Task03_Image_Classification_from_Scratch_Output

August 6, 2022

1 Logistic Regression with a Neural Network

You will build a logistic regression classifier to recognize cats. This assignment will step you through how to do this with a Neural Network mindset, and so will also hone your intuitions about deep learning.

[1]:
```
!git clone https://github.com/SanVik2000/EE5179-Final.git
```

```
fatal: destination path 'EE5179-Final' already exists and is not an empty
directory.
```

[2]:
```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage

%matplotlib inline
```

[3]:
```
def load_dataset():
    train_dataset = h5py.File('/content/EE5179-Final/Tutorial2/train_catvnoncat.
↪h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train␣
↪set features
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train␣
↪set labels

    test_dataset = h5py.File('/content/EE5179-Final/Tutorial2/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.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig,␣
↪test_set_y_orig, classes
```

1.1 2 - Overview of the Problem set

Problem Statement: You are given a dataset containing: * a training set of images labeled as cat (y=1) or non-cat (y=0) * a test set of images labeled as cat or non-cat * each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB). Thus, each image is square (height = num_px) and (width = num_px).

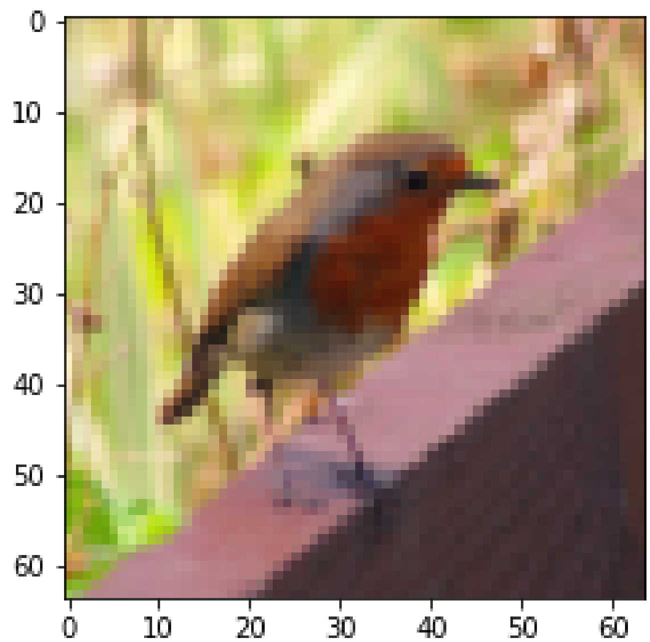
You will build a simple image-recognition algorithm that can correctly classify pictures as cat or non-cat.

Let's get more familiar with the dataset. Load the data by running the following code.

[4]:
```
# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes =␣
↪load_dataset()
```

[5]:
```
# Example of a picture
index =10
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[0, index]) + ", it's a '" + classes[np.
↪squeeze(train_set_y[:, index])].decode("utf-8") +  "' picture.")
```

```
y = 0, it's a 'non-cat' picture.
```



[6]:

```
m_train = train_set_x_orig.shape[0]
m_test =  test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)
```

Expected Output for m_train, m_test and num_px:

m_train

209

m_test

50

num_px

64

For convenience, you should now reshape images of shape (num_px, num_px, 3) in a numpy-array of shape (num_px $*$ num_px $*$ 3, 1).

Exercise: Reshape the training and test data sets so that images of size (num_px, num_px, 3) are flattened into single vectors of shape (num_px $*$ num_px $*$ 3, 1).

A trick when you want to flatten a matrix X of shape (a,b,c,d) to a matrix X_flatten of shape (b$*$c$*$d, a) is to use:

```
X_flatten = X.reshape(X.shape[0], -1).T      # X.T is the transpose of X
```

[7]:

```
# Reshape the training and test examples

train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1)
```

```python
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1)
train_set_y = train_set_y.reshape(train_set_y.shape[1], -1)
test_set_y = test_set_y.reshape(test_set_y.shape[1], -1)

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
train_set_x_flatten shape: (209, 12288)
train_set_y shape: (209, 1)
test_set_x_flatten shape: (50, 12288)
test_set_y shape: (50, 1)
```

Expected Output:

train__set__x__flatten shape

(209, 12288)

train__set__y shape

(209, 1)

test__set__x__flatten shape

(50, 12288)

test__set__y shape

(50, 1)

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.

One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you substract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

Let's standardize our dataset.

[8]:
```python
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

What you need to remember:

Common steps for pre-processing a new dataset are: - Figure out the dimensions and shapes of the problem (m__train, m__test, num__px, …) - Reshape the datasets such that each example is now a vector of size (m__train, num__px * num__px * 3) - "Standardize" the data

1.2 3 - General Architecture of the learning algorithm

It's time to design a simple algorithm to distinguish cat images from non-cat images.

You will build a Logistic Regression, using a Neural Network mindset. The following Figure explains why **Logistic Regression is actually a very simple Neural Network!**

Mathematical expression of the algorithm:

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

$$z^{(i)} = x^{(i)} w^T + b \tag{1}$$

$$\hat{y}^{(i)} = a^{(i)} = sigmoid(z^{(i)}) \tag{2}$$

$$\mathcal{L}(a^{(i)}, y^{(i)}) = -y^{(i)} \log(a^{(i)}) - (1 - y^{(i)}) \log(1 - a^{(i)}) \tag{3}$$

The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^{m} \mathcal{L}(a^{(i)}, y^{(i)}) \tag{6}$$

Key steps: In this exercise, you will carry out the following steps: - Initialize the parameters of the model - Learn the parameters for the model by minimizing the cost
- Use the learned parameters to make predictions (on the test set) - Analyse the results and conclude

1.3 4 - Building the parts of our algorithm

The main steps for building a Neural Network are: 1. Define the model structure (such as number of input features) 2. Initialize the model's parameters 3. Loop: - Calculate current loss (forward propagation) - Calculate current gradient (backward propagation) - Update parameters (gradient descent)

You often build 1-3 separately and integrate them into one function we call `model()`.

1.3.1 4.1 - Helper functions

Exercise: Using your code from "Task02", implement `sigmoid()`. As you've seen in the figure above, you need to compute $sigmoid(xw^T + b) = \frac{1}{1+e^{-(xw^T+b)}}$ to make predictions. Use np.exp().

[9]:
```
# GRADED FUNCTION: sigmoid

def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(z)
    """

    ### START CODE HERE ### ( 1 line of code)
    s = 1/(1+np.exp(-z))
    ### END CODE HERE ###
```

```
    return s
```

[10]:
```
print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))
```

```
sigmoid([0, 2]) = [0.5        0.88079708]
```

Expected Output:

sigmoid([0, 2])

[0.5 0.88079708]

1.3.2 4.2 - Initializing parameters

Exercise: Implement parameter initialization in the cell below. You have to initialize w as a vector of zeros. If you don't know what numpy function to use, look up np.zeros() in the Numpy library's documentation.

[11]:
```
# GRADED FUNCTION: initialize_with_zeros

def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (1, dim) for w and␣
 ↪initializes b to 0.

    Argument:
    dim -- size of the w vector we want (or number of parameters in this case)

    Returns:
    w -- initialized vector of shape (dim, 1)
    b -- initialized scalar (corresponds to the bias)
    """

    ### START CODE HERE ### (  1 line of code)
    w = np.zeros((1, dim))
    b = 0
    ### END CODE HERE ###

    assert(w.shape == (1, dim))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

[12]:
```
dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

```
w = [[0. 0.]]
b = 0
```

Expected Output:

** w **

[[0. 0.]]

** b **

0

For image inputs, w will be of shape (1, num_px × num_px × 3).

1.3.3 4.3 - Forward and Backward propagation

Now that your parameters are initialized, you can do the "forward" and "backward" propagation steps for learning the parameters.

Exercise: Implement a function `propagate()` that computes the cost function and its gradient.

Hints:

Forward Propagation: - You get X - You compute $A = \sigma(Xw^T + b) = (a^{(1)}, a^{(2)}, ..., a^{(m-1)}, a^{(m)})$ - You calculate the cost function: $J = -\frac{1}{m}\sum_{i=1}^{m} y^{(i)}\log(a^{(i)}) + (1 - y^{(i)})\log(1 - a^{(i)})$

Here are the two formulas you will be using:

$$\frac{\partial J}{\partial w} = \frac{1}{m}(A - Y)^T X \tag{7}$$

$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)} - y^{(i)}) \tag{8}$$

[13]:
```python
# GRADED FUNCTION: propagate
def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation explained␣
 ↪above

    Arguments:
    w -- weights, a numpy array of size (1, num_px * num_px * 3)
    b -- bias, a scalar
    X -- data of size (number of examples, num_px * num_px * 3)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size␣
 ↪(number of examples, 1)

    Return:
    cost -- negative log-likelihood cost for logistic regression
    dw -- gradient of the loss with respect to w, thus same shape as w
    db -- gradient of the loss with respect to b, thus same shape as b
    """
```

```
    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    ### START CODE HERE ### ( 2 lines of code)
    A = sigmoid(np.dot(X, w.T) + b)              # compute activation
    cost = np.sum(((- np.log(A))*Y + (-np.log(1-A))*(1-Y)))/m  # compute cost
    ### END CODE HERE ###

    # BACKWARD PROPAGATION (TO FIND GRAD)
    ### START CODE HERE ### ( 2 lines of code)
    dw = (np.dot((A-Y).T, X))/m
    db = (np.sum(A-Y))/m
    ### END CODE HERE ###

    assert(dw.shape == w.shape)
    assert(db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"dw": dw,
             "db": db}

    return grads, cost
```

[14]:
```
w, b, X, Y = np.array([[1.,2.]]), 2., np.array([[1.,2.],[-1.,3.],[4.,-3.2]]),␣
↪np.array([[1],[0],[1]])
print("W Shape : " , w.shape)
print("X Shape : " , X.shape)
print("Y Shape : " , Y.shape, "\n")

grads, cost = propagate(w, b, X, Y)
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
print ("cost = " + str(cost))
```

```
W Shape :  (1, 2)
X Shape :  (3, 2)
Y Shape :  (3, 1)

dw = [[-1.69737532  2.45562263]]
db = 0.1997451187493734
cost = 3.9574190926537742
```

Expected Output:

** dw **

[[0.99845601] [2.39507239]]

** db **

0.00145557813678

** cost **

5.801545319394553

1.3.4 4.4 - Optimization

- You have initialized your parameters.
- You are also able to compute a cost function and its gradient.
- Now, you want to update the parameters using gradient descent.

Exercise: Write down the optimization function. The goal is to learn w and b by minimizing the cost function J. For a parameter θ, the update rule is \$ = - d \$, where α is the learning rate.

[15]:
```python
# GRADED FUNCTION: optimize

def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):
    """
    This function optimizes w and b by running a gradient descent algorithm

    Arguments:
    w -- weights, a numpy array of size (1, num_px * num_px * 3)
    b -- bias, a scalar
    X -- data of shape (number of examples, num_px * num_px * 3)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape␣
↪(number of examples, 1)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with␣
↪respect to the cost function
    costs -- list of all the costs computed during the optimization, this will␣
↪be used to plot the learning curve.

    Tips:
    You basically need to write down two steps and iterate through them:
        1) Calculate the cost and the gradient for the current parameters. Use␣
↪propagate().
        2) Update the parameters using gradient descent rule for w and b.
    """

    costs = []

    for i in range(num_iterations):
```

```
        # Cost and gradient calculation ( 1-4 lines of code)
        ### START CODE HERE ###
        grads, cost = propagate(w, b, X, Y)
        ### END CODE HERE ###

        # Retrieve derivatives from grads
        dw = grads["dw"]
        db = grads["db"]

        # update rule ( 2 lines of code)
        ### START CODE HERE ###
        w = w - (learning_rate*dw)
        b = b - (learning_rate*db)
        ### END CODE HERE ###

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

        # Print the cost every 100 training iterations
        if print_cost and i % 100 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    params = {"w": w,
              "b": b}

    grads = {"dw": dw,
             "db": db}

    return params, grads, costs
```

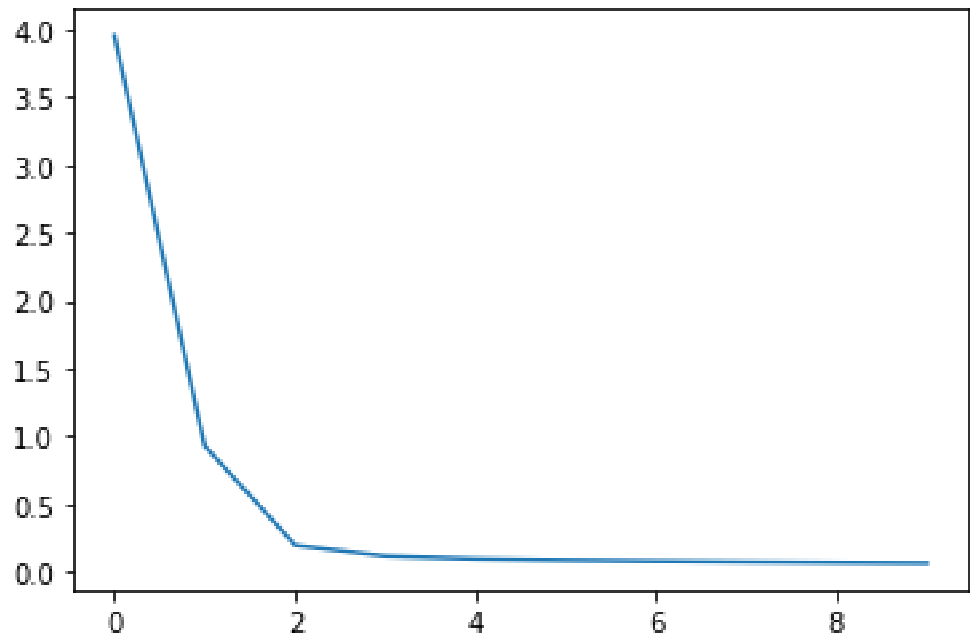
[16]:
```
params, grads, costs = optimize(w, b, X, Y, num_iterations= 1000, learning_rate␣
↪= 0.009, print_cost = False)

print ("w = " + str(params["w"]))
print ("b = " + str(params["b"]))
print ("dw = " + str(grads["dw"]))
print ("db = " + str(grads["db"]))
```

```
w = [[ 2.46915585 -0.59357113]]
b = 1.322892868548117
dw = [[-0.05986524  0.00745058]]
db = -0.008994333545665381
```

[17]:
```
import matplotlib.pyplot as plt
plt.plot(costs)
```

[17]: [<matplotlib.lines.Line2D at 0x7f9cc3156b50>]



Exercise: The previous function will output the learned w and b. We are able to use w and b to predict the labels for a dataset X. Implement the `predict()` function. There are two steps to computing predictions:

1. Calculate $\hat{Y} = A = \sigma(Xw^T + b)$
2. Convert the entries of a into 0 (if activation <= 0.5) or 1 (if activation > 0.5), stores the predictions in a vector `Y_prediction`. If you wish, you can use an `if`/`else` statement in a `for` loop (though there is also a way to vectorize this).

[18]:
```
# GRADED FUNCTION: predict

def predict(w, b, X):
    '''
    Predict whether the label is 0 or 1 using learned logistic regression␣
 ↪parameters (w, b)

    Arguments:
    w -- weights, a numpy array of size (1, num_px * num_px * 3)
    b -- bias, a scalar
    X -- data of size (number of examples, num_px * num_px * 3)
```

```
    Returns:
    Y_prediction -- a numpy array (vector) containing all predictions (0/1) for␣
 ↪the examples in X
    '''

    m = X.shape[0]
    Y_prediction = np.zeros((m, 1))

    # Compute vector "A" predicting the probabilities of a cat being present in␣
 ↪the picture
    ### START CODE HERE ### ( 1 line of code)
    A = sigmoid(np.dot(X, w.T) + b)           # Dimentions = (m, 1)
    ### END CODE HERE ###

    #### VECTORISED IMPLEMENTATION ####
    Y_prediction = (A >= 0.5) * 1.0

    assert(Y_prediction.shape == (m, 1))

    return Y_prediction
```

[19]:
```
w = np.array([[2.46915585, -0.59357113]])
b = 1.322892868548117
X = np.array([[1.,-1.1],[-3.2,1.2],[2.,0.1]])
print ("predictions = " + str(predict(w, b, X)))
```

```
predictions = [[1.]
 [0.]
 [1.]]
```

What to remember: You've implemented several functions that: - Initialize (w,b) - Optimize the loss iteratively to learn parameters (w,b): - computing the cost and its gradient - updating the parameters using gradient descent - Use the learned (w,b) to predict the labels for a given set of examples

1.4 5 - Merge all functions into a model

You will now see how the overall model is structured by putting together all the building blocks (functions implemented in the previous parts) together, in the right order.

[20]:
```
# GRADED FUNCTION: model

def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000,␣
 ↪learning_rate = 0.5, print_cost = False):
    """
    Builds the logistic regression model by calling the function you've␣
 ↪implemented previously
```

```
    Arguments:
    X_train -- training set represented by a numpy array of shape (m_train,␣
↪num_px * num_px * 3)
    Y_train -- training labels represented by a numpy array (vector) of shape␣
↪(m_train, 1)
    X_test -- test set represented by a numpy array of shape (m_test, num_px *␣
↪num_px * 3)
    Y_test -- test labels represented by a numpy array (vector) of shape␣
↪(m_test, 1)
    num_iterations -- hyperparameter representing the number of iterations to␣
↪optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used in the␣
↪update rule of optimize()
    print_cost -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the model.
    """

    ### START CODE HERE ###

    # initialize parameters with zeros ( 1 line of code)
    w, b = initialize_with_zeros(X_train.shape[1])
    print("W Shape : " , w.shape)
    print("X_Train Shape : " , X_train.shape)

    # Gradient descent ( 1 line of code)
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations,␣
↪learning_rate, print_cost)

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples ( 2 lines of code)
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)
    ### END CODE HERE ###

    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train␣
↪- Y_train)) * 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test -␣
↪Y_test)) * 100))
```

```
    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

Run the following cell to train your model.

[21]:
```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations =
 ↪20000, learning_rate = 0.05, print_cost = True)
```

```
W Shape :  (1, 12288)
X_Train Shape :  (209, 12288)
Cost after iteration 0: 0.011789
Cost after iteration 100: 0.010170
Cost after iteration 200: 0.009613
Cost after iteration 300: 0.009189
Cost after iteration 400: 0.008839
Cost after iteration 500: 0.008536
Cost after iteration 600: 0.008267
Cost after iteration 700: 0.008026
Cost after iteration 800: 0.007805
Cost after iteration 900: 0.007602
Cost after iteration 1000: 0.007414
Cost after iteration 1100: 0.007238
Cost after iteration 1200: 0.007074
Cost after iteration 1300: 0.006919
Cost after iteration 1400: 0.006773
Cost after iteration 1500: 0.006634
Cost after iteration 1600: 0.006503
Cost after iteration 1700: 0.006378
Cost after iteration 1800: 0.006259
Cost after iteration 1900: 0.006145
Cost after iteration 2000: 0.006036
Cost after iteration 2100: 0.005931
Cost after iteration 2200: 0.005831
Cost after iteration 2300: 0.005735
Cost after iteration 2400: 0.005642
Cost after iteration 2500: 0.005553
Cost after iteration 2600: 0.005467
Cost after iteration 2700: 0.005384
Cost after iteration 2800: 0.005303
Cost after iteration 2900: 0.005226
Cost after iteration 3000: 0.005150
```

```
Cost after iteration 3100: 0.005078
Cost after iteration 3200: 0.005007
Cost after iteration 3300: 0.004939
Cost after iteration 3400: 0.004872
Cost after iteration 3500: 0.004808
Cost after iteration 3600: 0.004745
Cost after iteration 3700: 0.004684
Cost after iteration 3800: 0.004625
Cost after iteration 3900: 0.004567
Cost after iteration 4000: 0.004511
Cost after iteration 4100: 0.004456
Cost after iteration 4200: 0.004402
Cost after iteration 4300: 0.004350
Cost after iteration 4400: 0.004300
Cost after iteration 4500: 0.004250
Cost after iteration 4600: 0.004202
Cost after iteration 4700: 0.004154
Cost after iteration 4800: 0.004108
Cost after iteration 4900: 0.004063
Cost after iteration 5000: 0.004019
Cost after iteration 5100: 0.003976
Cost after iteration 5200: 0.003933
Cost after iteration 5300: 0.003892
Cost after iteration 5400: 0.003852
Cost after iteration 5500: 0.003812
Cost after iteration 5600: 0.003773
Cost after iteration 5700: 0.003735
Cost after iteration 5800: 0.003698
Cost after iteration 5900: 0.003662
Cost after iteration 6000: 0.003626
Cost after iteration 6100: 0.003591
Cost after iteration 6200: 0.003556
Cost after iteration 6300: 0.003523
Cost after iteration 6400: 0.003489
Cost after iteration 6500: 0.003457
Cost after iteration 6600: 0.003425
Cost after iteration 6700: 0.003394
Cost after iteration 6800: 0.003363
Cost after iteration 6900: 0.003333
Cost after iteration 7000: 0.003303
Cost after iteration 7100: 0.003274
Cost after iteration 7200: 0.003245
Cost after iteration 7300: 0.003217
Cost after iteration 7400: 0.003189
Cost after iteration 7500: 0.003162
Cost after iteration 7600: 0.003135
Cost after iteration 7700: 0.003109
Cost after iteration 7800: 0.003083
```

```
Cost after iteration 7900: 0.003057
Cost after iteration 8000: 0.003032
Cost after iteration 8100: 0.003008
Cost after iteration 8200: 0.002983
Cost after iteration 8300: 0.002959
Cost after iteration 8400: 0.002936
Cost after iteration 8500: 0.002913
Cost after iteration 8600: 0.002890
Cost after iteration 8700: 0.002867
Cost after iteration 8800: 0.002845
Cost after iteration 8900: 0.002823
Cost after iteration 9000: 0.002802
Cost after iteration 9100: 0.002781
Cost after iteration 9200: 0.002760
Cost after iteration 9300: 0.002739
Cost after iteration 9400: 0.002719
Cost after iteration 9500: 0.002699
Cost after iteration 9600: 0.002679
Cost after iteration 9700: 0.002660
Cost after iteration 9800: 0.002640
Cost after iteration 9900: 0.002622
Cost after iteration 10000: 0.002603
Cost after iteration 10100: 0.002585
Cost after iteration 10200: 0.002566
Cost after iteration 10300: 0.002548
Cost after iteration 10400: 0.002531
Cost after iteration 10500: 0.002513
Cost after iteration 10600: 0.002496
Cost after iteration 10700: 0.002479
Cost after iteration 10800: 0.002462
Cost after iteration 10900: 0.002446
Cost after iteration 11000: 0.002430
Cost after iteration 11100: 0.002413
Cost after iteration 11200: 0.002397
Cost after iteration 11300: 0.002382
Cost after iteration 11400: 0.002366
Cost after iteration 11500: 0.002351
Cost after iteration 11600: 0.002336
Cost after iteration 11700: 0.002321
Cost after iteration 11800: 0.002306
Cost after iteration 11900: 0.002291
Cost after iteration 12000: 0.002277
Cost after iteration 12100: 0.002263
Cost after iteration 12200: 0.002249
Cost after iteration 12300: 0.002235
Cost after iteration 12400: 0.002221
Cost after iteration 12500: 0.002207
Cost after iteration 12600: 0.002194
```

```
Cost after iteration 12700: 0.002181
Cost after iteration 12800: 0.002168
Cost after iteration 12900: 0.002155
Cost after iteration 13000: 0.002142
Cost after iteration 13100: 0.002129
Cost after iteration 13200: 0.002117
Cost after iteration 13300: 0.002104
Cost after iteration 13400: 0.002092
Cost after iteration 13500: 0.002080
Cost after iteration 13600: 0.002068
Cost after iteration 13700: 0.002056
Cost after iteration 13800: 0.002044
Cost after iteration 13900: 0.002033
Cost after iteration 14000: 0.002021
Cost after iteration 14100: 0.002010
Cost after iteration 14200: 0.001999
Cost after iteration 14300: 0.001988
Cost after iteration 14400: 0.001977
Cost after iteration 14500: 0.001966
Cost after iteration 14600: 0.001955
Cost after iteration 14700: 0.001944
Cost after iteration 14800: 0.001934
Cost after iteration 14900: 0.001923
Cost after iteration 15000: 0.001913
Cost after iteration 15100: 0.001903
Cost after iteration 15200: 0.001893
Cost after iteration 15300: 0.001883
Cost after iteration 15400: 0.001873
Cost after iteration 15500: 0.001863
Cost after iteration 15600: 0.001853
Cost after iteration 15700: 0.001844
Cost after iteration 15800: 0.001834
Cost after iteration 15900: 0.001825
Cost after iteration 16000: 0.001815
Cost after iteration 16100: 0.001806
Cost after iteration 16200: 0.001797
Cost after iteration 16300: 0.001788
Cost after iteration 16400: 0.001779
Cost after iteration 16500: 0.001770
Cost after iteration 16600: 0.001761
Cost after iteration 16700: 0.001752
Cost after iteration 16800: 0.001744
Cost after iteration 16900: 0.001735
Cost after iteration 17000: 0.001727
Cost after iteration 17100: 0.001718
Cost after iteration 17200: 0.001710
Cost after iteration 17300: 0.001702
Cost after iteration 17400: 0.001694
```

```
Cost after iteration 17500: 0.001686
Cost after iteration 17600: 0.001678
Cost after iteration 17700: 0.001670
Cost after iteration 17800: 0.001662
Cost after iteration 17900: 0.001654
Cost after iteration 18000: 0.001646
Cost after iteration 18100: 0.001639
Cost after iteration 18200: 0.001631
Cost after iteration 18300: 0.001623
Cost after iteration 18400: 0.001616
Cost after iteration 18500: 0.001609
Cost after iteration 18600: 0.001601
Cost after iteration 18700: 0.001594
Cost after iteration 18800: 0.001587
Cost after iteration 18900: 0.001580
Cost after iteration 19000: 0.001572
Cost after iteration 19100: 0.001565
Cost after iteration 19200: 0.001558
Cost after iteration 19300: 0.001552
Cost after iteration 19400: 0.001545
Cost after iteration 19500: 0.001538
Cost after iteration 19600: 0.001531
Cost after iteration 19700: 0.001525
Cost after iteration 19800: 0.001518
Cost after iteration 19900: 0.001511
train accuracy: 99.52153110047847 %
test accuracy: 68.0 %
```

Expected Output:

Cost after iteration 0

0.693147

⋮

⋮

Train Accuracy

99.04306220095694 %

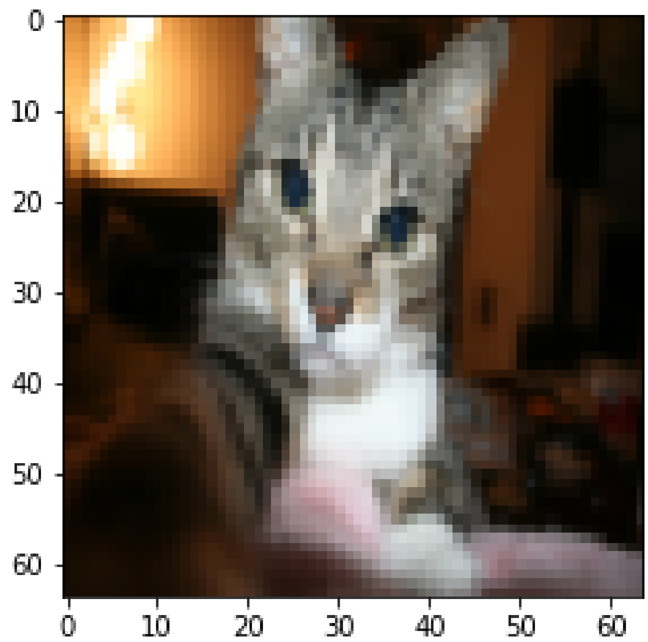
Test Accuracy

70.0 %

Comment: Training accuracy is close to 100%. This is a good sanity check: your model is working and has high enough capacity to fit the training data. Test accuracy is 68%. It is actually not bad for this simple model, given the small dataset we used and that logistic regression is a linear classifier.

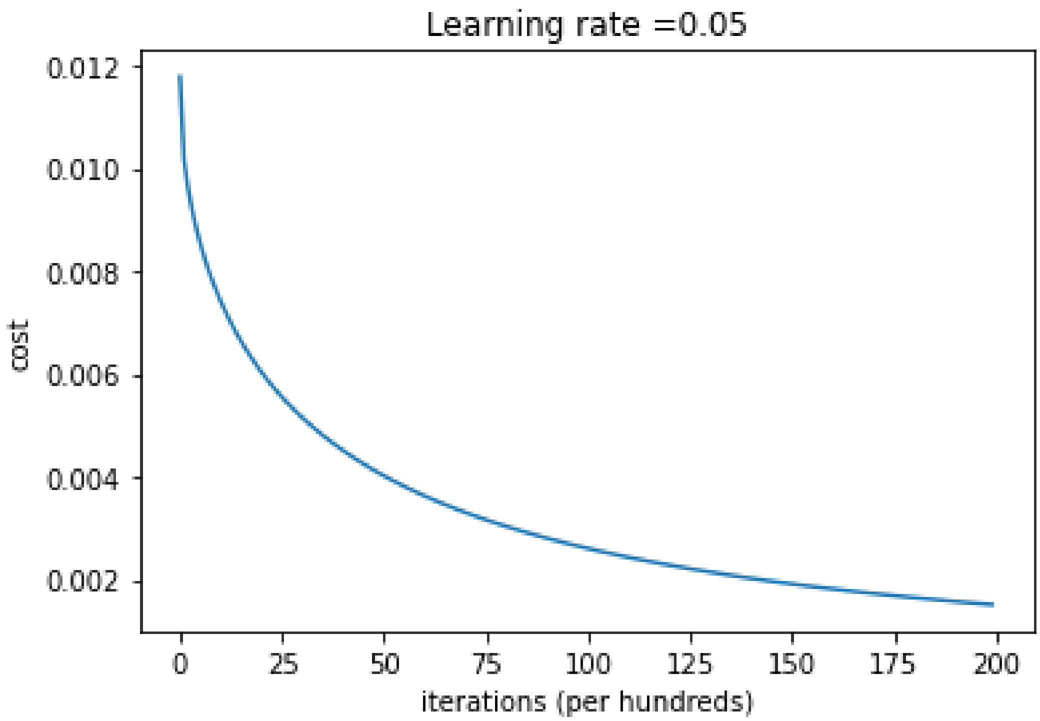
[22]:
```
# Example of a picture that was wrongly classified.
index = 25
plt.imshow(test_set_x[index, :].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[index, 0]) + ", you predicted that it is a \"" +␣
↪classes[int(d["Y_prediction_test"][index][0])].decode("utf-8") +  "\"␣
↪picture.")
```

```
y = 1, you predicted that it is a "cat" picture.
```



Let's also plot the cost function and the gradients.

[23]:
```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```



Interpretation: You can see the cost decreasing. It shows that the parameters are being learned. However, you see that you could train the model even more on the training set. Try to increase the number of iterations in the cell above and rerun the cells. You might see that the training set accuracy goes up, but the test set accuracy goes down. This is called overfitting.