Logistic Regression with a Neural Network mindset

Welcome to your first (required) programming exercise. This exercise will guide you through a logistic regression classifier to recognize cats. This assignment will take you through the steps to build a simple feature-based recognizer, and as an extra bonus your exclusive dream deep learning framework!

Instructions:

- Do not use loops (for/while) in your code, unless the instructions explicitly ask you to do so.

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 - Packages

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

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

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        import h5py
        import scipy
        from PIL import Image
        from scipy.ndimage import ndimage
        from lr_utils import load_dataset

        %matplotlib inline
```

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 = 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.

```
In [2]: # Loading the data (cat/non-cat)
        train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()
```

We add "_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 values index and re-run to see other images.

```
In [3]: # 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 you see bugs your matrix/vector dimensions straight you will go a long way toward eliminating many bugs.

Exercise: 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]: #(m_train = m_train_set_x_orig.shape[0])
        #(m_test = m_test_set_x_orig.shape[0])
        #(num_px = m_train_set_x_orig.shape[1])
```

```
In [5]: m_train = 209
        m_test = 50
        num_px = 64
        print("m_train: {0:d}".
              format(m_train))
        print("m_test: {0:d}".
              format(m_test))
        print("num_px: {0:d}".
              format(num_px))
        print("train_image_is_square: {0:d}".
              format(int(num_px == train_set_x_orig.shape[1])))
        print("train_image_is_color: {0:d}".
              format(int(train_set_x_orig.shape[3] == 3)))
        print("train_set_x_shape: {0:s}".
              format(str(train_set_x_orig.shape)))
        print("train_set_y_shape: {0:s}".
              format(str(train_set_y.shape)))
        print("test_set_x_shape: {0:s}".
              format(str(test_set_x_orig.shape)))
        print("test_set_y_shape: {0:s}".
              format(str(test_set_y.shape)))
```

```
m_train: 209
m_test: 50
num_px: 64
train_image_is_square: 1
train_image_is_color: 1
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: 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 [6]: # Reshape the training and test examples
        ndef shape(train_set_x_orig, m_train, m_test, num_px):
        ndef shape(train_set_x_orig, m_train, m_test, num_px):
        train_set_x_flatten = train_set_x_orig.reshape(m_train, -1).T
        test_set_x_flatten = test_set_x_orig.reshape(m_test, -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))
        print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0:5]))
```

```
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]
 [43 67 115 72 78]
 [ 1 19 62 20 10]
 [87 96 148 94 153]
 [ 0 43 150 43 62]]
```

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

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and the value of each pixel is a third number in the range of zero to one for each channel.

We usually preprocess the dataset by centering and standardizing it.

For the purposes of this assignment, it's more convenient and sufficient to simply divide every row of the dataset by 255 (the maximum value of a pixel channel).

Let's standardize our dataset.

```
In [7]: 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 shape (num_px * num_px * 3, 1)
- "Standardize" the data

3 - General Architecture of the learning algorithm

It is time to design a simple image recognition algorithm that can correctly classify pictures as cat or non-cat.

You will build a logistic classifier, 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 $x^{(i)}$:

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

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

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

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)
- Analyze 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 inputs 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: Using your code from "Python Basics", implement sigmoid(). As you've seen in the figure above, you need to compute $sigmoid(z) = \frac{1}{1 + e^{-z}}$ for z = np.dot(w.T,x)+b, to make predictions. Use np.exp().

```
In [8]: # 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)

            ndef sigmoid(z):
            s = 1.0 / (1.0 + np.exp(-z))
            return s
```

```
In [9]: print ("sigmoid(0) = " + str(sigmoid(0)))
        print ("sigmoid(9.2) = " + str(sigmoid(9.2)))
```

```
sigmoid(0) = 0.5
sigmoid(9.2) = 0.999898987822
```

Expected Output:

sigmoid(0) =	0.5
sigmoid(9.2) =	0.999898987822

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

4.2 - 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 $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, ..., a^{(m-1)}, a^{(m)})$
- You calculate the cost function: $J = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log(a^{(i)})+(1-y^{(i)})\log(1-a^{(i)}))$ (1)
$$\frac{\partial J}{\partial w} = \frac{1}{m}X(A-Y)^T \tag{7}$$
$$\frac{\partial J}{\partial b} = \frac{1}{m}\sum_{i=1}^{m}(a^{(i)}-y^{(i)}) \tag{8}$$

```
In [10]: # 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

             Tips:
             - Write your code step by step for the propagation. np.log(), np.dot()

             m = X.shape[1]

             # FORWARD PROPAGATION (FROM X TO COST)
             ndef propagate(w, b, X, Y):
             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

             # BACKWARD PROPAGATION (TO FIND GRAD)
             ndef propagate(w, b, X, Y):
             dw = 1/m*(np.dot(X,(A-Y).T))
             db = 1/m*(np.sum(A-Y))
             cost = np.squeeze(cost)

             assert(cost.shape == ())

             # GRADIENT DICTIONARY
             grads = {"dw": dw,
                      "db": db}

             return grads, cost
```

```
In [11]: w, b, X, Y = np.array([[1.],[2.]]), 2., np.array([[1.,2.,-1.],[3.,4.,-3.2]]), np.array([[1,0,1]])
         grads, cost = propagate(w, b, X, Y)
         print ("dw = " + str(grads["dw"]))
         print ("db = " + str(grads["db"]))
         print ("cost = " + str(cost))
```

```
dw = [[ 0.99993216]
 [ 1.99980262]]
db = 0.499935230625
cost = 6.000064773192205
```

Expected Output:

dw =	[[0.99993216] [1.99980262]]
db =	0.499935230625
cost =	6.000064773192205

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

```
In [12]: # GRADED FUNCTION: optimize

         ndef 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 (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 done
             learning_rate -- learning rate of the gradient descent update rule
             print_cost -- True to print the loss every 100 steps

             Return:
             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 things: cost and gradients.
             It includes the loss and the gradient for the cost function parameters. Not propagate().
             It takes the gradients and updates the parameters. Gradient descent rule is: θ = θ - α dθ

             costs = []

             for i in range(num_iterations):

                 # Cost and gradient calculation (≈ 1-2 lines of code)
                 grads, cost = propagate(w, b, X, Y)

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

                 # gradient rule (≈ 2 lines of code)
                 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 examples
                 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
```

```
In [13]: 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"]))
```

```
w = [[0.19033591]
 [0.12259159]]
b = 1.92535983008
dw = [[ 0.67752042]
 [ 1.41625495]]
db = 0.219633398303
```

Expected Output:

w =	[[0.19033591] [0.12259159]]
b =	1.92535983008
dw =	[[0.67752042] [1.41625495]]
db =	0.219633398303

```
In [14]: params, grads, costs = optimize(w, b, X, Y, num_iterations=100, learning_rate = 0.009, print_cost = True)
         print ("w = " + str(params["w"]))
         print ("b = " + str(params["b"]))
         print ("dw = " + str(grads["dw"]))
         print ("db = " + str(grads["db"]))
```

```
Cost after iteration 0: 5.801545
Cost after iteration 100: 1.000372
w = [[0.19033591]
 [0.12259159]]
b = 1.92535983008
dw = [[ 0.67752042]
 [ 1.41625495]]
db = 0.219633398303
```

Expected Output:

Cost after iteration 0:	5.801545
Cost after iteration 100:	1.000372
w =	[[0.19033591] [0.12259159]]
b =	1.92535983008
dw =	[[0.67752042] [1.41625495]]
db =	0.219633398303

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 is 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 like, you can use if/else statements in a for loop (though there is also a way to vectorize this).

```
In [15]: # 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)

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

             ndef predict(w, b, X):
             for i in range(A.shape[1]):

                 # Convert probabilities A[0,i] to actual predictions p[0,i]
                 ndef predict(w, b, X):
                 if A[0,i] > 0.5:
                     ndef predict(w, b, X):
                     Y_prediction[0,i] = 1
                     else:
                     Y_prediction[0,i] = 0

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

             return Y_prediction
```

```
In [16]: w = np.array([[0.1124579],[0.23475715]])
         b = -0.29745588
         print ("predictions = " + str(predict(w, b, X)))
```

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

Expected Output:

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

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: Implement the model function. Use the following notation:

- Y_prediction for your predictions on the test set
- Y_prediction_train for your predictions on the train set
- w, costs, grads for the outputs of optimize()

```
In [17]: # GRADED FUNCTION: model

         ndef 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 functions you've implemented previously

             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 -- True to print the cost every 100 iterations

             Return:
             d -- dictionary containing information about the model.
             '''

             ndef model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):
             w, b = np.zeros(X_train.shape[0], 1), 0

             # Gradient descent (≈ 1 line of code)
             ndef model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):
             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)
             ndef model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):
             Y_prediction_test = predict(w, b, X_test)
             ndef model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):
             Y_prediction_train = predict(w, b, X_train)

             ndef model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_rate = 0.5, print_cost = False):
             # 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.

```
In [18]: m_train,m_test,num_px = train_set_x.shape[1], test_set_x.shape[1], train_set_x.shape[2]
         d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 2000, learning_rate = 0.01, print_cost =
         True)
```

Expected Output:

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.584508
Cost after iteration 200: 0.466949
Cost after iteration 300: 0.376069
Cost after iteration 400: 0.331157
Cost after iteration 500: 0.303771
Cost after iteration 600: 0.268423
Cost after iteration 700: 0.239711
Cost after iteration 800: 0.226792
Cost after iteration 900: 0.211521
Cost after iteration 1000: 0.202304
Cost after iteration 1100: 0.192774
Cost after iteration 1200: 0.183841
Cost after iteration 1300: 0.175890
Cost after iteration 1400: 0.168501
Cost after iteration 1500: 0.161189
Cost after iteration 1600: 0.154113
Cost after iteration 1700: 0.147366
Cost after iteration 1800: 0.141696
Cost after iteration 1900: 0.136790
Cost after iteration 2000: 0.132213
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 error is 68%. It is actually not bad for this simple model, given the small dataset used and that logistic regression is a linear classifier. But no worries, you'll see how we can do even better later.

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 training and test sets, you can also begin to see how this model can be used on predictions on pictures of the test set.

```
In [19]: # Example of a picture that was wrongly classified.
         index = 5
         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 \"" + classes[d["Y_prediction_test"",index
         ].decode("utf-8") + "\" picture.")
```

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


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

```
In [20]: # 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. Find the best value for the learning rate and plot again, you will find that set accuracy goes down. This is called overfitting.

6 - Further analysis (optional/ungraded exercise)

Congratulations on building your first image classification model. Let's further 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 optimal value. That's why it is useful to see a range of different learning rates.

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 [21]: # Training curve for different learning rates
         learning_rates = [0.01, 0.001, 0.0001]
         models = {}
         for i in learning_rates:
             print ("learning rate is: " + str(i))
             models[str(i)] = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 1500, learning_rate = i,
             print_cost = False)
             print ('\n' + "-------------------------------------------------------" + '\n')

         for i in learning_rates:
             plt.plot(np.squeeze(models[str(i)]["costs"]), label= str(models[str(i)]["learning_rate"]))

         plt.ylabel('cost')
         plt.xlabel('iterations')

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


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 cost of even 1 means that a single mislabeled pixel has enough power to keep the model from perfectly fitting the training dataset.
- A lower learning rate (0.001) means that the cost keeps decreasing, but it shows that we could have chosen a better learning rate.
- For the learning rate 0.0001, the cost still shows an downward trend, but it makes too much time to converge.
- Overall, a learning rate that is too high may hinder convergence. And a value that is too low is not optimal since it slows the learning process.

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. Add your image to this Jupyter Notebook directory, in the folder named "images"
2. Change your image's name in the following code
3. Run the code and see if the algorithm is right (1 = cat, 0 = non-cat)!

```
In [22]: ## START CODE HERE ##
         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))
         my_image = scipy.misc.imresize(image, size=(num_px,num_px),interp='bicubic').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.squeeze(my_predi
         cted_image)),].decode("utf-8") + "\" 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 hyperparameter) can lead to a much better model.

Hope to see more examples of this kind from you in your future work!

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 dataset, or divide each row by its standard deviation)

Bibliography:

- http://www.wildml.com/2015/09/implementing-a-neural-network-from-scratch/
- https://stats.stackexchange.com/questions/211889/understanding-feature-scaling-in-machine-learning, suggesting the standard deviation