

Logistic Regression with a Neural Network mindset

Welcome to your first (required) programming assignment! 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 will also hone your intuitions about deep learning.

****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.

Important Note on Submission to the AutoGrader

Before submitting your assignment to the AutoGrader, please make sure you are not doing the following:

1. You have not added any `_extra_`print`` statement(s) in the assignment.
2. You have not added any `_extra_`code`` cell(s) in the assignment.
3. You have not changed any of the function parameters.
4. You are not using any global variables inside your graded exercises. Unless specifically instructed to do so, please refrain from it and use the local variables instead.
5. You are not changing the assignment code where it is not required, like creating `_extra_`variables``.

If you do any of the following, you will get something like, ``Grader Error: Grader feedback not found`` (or similarly unexpected) error upon submitting your assignment. Before asking for help/debugging the errors in your assignment, check for these first. If this is the case, and you don't remember the changes you have made, you can get a fresh copy of the assignment by following these [\[instructions\]](https://www.coursera.org/learn/neural-networks-deep-learning/supplement/ilwon/h-ow-to-refresh-your-workspace).
(<https://www.coursera.org/learn/neural-networks-deep-learning/supplement/ilwon/h-ow-to-refresh-your-workspace>).

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, let's run the cell below to import all the packages that you will need during this assignment.

- [numpy \(https://numpy.org/doc/1.20/\)](https://numpy.org/doc/1.20/) is the fundamental package for scientific computing with Python.
- [h5py \(http://www.h5py.org\)](http://www.h5py.org) is a common package to interact with a dataset that is stored on an H5 file.
- [matplotlib \(http://matplotlib.org\)](http://matplotlib.org) is a famous library to plot graphs in Python.
- [PIL \(https://pillow.readthedocs.io/en/stable/\)](https://pillow.readthedocs.io/en/stable/) and [scipy \(https://www.scipy.org/\)](https://www.scipy.org/) are used here to test your model with your own picture at the end.

In []: `### v1.2`

```
In [1]: 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
```

2 - Overview of the Problem set

Problem Statement: You 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`

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

We added "`_orig`" 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.

```
In [ ]: # 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
```



Many software bugs in deep learning come from having matrix/vector dimensions that don't fit. If you can keep your matrix/vector dimensions straight you 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]`.

```
In [4]: import numpy as np
import h5py
from lr_utils import load_dataset # Make sure this import is correct

# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_data(...)

# Now, you can safely access the dataset's properties
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1] # or train_set_x_orig.shape[2], since height and width are the same

# Output the values
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). 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 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
```

```
In [5]: # Reshape the training and test examples
# Flatten the images into vectors (num_px * num_px * 3, 1)
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

# Standardize the dataset by dividing by 255 (pixel values range from 0 to 255)
train_set_x = train_set_x_flatten / 255.
test_set_x = test_set_x_flatten / 255.

# Check that the first 10 pixels of the second image are in the correct place
assert np.alltrue(train_set_x_flatten[0:10, 1] == [196, 192, 190, 193, 186, 188, 193, 188, 193, 193])
assert np.alltrue(test_set_x_flatten[0:10, 1] == [115, 110, 111, 137, 129, 129, 146, 147, 148, 149])

# Output the shapes of the resulting datasets
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 center and standardize your dataset, meaning that you subtract 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.

```
In [6]: 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:

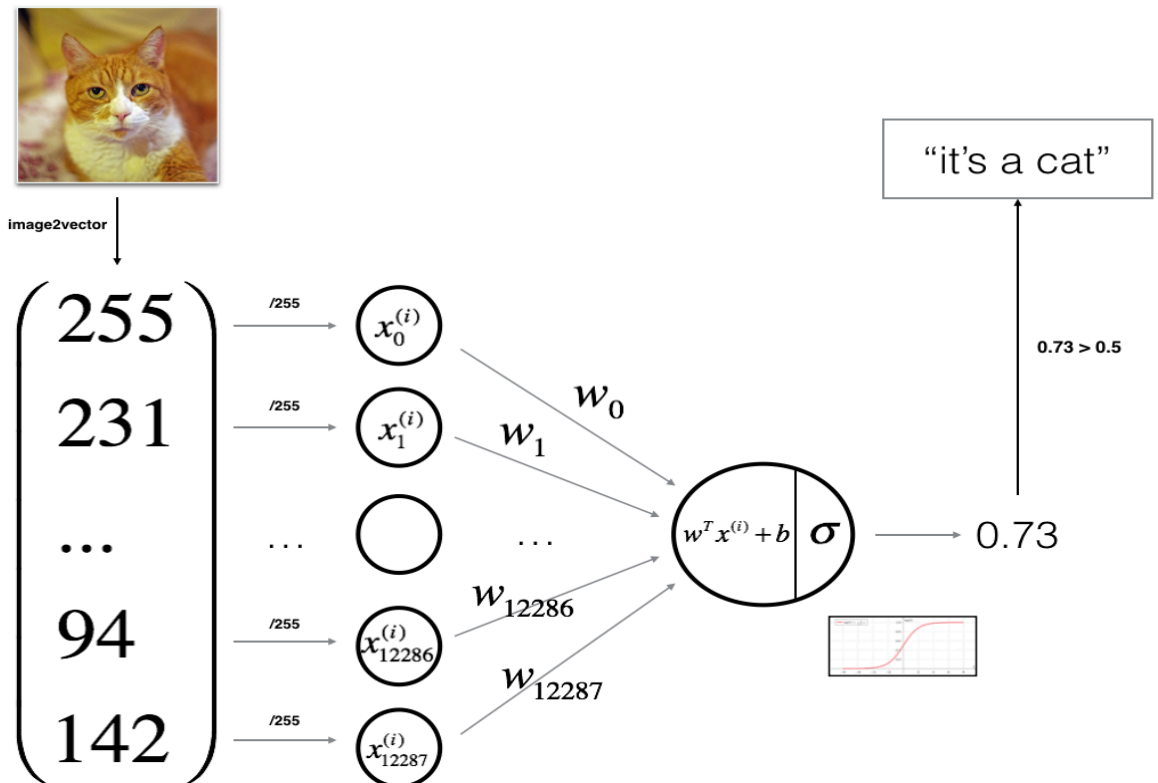
- Figure out the dimensions and shapes of the problem (m_{train} , m_{test} , num_px , ...)

- Reshape the datasets such that each example is now a vector of size (num_px * num_px * 3, 1)
- "Standardize" the data

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 $x^{(i)}$:

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

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

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

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, 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

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()` .

4.1 - Helper functions

Exercise 3 - sigmoid

Using your code from "Python Basics", implement `sigmoid()` . 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()`.

```
In [7]: import numpy as np

# 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)
    """
    # Compute the sigmoid function using np.exp()
    s = 1 / (1 + np.exp(-z))

    return s

# Testing the sigmoid function
print("sigmoid([0, 2]) = " + str(sigmoid(np.array([0, 2]))))

# Running the sigmoid test with the sigmoid function
x = np.array([0.5, 0, 2.0])
output = sigmoid(x)
print(output)
```

```
sigmoid([0, 2]) = [0.5          0.88079708]
[0.62245933 0.5          0.88079708]
```

```
In [8]: print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2])))  
  
sigmoid_test(sigmoid)  
  
sigmoid([0, 2]) = [0.5          0.88079708]  
All tests passed!
```

```
In [9]: 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. If you don't know what numpy function to use, look up `np.zeros()` in the Numpy library's documentation.


```
In [10]: import numpy as np

# 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
    the bias b as a scalar.

    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
    """
    # Initialize w as a zero vector of shape (dim, 1)
    w = np.zeros((dim, 1))

    # Initialize b to 0
    b = 0.0

    return w, b

# Example: Test the function
dim = 2
w, b = initialize_with_zeros(dim)

# Check if b is a float and print the results
assert type(b) == float
print("w = " + str(w))
print("b = " + str(b))

# Test functions (you can assume these are defined to check correctness)
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!

```
In [11]: 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 your parameters are initialized, you can do the "forward" and "backward" propagation steps for learning the parameters.

Exercise 5 - propagate

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

Hints:

Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \dots, a^{(m-1)}, a^{(m)})$
- 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:

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

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


```

In [12]: import numpy as np

# GRADED FUNCTION: propagate
def propagate(w, b, X, Y):
    """
    Implement the cost function and its gradient for the propagation explained
    in the video.

    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)

    Returns:
    grads -- dictionary containing the gradients of the weights and bias
             (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
    cost -- negative log-likelihood cost for logistic regression
    """

    m = X.shape[1] # Number of examples

    # FORWARD PROPAGATION (FROM X TO COST)
    # Compute activation A using the sigmoid function
    z = np.dot(w.T, X) + b # (1, m)
    A = 1 / (1 + np.exp(-z)) # Apply sigmoid

    # Compute cost function
    cost = -np.mean(Y * np.log(A) + (1 - Y) * np.log(1 - A))

    # BACKWARD PROPAGATION (TO FIND GRADIENTS)
    # Gradient of the cost function with respect to w
    dw = np.dot(X, (A - Y).T) / m # Shape (num_px * num_px * 3, 1)

    # Gradient of the cost function with respect to b
    db = np.sum(A - Y) / m # Shape (1,)

    # Store gradients in a dictionary
    grads = {"dw": dw, "db": db}

    # Return both the gradients and the cost
    return grads, cost

# Test example
w = np.array([[1.], [2.]]) # 2 features, 1 column
b = 1.5
X = np.array([[1., -2., -1.], [3., 0.5, -3.2]]) # 2 features, 3 examples
Y = np.array([[1, 1, 0]]) # True labels for the 3 examples

# Call the propagate function
grads, cost = propagate(w, b, X, Y)

# Check the result
assert type(grads["dw"]) == np.ndarray
assert grads["dw"].shape == (2, 1) # Shape of the gradient w.r.t w
assert type(grads["db"]) == np.float64 # Scalar for the gradient w.r.t b

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

```

```
dw = [[ 0.25071532]
      [-0.06604096]]
db = -0.12500404500439652
cost = 0.15900537707692405
```

```
In [13]: w = np.array([[1.], [2]])
        b = 1.5

        # X is using 3 examples, with 2 features each
        # Each example is stacked column-wise
        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 = -0.12500404500439652
cost = 0.15900537707692405
All tests passed!
```

Expected output

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

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 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 $\theta = \theta - \alpha d\theta$, where α is the learning rate.

```

In [16]: import copy

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, num_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 parameters
    costs -- list of all the costs computed during the optimization, this will have a length of num_iterations
    """

    w = copy.deepcopy(w)
    b = copy.deepcopy(b)

    costs = []

    for i in range(num_iterations):
        # Cost and gradient calculation
        grads, cost = propagate(w, b, X, Y) # Assuming the propagate function returns a dictionary of gradients and a scalar cost

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

        # Update rule
        w = w - learning_rate * dw
        b = b - learning_rate * db

        # 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

```

```
In [17]: params, grads, costs = optimize(w, b, X, Y, num_iterations=100, learning_rate=0.01)

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 = [0.15900537707692405]
All tests passed!
```

Exercise 7 - predict

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 $\hat{Y} = A = \sigma(w^T X + b)$
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).

```

In [18]: import numpy as np

def sigmoid(z):
    """Sigmoid activation function"""
    return 1 / (1 + np.exp(-z))

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

    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] # number of examples
    Y_prediction = np.zeros((1, m)) # Initialize predictions array
    w = w.reshape(X.shape[0], 1) # Reshape w to match the dimensions

    # Compute vector A, predicting the probabilities of a cat being present in the image
    A = sigmoid(np.dot(w.T, X) + b) # Sigmoid activation

    # Convert probabilities A[0, i] to actual predictions
    for i in range(A.shape[1]):
        if A[0, i] > 0.5:
            Y_prediction[0, i] = 1
        else:
            Y_prediction[0, i] = 0

    return Y_prediction

# Example input
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)))

```

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

```

In [19]: 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

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.

Exercise 8 - model

Implement the model function. Use the following notation:

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


```

In [20]: import numpy as np

def model(X_train, Y_train, X_test, Y_test, num_iterations=2000, learning_rate=0.01,
         print_cost=True):
    """
    Builds the logistic regression model by calling the function you've implemented in previous weeks.

    Arguments:
    X_train -- training set represented by a numpy array of shape (num_px * num_px * 3, num_train)
    Y_train -- training labels represented by a numpy array (vector) of shape (num_train)
    X_test -- test set represented by a numpy array of shape (num_px * num_px * 3, num_test)
    Y_test -- test labels represented by a numpy array (vector) of shape (num_test)
    num_iterations -- hyperparameter representing the number of iterations to optimize the parameters
    learning_rate -- hyperparameter representing the learning rate used in the optimization process
    print_cost -- Set to True to print the cost every 100 iterations

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

    # Initialize parameters
    w = np.zeros((X_train.shape[0], 1)) # Initialize weights as zeros
    b = 0 # Initialize bias to zero

    # Gradient descent
    params, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost)

    # Retrieve parameters
    w = params["w"]
    b = params["b"]

    # Make predictions on test and train sets
    Y_prediction_train = predict(w, b, X_train)
    Y_prediction_test = predict(w, b, X_test)


    # Print train/test Errors
    if print_cost:
        train_accuracy = 100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100
        test_accuracy = 100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100
        print(f"train accuracy: {train_accuracy} %")
        print(f"test accuracy: {test_accuracy} %")

    # Create a dictionary to store model results
    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

# Example usage with some training and test sets
# Assuming train_set_x, train_set_y, test_set_x, test_set_y are preloaded
logistic_regression_model = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations=2000, learning_rate=0.01, 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 %
```

```
In [22]: from public_tests import *

model_test(model)
```

All tests passed!

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

```
In [23]: logistic_regression_model = model(train_set_x, train_set_y, test_set_x, test_s
```

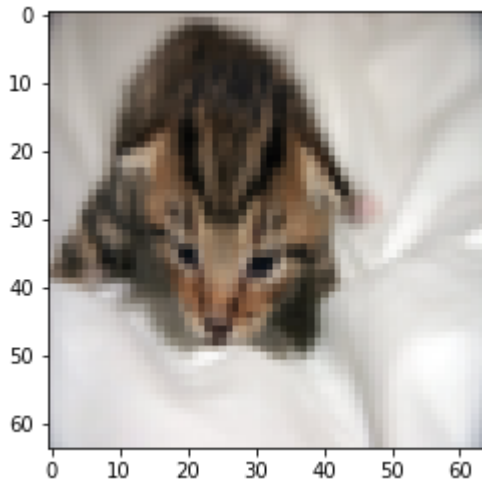
```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376007
Cost after iteration 400: 0.331463
Cost after iteration 500: 0.303273
Cost after iteration 600: 0.279880
Cost after iteration 700: 0.260042
Cost after iteration 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.

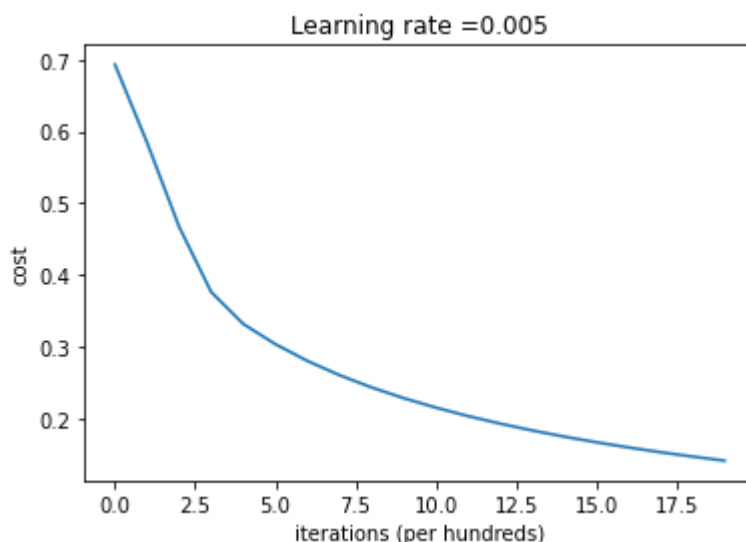
```
In [24]: # Example of a picture that was wrongly classified.
index = 1
plt.imshow(test_set_x[:, index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", you predicted that it is a \"")
```

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



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

```
In [31]: # 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: 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.

6 - Further analysis (optional/ungraded exercise)

Congratulations on building your first image classification model. Let's analyze it further, and examine possible choices for the learning rate α .

Choice of learning rate

Reminder: In order for Gradient Descent to work you 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.

```
In [27]: import matplotlib.pyplot as plt
import numpy as np

# Different Learning rates to analyze
learning_rates = [0.01, 0.001, 0.0001]
models = {}

# Train a model for each learning rate
for lr in learning_rates:
    print(f"Training a model with learning rate: {lr}")
    models[str(lr)] = model(train_set_x, train_set_y, test_set_x, test_set_y,
    print('\n' + "-----" + '

# Plot the costs for each learning rate
for lr in learning_rates:
    plt.plot(np.squeeze(models[str(lr)]["costs"]), label=f"Learning rate: {lr}")

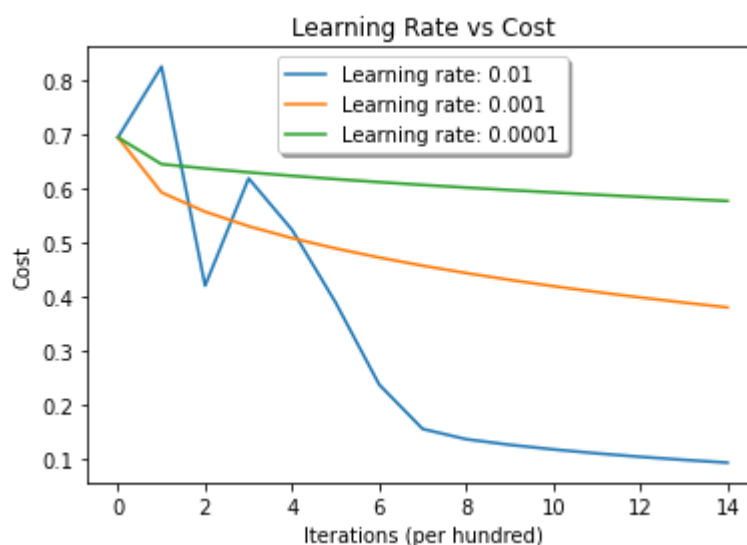
plt.ylabel('Cost')
plt.xlabel('Iterations (per hundred)')
plt.title("Learning Rate vs Cost")
plt.legend(loc='upper center', shadow=True)

# Display the plot
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.
- In deep learning, we usually recommend that you:
 - Choose the learning rate that better minimizes the cost function.
 - If your model overfits, use other techniques to reduce overfitting. (We'll talk about this in later videos.)

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

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

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

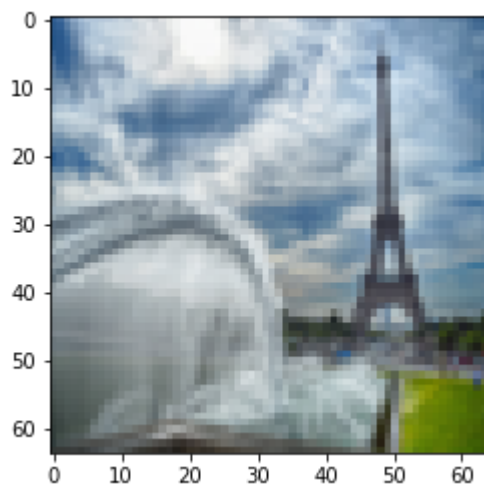
```
In [28]: # change this to the name of your image file
my_image = "my_image.jpg" # Replace with the actual filename of your image

# We preprocess the image to fit your algorithm.
fname = "images/" + my_image
image = np.array(Image.open(fname).resize((num_px, num_px))) # Resize to match
plt.imshow(image)
plt.show()

# Normalize the image and reshape it to the correct format
image = image / 255.
image = image.reshape((1, num_px * num_px * 3)).T

# Make a prediction using the trained model
my_predicted_image = predict(logistic_regression_model["w"], logistic_regression_model["b"], image)

# Output the prediction result
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a " +
      classes[int(np.squeeze(my_predicted_image))].decode("utf-8") + " picture")
```



$y = 0.0$, your algorithm predicts that it is a "non-cat" picture.

What to remember from this assignment:

1. Preprocessing the dataset is important.
2. You implemented each function separately: `initialize()`, `propagate()`, `optimize()`. Then you built a model().
3. Tuning the learning rate (which is an example of a "hyperparameter") can make a big difference to the algorithm. You will see more examples of this later in this course!

Finally, if you'd like, we invite you to try different things on this Notebook. Make sure you submit before trying anything. Once you submit, things you can play with include:

- Play with the learning rate and the number of iterations
- Try different initialization methods and compare the results
- Test other preprocessings (center the data, or divide each row by its standard deviation)

Bibliography:

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