Deep Neural Network for Image Classification: Application

You will use the functions you'd implemented in the Part 1 of the assignment to build a deep network, and apply it to cat vs non-cat classification. Hopefully, you will see an improvement in accuracy relative to your previous logistic regression implementation.

After this assignment you will be able to:

- Build and apply a deep neural network to supervised learning.

Let's get started!

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.
- 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 will help us grade your work.

In [1]:
```
import time
import numpy as np
import h5py
import matplotlib.pyplot as plt
import matplotlib.image as img
import scipy
from PIL import Image
from scipy import ndimage
#from dnn_app_utils_v3 import *

%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plo
ts
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

%load_ext autoreload
%autoreload 2

np.random.seed(1)
```

2 - Dataset

You will use the same "Cat vs non-Cat" dataset as in "Logistic Regression" Assignment (Assignment 1). The model you had built had 70% test accuracy on classifying cats vs non-cats images. Hopefully, your new model will perform a 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.

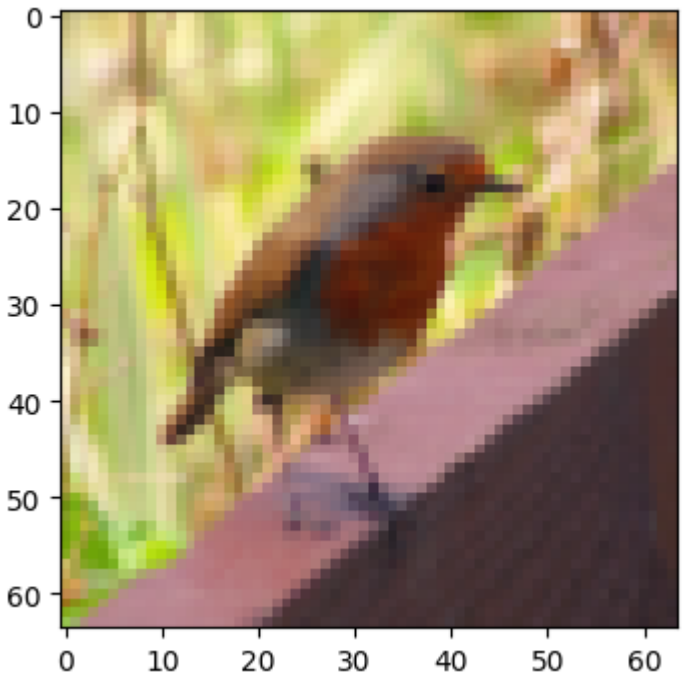
In [2]:
```
def load_data():
    train_dataset = h5py.File('../../data/train_catvnoncat.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # yo
ur train set features
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # yo
ur train set labels

    test_dataset = h5py.File('../../data/test_catvnoncat.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your
 test set features
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your
 test set labels

    classes = np.array(test_dataset["list_classes"][:]) # the list of
classes

    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.
shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.sha
pe[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_
set_y_orig, classes
```

In [3]:
```
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 see other images.

In [4]:
```
# Example of a picture
index = 10
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.
```



In [5]:
```
# 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.

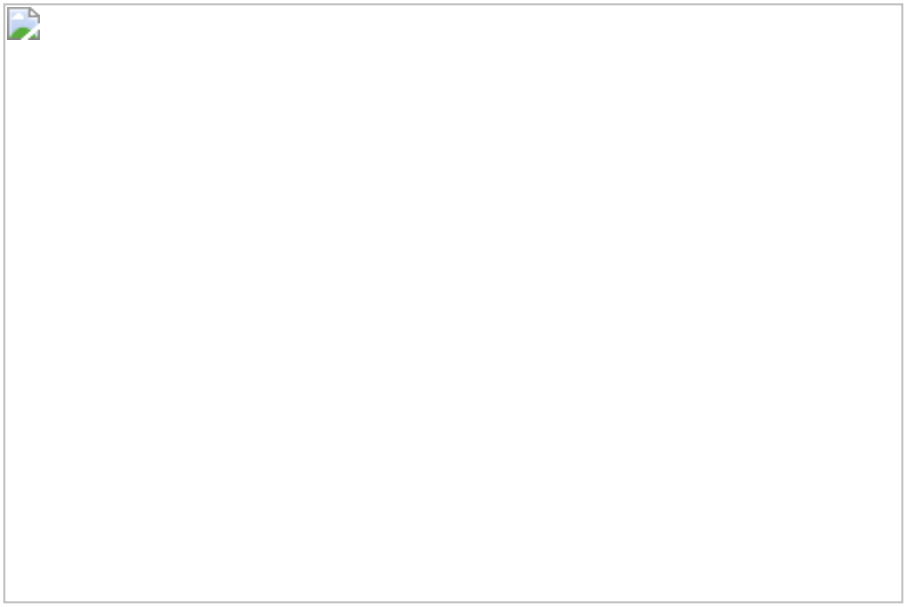


Figure 1: Image to vector conversion.

Exercise 1: Flattern and array the samples in columns

In [6]:
```
# 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)
```

$12,288$ equals $64 \times 64 \times 3$ which is the size of one reshaped image vector.

3 - Architecture of your model

Now that you are familiar with the dataset, it is time to build a deep neural network to distinguish cat images from non-cat images.

You will build two different models:

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

You will then compare the performance of these models, and also try out different values for L.

Let's look at the two architectures.

3.1 - 2-layer neural network

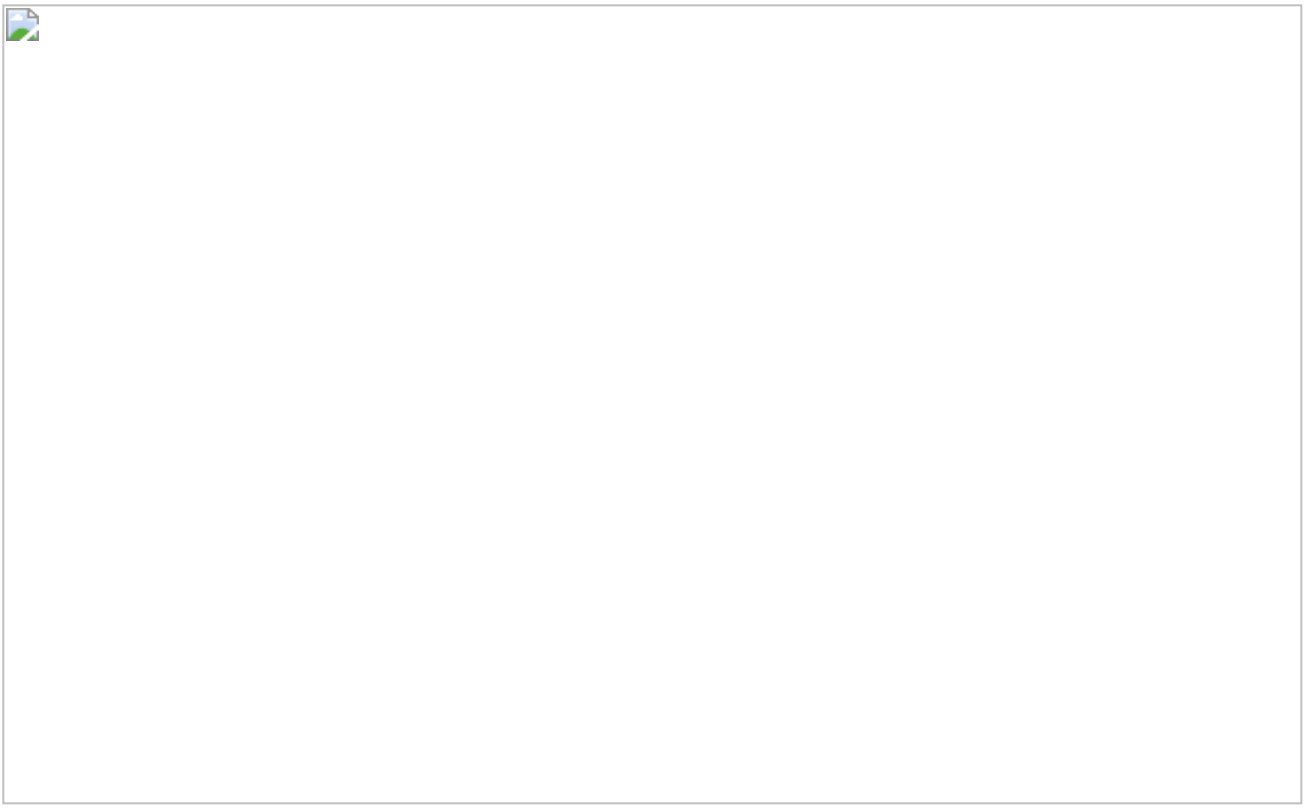


Figure 2: 2-layer neural network.
The model can be summarized as: ***INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT***.

Detailed Architecture of figure 2:

- The input is a (64,64,3) image which is flattened to a vector of size $(12288, 1)$.
- The corresponding vector: $[x_0, x_1, \ldots, x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ of size $(n^{[1]}, 12288)$.
- You then add a bias term and take its relu to get the following vector: $[a_0^{[1]}, a_1^{[1]}, \ldots, a_{n^{[1]}-1}^{[1]}]^T$.
- You then repeat the same process.
- You multiply the resulting vector by $W^{[2]}$ and add your intercept (bias).
- Finally, you take the sigmoid of the result. If it is greater than 0.5, you classify it to be a cat.

3.2 - L-layer deep neural network

It is hard to represent an L-layer deep neural network with the above representation. However, here is a simplified network representation:

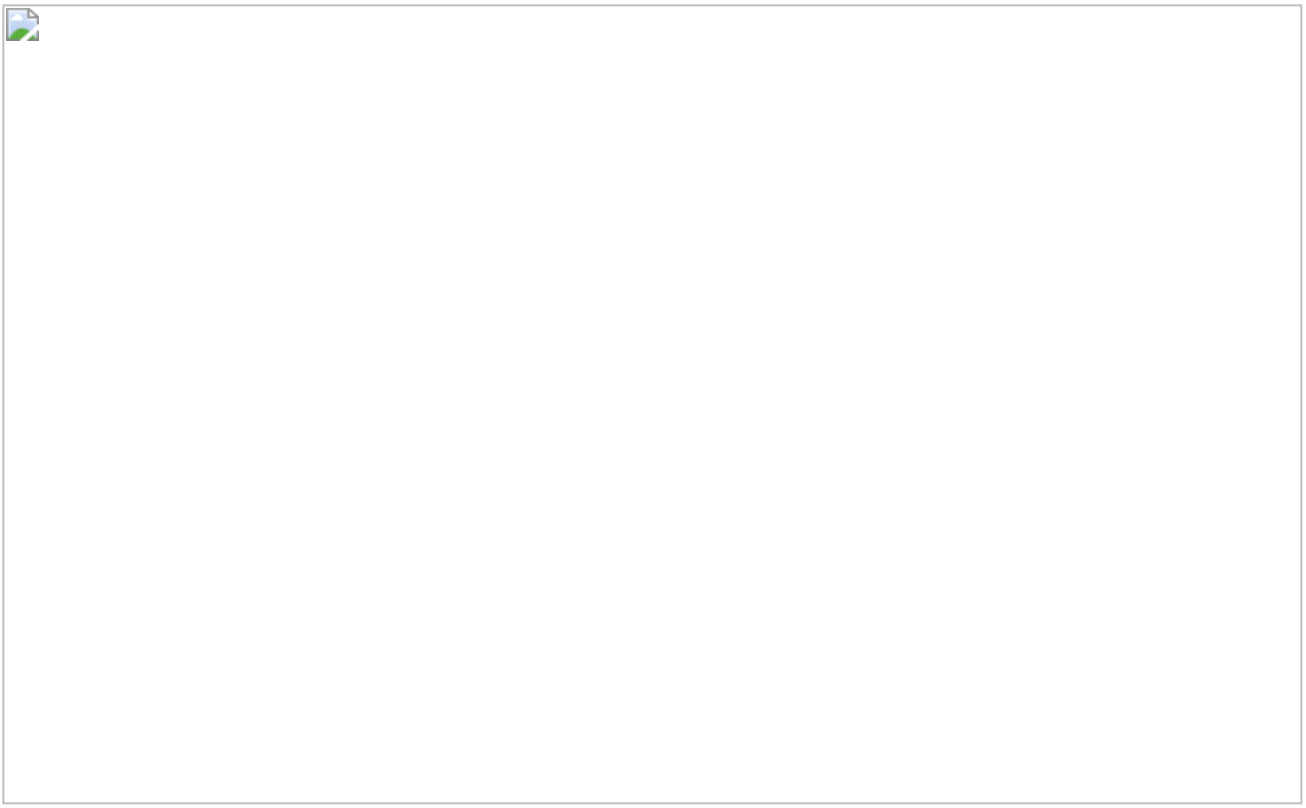


Figure 3: L-layer neural network.
The model can be summarized as: ***[LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID***

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: $[x_0, x_1, \ldots, x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ and then you add the intercept $b^{[1]}$. The result is called the linear unit.
- Next, you take the relu of the linear unit. This process could be repeated several times for each $(W^{[l]}, b^{[l]})$ depending on the model architecture.
- Finally, you take the sigmoid of the final linear unit. If it is greater than 0.5, you classify it to be a cat.

3.3 - General methodology

As usual you will 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

Let's now implement those two models!

4 - Two-layer neural network

Exercise 2: 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 you may need 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 [7]:
```
### 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)
```

In [8]:
```
def linear_activation_forward(A_prev, W, b, activation):
    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 o
f previous layer, number of examples)
        W -- weights matrix: numpy array of shape (size of current la
yer, size of previous layer)
        b -- bias vector, numpy array of shape (size of the current l
ayer, 1)

        Returns:
        Z -- the input of the activation function, also called pre-ac
tivation parameter
        cache -- a python tuple containing "A", "W" and "b" ; stored
 for computing the backward pass efficiently
        """

        ### START CODE HERE ###
        Z = np.dot(W, A) + b
        ### END CODE HERE ###

        assert(Z.shape == (W.shape[0], A.shape[1]))
        cache = (A, W, b)

        return Z, cache

    def sigmoid(Z):
        """
        Implements the sigmoid activation in numpy

        Arguments:
        Z -- numpy array of any shape

        Returns:
        A -- output of sigmoid(z), same shape as Z
        cache -- returns Z as well, useful during backpropagation
        """

        A = 1/(1+np.exp(-Z))
        cache = Z

        return A, cache

    def relu(Z):
        """
        Implement the RELU function.
        Arguments:
        Z -- Output of the linear layer, of any shape
        Returns:
        A -- Post-activation parameter, of the same shape as Z
        cache -- a python dictionary containing "A" ; stored for comp
uting the backward pass efficiently
        """
```

```
        A = np.maximum(0,Z)

        assert(A.shape == Z.shape)

        cache = Z
        return A, cache

    """
    Implement the forward propagation for the LINEAR->ACTIVATION laye
r

    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 laye
r, 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":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ###
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
        ### END CODE HERE ###

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### (≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)
        ### END CODE HERE ###

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache
```

In [9]:
```
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 p
ropagation in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (o
f the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), s
ame shape as W
    db -- Gradient of the cost with respect to b (current layer l), s
ame shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]
    ### START CODE HERE ###
    dA_prev = np.dot(W.T, dZ)
    dW = np.dot(dZ, A_prev.T) / m
    db = np.reshape(np.sum(dZ, axis = 1) / m, b.shape)
    ### END CODE HERE ###

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db

def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.
    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation ef
ficiently
    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct ob
ject.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):
```

```
    """
    Implement the backward propagation for a single SIGMOID unit.
    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation ef
ficiently
    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)

    return dZ
def linear_activation_backward(dA, cache, activation):

    """
    Implement the backward propagation for the LINEAR->ACTIVATION lay
er.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we stor
e 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 (o
f the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), s
ame shape as W
    db -- Gradient of the cost with respect to b (current layer l), s
ame shape as b
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        ### START CODE HERE ###
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    elif activation == "sigmoid":
        ### START CODE HERE ###
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    return dA_prev, dW, db
```

In [10]:
```
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)
    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))
    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))
    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}
    return parameters
```

In [11]:
```
def compute_cost(AL, Y):
    """
    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.
    ### START CODE HERE ###
    cost = -np.sum((Y * np.log(AL) + (1 - Y) * np.log(1 - AL)))/m
    #cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))
    ### END CODE HERE ###

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

    return cost

def update_parameters(parameters, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- 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)] = ...
    """

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

    # Update rule for each parameter. Use a for loop.
    ### START CODE HERE ###
    for l in range(L):
        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)]
    ### END CODE HERE ###
    return parameters
```

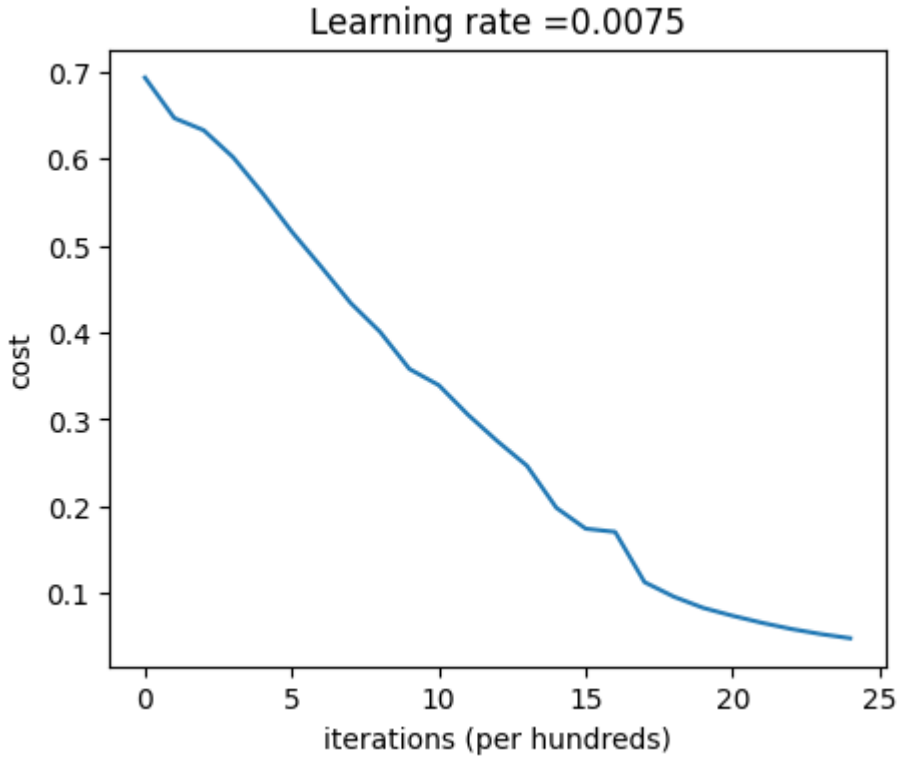
In [12]:
```
# GRADED FUNCTION: two_layer_model
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_it
erations = 3000, print_cost=False):
    """
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGM
OID.

    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 rul
e
    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 co
st
    m = X.shape[1]                                 # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functio
ns you'd previously implemented
    ### START CODE HERE ###
    parameters = initialize_parameters(n_x, n_h, n_y)
    ### END CODE 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. I
nputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2".
        ### START CODE HERE ### (≈ 2 lines of code)
        A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")
        ### END CODE HERE ###

        # Compute cost
        ### START CODE HERE ###
        cost = compute_cost(A2, Y)
        ### END CODE HERE ###
```

```
        # Initializing backward propagation
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Output
s: "dA1, dW2, db2; also dA0 (not used), dW1, db1".
        ### START CODE HERE ###
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, "sigm
oid")
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "rel
u")
        ### END CODE 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.
        ### START CODE HERE ###
        parameters = update_parameters(parameters, grads, learning_ra
te)
        ### END CODE 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 training example
        if print_cost and i % 100 == 0:
            print("Cost after iteration {}: {}".format(i, np.squeeze(
cost)))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per hundreds)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

Run the cell below to train your parameters. See if your model runs. The cost should be decreasing. It may take up to 5 minutes to run 2500 iterations. Check if the "Cost after iteration 0" matches the expected output below, if not click on the square (■) on the upper bar of the notebook to stop the cell and try to find your error.

In [13]:
```
parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_
h, n_y), num_iterations = 2500, print_cost=True)
```

```
Cost after iteration 0: 0.6930497356599891
Cost after iteration 100: 0.6464320953428849
Cost after iteration 200: 0.6325140647912677
Cost after iteration 300: 0.6015024920354665
Cost after iteration 400: 0.5601966311605748
Cost after iteration 500: 0.515830477276473
Cost after iteration 600: 0.4754901313943326
Cost after iteration 700: 0.4339163151225749
Cost after iteration 800: 0.400797753620389
Cost after iteration 900: 0.3580705011323798
Cost after iteration 1000: 0.3394281538366412
Cost after iteration 1100: 0.30527536361962637
Cost after iteration 1200: 0.27491377282130197
Cost after iteration 1300: 0.24681768210614835
Cost after iteration 1400: 0.19850735037466116
Cost after iteration 1500: 0.17448318112556643
Cost after iteration 1600: 0.17080762978095307
Cost after iteration 1700: 0.11306524562164758
Cost after iteration 1800: 0.09629426845937161
Cost after iteration 1900: 0.0834261795972686
Cost after iteration 2000: 0.07439078704319077
Cost after iteration 2100: 0.06630748132267925
Cost after iteration 2200: 0.05919329501038164
Cost after iteration 2300: 0.0533614034856055
Cost after iteration 2400: 0.04855478562877013
```



Expected Output:

Cost after iteration 0	0.69...
Cost after iteration 100	0.64...
...	...
Cost after iteration 2400	0.04...

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 [14]:
```
def predict(X, y, parameters):
    m = X.shape[1]
    A1, _ = linear_activation_forward(X, parameters['W1'], parameters
['b1'], "relu")
    p, _ = linear_activation_forward(A1, parameters['W2'], parameters
['b2'], "sigmoid")
    p = np.where(p >= 0.5, 1, 0)
    p = np.reshape(p, y.shape)
    print("Accuracy: "  + str(np.sum((p == y))/m))
    return p
predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 1.0
```

Expected Output:

Accuracy	1.0

In [15]:
```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.72
```

Expected Output:

Accuracy	0.72

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 we will talk about it in the next course. Early stopping is a way to prevent overfitting.

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 L-layer model.

5 - L-layer Neural Network

Exercise 3: Use the helper functions you have implemented previously to build an L-layer neural network with the following structure: *[LINEAR -> RELU]$\times$(L-1) -> LINEAR -> SIGMOID*. The functions you may need 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 [16]:
```
### CONSTANTS ###
layers_dims = [12288, 20, 20, 10, 5, 1] #  4-layer model
```

In [17]:
```
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of ea
ch layer in our network

    Returns:
    parameters -- python dictionary containing your parameters "W1",
 "b1", ..., "WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l], laye
r_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(1)
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], lay
er_dims[l-1]) / np.sqrt(layer_dims[l-1]) #*0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l], laye
r_dims[l-1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters
```

In [18]:
```
# GRADED FUNCTION: L_model_forward

def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEA
R->SIGMOID computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- last post-activation value
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there are
L-1 of them, indexed from 0 to L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2                  # number of layers in t
he neural network

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" l
ist.
    for l in range(1, L):
        A_prev = A
        ### START CODE HERE ###
        A, cache = linear_activation_forward(A_prev, parameters['W' +
str(l)], parameters['b' + str(l)], "relu")
        caches.append(cache)
        ### END CODE HERE ###

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    ### START CODE HERE ###
    AL, cache = linear_activation_forward(A, parameters['W' + str(L
)], parameters['b' + str(L)], "sigmoid")
    caches.append(cache)
    ### END CODE HERE ###

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

    return AL, caches
```

In [19]:
```
# 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_mo
del_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 "rel
u" (there are (L-1) or them, indexes from 0 to L-2)
                the cache of linear_activation_forward() with "sigmoi
d" (there is one, index 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
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, cache
s". Outputs: "grads["dAL"], grads["dWL"], grads["dbL"]
    current_cache = caches[L-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)]
= linear_activation_backward(dAL, current_cache, activation = "sigmoi
d")

    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(g
rads["dA" + str(l + 2)], current_cache, activation = "relu")
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

In [20]:
```
# GRADED FUNCTION: L_layer_model

def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):#lr was 0.009
    """
    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.
    ### START CODE HERE ###
    parameters = initialize_parameters_deep(layers_dims)
    ### END CODE HERE ###

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
        ### START CODE HERE ###
        AL, caches = L_model_forward(X, parameters)
        ### END CODE HERE ###

        # Compute cost.
        ### START CODE HERE ###
        cost = compute_cost(AL, Y)
        ### END CODE HERE ###

        # Backward propagation.
        ### START CODE HERE ###
        grads = L_model_backward(AL, Y, caches)
        ### END CODE HERE ###

        # Update parameters.
        ### START CODE HERE ###
        parameters = update_parameters(parameters, grads, learning_rate)
```

```
        ### END CODE HERE ###

        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))
        if print_cost and i % 100 == 0:
            costs.append(cost)
            for g in grads:
                print(g, np.max(np.abs(grads[g])))

    # plot the cost
    if print_cost:
        plt.plot(np.squeeze(costs))
        plt.ylabel('cost')
        plt.xlabel('iterations (per hundreds)')
        plt.title("Learning rate =" + str(learning_rate))
        plt.show()

    return parameters
```

You will now train the model as a 4-layer neural network.

Run the cell below to train your model. The cost should decrease on every iteration. It may take up to 5 minutes to run 2500 iterations. Check if the "Cost after iteration 0" matches the expected output below, if not click on the square (■) on the upper bar of the notebook to stop the cell and try to find your error.

In [21]:
```
parameters = L_layer_model(train_x, train_y, layers_dims, num_iterati
ons = 2500, print_cost = True)
```

```
Cost after iteration 0: 0.695534
dA5 0.35374838994301705
dW5 0.019327551390066804
db5 0.1587296315569707
dA4 0.13378455512396414
dW4 0.013085842243508606
db4 0.04737981665387609
dA3 0.11229576241090249
dW3 0.012167305908332492
db3 0.01984152431486486
dA2 0.09478545627029686
dW2 0.0038064213092023403
db2 0.006629289474555379
dA1 0.004351701754290881
dW1 0.0022377010422810913
db1 0.0032257082621592456
Cost after iteration 100: 0.666574
dA5 0.3841302212237409
dW5 0.04485185619779358
db5 0.10642941653248263
dA4 0.14612058451291224
dW4 0.030288639897190803
db4 0.02446849059170328
dA3 0.07374958101539829
dW3 0.01332267458676473
db3 0.006232440823946565
dA2 0.05086671501590228
dW2 0.007285450275922996
db2 0.0034927677751364515
dA1 0.003239724493784079
dW1 0.0012993031235424089
db1 0.0015457173120339129
Cost after iteration 200: 0.646906
dA5 0.4385770225126345
dW5 0.04733130534221402
db5 0.04997058206152563
dA4 0.16795013547118925
dW4 0.026595032618242514
db4 0.013657505499481734
dA3 0.09622234203568363
dW3 0.010657886843813429
db3 0.0034213205692776643
dA2 0.06223738269736234
dW2 0.005545861173587549
db2 0.0018448351756057534
dA1 0.003974831586549866
dW1 0.0011197823337187115
db1 0.0010435757591124805
Cost after iteration 300: 0.635519
dA5 0.4621634437321409
dW5 0.036950405167095184
db5 0.023487595851315643
dA4 0.18276518259898292
dW4 0.024286858316841003
db4 0.007102460299234173
dA3 0.14632717614904608
dW3 0.013206266240512781
```

```
db3 0.003176674430854858
dA2 0.07596927790737884
dW2 0.010851420252322752
db2 0.002965473478180183
dA1 0.005005119319565012
dW1 0.001953953842175162
db1 0.0011801447279721127
Cost after iteration 400: 0.607224
dA5 0.4938697939738176
dW5 0.06922502561359463
db5 0.01760008338238167
dA4 0.21248700289054162
dW4 0.05224956607189524
db4 0.007656751302422092
dA3 0.13246434747556124
dW3 0.023082174308048584
db3 0.002271341998862142
dA2 0.11864995269006792
dW2 0.010209465915242254
db2 0.0012486949405474863
dA1 0.007769926849623558
dW1 0.003317103863129133
db1 0.0009479505591508892
Cost after iteration 500: 0.564695
dA5 0.5464450734596038
dW5 0.08815765098306523
db5 0.0016417844911477135
dA4 0.2876902064323085
dW4 0.08441077802503208
db4 0.017080967399644265
dA3 0.17678631973018438
dW3 0.03746275630468873
db3 0.0024178493733236395
dA2 0.18040246595193188
dW2 0.018289357480136513
db2 0.0017557269956313326
dA1 0.012006238430503023
dW1 0.004139356948429502
db1 0.0011748575140555066
Cost after iteration 600: 0.490628
dA5 0.6244583257243751
dW5 0.11775077099136537
db5 0.00426467115162155
dA4 0.41042881210273957
dW4 0.11845108884530163
db4 0.020910782501014438
dA3 0.281799239018526
dW3 0.06124317009076699
db3 0.00726432352330203
dA2 0.2736662630107207
dW2 0.035722060782273725
db2 0.004112515476013503
dA1 0.022505843513616636
dW1 0.005936331857134263
db1 0.0037245181320506803
Cost after iteration 700: 0.490949
dA5 0.7168381063755909
```

```
dW5 0.2717961589162799
db5 0.13609192944646767
dA4 0.5359658900185466
dW4 0.5704298233111286
db4 0.12281329563273637
dA3 0.3733516032367663
dW3 0.5382456453644467
db3 0.08902947161021185
dA2 0.36464878049613253
dW2 0.4813075889112315
db2 0.059453557518226154
dA1 0.028794804321356175
dW1 0.03637668355811352
db1 0.0619853857246349
Cost after iteration 800: 0.445474
dA5 0.7337492915052766
dW5 0.3619964846742659
db5 0.14920254050455184
dA4 0.5749161159355941
dW4 0.6575288436594955
db4 0.12622429722525272
dA3 0.4090148197756949
dW3 0.5757656486661492
db3 0.09779910413905359
dA2 0.39502812305416146
dW2 0.5985790632398492
db2 0.06692519072113035
dA1 0.03547320856895203
dW1 0.04158439008434522
db1 0.07160420093566412
Cost after iteration 900: 0.415454
dA5 0.7511174375901812
dW5 0.42187448217113166
db5 0.15503479170195336
dA4 0.6119264931516706
dW4 0.6987292963650137
db4 0.13052164786178758
dA3 0.5090612925194223
dW3 0.643158765480034
db3 0.1056521474424864
dA2 0.44109372426420507
dW2 0.6571788819092579
db2 0.07299732668200945
dA1 0.04453018399248714
dW1 0.045861856843826175
db1 0.07862355491329275
Cost after iteration 1000: 0.348942
dA5 0.7684082715012277
dW5 0.3808782735917051
db5 0.13865443761180887
dA4 0.6380585814735561
dW4 0.6411657218738607
db4 0.12134126739872175
dA3 0.4926551895011954
dW3 0.6192962780409283
db3 0.10145723642720135
dA2 0.5117422317399258
```

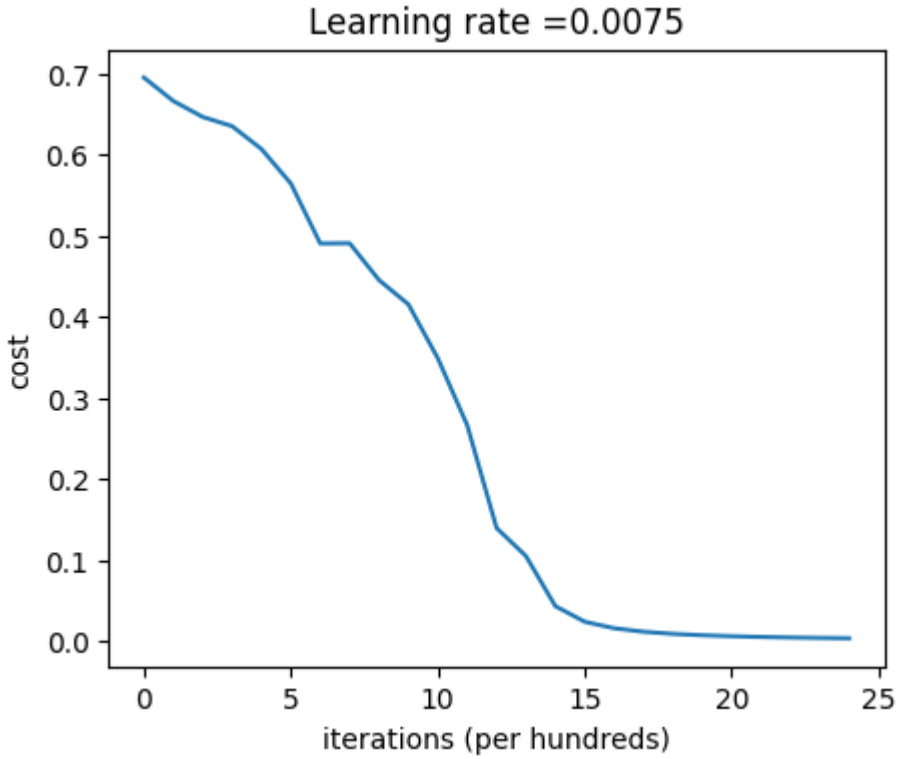
```
dW2 0.6519139664799513
db2 0.0712837308932569
dA1 0.0502018962455836
dW1 0.044926785601417094
db1 0.07720166596432192
Cost after iteration 1100: 0.266278
dA5 0.7934453191878471
dW5 0.2924392474042371
db5 0.10992852348203253
dA4 0.6816548842749638
dW4 0.527671778988213
db4 0.10040617044615095
dA3 0.5277480847791297
dW3 0.5336102900126958
db3 0.08792378387110955
dA2 0.5732883187320131
dW2 0.5748767570966902
db2 0.061551490685837915
dA1 0.05427947479720967
dW1 0.039366248482980495
db1 0.06830743996453414
Cost after iteration 1200: 0.139270
dA5 0.8014165048312399
dW5 0.1604602407154827
db5 0.03761651203257863
dA4 0.7419805836080308
dW4 0.19342132039363094
db4 0.040402340886154284
dA3 0.6037000391548429
dW3 0.20408320827476806
db3 0.03573156214337886
dA2 0.634797530023418
dW2 0.26556827799733024
db2 0.027286522037162447
dA1 0.06853582999358217
dW1 0.017989898315163563
db1 0.029053052646243388
Cost after iteration 1300: 0.104633
dA5 0.7025726018204885
dW5 0.22080279817909187
db5 0.0436850768158162
dA4 0.6777519166530193
dW4 0.21070974889958885
db4 0.03566443548070321
dA3 0.5550334094128684
dW3 0.24592041823425284
db3 0.036164811489520575
dA2 0.8363264603152133
dW2 0.2672299196592058
db2 0.02728736987174127
dA1 0.09521369485397282
dW1 0.0177410558921742
db1 0.03066288433832888
Cost after iteration 1400: 0.042788
dA5 0.4936519810727402
dW5 0.09325672059899924
db5 0.010385569113669224
```

```
dA4 0.5089024629981465
dW4 0.06382581543154528
db4 0.01022691734304244
dA3 0.47615311467827676
dW3 0.07892175549061761
db3 0.00922691067725878
dA2 0.6122060375496715
dW2 0.08048243259772106
db2 0.005861047137840615
dA1 0.06825423113134665
dW1 0.005727419012504868
db1 0.00966595527600655
Cost after iteration 1500: 0.023529
dA5 0.22901883765419798
dW5 0.04667331988787782
db5 0.0023094804298441443
dA4 0.24665354559583855
dW4 0.021608678980541734
db4 0.0035169052031947755
dA3 0.23574020792855296
dW3 0.020134614376803434
db3 0.001984044842163217
dA2 0.34692770196306527
dW2 0.011409374783674027
db2 0.0018123311793044052
dA1 0.048584490554130076
dW1 0.001833992434791476
db1 0.002244307914712665
Cost after iteration 1600: 0.015534
dA5 0.19225059892989782
dW5 0.030847179481601865
db5 0.0007990271839967303
dA4 0.16649372078320362
dW4 0.01865453114001944
db4 0.001955209602610216
dA3 0.1615422406946419
dW3 0.011674381963842953
db3 0.0008948782417904616
dA2 0.24379085400214498
dW2 0.00515978461174237
db2 0.0010989810672953333
dA1 0.03582585683957158
dW1 0.0010884646888091251
db1 0.0012644258179329783
Cost after iteration 1700: 0.011208
dA5 0.17007285238391645
dW5 0.023814064359011528
db5 0.0005495720719404382
dA4 0.12880263873900125
dW4 0.0145467448008679
db4 0.0015543734582337695
dA3 0.12637445050058352
dW3 0.008654207442048376
db3 0.0007340251539654796
dA2 0.19418715910360276
dW2 0.004735287228510691
db2 0.0009460610297835939
```

```
dA1 0.02864645565827826
dW1 0.001014520406687742
db1 0.001358726504597038
Cost after iteration 1800: 0.008589
dA5 0.15165164481725565
dW5 0.01841998586128916
db5 0.00026944691838343203
dA4 0.10108813022416593
dW4 0.012514612597499856
db4 0.0011698975751955711
dA3 0.1033503565261997
dW3 0.0077586209281958855
db3 0.0006143117988866208
dA2 0.1534754589293571
dW2 0.0043391492488492394
db2 0.0007182498273447753
dA1 0.022835281796281682
dW1 0.0007323032324472073
db1 0.0007861100225602539
Cost after iteration 1900: 0.006863
dA5 0.135754430721411
dW5 0.015316728417978896
db5 0.00022862364720293104
dA4 0.08550279351452787
dW4 0.010410075696511834
db4 0.001039793854127651
dA3 0.09409172261531455
dW3 0.006506815631158309
db3 0.0006919622849827376
dA2 0.12722147178522136
dW2 0.004193842748880152
db2 0.0008330349549752114
dA1 0.01893699897714539
dW1 0.0005979961480494602
db1 0.0005831913533427244
Cost after iteration 2000: 0.005660
dA5 0.12217724666655809
dW5 0.013333368452223479
db5 0.000262927329956498
dA4 0.07781262322460523
dW4 0.008499031489258261
db4 0.0009497734085486453
dA3 0.08026010496825631
dW3 0.005113685840176078
db3 0.0005582468147087181
dA2 0.11163315597548351
dW2 0.002506195641898018
db2 0.0006219804146349948
dA1 0.01667078845770083
dW1 0.0005478155144425769
db1 0.0006861882107203521
Cost after iteration 2100: 0.004778
dA5 0.1107471078666583
dW5 0.01111737830365961
db5 0.0001477733511562703
dA4 0.07120412974412775
dW4 0.007838756260686818
```

```
db4 0.0008208396640168166
dA3 0.07884663508885875
dW3 0.004965382387360795
db3 0.0005434547465258451
dA2 0.10021995550717323
dW2 0.003596840830877292
db2 0.0006618906314030045
dA1 0.013636068516771481
dW1 0.0006234900921479007
db1 0.0005494243682686438
Cost after iteration 2200: 0.004106
dA5 0.10097679316969585
dW5 0.01003633291771528
db5 0.00017932989117258732
dA4 0.06545621366671278
dW4 0.006603244348453574
db4 0.0007360291419465256
dA3 0.06794881041177996
dW3 0.004027737041590938
db3 0.00045123453233928415
dA2 0.08187369501790998
dW2 0.002241365757887161
db2 0.0004931966099772177
dA1 0.012331902606796471
dW1 0.00044014523292959734
db1 0.0006312768945380782
Cost after iteration 2300: 0.003584
dA5 0.09254969963235121
dW5 0.00899925944671909
db5 0.00017200261187732172
dA4 0.06042666389392904
dW4 0.005806860820443784
db4 0.0006765302010348467
dA3 0.06289211582572428
dW3 0.0034449983133769094
db3 0.000401377070363201
dA2 0.07279927319439866
dW2 0.0017955027460971483
db2 0.0004456677370236041
dA1 0.010988299924156316
dW1 0.0004090844644939317
db1 0.00062765945827865
Cost after iteration 2400: 0.003163
dA5 0.08521314141847368
dW5 0.007987154336450208
db5 0.00013832632366778306
dA4 0.055993532406812485
dW4 0.005326775651147762
db4 0.0006011016415007826
dA3 0.05841443008249564
dW3 0.003218764109697149
db3 0.00038697148643146445
dA2 0.06714698987977187
dW2 0.0018895731498398687
db2 0.0004005285999761895
dA1 0.00968751222571238
```

```
dW1 0.00032260850340110644
db1 0.00041763367143268846
```



Expected Output:

Cost after iteration 0	0.69...
Cost after iteration 100	0.66...
...	...
Cost after iteration 2400	0.003...

In [22]:
```
def predict_deep(X, y, parameters):
    """
    This function is used to predict the results of a  L-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
    n = len(parameters) // 2 # number of layers in the neural network
    p = np.zeros((1, m),dtype=int)

    # Forward propagation
    probas, caches = L_model_forward(X, parameters)


    # convert probas to 0/1 predictions
    for i in range(0, probas.shape[1]):
        if probas[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0

    #print results
    #print ("predictions: " + str(p))
    #print ("true labels: " + str(y))
    accuracy = str(np.sum(p == y)/float(m))
    print("Accuracy: %s" % accuracy)

    return p, accuracy
```

In [23]:
```
pred_train, _ = predict_deep(train_x, train_y, parameters)
```

```
Accuracy: 1.0
```

Train Accuracy greater than 0.95

In [24]:
```
pred_test, _ = predict_deep(test_x, test_y, parameters)
```

```
Accuracy: 0.82
```

Expected Output:

Test Accuracy greater than 0.7

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 good performance for this task. Nice job!

6) Results Analysis

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

In [25]:
```
def print_mislabeled_images(classes, X, y, p, target_size = (64, 64, 3)):
    """
    Plots images where predictions and truth were different.
    X -- dataset
    y -- true labels
    p -- predictions
    """
    a = p + y
    mislabeled_indices = np.asarray(np.where(a == 1))
    plt.rcParams['figure.figsize'] = (40.0, 40.0) # set default size of plots
    num_images = len(mislabeled_indices[0])
    for i in range(num_images):
        index = mislabeled_indices[1][i]

        plt.subplot(2, num_images, i + 1)
        plt.imshow(X[:,index].reshape(*target_size), interpolation='nearest')
        plt.axis('off')
        plt.title("Prediction: " + classes[int(p[0,index])].decode("utf-8") + " \n Class: " + classes[y[0,index]].decode("utf-8"))
```

In [26]:
```
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 data (12 additional points)

Use the dataset of fingerprints created in Assignment 2 and test the previously implemented classifier. Note that this is a two-class problem, so if your dataset has three or more classes, select two.

Report a pdf file in Moodle where you discuss:

- Compare the results obtained with logistic regression and the Deep Network implemented in this assignment. (6 points)
- Test several iterations, learning rates, number of hidden layers and units, and discuss the results. (6 points)

In [27]:
```
def get_dataset(train_size = 500, test_size = 100):
    train_x_o, train_y = np.zeros((train_size, 32, 32, 3)), np.zeros
((train_size))
    test_x_o, test_y = np.zeros((test_size, 32, 32, 3)), np.zeros((te
st_size))
    for i in range(train_size):
        path, isCat = None, None
        if np.random.uniform() < 0.5:
            path = "../../data/cifar10/train/airplane/" + f"{(i + 1):
04d}" + ".png"
            isCat = False
        else:
            path = "../../data/cifar10/train/cat/" + f"{(i + 1):04d}"
+ ".png"
            isCat = True
        train_x_o[i] = img.imread(path)
        train_y[i] = isCat
    for i in range(test_size):
        path, isCat = None, None
        if np.random.uniform() < 0.5:
            path = "../../data/cifar10/test/airplane/" + f"{(i + 1):0
4d}" + ".png"
            isCat = False
        else:
            path = "../../data/cifar10/test/cat/" + f"{(i + 1):04d}"
+ ".png"
            isCat = True
        test_x_o[i] = img.imread(path)
        test_y[i] = isCat
    test_y = np.expand_dims(test_y, 0)
    train_y = np.expand_dims(train_y, 0)
    return train_x_o, train_y, test_x_o, test_y
```

In [28]:
```
train_x_o, train_y, test_x_o, test_y = get_dataset()
num_px = 32
print(train_x_o.shape, train_y.shape, test_x_o.shape, test_y.shape, s
ep = "\n")
```

```
(500, 32, 32, 3)
(1, 500)
(100, 32, 32, 3)
(1, 100)
```

In [29]:
```
# Example of a picture
index = np.random.randint(1, 500)
plt.imshow(train_x_o[index])
print(train_x_o.shape)
print("It's a", "cat" if train_y[0, index] else "non-cat")
```

```
(500, 32, 32, 3)
It's a non-cat
```

In [30]:
```
train_x = train_x_o.reshape(train_x_o.shape[0], -1).T
test_x = test_x_o.reshape(test_x_o.shape[0], -1).T
```

In [33]:
```
layers_dims = [3072, 20, 20, 10, 5, 1] #  4-layer model
hyper = [[1000, 0.001], [1000, 0.005], [1000, 0.0075], [3000, 0.001],
[3000, 0.005], [3000, 0.0075], [5000, 0.001], [5000, 0.005], [5000,
0.0075]]
for h in hyper:
    e, lr = h
    parameters = L_layer_model(train_x, train_y, layers_dims, num_ite
rations = e, print_cost = False, learning_rate = lr)
    pred_train, tr_acc = predict_deep(train_x, train_y, parameters)
    pred_test, te_acc = predict_deep(test_x, test_y, parameters)
    print(f'epochs: {e} \trate: {lr} \tTrain Accuracy: {tr_acc} \tTes
t Accuracy: {te_acc}')
```

```
Accuracy: 0.702
Accuracy: 0.76
epochs: 1000    rate: 0.001     Train Accuracy: 0.702   Test Accurac
y: 0.76
Accuracy: 0.844
Accuracy: 0.82
epochs: 1000    rate: 0.005     Train Accuracy: 0.844   Test Accurac
y: 0.82
Accuracy: 0.852
Accuracy: 0.83
epochs: 1000    rate: 0.0075    Train Accuracy: 0.852   Test Accurac
y: 0.83
Accuracy: 0.846
Accuracy: 0.84
epochs: 3000    rate: 0.001     Train Accuracy: 0.846   Test Accurac
y: 0.84
Accuracy: 0.874
Accuracy: 0.79
epochs: 3000    rate: 0.005     Train Accuracy: 0.874   Test Accurac
y: 0.79
Accuracy: 0.98
Accuracy: 0.88
epochs: 3000    rate: 0.0075    Train Accuracy: 0.98    Test Accurac
y: 0.88
Accuracy: 0.894
Accuracy: 0.83
epochs: 5000    rate: 0.001     Train Accuracy: 0.894   Test Accurac
y: 0.83
Accuracy: 0.982
Accuracy: 0.86
epochs: 5000    rate: 0.005     Train Accuracy: 0.982   Test Accurac
y: 0.86
Accuracy: 0.99
Accuracy: 0.86
epochs: 5000    rate: 0.0075    Train Accuracy: 0.99    Test Accurac
y: 0.86
```