Logistic Regression with a Neural Network mindset

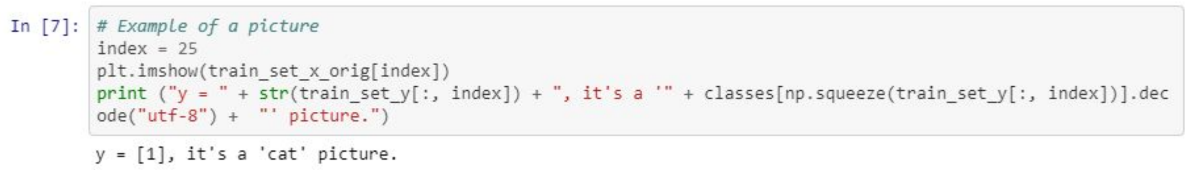
Problem Statement:

We are given a dataset containing a training set of images labeled as cat or non-cat and a test set of images labeled as cat or non-cat.
All the images are RGB square images.

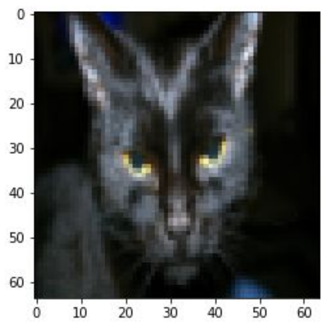
Required:
build a simple image-recognition algorithm that can correctly classify pictures as cat or non-cat**.**

Steps:
We started by Loading the data which are divided into four sections. Two of them are for the train and test which we want to process them, and the other two data sets are already processed to use them as a reference after we finish the process of the first two sets.

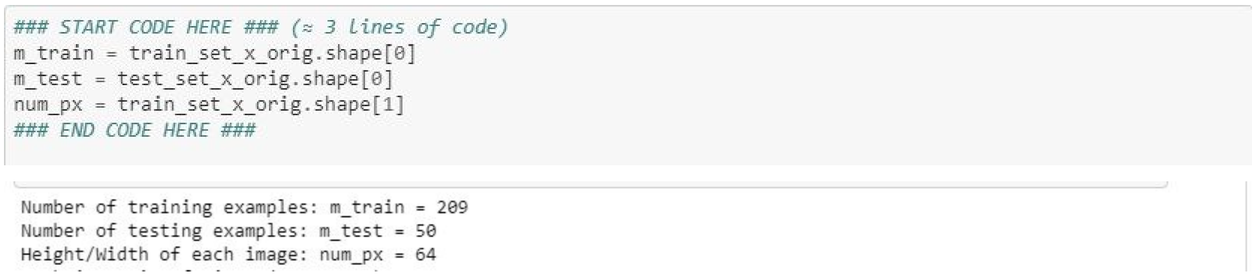
The following picture is from our trained dataset we are sure that it's correct:



```
In [7]: # 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])].dec
        ode("utf-8") +  "' picture.")

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



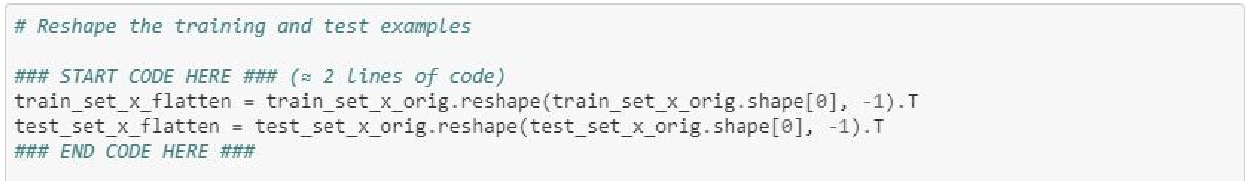
Then we used the numpy package to find the number of images we for the training and testing data sets and the dimension of each picture use the following code:



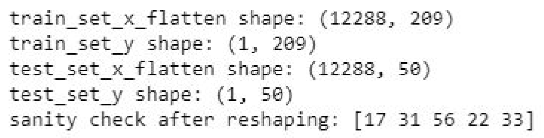
```
### START CODE HERE ### (≈ 3 lines of code)
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]
### END CODE HERE ###
```

```
Number of training examples: m_train = 209
Number of testing examples: m_test = 50
Height/Width of each image: num_px = 64
```

Then we needed to reshape training and test data sets so that images are flattened into single vectors

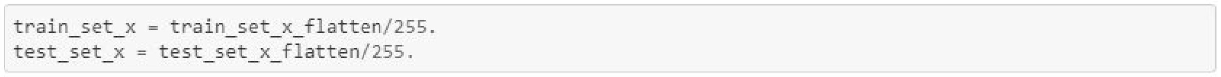


```python
# Reshape the training and test examples

### START CODE HERE ### (≈ 2 lines of code)
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
### END CODE HERE ###
```



```
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)
sanity check after reshaping: [17 31 56 22 33]
```

In order 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 so we needed to standardize or normalize our dataset.

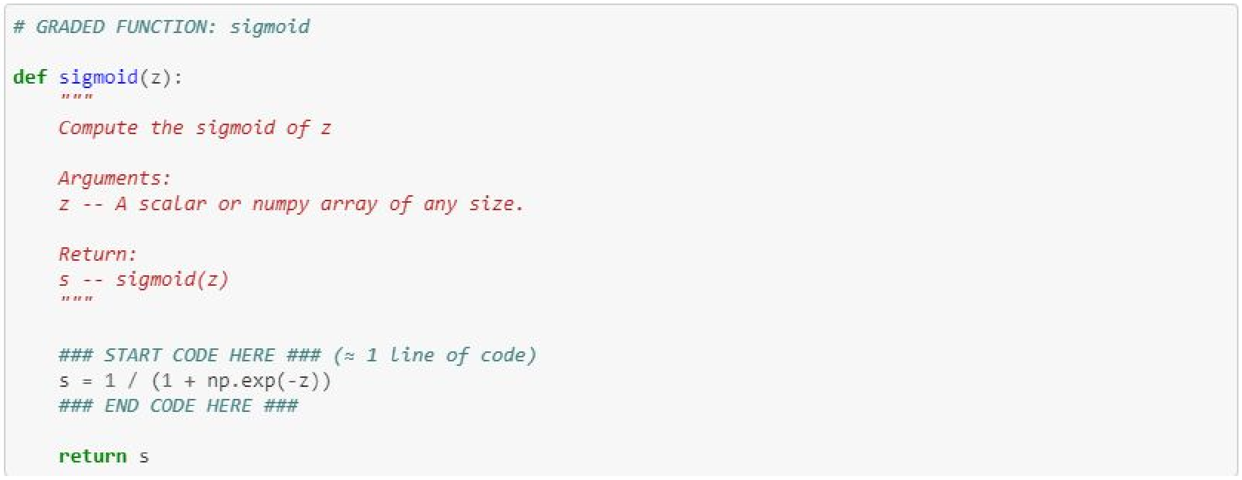


```python
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

In order to implement our learning algorithm we need the following steps:

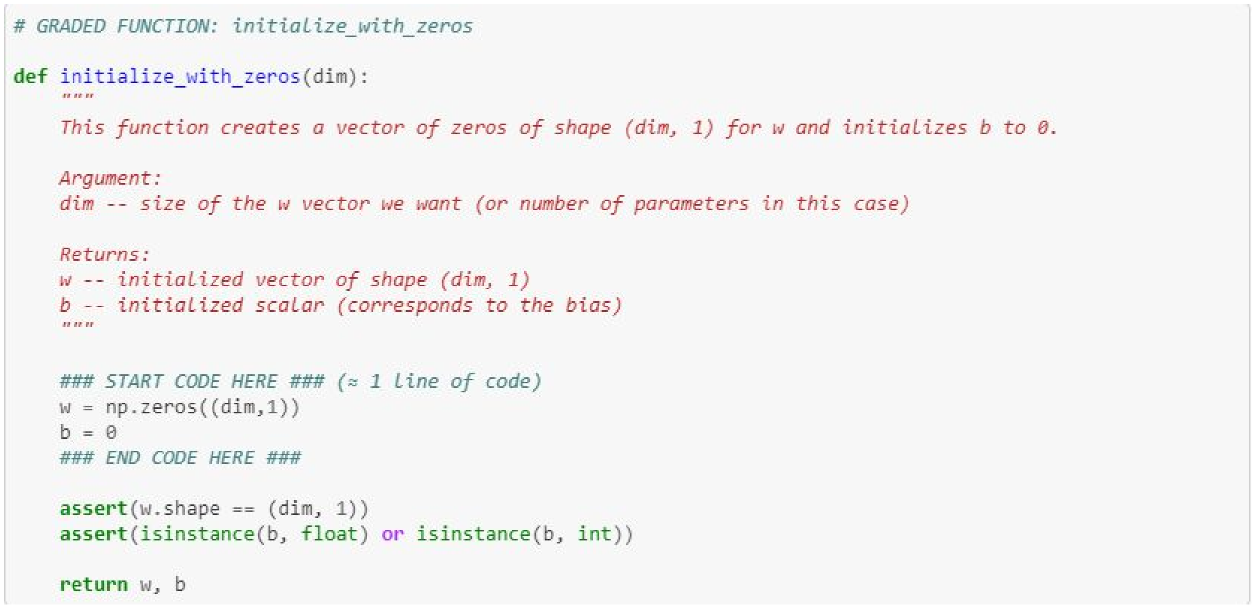
1. **Define the model structure (such as number of input features)**
2. **Initialize the model's parameters**
3. **Loop:**
    - **Calculate current loss (forward propagation)**
    - **Calculate current gradient (backward propagation)**
    - **Update parameters (gradient descent)**

So we started by building a function to calculate the sigmoid function using the numpy package :



```python
# 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
```

Then we needed to build a function to initialize the parameters of our learning algorithm to zeros



```python
# 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)
    """

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

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

    return w, b
```

After initializing our parameters we needed to do the "forward" and "backward" propagation steps for learning the parameters in order to our cost function and its gradient

```
    # FORWARD PROPAGATION (FROM X TO COST)
    ### START CODE HERE ### (≈ 2 lines of code)
    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), axis = 1)      # compute cost
    ### END CODE HERE ###

    # BACKWARD PROPAGATION (TO FIND GRAD)
    ### START CODE HERE ### (≈ 2 lines of code)
    dw = (1/m) * np.dot(X, (A-Y).T)
    db = (1/m) * np.sum(A - Y, axis = 1)
    ### END CODE HERE ###
```

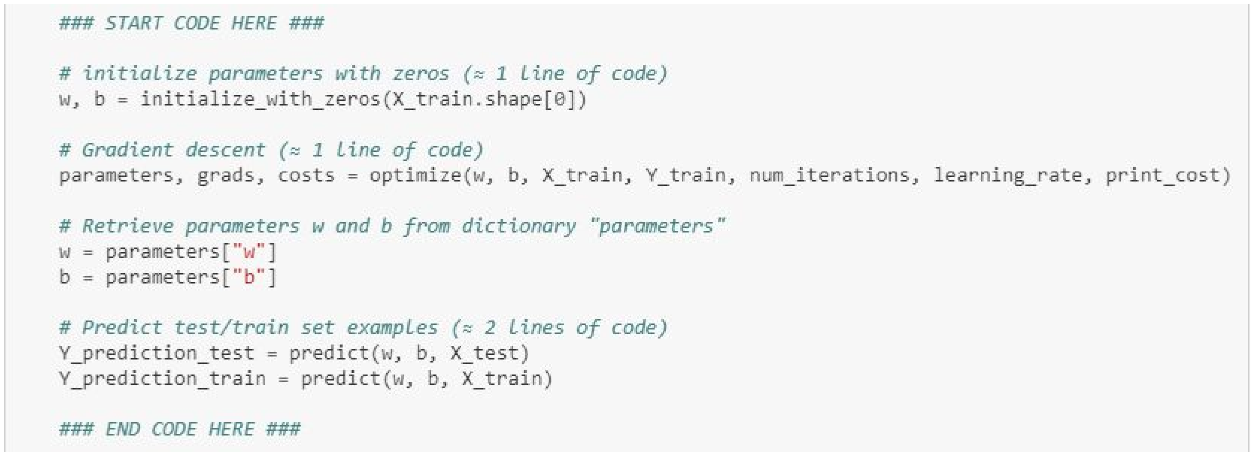
In order to update the parameters using gradient descent we implemented an optimization function for it to learn w and b by minimizing the cost function J:

```
        # Cost and gradient calculation (≈ 1-4 lines of code)
        ### START CODE HERE ###
        m = X.shape[1]

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

        dw = None
        db = None

        dw = (1/m) * np.dot(X, (A-Y).T)
        db = (1/m) * np.sum(A - Y, axis = 1)

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

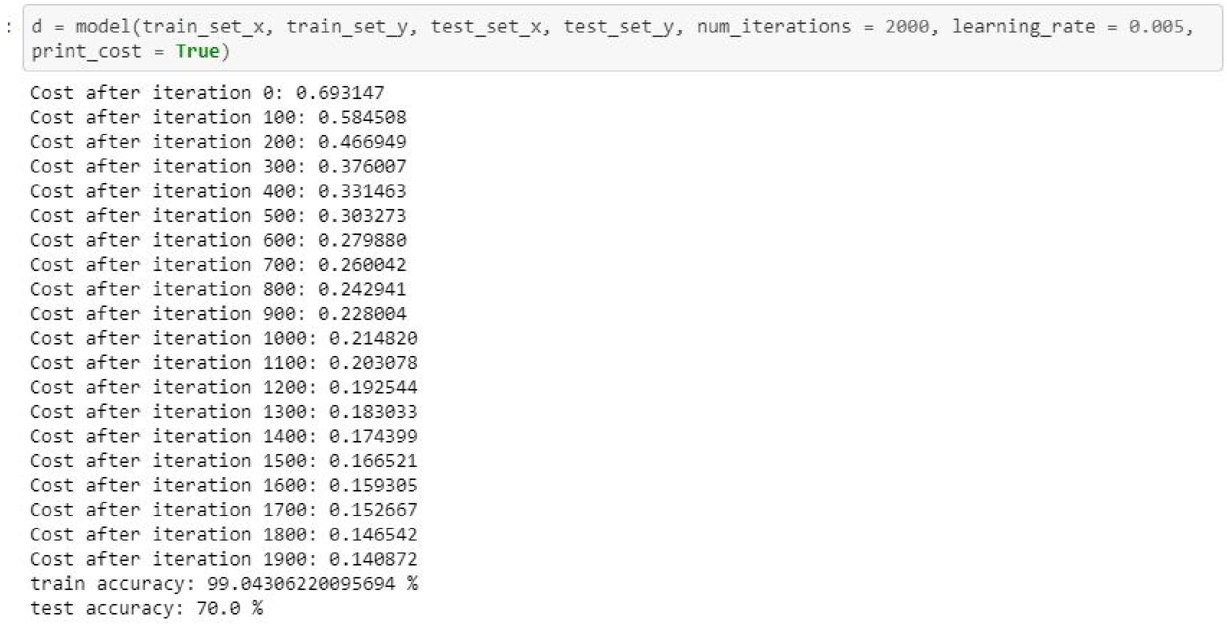
The using our parameter w and b to predict the labels for our dataset by Implement the predict() function:

```
    # 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(w.T,X) + b)
    ### END CODE HERE ###

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

        # Convert probabilities A[0,i] to actual predictions p[0,i]
        ### START CODE HERE ### (≈ 4 lines of code)
        if A[0,i] <= 0.5:
            Y_prediction[0,i] = 0
        else:
            Y_prediction[0,i] = 1
        pass
        ### END CODE HERE ###
```

Then we merged our all function into a structured model so that we can train our model:



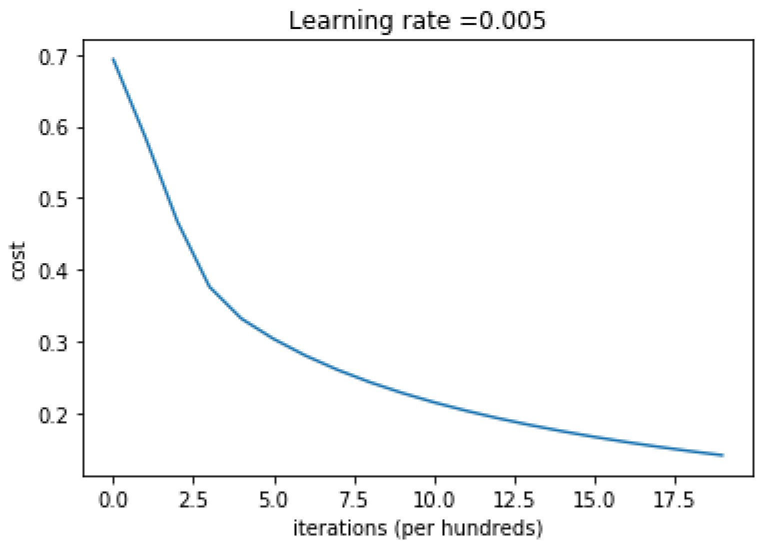
```
### START CODE HERE ###

# initialize parameters with zeros (≈ 1 line of code)
w, b = initialize_with_zeros(X_train.shape[0])

# 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 ###
```



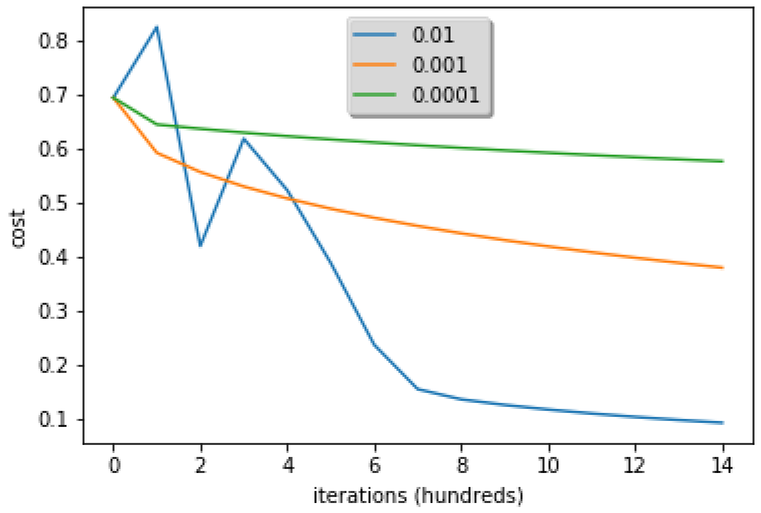
```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, learning_rate = 0.005,
print_cost = True)
```

```
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 %
```

Plot for the cost function and the gradients.



In order for Gradient Descent to work we must choose the learning rate wisely. The learning rate alpha 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



Finally we test our model by injecting a non photo cat to see the result:

```python
## START CODE HERE ## (PUT YOUR IMAGE NAME)
my_image = "my_image.jpg"   # change this to the name of your image file
## END CODE HERE ##

# We preprocess the image to fit your algorithm.
fname = "images/" + my_image
image = np.array(ndimage.imread(fname, flatten=False))
image = image/255.
my_image = scipy.misc.imresize(image, size=(num_px,num_px)).reshape((1, num_px*num_px*3)).T
my_predicted_image = predict(d["w"], d["b"], my_image)

plt.imshow(image)
print("y = " + str(np.squeeze(my_predicted_image)) + ", your algorithm predicts a \"" + classes[int(np.squu
eeze(my_predicted_image)),].decode("utf-8") +  "\" picture.")
```

```
y = 0.0, your algorithm predicts a "non-cat" picture.
```