑑𝑊[𝑙],𝑑𝑏[𝑙],𝑑𝐴[𝑙−1])(dW[l],db[l],dA[l−1]).

**Figure 4**

The three outputs (𝑑𝑊[𝑙],𝑑𝑏[𝑙],𝑑𝐴[𝑙−1])(dW[l],db[l],dA[l−1]) are computed using the input 𝑑𝑍[𝑙]dZ[l].

Here are the formulas you need:

𝑑𝑊[𝑙]=∂∂𝑊[𝑙]=1𝑚𝑑𝑍[𝑙]𝐴[𝑙−1]𝑇(8)(8)dW[l]=∂J∂W[l]=1mdZ[l]A[l−1]T

𝑑𝑏[𝑙]=∂∂𝑏[𝑙]=1𝑚∑𝑖=1𝑚𝑑𝑍[𝑙](𝑖)(9)(9)db[l]=∂J∂b[l]=1m∑i=1mdZ[l](i)

𝑑𝐴[𝑙−1]=∂∂𝐴[𝑙−1]=𝑊[𝑙]𝑇𝑑𝑍[𝑙](10)(10)dA[l−1]=∂L∂A[l−1]=W[l]TdZ[l]

𝐴[𝑙−1]𝑇A[l−1]T is the transpose of 𝐴[𝑙−1]A[l−1].

### Exercise 7 - linear\_backward

Use the 3 formulas above to implement linear\_backward().

**Hint**:

* In numpy you can get the transpose of an ndarray A using A.T or A.transpose()

In [15]:

*# GRADED FUNCTION: linear\_backward*

​

**def** linear\_backward(dZ, cache):

"""

Implement the linear portion of backward propagation for a single layer (layer l)

​

Arguments:

dZ -- Gradient of the cost with respect to the linear output (of current layer l)

cache -- tuple of values (A\_prev, W, b) coming from the forward propagation in the current layer

​

Returns:

dA\_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A\_prev

dW -- Gradient of the cost with respect to W (current layer l), same shape as W

db -- Gradient of the cost with respect to b (current layer l), same shape as b

"""

A\_prev, W, b **=** cache

m **=** A\_prev.shape[1]

​

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

*# dW = ...*

*# db = ... sum by the rows of dZ with keepdims=True*

*# dA\_prev = ...*

*# YOUR CODE STARTS HERE*

dW **=** (1**/**m)**\***np.dot(dZ,A\_prev.T)

db **=** (1**/**m)**\*** np.sum(dZ, axis **=** 1,keepdims **=** **True** )

dA\_prev **=** np.dot(W.T, dZ)

*# YOUR CODE ENDS HERE*

**return** dA\_prev, dW, db

In [16]:

t\_dZ, t\_linear\_cache **=** linear\_backward\_test\_case()

t\_dA\_prev, t\_dW, t\_db **=** linear\_backward(t\_dZ, t\_linear\_cache)

​

print("dA\_prev: " **+** str(t\_dA\_prev))

print("dW: " **+** str(t\_dW))

print("db: " **+** str(t\_db))

​

linear\_backward\_test(linear\_backward)

dA\_prev: [[-1.15171336 0.06718465 -0.3204696 2.09812712]

[ 0.60345879 -3.72508701 5.81700741 -3.84326836]

[-0.4319552 -1.30987417 1.72354705 0.05070578]

[-0.38981415 0.60811244 -1.25938424 1.47191593]

[-2.52214926 2.67882552 -0.67947465 1.48119548]]

dW: [[ 0.07313866 -0.0976715 -0.87585828 0.73763362 0.00785716]

[ 0.85508818 0.37530413 -0.59912655 0.71278189 -0.58931808]

[ 0.97913304 -0.24376494 -0.08839671 0.55151192 -0.10290907]]

db: [[-0.14713786]

[-0.11313155]

[-0.13209101]]

All tests passed.

**Expected Output**:

dA\_prev: [[-1.15171336 0.06718465 -0.3204696 2.09812712]

[ 0.60345879 -3.72508701 5.81700741 -3.84326836]

[-0.4319552 -1.30987417 1.72354705 0.05070578]

[-0.38981415 0.60811244 -1.25938424 1.47191593]

[-2.52214926 2.67882552 -0.67947465 1.48119548]]

dW: [[ 0.07313866 -0.0976715 -0.87585828 0.73763362 0.00785716]

[ 0.85508818 0.37530413 -0.59912655 0.71278189 -0.58931808]

[ 0.97913304 -0.24376494 -0.08839671 0.55151192 -0.10290907]]

db: [[-0.14713786]

[-0.11313155]

[-0.13209101]]

### 6.2 - Linear-Activation Backward

Next, you will create a function that merges the two helper functions: **linear\_backward** and the backward step for the activation **linear\_activation\_backward**.

To help you implement linear\_activation\_backward, two backward functions have been provided:

* **sigmoid\_backward**: Implements the backward propagation for SIGMOID unit. You can call it as follows:

dZ **=** sigmoid\_backward(dA, activation\_cache)

* **relu\_backward**: Implements the backward propagation for RELU unit. You can call it as follows:

dZ **=** relu\_backward(dA, activation\_cache)

If 𝑔(.)g(.) is the activation function, sigmoid\_backward and relu\_backward compute

𝑑𝑍[𝑙]=𝑑𝐴[𝑙]∗𝑔′(𝑍[𝑙]).(11)(11)dZ[l]=dA[l]∗g′(Z[l]).

### Exercise 8 - linear\_activation\_backward

Implement the backpropagation for the LINEAR->ACTIVATION layer.

In [17]:

*# GRADED FUNCTION: linear\_activation\_backward*

​

**def** linear\_activation\_backward(dA, cache, activation):

"""

Implement the backward propagation for the LINEAR->ACTIVATION layer.

Arguments:

dA -- post-activation gradient for current layer l

cache -- tuple of values (linear\_cache, activation\_cache) we store for computing backward propagation efficiently

activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

Returns:

dA\_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A\_prev

dW -- Gradient of the cost with respect to W (current layer l), same shape as W

db -- Gradient of the cost with respect to b (current layer l), same shape as b

"""

linear\_cache, activation\_cache **=** cache

**if** activation **==** "relu":

*#(≈ 2 lines of code)*

*# dZ = ...*

*# dA\_prev, dW, db = ...*

*# YOUR CODE STARTS HERE*

dZ **=** relu\_backward(dA, activation\_cache)

dA\_prev, dW, db **=** linear\_backward(dZ, linear\_cache)

*# YOUR CODE ENDS HERE*

**elif** activation **==** "sigmoid":

*#(≈ 2 lines of code)*

*# dZ = ...*

*# dA\_prev, dW, db = ...*

*# YOUR CODE STARTS HERE*

dZ **=** sigmoid\_backward(dA, activation\_cache)

dA\_prev, dW, db **=** linear\_backward(dZ, linear\_cache)

*# YOUR CODE ENDS HERE*

**return** dA\_prev, dW, db

In [18]:

t\_dAL, t\_linear\_activation\_cache **=** linear\_activation\_backward\_test\_case()

​

t\_dA\_prev, t\_dW, t\_db **=** linear\_activation\_backward(t\_dAL, t\_linear\_activation\_cache, activation **=** "sigmoid")

print("With sigmoid: dA\_prev = " **+** str(t\_dA\_prev))

print("With sigmoid: dW = " **+** str(t\_dW))

print("With sigmoid: db = " **+** str(t\_db))

​

t\_dA\_prev, t\_dW, t\_db **=** linear\_activation\_backward(t\_dAL, t\_linear\_activation\_cache, activation **=** "relu")

print("With relu: dA\_prev = " **+** str(t\_dA\_prev))

print("With relu: dW = " **+** str(t\_dW))

print("With relu: db = " **+** str(t\_db))

​

linear\_activation\_backward\_test(linear\_activation\_backward)

With sigmoid: dA\_prev = [[ 0.11017994 0.01105339]

[ 0.09466817 0.00949723]

[-0.05743092 -0.00576154]]

With sigmoid: dW = [[ 0.10266786 0.09778551 -0.01968084]]

With sigmoid: db = [[-0.05729622]]

With relu: dA\_prev = [[ 0.44090989 0. ]

[ 0.37883606 0. ]

[-0.2298228 0. ]]

With relu: dW = [[ 0.44513824 0.37371418 -0.10478989]]

With relu: db = [[-0.20837892]]

All tests passed.

**Expected output:**

With sigmoid: dA\_prev = [[ 0.11017994 0.01105339]

[ 0.09466817 0.00949723]

[-0.05743092 -0.00576154]]

With sigmoid: dW = [[ 0.10266786 0.09778551 -0.01968084]]

With sigmoid: db = [[-0.05729622]]

With relu: dA\_prev = [[ 0.44090989 0. ]

[ 0.37883606 0. ]

[-0.2298228 0. ]]

With relu: dW = [[ 0.44513824 0.37371418 -0.10478989]]

With relu: db = [[-0.20837892]]

### 6.3 - L-Model Backward

Now you will implement the backward function for the whole network!

Recall that when you implemented the L\_model\_forward function, at each iteration, you stored a cache which contains (X,W,b, and z). In the back propagation module, you'll use those variables to compute the gradients. Therefore, in the L\_model\_backward function, you'll iterate through all the hidden layers backward, starting from layer 𝐿L. On each step, you will use the cached values for layer 𝑙l to backpropagate through layer 𝑙l. Figure 5 below shows the backward pass.

**Figure 5**: Backward pass

**Initializing backpropagation**:

To backpropagate through this network, you know that the output is: 𝐴[𝐿]=𝜎(𝑍[𝐿])A[L]=σ(Z[L]). Your code thus needs to compute dAL =∂∂𝐴[𝐿]=∂L∂A[L]. To do so, use this formula (derived using calculus which, again, you don't need in-depth knowledge of!):

dAL **=** **-** (np.divide(Y, AL) **-** np.divide(1 **-** Y, 1 **-** AL)) *# derivative of cost with respect to AL*

You can then use this post-activation gradient dAL to keep going backward. As seen in Figure 5, you can now feed in dAL into the LINEAR->SIGMOID backward function you implemented (which will use the cached values stored by the L\_model\_forward function).

After that, you will have to use a for loop to iterate through all the other layers using the LINEAR->RELU backward function. You should store each dA, dW, and db in the grads dictionary. To do so, use this formula :

𝑔𝑟𝑎𝑑𝑠["𝑑𝑊"+𝑠𝑡𝑟(𝑙)]=𝑑𝑊[𝑙](15)(15)grads["dW"+str(l)]=dW[l]

For example, for 𝑙=3l=3 this would store 𝑑𝑊[𝑙]dW[l] in grads["dW3"].

### Exercise 9 - L\_model\_backward

Implement backpropagation for the \*[LINEAR->RELU] ×× (L-1) -> LINEAR -> SIGMOID\* model.

In [40]:

*# GRADED FUNCTION: L\_model\_backward*

​

**def** L\_model\_backward(AL, Y, caches):

"""

Implement the backward propagation for the [LINEAR->RELU] \* (L-1) -> LINEAR -> SIGMOID group

Arguments:

AL -- probability vector, output of the forward propagation (L\_model\_forward())

Y -- true "label" vector (containing 0 if non-cat, 1 if cat)

caches -- list of caches containing:

every cache of linear\_activation\_forward() with "relu" (it's caches[l], for l in range(L-1) i.e l = 0...L-2)

the cache of linear\_activation\_forward() with "sigmoid" (it's caches[L-1])

Returns:

grads -- A dictionary with the gradients

grads["dA" + str(l)] = ...

grads["dW" + str(l)] = ...

grads["db" + str(l)] = ...

"""

grads **=** {}

L **=** len(caches) *# the number of layers*

m **=** AL.shape[1]

Y **=** Y.reshape(AL.shape) *# after this line, Y is the same shape as AL*

*# Initializing the backpropagation*

*#(1 line of code)*

*# dAL = ...*

*# YOUR CODE STARTS HERE*

dAL **=** **-** (np.divide(Y, AL) **-** np.divide(1 **-** Y, 1 **-** AL))

*# YOUR CODE ENDS HERE*

*# Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "dAL, current\_cache". Outputs: "grads["dAL-1"], grads["dWL"], grads["dbL"]*

*#(approx. 5 lines)*

*# current\_cache = ...*

*# dA\_prev\_temp, dW\_temp, db\_temp = ...*

*# grads["dA" + str(L-1)] = ...*

*# grads["dW" + str(L)] = ...*

*# grads["db" + str(L)] = ...*

*# YOUR CODE STARTS HERE*

current\_cache **=** caches[L**-**1]

dA\_prev\_temp, dW\_temp, db\_temp **=** linear\_activation\_backward(dAL, current\_cache, activation **=** "sigmoid")

grads["dA" **+** str(L**-**1)] **=** dA\_prev\_temp

grads["dW" **+** str(L)] **=** dW\_temp

grads["db" **+** str(L)] **=** db\_temp

*# YOUR CODE ENDS HERE*

*# Loop from l=L-2 to l=0*

**for** l **in** reversed(range(L**-**1)):

*# lth layer: (RELU -> LINEAR) gradients.*

*# Inputs: "grads["dA" + str(l + 1)], current\_cache". Outputs: "grads["dA" + str(l)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]*

*#(approx. 5 lines)*

*# current\_cache = ...*

*# dA\_prev\_temp, dW\_temp, db\_temp = ...*

*# grads["dA" + str(l)] = ...*

*# grads["dW" + str(l + 1)] = ...*

*# grads["db" + str(l + 1)] = ...*

*# YOUR CODE STARTS HERE*

current\_cache **=** caches[l]

dA\_prev\_temp, dW\_temp, db\_temp **=** linear\_activation\_backward(grads["dA" **+** str(l **+** 1)], current\_cache, activation **=** "relu")

grads["dA" **+** str(l)] **=** dA\_prev\_temp

grads["dW" **+** str(l **+** 1)] **=** dW\_temp

grads["db" **+** str(l **+** 1)] **=** db\_temp

​

**return** grads

In [41]:

t\_AL, t\_Y\_assess, t\_caches **=** L\_model\_backward\_test\_case()

grads **=** L\_model\_backward(t\_AL, t\_Y\_assess, t\_caches)

​

print("dA0 = " **+** str(grads['dA0']))

print("dA1 = " **+** str(grads['dA1']))

print("dW1 = " **+** str(grads['dW1']))

print("dW2 = " **+** str(grads['dW2']))

print("db1 = " **+** str(grads['db1']))

print("db2 = " **+** str(grads['db2']))

​

L\_model\_backward\_test(L\_model\_backward)

dA0 = [[ 0. 0.52257901]

[ 0. -0.3269206 ]

[ 0. -0.32070404]

[ 0. -0.74079187]]

dA1 = [[ 0.12913162 -0.44014127]

[-0.14175655 0.48317296]

[ 0.01663708 -0.05670698]]

dW1 = [[0.41010002 0.07807203 0.13798444 0.10502167]

[0. 0. 0. 0. ]

[0.05283652 0.01005865 0.01777766 0.0135308 ]]

dW2 = [[-0.39202432 -0.13325855 -0.04601089]]

db1 = [[-0.22007063]

[ 0. ]

[-0.02835349]]

db2 = [[0.15187861]]

All tests passed.

**Expected output:**

dA0 = [[ 0. 0.52257901]

[ 0. -0.3269206 ]

[ 0. -0.32070404]

[ 0. -0.74079187]]

dA1 = [[ 0.12913162 -0.44014127]

[-0.14175655 0.48317296]

[ 0.01663708 -0.05670698]]

dW1 = [[0.41010002 0.07807203 0.13798444 0.10502167]

[0. 0. 0. 0. ]

[0.05283652 0.01005865 0.01777766 0.0135308 ]]

dW2 = [[-0.39202432 -0.13325855 -0.04601089]]

db1 = [[-0.22007063]

[ 0. ]

[-0.02835349]]

db2 = [[0.15187861]]

### 6.4 - Update Parameters

In this section, you'll update the parameters of the model, using gradient descent:

𝑊[𝑙]=𝑊[𝑙]−𝛼 𝑑𝑊[𝑙](16)(16)W[l]=W[l]−α dW[l]

𝑏[𝑙]=𝑏[𝑙]−𝛼 𝑑𝑏[𝑙](17)(17)b[l]=b[l]−α db[l]

where 𝛼α is the learning rate.

After computing the updated parameters, store them in the parameters dictionary.

### Exercise 10 - update\_parameters

Implement update\_parameters() to update your parameters using gradient descent.

**Instructions**: Update parameters using gradient descent on every 𝑊[𝑙]W[l] and 𝑏[𝑙]b[l] for 𝑙=1,2,...,𝐿l=1,2,...,L.

In [42]:

*# GRADED FUNCTION: update\_parameters*

​

**def** update\_parameters(params, grads, learning\_rate):

"""

Update parameters using gradient descent

Arguments:

params -- python dictionary containing your parameters

grads -- python dictionary containing your gradients, output of L\_model\_backward

Returns:

parameters -- python dictionary containing your updated parameters

parameters["W" + str(l)] = ...

parameters["b" + str(l)] = ...

"""

parameters **=** params.copy()

L **=** len(parameters) **//** 2 *# number of layers in the neural network*

​

*# Update rule for each parameter. Use a for loop.*

*#(≈ 2 lines of code)*

**for** l **in** range(L):

*# parameters["W" + str(l+1)] = ...*

*# parameters["b" + str(l+1)] = ...*

*# YOUR CODE STARTS HERE*

parameters["W" **+** str(l**+**1)] **=** parameters["W" **+** str(l**+**1)] **-** learning\_rate **\*** grads["dW" **+** str(l**+**1)]

parameters["b" **+** str(l**+**1)] **=** parameters["b" **+** str(l**+**1)] **-** learning\_rate **\*** grads["db" **+** str(l**+**1)]

*# YOUR CODE ENDS HERE*

**return** parameters

In [43]:

t\_parameters, grads **=** update\_parameters\_test\_case()

t\_parameters **=** update\_parameters(t\_parameters, grads, 0.1)

​

print ("W1 = "**+** str(t\_parameters["W1"]))

print ("b1 = "**+** str(t\_parameters["b1"]))

print ("W2 = "**+** str(t\_parameters["W2"]))

print ("b2 = "**+** str(t\_parameters["b2"]))

​

update\_parameters\_test(update\_parameters)

W1 = [[-0.59562069 -0.09991781 -2.14584584 1.82662008]

[-1.76569676 -0.80627147 0.51115557 -1.18258802]

[-1.0535704 -0.86128581 0.68284052 2.20374577]]

b1 = [[-0.04659241]

[-1.28888275]

[ 0.53405496]]

W2 = [[-0.55569196 0.0354055 1.32964895]]

b2 = [[-0.84610769]]

All tests passed.

**Expected output:**

W1 = [[-0.59562069 -0.09991781 -2.14584584 1.82662008]

[-1.76569676 -0.80627147 0.51115557 -1.18258802]

[-1.0535704 -0.86128581 0.68284052 2.20374577]]

b1 = [[-0.04659241]

[-1.28888275]

[ 0.53405496]]

W2 = [[-0.55569196 0.0354055 1.32964895]]

b2 = [[-0.84610769]]

## Deep Neural Network: Application

## 1 - Packages

Begin by importing all the packages you'll need during this assignment.

* [numpy](https://www.numpy.org/) is the fundamental package for scientific computing with Python.
* [matplotlib](http://matplotlib.org/) is a library to plot graphs in Python.
* [h5py](http://www.h5py.org/) is a common package to interact with a dataset that is stored on an H5 file.
* [PIL](http://www.pythonware.com/products/pil/) and [scipy](https://www.scipy.org/) are used here to test your model with your own picture at the end.
* dnn\_app\_utils provides the functions implemented in the "Building your Deep Neural Network: Step by Step" assignment to this notebook.
* np.random.seed(1) is used to keep all the random function calls consistent. It helps grade your work - so please don't change it!

In [1]:

**import** time

**import** numpy **as** np

**import** h5py

**import** matplotlib.pyplot **as** plt

**import** scipy

**from** PIL **import** Image

**from** scipy **import** ndimage

**from** dnn\_app\_utils\_v3 **import** **\***

**from** public\_tests **import** **\***

​

**%**matplotlib inline

plt.rcParams['figure.figsize'] **=** (5.0, 4.0) *# set default size of plots*

plt.rcParams['image.interpolation'] **=** 'nearest'

plt.rcParams['image.cmap'] **=** 'gray'

​

**%**load\_ext autoreload

**%**autoreload 2

​

np.random.seed(1)

**2 - Load and Process the Dataset**

You'll be using the same "Cat vs non-Cat" dataset as in "Logistic Regression as a Neural Network" (Assignment 2). The model you built back then had 70% test accuracy on classifying cat vs non-cat images. Hopefully, your new model will perform even better!

**Problem Statement**: You are given a dataset ("data.h5") containing:

- a training set of `m\_train` images labelled as cat (1) or non-cat (0)

- a test set of `m\_test` images labelled as cat and non-cat

- each image is of shape (num\_px, num\_px, 3) where 3 is for the 3 channels (RGB).

Let's get more familiar with the dataset. Load the data by running the cell below.

In [2]:

train\_x\_orig, train\_y, test\_x\_orig, test\_y, classes **=** load\_data()

The following code will show you an image in the dataset. Feel free to change the index and re-run the cell multiple times to check out other images.

In [3]:

*# Example of a picture*

index **=** 12

plt.imshow(train\_x\_orig[index])

print ("y = " **+** str(train\_y[0,index]) **+** ". It's a " **+** classes[train\_y[0,index]].decode("utf-8") **+** " picture.")

y = 0. It's a non-cat picture.

A picture containing text

Description automatically generated

In [4]:

*# Explore your dataset*

m\_train **=** train\_x\_orig.shape[0]

num\_px **=** train\_x\_orig.shape[1]

m\_test **=** test\_x\_orig.shape[0]

​

print ("Number of training examples: " **+** str(m\_train))

print ("Number of testing examples: " **+** str(m\_test))

print ("Each image is of size: (" **+** str(num\_px) **+** ", " **+** str(num\_px) **+** ", 3)")

print ("train\_x\_orig shape: " **+** str(train\_x\_orig.shape))

print ("train\_y shape: " **+** str(train\_y.shape))

print ("test\_x\_orig shape: " **+** str(test\_x\_orig.shape))

print ("test\_y shape: " **+** str(test\_y.shape))

Number of training examples: 209

Number of testing examples: 50

Each image is of size: (64, 64, 3)

train\_x\_orig shape: (209, 64, 64, 3)

train\_y shape: (1, 209)

test\_x\_orig shape: (50, 64, 64, 3)

test\_y shape: (1, 50)

As usual, you reshape and standardize the images before feeding them to the network. The code is given in the cell below.

**Figure 1**: Image to vector conversion.

In [5]:

*# Reshape the training and test examples*

train\_x\_flatten **=** train\_x\_orig.reshape(train\_x\_orig.shape[0], **-**1).T *# The "-1" makes reshape flatten the remaining dimensions*

test\_x\_flatten **=** test\_x\_orig.reshape(test\_x\_orig.shape[0], **-**1).T

​

*# Standardize data to have feature values between 0 and 1.*

train\_x **=** train\_x\_flatten**/**255.

test\_x **=** test\_x\_flatten**/**255.

​

print ("train\_x's shape: " **+** str(train\_x.shape))

print ("test\_x's shape: " **+** str(test\_x.shape))

train\_x's shape: (12288, 209)

test\_x's shape: (12288, 50)

**Note**: 12,28812,288 equals 64×64×364×64×3, which is the size of one reshaped image vector.

**3 - Model Architecture**

**3.1 - 2-layer Neural Network**

Now that you're familiar with the dataset, it's time to build a deep neural network to distinguish cat images from non-cat images!

You're going to build two different models:

* A 2-layer neural network
* An L-layer deep neural network

Then, you'll compare the performance of these models, and try out some different values for 𝐿L.

Let's look at the two architectures:

**Figure 2**: 2-layer neural network.  
The model can be summarized as: INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT.

**Detailed Architecture of Figure 2**:

* The input is a (64,64,3) image which is flattened to a vector of size (12288,1)(12288,1).
* The corresponding vector: [𝑥0,𝑥1,...,𝑥12287]𝑇[x0,x1,...,x12287]T is then multiplied by the weight matrix 𝑊[1]W[1] of size (𝑛[1],12288)(n[1],12288).
* Then, add a bias term and take its relu to get the following vector: [𝑎[1]0,𝑎[1]1,...,𝑎[1]𝑛[1]−1]𝑇[a0[1],a1[1],...,an[1]−1[1]]T.
* Repeat the same process.
* Multiply the resulting vector by 𝑊[2]W[2] and add the intercept (bias).
* Finally, take the sigmoid of the result. If it's greater than 0.5, classify it as a cat.

**3.2 - L-layer Deep Neural Network**

It's pretty difficult to represent an L-layer deep neural network using the above representation. However, here is a simplified network representation:

**Figure 3**: L-layer neural network.  
The model can be summarized as: [LINEAR -> RELU] ×× (L-1) -> LINEAR -> SIGMOID

**Detailed Architecture of Figure 3**:

* The input is a (64,64,3) image which is flattened to a vector of size (12288,1).
* The corresponding vector: [𝑥0,𝑥1,...,𝑥12287]𝑇[x0,x1,...,x12287]T is then multiplied by the weight matrix 𝑊[1]W[1] and then you add the intercept 𝑏[1]b[1]. The result is called the linear unit.
* Next, take the relu of the linear unit. This process could be repeated several times for each (𝑊[𝑙],𝑏[𝑙])(W[l],b[l]) depending on the model architecture.
* Finally, take the sigmoid of the final linear unit. If it is greater than 0.5, classify it as a cat.

**3.3 - General Methodology**

As usual, you'll follow the Deep Learning methodology to build the model:

1. Initialize parameters / Define hyperparameters
2. Loop for num\_iterations: a. Forward propagation b. Compute cost function c. Backward propagation d. Update parameters (using parameters, and grads from backprop)
3. Use trained parameters to predict labels

Now go ahead and implement those two models!

**4 - Two-layer Neural Network**

**Exercise 1 - two\_layer\_model**

Use the helper functions you have implemented in the previous assignment to build a 2-layer neural network with the following structure: *LINEAR -> RELU -> LINEAR -> SIGMOID*. The functions and their inputs are:

**def** initialize\_parameters(n\_x, n\_h, n\_y):

**...**

**return** parameters

**def** linear\_activation\_forward(A\_prev, W, b, activation):

**...**

**return** A, cache

**def** compute\_cost(AL, Y):

**...**

**return** cost

**def** linear\_activation\_backward(dA, cache, activation):

**...**

**return** dA\_prev, dW, db

**def** update\_parameters(parameters, grads, learning\_rate):

**...**

**return** parameters

In [6]:

*### CONSTANTS DEFINING THE MODEL ####*

n\_x **=** 12288 *# num\_px \* num\_px \* 3*

n\_h **=** 7

n\_y **=** 1

layers\_dims **=** (n\_x, n\_h, n\_y)

learning\_rate **=** 0.0075

In [7]:

*# GRADED FUNCTION: two\_layer\_model*

​

**def** two\_layer\_model(X, Y, layers\_dims, learning\_rate **=** 0.0075, num\_iterations **=** 3000, print\_cost**=False**):

"""

Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

Arguments:

X -- input data, of shape (n\_x, number of examples)

Y -- true "label" vector (containing 1 if cat, 0 if non-cat), of shape (1, number of examples)

layers\_dims -- dimensions of the layers (n\_x, n\_h, n\_y)

num\_iterations -- number of iterations of the optimization loop

learning\_rate -- learning rate of the gradient descent update rule

print\_cost -- If set to True, this will print the cost every 100 iterations

Returns:

parameters -- a dictionary containing W1, W2, b1, and b2

"""

np.random.seed(1)

grads **=** {}

costs **=** [] *# to keep track of the cost*

m **=** X.shape[1] *# number of examples*

(n\_x, n\_h, n\_y) **=** layers\_dims

*# Initialize parameters dictionary, by calling one of the functions you'd previously implemented*

*#(≈ 1 line of code)*

*# parameters = ...*

*# YOUR CODE STARTS HERE*

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

*# YOUR CODE ENDS HERE*

*# Get W1, b1, W2 and b2 from the dictionary parameters.*

W1 **=** parameters["W1"]

b1 **=** parameters["b1"]

W2 **=** parameters["W2"]

b2 **=** parameters["b2"]

*# Loop (gradient descent)*

​

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

​

*# Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2".*

*#(≈ 2 lines of code)*

*# A1, cache1 = ...*

*# A2, cache2 = ...*

*# YOUR CODE STARTS HERE*

A1, cache1 **=** linear\_activation\_forward(X, W1, b1, activation **=** "relu")

A2, cache2 **=** linear\_activation\_forward(A1, W2, b2, activation **=** "sigmoid")

*# YOUR CODE ENDS HERE*

*# Compute cost*

*#(≈ 1 line of code)*

*# cost = ...*

*# YOUR CODE STARTS HERE*

cost **=** compute\_cost(A2, Y)

*# YOUR CODE ENDS HERE*

*# Initializing backward propagation*

dA2 **=** **-** (np.divide(Y, A2) **-** np.divide(1 **-** Y, 1 **-** A2))

*# Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also dA0 (not used), dW1, db1".*

*#(≈ 2 lines of code)*

*# dA1, dW2, db2 = ...*

*# dA0, dW1, db1 = ...*

*# YOUR CODE STARTS HERE*

dA1, dW2, db2 **=** linear\_activation\_backward(dA2, cache2, activation**=** "sigmoid")

dA0, dW1, db1 **=** linear\_activation\_backward(dA1, cache1, activation **=** "relu")

*# YOUR CODE ENDS HERE*

*# Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] to db2*

grads['dW1'] **=** dW1

grads['db1'] **=** db1

grads['dW2'] **=** dW2

grads['db2'] **=** db2

*# Update parameters.*

*#(approx. 1 line of code)*

*# parameters = ...*

*# YOUR CODE STARTS HERE*

parameters **=** update\_parameters(parameters, grads, learning\_rate)

*# YOUR CODE ENDS HERE*

​

*# Retrieve W1, b1, W2, b2 from parameters*

W1 **=** parameters["W1"]

b1 **=** parameters["b1"]

W2 **=** parameters["W2"]

b2 **=** parameters["b2"]

*# Print the cost every 100 iterations*

**if** print\_cost **and** i **%** 100 **==** 0 **or** i **==** num\_iterations **-** 1:

print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))

**if** i **%** 100 **==** 0 **or** i **==** num\_iterations:

costs.append(cost)

​

**return** parameters, costs

​

**def** plot\_costs(costs, learning\_rate**=**0.0075):

plt.plot(np.squeeze(costs))

plt.ylabel('cost')

plt.xlabel('iterations (per hundreds)')

plt.title("Learning rate =" **+** str(learning\_rate))

plt.show()

In [8]:

parameters, costs **=** two\_layer\_model(train\_x, train\_y, layers\_dims **=** (n\_x, n\_h, n\_y), num\_iterations **=** 2, print\_cost**=False**)

​

print("Cost after first iteration: " **+** str(costs[0]))

​

two\_layer\_model\_test(two\_layer\_model)

Cost after iteration 1: 0.6926114346158595

Cost after first iteration: 0.693049735659989

Cost after iteration 1: 0.6915746967050506

Cost after iteration 1: 0.6915746967050506

Cost after iteration 1: 0.6915746967050506

Cost after iteration 2: 0.6524135179683452

All tests passed.

**Expected output:**

cost after iteration 1 must be around 0.69

**4.1 - Train the model**

If your code passed the previous cell, run the cell below to train your parameters.

* The cost should decrease on every iteration.
* It may take up to 5 minutes to run 2500 iterations.

In [9]:

parameters, costs **=** two\_layer\_model(train\_x, train\_y, layers\_dims **=** (n\_x, n\_h, n\_y), num\_iterations **=** 2500, print\_cost**=True**)

plot\_costs(costs, learning\_rate)

Cost after iteration 0: 0.693049735659989

Cost after iteration 100: 0.6464320953428849

Cost after iteration 200: 0.6325140647912677

Cost after iteration 300: 0.6015024920354665

Cost after iteration 400: 0.5601966311605747

Cost after iteration 500: 0.5158304772764729

Cost after iteration 600: 0.4754901313943325

Cost after iteration 700: 0.43391631512257495

Cost after iteration 800: 0.4007977536203886

Cost after iteration 900: 0.3580705011323798

Cost after iteration 1000: 0.3394281538366413

Cost after iteration 1100: 0.30527536361962654

Cost after iteration 1200: 0.2749137728213015

Cost after iteration 1300: 0.2468176821061484

Cost after iteration 1400: 0.19850735037466102

Cost after iteration 1500: 0.17448318112556638

Cost after iteration 1600: 0.1708076297809692

Cost after iteration 1700: 0.11306524562164715

Cost after iteration 1800: 0.09629426845937156

Cost after iteration 1900: 0.0834261795972687

Cost after iteration 2000: 0.07439078704319085

Cost after iteration 2100: 0.06630748132267933

Cost after iteration 2200: 0.05919329501038172

Cost after iteration 2300: 0.053361403485605606

Cost after iteration 2400: 0.04855478562877019

Cost after iteration 2499: 0.04421498215868956

Chart, line chart

Description automatically generated

**Expected Output**:

|  |  |
| --- | --- |
| **Cost after iteration 0** | 0.6930497356599888 |
| **Cost after iteration 100** | 0.6464320953428849 |
| **...** | ... |
| **Cost after iteration 2499** | 0.04421498215868956 |

**Nice!** You successfully trained the model. Good thing you built a vectorized implementation! Otherwise it might have taken 10 times longer to train this.

Now, you can use the trained parameters to classify images from the dataset. To see your predictions on the training and test sets, run the cell below.

In [10]:

predictions\_train **=** predict(train\_x, train\_y, parameters)

Accuracy: 0.9999999999999998

**Expected Output**:

|  |  |
| --- | --- |
| **Accuracy** | 0.9999999999999998 |

In [11]:

predictions\_test **=** predict(test\_x, test\_y, parameters)

Accuracy: 0.72

**Expected Output**:

|  |  |
| --- | --- |
| **Accuracy** | 0.72 |

**Congratulations! It seems that your 2-layer neural network has better performance (72%) than the logistic regression implementation (70%, assignment week 2). Let's see if you can do even better with an**𝐿L**-layer model.**

**Note**: You may notice that running the model on fewer iterations (say 1500) gives better accuracy on the test set. This is called "early stopping" and you'll hear more about it in the next course. Early stopping is a way to prevent overfitting.

**5 - L-layer Neural Network**

**Exercise 2 - L\_layer\_model**

Use the helper functions you implemented previously to build an 𝐿L-layer neural network with the following structure: \*[LINEAR -> RELU]××(L-1) -> LINEAR -> SIGMOID\*. The functions and their inputs are:

**def** initialize\_parameters\_deep(layers\_dims):

**...**

**return** parameters

**def** L\_model\_forward(X, parameters):

**...**

**return** AL, caches

**def** compute\_cost(AL, Y):

**...**

**return** cost

**def** L\_model\_backward(AL, Y, caches):

**...**

**return** grads

**def** update\_parameters(parameters, grads, learning\_rate):

**...**

**return** parameters

In [12]:

*### CONSTANTS ###*

layers\_dims **=** [12288, 20, 7, 5, 1] *# 4-layer model*

In [13]:

*# GRADED FUNCTION: L\_layer\_model*

​

**def** L\_layer\_model(X, Y, layers\_dims, learning\_rate **=** 0.0075, num\_iterations **=** 3000, print\_cost**=False**):

"""

Implements a L-layer neural network: [LINEAR->RELU]\*(L-1)->LINEAR->SIGMOID.

Arguments:

X -- data, numpy array of shape (num\_px \* num\_px \* 3, number of examples)

Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of examples)

layers\_dims -- list containing the input size and each layer size, of length (number of layers + 1).

learning\_rate -- learning rate of the gradient descent update rule

num\_iterations -- number of iterations of the optimization loop

print\_cost -- if True, it prints the cost every 100 steps

Returns:

parameters -- parameters learnt by the model. They can then be used to predict.

"""

​

np.random.seed(1)

costs **=** [] *# keep track of cost*

*# Parameters initialization.*

*#(≈ 1 line of code)*

*# parameters = ...*

*# YOUR CODE STARTS HERE*

parameters **=** initialize\_parameters\_deep(layers\_dims)

*# YOUR CODE ENDS HERE*

*# Loop (gradient descent)*

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

​

*# Forward propagation: [LINEAR -> RELU]\*(L-1) -> LINEAR -> SIGMOID.*

*#(≈ 1 line of code)*

*# AL, caches = ...*

*# YOUR CODE STARTS HERE*

AL, caches **=** L\_model\_forward(X, parameters)

*# YOUR CODE ENDS HERE*

*# Compute cost.*

*#(≈ 1 line of code)*

*# cost = ...*

*# YOUR CODE STARTS HERE*

cost **=** compute\_cost(AL, Y)

*# YOUR CODE ENDS HERE*

*# Backward propagation.*

*#(≈ 1 line of code)*

*# grads = ...*

*# YOUR CODE STARTS HERE*

grads **=** L\_model\_backward(AL, Y, caches)

*# YOUR CODE ENDS HERE*

*# Update parameters.*

*#(≈ 1 line of code)*

*# parameters = ...*

*# YOUR CODE STARTS HERE*

parameters **=** update\_parameters(parameters, grads, learning\_rate)

*# YOUR CODE ENDS HERE*

*# Print the cost every 100 iterations*

**if** print\_cost **and** i **%** 100 **==** 0 **or** i **==** num\_iterations **-** 1:

print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))

**if** i **%** 100 **==** 0 **or** i **==** num\_iterations:

costs.append(cost)

**return** parameters, costs

In [14]:

parameters, costs **=** L\_layer\_model(train\_x, train\_y, layers\_dims, num\_iterations **=** 1, print\_cost **=** **False**)

​

print("Cost after first iteration: " **+** str(costs[0]))

​

L\_layer\_model\_test(L\_layer\_model)

Cost after iteration 0: 0.7717493284237686

Cost after first iteration: 0.7717493284237686

Cost after iteration 1: 0.7070709008912569

Cost after iteration 1: 0.7070709008912569

Cost after iteration 1: 0.7070709008912569

Cost after iteration 2: 0.7063462654190897

All tests passed.

**5.1 - Train the model**

If your code passed the previous cell, run the cell below to train your model as a 4-layer neural network.

* The cost should decrease on every iteration.
* It may take up to 5 minutes to run 2500 iterations.

In [15]:

parameters, costs **=** L\_layer\_model(train\_x, train\_y, layers\_dims, num\_iterations **=** 2500, print\_cost **=** **True**)

Cost after iteration 0: 0.7717493284237686

Cost after iteration 100: 0.6720534400822914

Cost after iteration 200: 0.6482632048575212

Cost after iteration 300: 0.6115068816101356

Cost after iteration 400: 0.5670473268366111

Cost after iteration 500: 0.5401376634547801

Cost after iteration 600: 0.5279299569455267

Cost after iteration 700: 0.4654773771766851

Cost after iteration 800: 0.369125852495928

Cost after iteration 900: 0.39174697434805344

Cost after iteration 1000: 0.31518698886006163

Cost after iteration 1100: 0.2726998441789385

Cost after iteration 1200: 0.23741853400268137

Cost after iteration 1300: 0.19960120532208644

Cost after iteration 1400: 0.18926300388463307

Cost after iteration 1500: 0.16118854665827753

Cost after iteration 1600: 0.14821389662363316

Cost after iteration 1700: 0.13777487812972944

Cost after iteration 1800: 0.1297401754919012

Cost after iteration 1900: 0.12122535068005211

Cost after iteration 2000: 0.11382060668633713

Cost after iteration 2100: 0.10783928526254133

Cost after iteration 2200: 0.10285466069352679

Cost after iteration 2300: 0.10089745445261786

Cost after iteration 2400: 0.09287821526472398

Cost after iteration 2499: 0.08843994344170202

**Expected Output**:

|  |  |
| --- | --- |
| **Cost after iteration 0** | 0.771749 |
| **Cost after iteration 100** | 0.672053 |
| **...** | ... |
| **Cost after iteration 2499** | 0.088439 |

In [16]:

pred\_train **=** predict(train\_x, train\_y, parameters)

Accuracy: 0.9856459330143539

**Expected Output**:

|  |  |
| --- | --- |
| **Train Accuracy** | 0.985645933014 |

In [17]:

pred\_test **=** predict(test\_x, test\_y, parameters)

Accuracy: 0.8

**Expected Output**:

|  |  |
| --- | --- |
| **Test Accuracy** | 0.8 |

**Congrats! It seems that your 4-layer neural network has better performance (80%) than your 2-layer neural network (72%) on the same test set.**

This is pretty good performance for this task. Nice job!

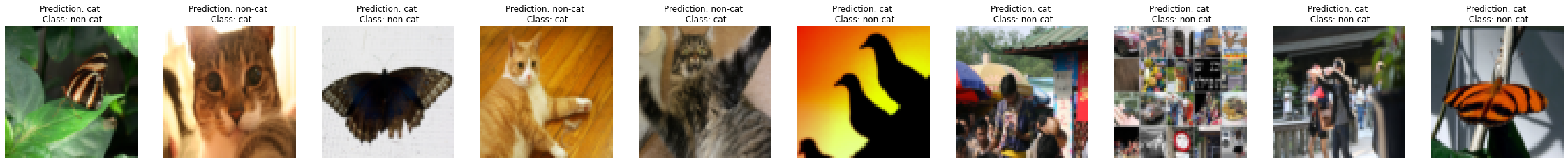
In the next course on "Improving deep neural networks," you'll be able to obtain even higher accuracy by systematically searching for better hyperparameters: learning\_rate, layers\_dims, or num\_iterations, for example.

**6 - Results Analysis**

First, take a look at some images the L-layer model labeled incorrectly. This will show a few mislabeled images.

In [18]:

print\_mislabeled\_images(classes, test\_x, test\_y, pred\_test)



**A few types of images the model tends to do poorly on include:**

* Cat body in an unusual position
* Cat appears against a background of a similar color
* Unusual cat color and species
* Camera Angle
* Brightness of the picture
* Scale variation (cat is very large or small in image)

**7 - Test with your own image (optional/ungraded exercise)**

From this point, if you so choose, you can use your own image to test the output of your model. To do that follow these steps:

1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.
2. Add your image to this Jupyter Notebook's directory, in the "images" folder
3. Change your image's name in the following code
4. Run the code and check if the algorithm is right (1 = cat, 0 = non-cat)!

In [21]:

*## START CODE HERE ##*

my\_image **=** "my\_image.jpg" *# change this to the name of your image file*

my\_label\_y **=** [1] *# the true class of your image (1 -> cat, 0 -> non-cat)*

*## END CODE HERE ##*

​

fname **=** "images/" **+** my\_image

image **=** np.array(Image.open(fname).resize((num\_px, num\_px)))

plt.imshow(image)

image **=** image **/** 255.

image **=** image.reshape((1, num\_px **\*** num\_px **\*** 3)).T

​

my\_predicted\_image **=** predict(image, my\_label\_y, parameters)

​

​

print ("y = " **+** str(np.squeeze(my\_predicted\_image)) **+** ", your L-layer model predicts a \"" **+** classes[int(np.squeeze(my\_predicted\_image)),].decode("utf-8") **+** "\" picture.")

Accuracy: 1.0

y = 1.0, your L-layer model predicts a "cat" picture.