

01_logistic_regression_as_a_neural_network

April 7, 2025

1 Logistic Regression as a Neural Network

In this exercise, we will build a logistic regression classifier to recognize cats. This assignment will step through how to do this from the perspective of constructing a neural network and will also introduce the approach we will take for building deeper networks.

Instructions: - Do not use loops (for/while) in your code, unless the instructions explicitly ask you to do so. - Use `np.dot(X,Y)` to calculate dot products.

You will learn to: - Build the general architecture of a learning algorithm, including: - Initializing parameters - Calculating the cost function and its gradient - Using an optimization algorithm (gradient descent) - Gather all three functions above into a main model function, in the right order.

1.1 Table of Contents

- 1 - Packages
- 2 - Overview of the Problem set
 - Exercise 1
 - Exercise 2
- 3 - General Architecture of the learning algorithm
- 4 - Building the parts of our algorithm
 - 4.1 - Helper functions
 - * Exercise 3 - sigmoid
 - 4.2 - Initializing parameters
 - * Exercise 4 - initialize_with_zeros
 - 4.3 - Forward and Backward propagation
 - * Exercise 5 - propagate
 - 4.4 - Optimization
 - * Exercise 6 - optimize
 - * Exercise 7 - predict
- 5 - Merge all functions into a model
 - Exercise 8 - model
- 6 - Further analysis (optional/ungraded exercise)
- 7 - Test with your own image (optional/ungraded exercise)

1 - Packages

First, run the cell below to import all the packages that you will need during this assignment. - [numpy](#) is the fundamental package for scientific computing with Python. - [h5py](#) is a common

package to interact with a dataset that is stored on an H5 file. - [matplotlib](#) is a famous library to plot graphs in Python. - [PIL](#) and [scipy](#) are used here to test your model with your own picture at the end.

```
[25]: %pip install h5py
```

```
Requirement already satisfied: h5py in  
/home/codespace/.python/current/lib/python3.12/site-packages (3.13.0)  
Requirement already satisfied: numpy>=1.19.3 in  
/home/codespace/.local/lib/python3.12/site-packages (from h5py) (2.2.4)  
Note: you may need to restart the kernel to use updated packages.
```

```
[26]: import numpy as np  
import copy  
import matplotlib.pyplot as plt  
import h5py  
import scipy  
from PIL import Image  
from scipy import ndimage  
from lr_utils import load_dataset  
from public_tests import *  
  
%matplotlib inline  
%load_ext autoreload  
%autoreload 2
```

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


```
%reload_ext autoreload
```

```
## 2 - Overview of the Problem set ##
```

Problem Statement: We are given a dataset (“data.h5”) containing: - a training set of `m_train` images labeled as cat (`y=1`) or non-cat (`y=0`) - a test set of `m_test` 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`).

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

In order to get familiar with the dataset, load the data by running the following code.

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

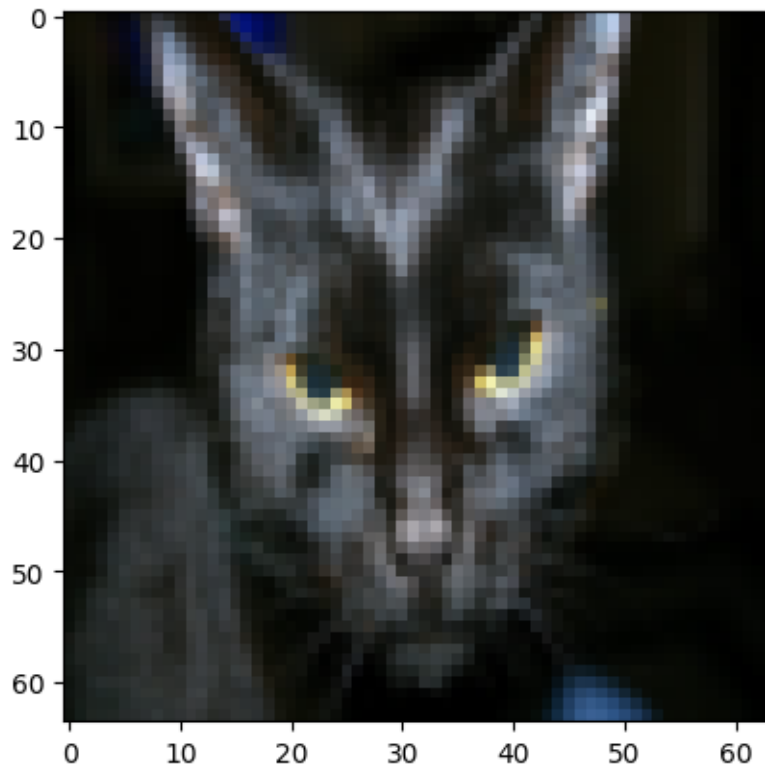
The suffix, “_orig”, is added at the end of image datasets (train and test) because we are going to preprocess them. After preprocessing, we will end up with `train_set_x` and `test_set_x` (the labels `train_set_y` and `test_set_y` don’t need any preprocessing).

Each line of your `train_set_x_orig` and `test_set_x_orig` is an array representing an image. You can visualize an example by running the following code. Feel free also to change the `index` value

and re-run to see other images.

```
[28]: # Example of a picture
index = 25
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a '" + classes[np.
↪squeeze(train_set_y[:, index])).decode("utf-8") + "' picture.")
```

y = [1], it's a 'cat' picture.



Many software bugs in deep learning come from having matrix/vector dimensions that don't fit. If we can keep our matrix/vector dimensions straight, this will go a long way toward eliminating many bugs.

Exercise 1 Find the values for:

- m_train (number of training examples)
- m_test (number of test examples)
- num_px (= height = width of a training image)

Remember that `train_set_x_orig` is a numpy-array of shape `(m_train, num_px, num_px, 3)`. For instance, you can access `m_train` by writing `train_set_x_orig.shape[0]`.

[29]: *# Your code in this next block should be structured as follows:*

```
# m_train = ....
# m_test = ....
# num_px = ...

# YOUR CODE STARTS HERE
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]

# YOUR CODE ENDS HERE

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
```

Now, reshape images of shape (num_px, num_px, 3) into a numpy-array of shape (num_px * num_px * 3, 1). After this, our training (and test) dataset is a numpy-array where each column

represents a flattened image. There should be `m_train` (respectively `m_test`) columns.

Exercise 2 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 we 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
```

```
[30]: # Reshape the training and test examples

# Your code in this block should have the following general structure:

# train_set_x_flatten = ...
# test_set_x_flatten = ...

# YOUR CODE STARTS HERE

train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T

# YOUR CODE ENDS HERE

# Check that the first 10 pixels of the second image are in the correct place
assert np.all(train_set_x_flatten[0:10, 1] == [196, 192, 190, 193, 186, 182, 188, 179, 174, 213]), "Wrong solution. Use (X.shape[0], -1).T."
assert np.all(test_set_x_flatten[0:10, 1] == [115, 110, 111, 137, 129, 129, 155, 146, 145, 159]), "Wrong solution. Use (X.shape[0], -1).T."

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: (12288, 209)
```

```
train_set_y shape: (1, 209)
```

```
test_set_x_flatten shape: (12288, 50)
```

```
test_set_y shape: (1, 50)
```

Expected Output:

```
train_set_x_flatten shape
```

```
(12288, 209)
```

```
train_set_y shape
```

```
(1, 209)
```

```
test_set_x_flatten shape
```

(12288, 50)

test_set_y shape

(1, 50)

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 use mean normalization - i.e., center and standardize your dataset by subtracting the mean of the whole numpy array from each example, and then dividing 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).

We scale our dataset as follows, then:

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

What 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 (num_px * num_px * 3, 1) - “Standardize” the data

3 - General Architecture of the learning algorithm

Now, we build a Logistic Regression model from the perspective of a neural network. The following Figure shows why we can think of Logistic Regression as actually a very simple Neural Network:

Mathematical expression of the algorithm:

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

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

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

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

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

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

Key steps: In this exercise, we 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 draw conclusions about the model.

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)

Often, we build the functions for steps 1-3 separately and then integrate them into one function we will call `model()`.

4.1 - Helper functions

Exercise 3 - sigmoid Implement the `sigmoid()` function, below. As you've seen in the figure above, you need to compute $\text{sigmoid}(z) = \frac{1}{1+e^{-z}}$ for $z = w^T x + b$ to make predictions. Use `np.exp()`.

```
[32]: # 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)
    """

    # YOUR CODE STARTS HERE

    s = 1 / (1 + np.exp(-z))

    # YOUR CODE ENDS HERE

    return s
```

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

sigmoid_test(sigmoid)
```

```
sigmoid([0, 2]) = [0.5          0.88079708]
All tests passed!
```

```
[34]: x = np.array([0.5, 0, 2.0])
output = sigmoid(x)
print(output)
```

```
[0.62245933 0.5          0.88079708]
```

4.2 - Initializing parameters

Exercise 4 - initialize_with_zeros Implement parameter initialization in the cell below. You have to initialize w as a vector of zeros.

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

def initialize_with_zeros(dim):
    """
    This function creates a vector of zeros of shape (dim, 1) 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) of type float
    """

    # YOUR CODE STARTS HERE

    w = np.zeros((dim, 1))
    b = 0.0

    # YOUR CODE ENDS HERE

    return w, b
```

```
[36]: dim = 2
w, b = initialize_with_zeros(dim)

assert type(b) == float
print ("w = " + str(w))
print ("b = " + str(b))

initialize_with_zeros_test_1(initialize_with_zeros)
initialize_with_zeros_test_2(initialize_with_zeros)
```

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

```
b = 0.0
```

```
First test passed!
```

```
Second test passed!
```

4.3 - Forward and Backward propagation

Now that the parameters w, b are initialized, we will do the “forward” and “backward” propagation

steps for learning the parameters.

Exercise 5 - propagate Implement a function `propagate()` that computes the cost function and the vector of gradients.

General Approach:

Forward Propagation: - You are given X , the matrix of input features for the m samples as an input to the `propagate()` function. - Compute $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \dots, a^{(m-1)}, a^{(m)})$ - Calculate the cost function: $J = -\frac{1}{m} \sum_{i=1}^m (y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)}))$

Back Propagation:

The formulas for the partial derivatives (derived in the lectures) are as follows:

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

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

Remember: No for loops!

```
[37]: # 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 (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1,
    ↪number of examples)

    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

    Important - Vectorization should be implemented here - which means using np.
    ↪dot(), np.log() and np.sum(), and
    and also, no for loops.

    """

    m = X.shape[1]
```

```

# FORWARD PROPAGATION (FROM X TO COST)
# YOUR CODE STARTS HERE

A = sigmoid(np.dot(w.T, X) + b) # compute activation
cost = - (1 / m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A)) #
↪compute cost

# YOUR CODE ENDS HERE

# BACKWARD PROPAGATION (TO FIND GRAD)
# YOUR CODE STARTS HERE

dw = (1 / m) * np.dot(X, (A - Y).T) # derivative with respect
↪to w
db = (1 / m) * np.sum(A - Y) # derivative with respect
↪to b

# YOUR CODE ENDS HERE

cost = np.squeeze(np.array(cost))

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

return grads, cost

```

```

[38]: w = np.array([[1.], [2]])
      b = 1.5
      X = np.array([[1., -2., -1.], [3., 0.5, -3.2]])
      Y = np.array([[1, 1, 0]])
      grads, cost = propagate(w, b, X, Y)

      assert type(grads["dw"]) == np.ndarray
      assert grads["dw"].shape == (2, 1)
      assert type(grads["db"]) == np.float64

      print ("dw = " + str(grads["dw"]))
      print ("db = " + str(grads["db"]))

```

```
print ("cost = " + str(cost))

propagate_test(propagate)
```

```
dw = [[ 0.25071532]
      [-0.06604096]]
db = -0.1250040450043965
cost = 0.15900537707692405
All tests passed!
```

Expected output

```
dw = [[ 0.25071532]
      [-0.06604096]]
db = -0.1250040450043965
cost = 0.15900537707692405
```

4.4 - Optimization

At this point we have written functions to:

- initialize the parameters, (`initialize_with_zeros()`)
- compute a cost function and its gradient, (`propagate()`)

Now, we need to write a function, `optimize()`, to update the parameters using gradient descent.

Exercise 6 - optimize 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 $\theta = \theta - \alpha \text{d}\theta$, where α is the learning rate.

```
[39]: # GRADED FUNCTION: optimize

def optimize(w, b, X, Y, num_iterations=100, learning_rate=0.009,
            print_cost=False):
    """
    This function optimizes w and b by running a gradient descent algorithm

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

"""

```
w = copy.deepcopy(w)
```

```
b = copy.deepcopy(b)
```

```
costs = []
```

```
for i in range(num_iterations):
```

```
    # Perform the cost and gradient calculation (use pre-existing function(s))
```

```
    # YOUR CODE STARTS HERE
```

```
    grads, cost = propagate(w, b, X, Y)
```

```
    # YOUR CODE ENDS HERE
```

```
    # Retrieve derivatives from grads
```

```
    dw = grads["dw"]
```

```
    db = grads["db"]
```

```
    # do the update to the parameters, w, b.
```

```
    # YOUR CODE STARTS HERE
```

```
    w = w - learning_rate * dw
```

```
    b = b - learning_rate * db
```

```
    # YOUR CODE ENDS HERE
```

```
    # Record the costs
```

```

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

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

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

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

    return params, grads, costs

```

```

[40]: params, grads, costs = optimize(w, b, X, Y, num_iterations=100, 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"]))
print("Costs = " + str(costs))

optimize_test(optimize)

```

```

w = [[0.80956046]
      [2.0508202 ]]
b = 1.5948713189708588
dw = [[ 0.17860505]
       [-0.04840656]]
db = -0.08888460336847771
Costs = [array(0.15900538)]
All tests passed!

```

Exercise 7 - predict The previous function outputs the learned w and b parameters. Now, we will 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 $\hat{Y} = A = \sigma(w^T X + b)$
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).

```

[41]: # 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 (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of size (num_px * num_px * 3, number of examples)

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

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

# Compute vector "A" predicting the probabilities of a cat being present in
↪ the picture
# YOUR CODE STARTS HERE

A = sigmoid(np.dot(w.T, X) + b)

# YOUR CODE ENDS HERE

for i in range(A.shape[1]):

    # Convert probabilities A[0,i] to actual predictions p[0,i]
    # YOUR CODE STARTS HERE

    Y_prediction[0, i] = 1 if A[0, i] > 0.5 else 0

    # YOUR CODE ENDS HERE

return Y_prediction

```

```

[42]: w = np.array([[0.1124579], [0.23106775]])
      b = -0.3
      X = np.array([[1., -1.1, -3.2], [1.2, 2., 0.1]])
      print ("predictions = " + str(predict(w, b, X)))

      predict_test(predict)

```

```
predictions = [[1. 1. 0.]]
All tests passed!
```

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

5 - Merge all functions into a model

Next, we will put together the building block functions we wrote above, into a single function called, `model()`.

Exercise 8 - model Implement the model function. Use the following notation:

- `Y_prediction_test` for the predictions on the test set
- `Y_prediction_train` for the predictions on the train set
- `parameters, grads, costs` for the outputs of `optimize()`

```
[43]: # 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 we just wrote.

    Arguments:
    X_train -- training set represented by a numpy array of shape (num_px *
    ↪num_px * 3, m_train)
    Y_train -- training labels represented by a numpy array (vector) of shape
    ↪(1, m_train)
    X_test -- test set represented by a numpy array of shape (num_px * num_px *
    ↪3, m_test)
    Y_test -- test labels represented by a numpy array (vector) of shape (1,
    ↪m_test)
    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.
    """

    # General Approach:

    # initialize parameters with zeros
```

```

# Find optimized values for the parameters, w, b.
# make predictions on both the test set and training set examples using the
↳ optimized parms, w, b.

# YOUR CODE STARTS HERE

#X_train.shape[0] gives the number of features (num_px * num_px * 3)
w, b = initialize_with_zeros(X_train.shape[0])

#Optimize parameters using gradient descent
parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations,
↳ learning_rate, print_cost)

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

#Predict on training and test sets
Y_prediction_train = predict(w, b, X_train)
Y_prediction_test = predict(w, b, X_test)

# YOUR CODE ENDS HERE

# Print train/test Errors
if print_cost:
    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

```

```

[44]: from public_tests import *

model_test(model)

```


All tests passed!

If you pass all the tests, run the following cell to train your model.

```
[45]: logistic_regression_model = model(train_set_x, train_set_y, test_set_x,
    ↪test_set_y, num_iterations=2000, learning_rate=0.005, print_cost=True)
```

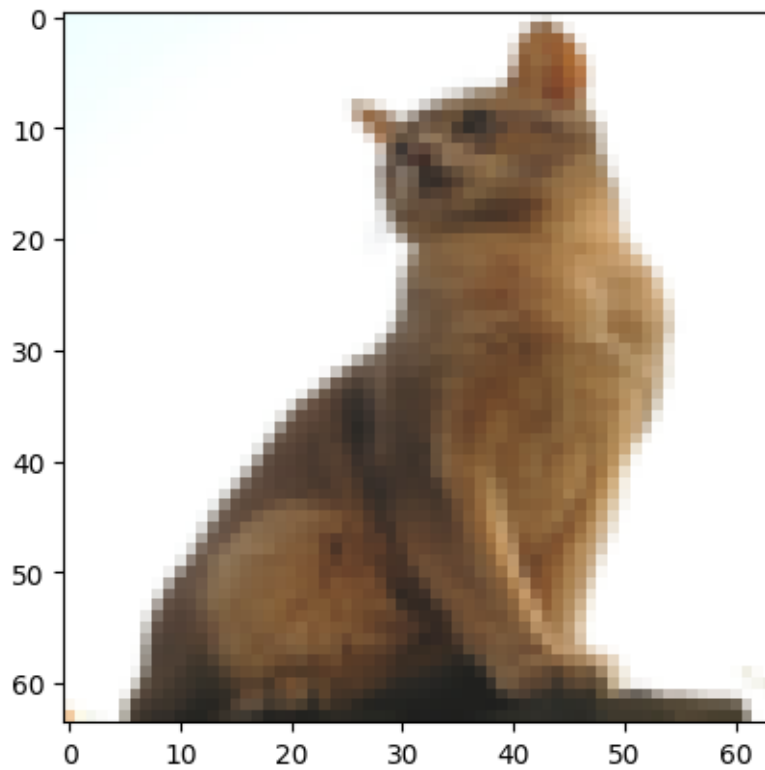
```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 800: 0.242941
Cost after iteration 900: 0.228004
Cost after iteration 1000: 0.214820
Cost after iteration 1100: 0.203078
Cost after iteration 1200: 0.192544
Cost after iteration 1300: 0.183033
Cost after iteration 1400: 0.174399
Cost after iteration 1500: 0.166521
Cost after iteration 1600: 0.159305
Cost after iteration 1700: 0.152667
Cost after iteration 1800: 0.146542
Cost after iteration 1900: 0.140872
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 70%. It is actually not bad for this simple model, given the small dataset we used and that logistic regression is a linear classifier. But no worries, you'll build an even better classifier next week!

Also, you see that the model is clearly overfitting the training data. Later in this specialization you will learn how to reduce overfitting, for example by using regularization. Using the code below (and changing the `index` variable) you can look at predictions on pictures of the test set.

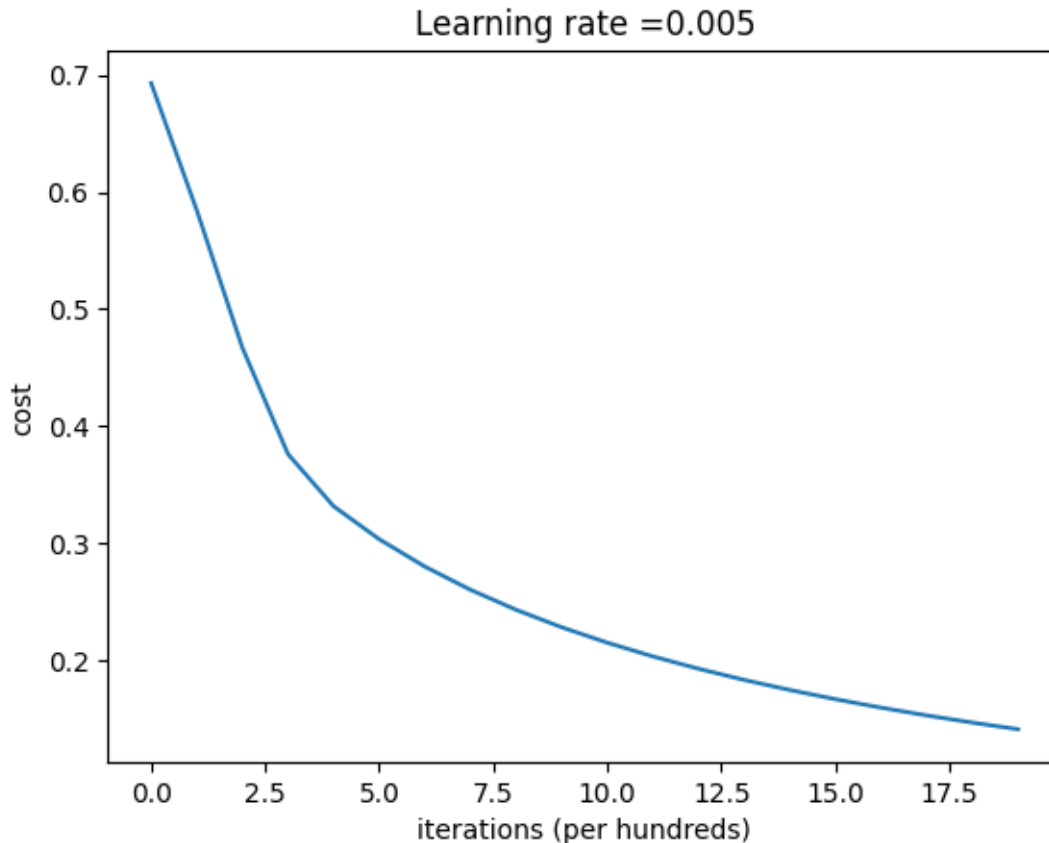
```
[46]: index = 6
plt.imshow(test_set_x[:, index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", the model predicted that this is_
    ↪a \"" +
    ↪classes[int(logistic_regression_model['Y_prediction_test'][0,index])].
    ↪decode("utf-8") + "\" picture.")
```

y = 1, the model predicted that this is a "non-cat" picture.



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

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



Interpretation: We can see the cost decreasing. It shows that the parameters are being learned. However, we also see that we 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. We might see that the training set accuracy goes up, but the test set accuracy goes down, in which case we have an instance of overfitting.

6 - Further analysis (optional/ungraded exercise)

Choice of learning rate **Reminder:** In order for Gradient Descent to work we must choose the learning rate wisely. The learning rate α 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.

```
[48]: 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)')

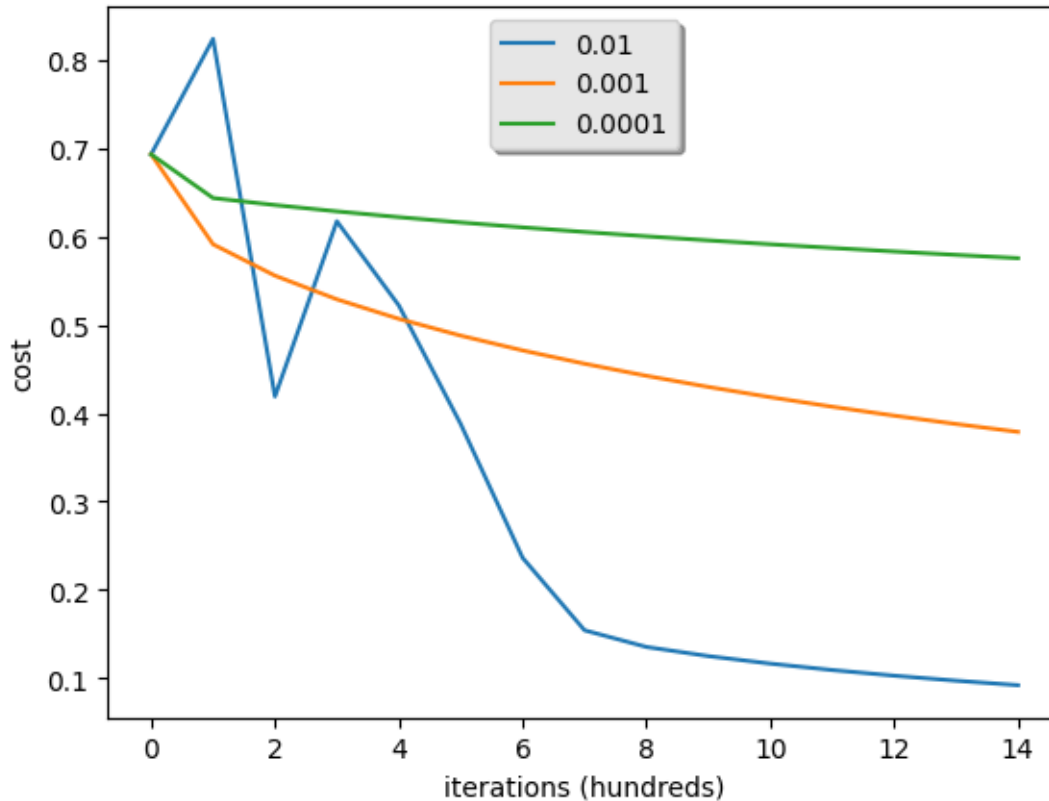
legend = plt.legend(loc='upper center', shadow=True)
frame = legend.get_frame()
frame.set_facecolor('0.90')
plt.show()

```

Training a model with learning rate: 0.01

Training a model with learning rate: 0.001

Training a model with learning rate: 0.0001



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.

What to remember from this assignment:

1. Preprocessing the dataset is important.
2. Software engineering best practice - we implemented and tested each function separately: `initialize()`, `propagate()`, and `optimize()`. Then, after verifying each piece works properly, we combined them into one function called, `model()`.
3. Tuning the learning rate (which is an example of a “hyperparameter”) can make a big difference to the algorithm.

Bibliography: - <http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/>
 - <https://stats.stackexchange.com/questions/211436/why-do-we-normalize-images-by-subtracting-the-datasets-image-mean-and-not-the-c>