DL-App

May 16, 2019

1 Deep Neural Network for Image Classification: Application

You will use use the functions you'd implemented in the previous assignment to build a deep network, and apply it to cat vs non-cat classification. Hopefully, you will see an improvement in accuracy relative to your previous logistic regression implementation.

After this assignment you will be able to: - Build and apply a deep neural network to supervised learning.

Let's get started!

1.1 1 - Packages

Let's first import all the packages that you will need during this assignment. - numpy is the fundamental package for scientific computing with Python. - matplotlib is a library to plot graphs in Python. - h5py is a common package to interact with a dataset that is stored on an H5 file. - PIL and scipy are used here to test your model with your own picture at the end. - dnn_app_utils provides the functions implemented in the "Building your Deep Neural Network: Step by Step" assignment to this notebook. - np.random.seed(1) is used to keep all the random function calls consistent. It will help us grade your work.

```
In [20]: import time
         import numpy as np
         import h5py
         import matplotlib.pyplot as plt
         import scipy
         import gc
         from PIL import Image
         from scipy import ndimage
         from dnn_app_utils_v3 import *

         %matplotlib inline
         plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
         plt.rcParams['image.interpolation'] = 'nearest'
         plt.rcParams['image.cmap'] = 'gray'

         %load_ext autoreload
         %autoreload 2

         np.random.seed(1)
```

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

1.2 2 - Dataset

You will use the same "Cat vs non-Cat" dataset as in "Logistic Regression as a Neural Network" (Assignment 2). The model you had built had 70% test accuracy on classifying cats vs non-cats images. Hopefully, your new model will perform a better!

Problem Statement: You are given a dataset ("data.h5") containing: - a training set of m_train images labelled as cat (1) or non-cat (0) - a test set of m_test images labelled as cat and non-cat - each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).

Let's get more familiar with the dataset. Load the data by running the cell below.

```
In [21]: train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
         print("train_x_orig: ", train_x_orig)
         print("train_y: ", train_y)
         print("test_x_orig: ", test_x_orig)
         print("test_y: ", test_y)
         print("classes: ", classes)
```

```
train_x_orig:  [[[[ 47  47  47]
   [ 64  64  64]
   [ 65  65  65]
   ...
   [  6   6   6]
   [  1   1   1]
   [  5   5   5]]

  [[ 47  47  47]
   [ 63  63  63]
   [ 65  65  65]
   ...
   [  3   3   3]
   [  0   0   0]
   [  3   3   3]]

  [[ 47  47  47]
   [ 64  64  64]
   [ 66  66  66]
   ...
   [  0   0   0]
   [  0   0   0]
   [  2   2   2]]

  ...

  [[  0   0   0]
```

```
  [  6   6   6]
  [  0   0   0]
  ...
  [  0   0   0]
  [  3   3   3]
  [  0   0   0]]

 [[  0   0   0]
  [  6   6   6]
  [  0   0   0]
  ...
  [  0   0   0]
  [  0   0   0]
  [  0   0   0]]

 [[  0   0   0]
  [  6   6   6]
  [  0   0   0]
  ...
  [  6   6   6]
  [  2   2   2]
  [  0   0   0]]]


[[[ 52  52  52]
  [ 36  36  36]
  [ 55  55  55]
  ...
  [111 111 111]
  [ 92  92  92]
  [201 201 201]]

 [[ 48  48  48]
  [ 52  52  52]
  [ 53  53  53]
  ...
  [ 90  90  90]
  [ 64  64  64]
  [166 166 166]]

 [[ 49  49  49]
  [ 53  53  53]
  [ 33  33  33]
  ...
  [ 67  67  67]
  [ 50  50  50]
  [153 153 153]]
```

```
  ...

  [[  6   6   6]
   [ 35  35  35]
   [ 54  54  54]
   ...
   [  5   5   5]
   [  2   2   2]
   [  0   0   0]]

  [[  6   6   6]
   [ 48  48  48]
   [ 63  63  63]
   ...
   [  3   3   3]
   [  1   1   1]
   [  0   0   0]]

  [[  4   4   4]
   [ 59  59  59]
   [ 74  74  74]
   ...
   [  0   0   0]
   [  1   1   1]
   [  2   2   2]]]


 [[[  0   0   0]
   [  3   3   3]
   [  5   5   5]
   ...
   [  8   8   8]
   [  0   0   0]
   [  3   3   3]]

  [[  0   0   0]
   [  0   0   0]
   [  0   0   0]
   ...
   [  1   1   1]
   [  0   0   0]
   [  5   5   5]]

  [[  4   4   4]
   [  4   4   4]
   [  0   0   0]
   ...
   [  8   8   8]
```

```
  [  0   0   0]
  [  5   5   5]]

 ...

 [[  0   0   0]
  [  0   0   0]
  [  2   2   2]
  ...
  [  4   4   4]
  [  0   0   0]
  [  0   0   0]]

 [[  0   0   0]
  [  0   0   0]
  [  0   0   0]
  ...
  [  0   0   0]
  [  0   0   0]
  [  0   0   0]]

 [[  3   3   3]
  [  2   2   2]
  [  1   1   1]
  ...
  [  0   0   0]
  [  0   0   0]
  [  2   2   2]]]


...


[[[ 43  43  43]
  [ 51  51  51]
  [ 62  62  62]
  ...
  [ 51  51  51]
  [ 41  41  41]
  [ 27  27  27]]

 [[ 51  51  51]
  [ 54  54  54]
  [ 64  64  64]
  ...
  [ 52  52  52]
  [ 31  31  31]
  [  9   9   9]]
```

```
 [[ 56  56  56]
  [ 55  55  55]
  [ 63  63  63]
  ...
  [ 42  42  42]
  [ 21  21  21]
  [  7   7   7]]

 ...

 [[ 39  39  39]
  [ 64  64  64]
  [140 140 140]
  ...
  [ 43  43  43]
  [ 43  43  43]
  [ 43  43  43]]

 [[ 41  41  41]
  [ 73  73  73]
  [150 150 150]
  ...
  [ 43  43  43]
  [ 43  43  43]
  [ 43  43  43]]

 [[ 42  42  42]
  [ 80  80  80]
  [156 156 156]
  ...
  [ 43  43  43]
  [ 43  43  43]
  [ 43  43  43]]]


[[[ 54  54  54]
  [ 32  32  32]
  [ 34  34  34]
  ...
  [ 25  25  25]
  [ 17  17  17]
  [ 31  31  31]]

 [[ 43  43  43]
  [ 46  46  46]
  [ 65  65  65]
  ...
```

```
 [ 23  23  23]
 [ 19  19  19]
 [ 34  34  34]]

[[ 46  46  46]
 [ 45  45  45]
 [ 53  53  53]
 ...
 [ 15  15  15]
 [ 15  15  15]
 [ 28  28  28]]

...

[[ 20  20  20]
 [ 12  12  12]
 [ 29  29  29]
 ...
 [ 20  20  20]
 [ 21  21  21]
 [ 21  21  21]]

[[ 22  22  22]
 [  9   9   9]
 [ 30  30  30]
 ...
 [ 31  31  31]
 [ 29  29  29]
 [ 28  28  28]]

[[ 23  23  23]
 [  8   8   8]
 [ 31  31  31]
 ...
 [ 43  43  43]
 [ 39  39  39]
 [ 36  36  36]]]


[[[ 24  24  24]
  [  0   0   0]
  [ 24  24  24]
  ...
  [123 123 123]
  [124 124 124]
  [125 125 125]]

 [[ 13  13  13]
```

```
   [ 13  13  13]
   [ 77  77  77]
   ...
   [131 131 131]
   [127 127 127]
   [125 125 125]]

  [[ 11  11  11]
   [ 31  31  31]
   [127 127 127]
   ...
   [139 139 139]
   [132 132 132]
   [127 127 127]]

  ...

  [[ 23  23  23]
   [ 22  22  22]
   [ 14  14  14]
   ...
   [ 26  26  26]
   [ 41  41  41]
   [ 46  46  46]]

  [[ 22  22  22]
   [ 21  21  21]
   [ 15  15  15]
   ...
   [ 40  40  40]
   [ 34  34  34]
   [ 25  25  25]]

  [[ 21  21  21]
   [ 20  20  20]
   [ 15  15  15]
   ...
   [ 45  45  45]
   [ 34  34  34]
   [ 55  55  55]]]]
train_y:  [[0 1 1 ... 1 1 1]]
test_x_orig:  [[[[  0   0   0]
   [  0   0   0]
   [  1   1   1]
   ...
   [ 65  65  65]
   [ 70  70  70]
   [ 54  54  54]]
```

```
 [[  0   0   0]
  [  0   0   0]
  [  1   1   1]
  ...
  [ 66  66  66]
  [ 70  70  70]
  [ 52  52  52]]

 [[  0   0   0]
  [  0   0   0]
  [  1   1   1]
  ...
  [ 68  68  68]
  [ 69  69  69]
  [ 50  50  50]]

 ...

 [[  0   0   0]
  [  0   0   0]
  [  0   0   0]
  ...
  [  2   2   2]
  [  0   0   0]
  [  3   3   3]]

 [[  0   0   0]
  [  0   0   0]
  [  0   0   0]
  ...
  [  1   1   1]
  [  0   0   0]
  [  3   3   3]]

 [[  0   0   0]
  [  0   0   0]
  [  0   0   0]
  ...
  [  1   1   1]
  [  0   0   0]
  [  3   3   3]]]


[[[ 26  26  26]
  [ 50  50  50]
  [ 63  63  63]
  ...
```

```
  [125 125 125]
  [111 111 111]
  [109 109 109]]

 [[ 19  19  19]
  [ 47  47  47]
  [ 61  61  61]
  ...
  [113 113 113]
  [ 75  75  75]
  [102 102 102]]

 [[ 16  16  16]
  [ 50  50  50]
  [ 65  65  65]
  ...
  [ 93  93  93]
  [ 44  44  44]
  [104 104 104]]

 ...

 [[  2   2   2]
  [  0   0   0]
  [ 14  14  14]
  ...
  [111 111 111]
  [ 26  26  26]
  [  3   3   3]]

 [[  0   0   0]
  [  2   2   2]
  [ 11  11  11]
  ...
  [104 104 104]
  [ 27  27  27]
  [  0   0   0]]

 [[  0   0   0]
  [  8   8   8]
  [ 10  10  10]
  ...
  [103 103 103]
  [ 33  33  33]
  [  3   3   3]]]


[[[  0   0   0]
```

```
 [  0   0   0]
 [  0   0   0]
 ...
 [  0   0   0]
 [  0   0   0]
 [  0   0   0]]

[[  0   0   0]
 [  0   0   0]
 [  0   0   0]
 ...
 [  0   0   0]
 [  0   0   0]
 [  0   0   0]]

[[  0   0   0]
 [  0   0   0]
 [  0   0   0]
 ...
 [  0   0   0]
 [  0   0   0]
 [  0   0   0]]

...

[[  0   0   0]
 [  0   0   0]
 [  1   1   1]
 ...
 [  2   2   2]
 [  1   1   1]
 [  0   0   0]]

[[  1   1   1]
 [  0   0   0]
 [  0   0   0]
 ...
 [  2   2   2]
 [  1   1   1]
 [  0   0   0]]

[[  2   2   2]
 [  0   0   0]
 [  0   0   0]
 ...
 [  2   2   2]
 [  1   1   1]
 [  0   0   0]]]
```

```
...

[[[  7   7   7]
  [ 21  21  21]
  [ 15  15  15]
  ...
  [  9   9   9]
  [ 11  11  11]
  [ 21  21  21]]

 [[ 12  12  12]
  [  3   3   3]
  [  3   3   3]
  ...
  [ 14  14  14]
  [ 15  15  15]
  [ 19  19  19]]

 [[ 12  12  12]
  [  2   2   2]
  [ 11  11  11]
  ...
  [ 17  17  17]
  [ 18  18  18]
  [ 16  16  16]]

 ...

 [[ 71  71  71]
  [ 54  54  54]
  [ 17  17  17]
  ...
  [ 22  22  22]
  [ 23  23  23]
  [ 24  24  24]]

 [[ 70  70  70]
  [ 61  61  61]
  [ 20  20  20]
  ...
  [ 22  22  22]
  [ 23  23  23]
  [ 25  25  25]]

 [[ 64  64  64]
```

```
  [ 61  61  61]
  [ 22  22  22]
  ...
  [ 22  22  22]
  [ 23  23  23]
  [ 25  25  25]]]


[[[ 22  22  22]
  [ 38  38  38]
  [ 45  45  45]
  ...
  [ 43  43  43]
  [ 26  26  26]
  [ 23  23  23]]

 [[ 21  21  21]
  [ 36  36  36]
  [ 46  46  46]
  ...
  [ 45  45  45]
  [ 31  31  31]
  [ 24  24  24]]

 [[ 20  20  20]
  [ 35  35  35]
  [ 46  46  46]
  ...
  [ 44  44  44]
  [ 34  34  34]
  [ 22  22  22]]

 ...

 [[  0   0   0]
  [  0   0   0]
  [  0   0   0]
  ...
  [  0   0   0]
  [  0   0   0]
  [  0   0   0]]

 [[  0   0   0]
  [  0   0   0]
  [  0   0   0]
  ...
  [  0   0   0]
  [  3   3   3]
```

```
  [  0   0   0]]

 [[  1   1   1]
  [  1   1   1]
  [  0   0   0]
  ...
  [  0   0   0]
  [  8   8   8]
  [  0   0   0]]]


[[[  6   6   6]
  [  2   2   2]
  [  3   3   3]
  ...
  [ 45  45  45]
  [ 19  19  19]
  [  9   9   9]]

 [[  2   2   2]
  [  0   0   0]
  [  2   2   2]
  ...
  [ 43  43  43]
  [ 16  16  16]
  [  6   6   6]]

 [[  0   0   0]
  [  0   0   0]
  [  1   1   1]
  ...
  [ 40  40  40]
  [ 13  13  13]
  [  4   4   4]]

 ...

 [[  2   2   2]
  [  0   0   0]
  [  2   2   2]
  ...
  [  0   0   0]
  [  0   0   0]
  [  2   2   2]]

 [[  1   1   1]
  [  0   0   0]
  [  8   8   8]
```

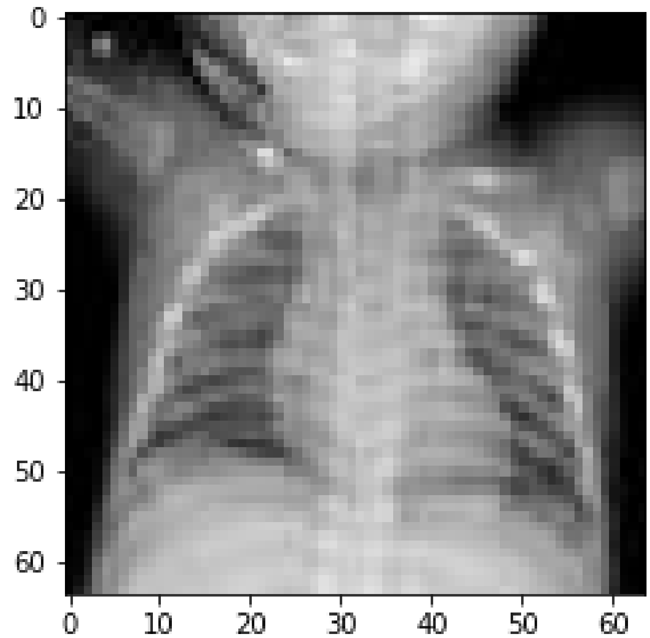
```
  ...
  [  0   0   0]
  [  0   0   0]
  [  3   3   3]]

 [[  0   0   0]
  [  0   0   0]
  [ 12  12  12]
  ...
  [  0   0   0]
  [  0   0   0]
  [  4   4   4]]]]
test_y:  [[0 0 0 1 0 1 0 0 1 1 1 0 1 1 1 1 1 1 0 1 1 1 0 0 0 1 1 1 0 1 1 1 0 1 1 1
  1 1 0 0 0 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 0 1 0 1 0 1 1 1 1
  0 1 1 1 1 1 1 0 0 1 1 1 1 0 0 1 0 0 1 1 0 1 1 1 1 1 1 0 0 0 1 1 1 0 0 1
  0 1 0 1 1 1 1 1 0 1 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1
  1 1 1 1 1 1 1 1 0 1 1 1 0 0 0 0 0 1 1 0 1 1 1 0 1 1 1 1 0 0 0 1 1 1 0 0
  1 1 1 0 0 0 1 1 0 1 1 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 1 1 1 1 0 1 0 1 1 1 1 1 1 1 1 1 1 0 1 1 1 0 1 1 1 0 1 1 1 1 1
  0 1 0 1 1 1 0 1 1 1 0 0 1 1 0 0 1 0 0 1 0 1 1 1 1 1 1 0 1 1 1 1 1 1 0 1
  0 1 0 1 1 0 0 1 1 1 1 1 0 1 0 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
  1 1 1 1 1 0 0 0 1 1 1 0 1 1 0 0 0 1 1 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 1 1
  1 1 1 0 0 0 1 1 1 1 1 1 0 1 0 1 1 0 1 1 0 0 1 1 0 1 1 0 0 0 0 1 1 1 1 1
  0 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 1 1 0 1 1 0 1 1 1 0 1 1 0 1 1 1 1
  1 0 1 0 1 0 1 1 0 1 1 1 1 0 1 1 1 1 0 1 1 1 1 1 1 1 1 1 1 0 0 1 1 1 0 0
  1 1 0 0 1 1 1 1 1 0 1 0 1 1 0 1 1 1 1 1 1 1 0 1 1 1 1 1 0 1 1 1 0 0 0 1
  1 1 1 0 0 1 1 1 0 1 1 0 0 1 0 1 0 1 1 1 1 1 1 0 1 1 1 1 0 1 1 1 0 1 1 1
  1 0 1 0 0 1 1 0 0 1 1 1 0 1 1 1 0 0 1 1 0 0 1 1 0 1 1 0 1 1 1 1 1 0 1 1
  1 0 1 1 1 0 1 1 0 1]]
classes:  ['NORMAL' 'PNEUMONIA']
```

The following code will show you an image in the dataset. Feel free to change the index and re-run the cell multiple times to see other images.

```
In [22]: # Example of a picture
         index = 10
         plt.imshow(train_x_orig[index])
         print ("y = " + str(train_y[0,index]) + ". It's a " + classes[train_y[0,index]] +  " p
```

```
y = 0. It's a NORMAL picture.
```




```
In [23]: # Explore your dataset
         m_train = train_x_orig.shape[0]
         num_px = train_x_orig.shape[1]
         m_test = test_x_orig.shape[0]

         print ("Number of training examples: " + str(m_train))
         print ("Number of testing examples: " + str(m_test))
         print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
         print ("train_x_orig shape: " + str(train_x_orig.shape))
         print ("train_y shape: " + str(train_y.shape))
         print ("test_x_orig shape: " + str(test_x_orig.shape))
         print ("test_y shape: " + str(test_y.shape))
```

```
Number of training examples: 5270
Number of testing examples: 586
Each image is of size: (64, 64, 3)
train_x_orig shape: (5270, 64, 64, 3)
train_y shape: (1, 5270)
test_x_orig shape: (586, 64, 64, 3)
test_y shape: (1, 586)
```

As usual, you reshape and standardize the images before feeding them to the network. The code is given in the cell below.

Figure 1: Image to vector conversion.

```
In [24]: # Reshape the training and test examples
         train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T   # The "-1" make
         test_x_flatten = test_x_orig.reshape(test_x_orig.shape[0], -1).T

         # Standardize data to have feature values between 0 and 1.
         train_x = train_x_flatten/255.
         test_x = test_x_flatten/255.

         print ("train_x's shape: " + str(train_x.shape))
         print ("test_x's shape: " + str(test_x.shape))
```

```
train_x's shape: (12288, 5270)
test_x's shape: (12288, 586)
```

$12,288$ equals $64 \times 64 \times 3$ which is the size of one reshaped image vector.

1.3 3 - Architecture of your model

Now that you are familiar with the dataset, it is time to build a deep neural network to distinguish cat images from non-cat images.

You will build two different models: - A 2-layer neural network - An L-layer deep neural network

You will then compare the performance of these models, and also try out different values for L.

Let's look at the two architectures.

1.3.1 3.1 - 2-layer neural network

Figure 2: 2-layer neural network. The model can be summarized as: ***INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT***.

Detailed Architecture of figure 2: - The input is a (64,64,3) image which is flattened to a vector of size $(12288, 1)$. - The corresponding vector: $[x_0, x_1, ..., x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ of size $(n^{[1]}, 12288)$. - You then add a bias term and take its relu to get the following vector: $[a_0^{[1]}, a_1^{[1]}, ..., a_{n^{[1]}-1}^{[1]}]^T$. - You then repeat the same process. - You multiply the resulting vector by $W^{[2]}$ and add your intercept (bias). - Finally, you take the sigmoid of the result. If it is greater than 0.5, you classify it to be a cat.

1.3.2 3.2 - L-layer deep neural network

It is hard to represent an L-layer deep neural network with the above representation. However, here is a simplified network representation:

Figure 3: L-layer neural network. The model can be summarized as: ***[LINEAR -> RELU]*** $\times$ ***(L-1) -> LINEAR -> SIGMOID***

Detailed Architecture of figure 3: - The input is a (64,64,3) image which is flattened to a vector of size (12288,1). - The corresponding vector: $[x_0, x_1, ..., x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ and then you add the intercept $b^{[1]}$. The result is called the linear unit. - Next, you take the relu of the linear unit. This process could be repeated several times for each $(W^{[l]}, b^{[l]})$

depending on the model architecture. - Finally, you take the sigmoid of the final linear unit. If it is greater than 0.5, you classify it to be a cat.

1.3.3 3.3 - General methodology

As usual you will follow the Deep Learning methodology to build the model: 1. Initialize parameters / Define hyperparameters 2. Loop for num_iterations: a. Forward propagation b. Compute cost function c. Backward propagation d. Update parameters (using parameters, and grads from backprop) 4. Use trained parameters to predict labels

Let's now implement those two models!

1.4 4 - Two-layer neural network

Question: Use the helper functions you have implemented in the previous assignment to build a 2-layer neural network with the following structure: *LINEAR -> RELU -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters(n_x, n_h, n_y):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_cost(AL, Y):
    ...
    return cost
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

In [25]:
```
### CONSTANTS DEFINING THE MODEL ####
n_x = 64*64*3     # num_px * num_px * 3
n_h = 3
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

In [7]:
```
# GRADED FUNCTION: two_layer_model

def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, p
    """
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number o
    layers_dims -- dimensions of the layers (n_x, n_h, n_y)
```

```
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- If set to True, this will print the cost every 100 iterations

    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    """

    np.random.seed(1)
    grads = {}
    costs = []                              # to keep track of the cost
    m = X.shape[1]                           # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions you'd previous
    ### START CODE HERE ### ( 1 line of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    ### END CODE HERE ###

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)

    for i in range(0, num_iterations):
        gc.collect()
        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1
        ### START CODE HERE ### ( 2 lines of code)
        A1, cache1 = linear_activation_forward(X, W1, b1, 'relu')
        A2, cache2 = linear_activation_forward(A1, W2, b2, 'sigmoid')
        ### END CODE HERE ###

        # Compute cost
        ### START CODE HERE ### ( 1 line of code)
        cost = compute_cost(A2, Y)
        ### END CODE HERE ###

        # Initializing backward propagation
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2
        ### START CODE HERE ### ( 2 lines of code)
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, 'sigmoid')
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, 'relu')
        ### END CODE HERE ###
```

```
        # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        # Update parameters.
        ### START CODE HERE ### (approx. 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate)
        ### END CODE HERE ###

        # Retrieve W1, b1, W2, b2 from parameters
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

        # Print the cost every 100 training example
        if print_cost and i % 100 == 0:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if print_cost and i % 100 == 0:
            costs.append(cost)

    # plot the cost

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

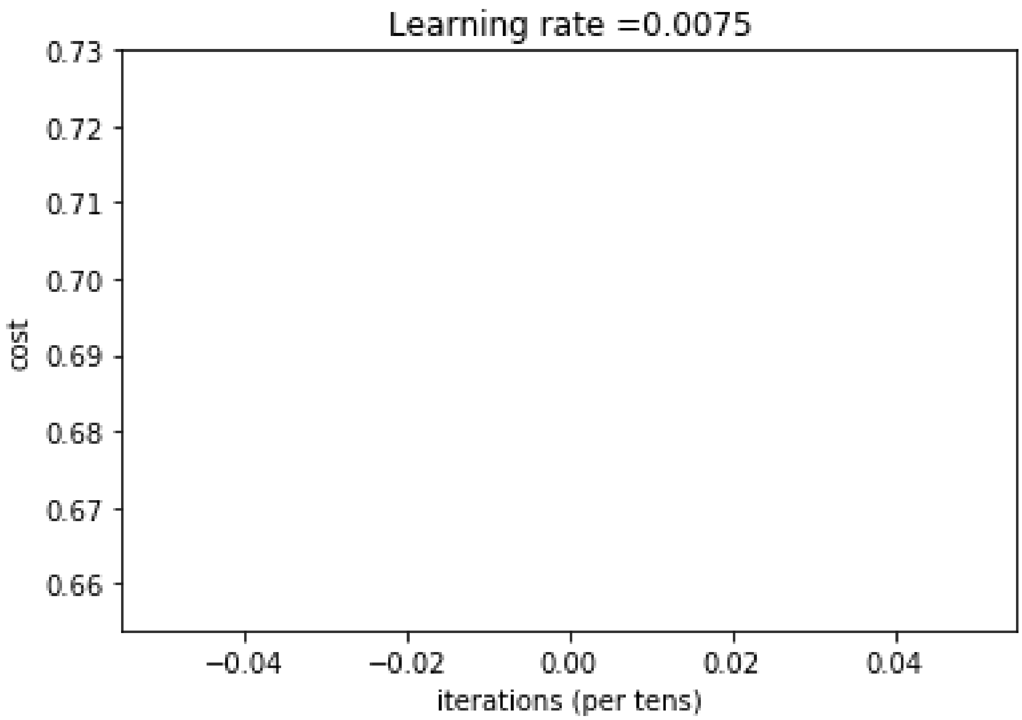
Run the cell below to train your parameters. See if your model runs. The cost should be decreasing. It may take up to 5 minutes to run 2500 iterations. Check if the "Cost after iteration 0" matches the expected output below, if not click on the square () on the upper bar of the notebook to stop the cell and try to find your error.

```
In [8]: gc.collect()

Out[8]: 2345

In [9]: parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y), num_itera

Cost after iteration 0: 0.6919219617594874
```



Expected Output:

```
<tr>
    <td> **Cost after iteration 0**</td>
    <td> 0.6930497356599888 </td>
</tr>
<tr>
    <td> **Cost after iteration 100**</td>
    <td> 0.6464320953428849 </td>
</tr>
<tr>
    <td> **...**</td>
    <td> ... </td>
</tr>
<tr>
    <td> **Cost after iteration 2400**</td>
    <td> 0.048554785628770206 </td>
</tr>
```

Good thing you built a vectorized implementation! Otherwise it might have taken 10 times longer to train this.

Now, you can use the trained parameters to classify images from the dataset. To see your predictions on the training and test sets, run the cell below.

```
In [10]: predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.7296015180265655
```

Expected Output:

```
<tr>
    <td> **Accuracy**</td>
    <td> 1.0 </td>
</tr>
```

```
In [11]: predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.7303754266211603
```

Expected Output:

```
<tr>
    <td> **Accuracy**</td>
    <td> 0.72 </td>
</tr>
```

Note: You may notice that running the model on fewer iterations (say 1500) gives better accuracy on the test set. This is called "early stopping" and we will talk about it in the next course. Early stopping is a way to prevent overfitting.

Congratulations! It seems that your 2-layer neural network has better performance (72%) than the logistic regression implementation (70%, assignment week 2). Let's see if you can do even better with an L-layer model.

1.5 5 - L-layer Neural Network

Question: Use the helper functions you have implemented previously to build an L-layer neural network with the following structure: *[LINEAR -> RELU]×(L-1) -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters_deep(layers_dims):
    ...
    return parameters
def L_model_forward(X, parameters):
    ...
    return AL, caches
def compute_cost(AL, Y):
    ...
    return cost
def L_model_backward(AL, Y, caches):
    ...
    return grads
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

In [26]:
```
### CONSTANTS ###
layers_dims = [12288, 20, 7, 5, 1] #  4-layer model
```

In [27]:
```
# GRADED FUNCTION: L_layer_model

def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, pr
    """
    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.

    Arguments:
    X -- data, numpy array of shape (number of examples, num_px * num_px * 3)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number
    layers_dims -- list containing the input size and each layer size, of length (num
    learning_rate -- learning rate of the gradient descent update rule
    num_iterations -- number of iterations of the optimization loop
    print_cost -- if True, it prints the cost every 100 steps

    Returns:
    parameters -- parameters learnt by the model. They can then be used to predict.
    """

    np.random.seed(1)
    costs = []                         # keep track of cost

    # Parameters initialization. ( 1 line of code)
    ### START CODE HERE ###
    parameters = initialize_parameters_deep(layers_dims)
    ### END CODE HERE ###

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
        ### START CODE HERE ### ( 1 line of code)
        AL, caches = L_model_forward(X, parameters)
        ### END CODE HERE ###

        # Compute cost.
        ### START CODE HERE ### ( 1 line of code)
        cost = compute_cost(AL, Y)
        ### END CODE HERE ###

        # Backward propagation.
        ### START CODE HERE ### ( 1 line of code)
        grads = L_model_backward(AL, Y, caches)
        ### END CODE HERE ###

        # Update parameters.
```

```
        ### START CODE HERE ### ( 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate)
        ### END CODE HERE ###

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

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters
```

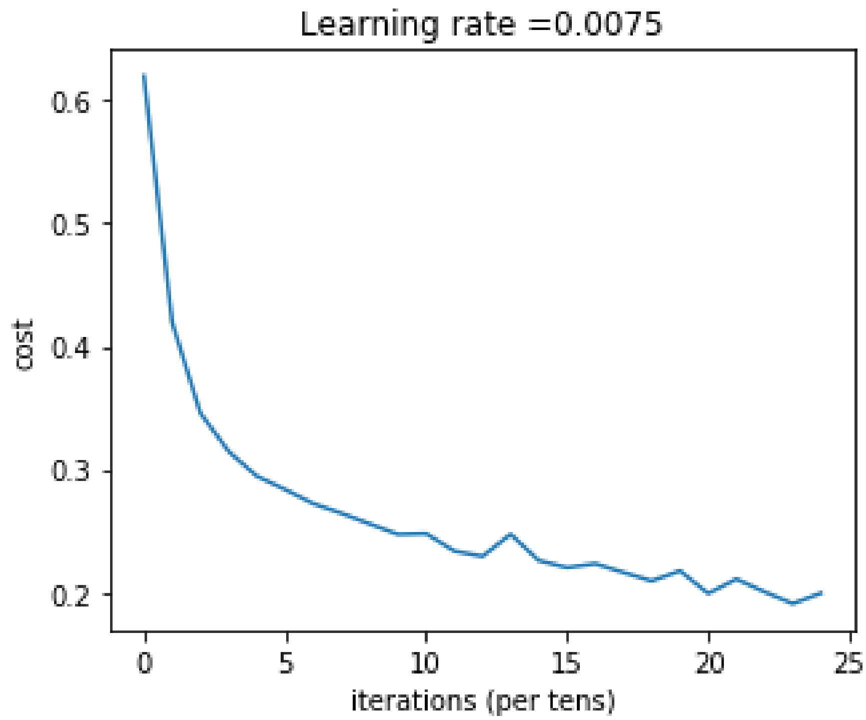
You will now train the model as a 4-layer neural network.

Run the cell below to train your model. The cost should decrease on every iteration. It may take up to 5 minutes to run 2500 iterations. Check if the "Cost after iteration 0" matches the expected output below, if not click on the square () on the upper bar of the notebook to stop the cell and try to find your error.

```
In [28]: parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations = 2500, prin
```

```
Cost after iteration 0: 0.619673
Cost after iteration 100: 0.419599
Cost after iteration 200: 0.345866
Cost after iteration 300: 0.314516
Cost after iteration 400: 0.294571
Cost after iteration 500: 0.283750
Cost after iteration 600: 0.272203
Cost after iteration 700: 0.264344
Cost after iteration 800: 0.255743
Cost after iteration 900: 0.247260
Cost after iteration 1000: 0.247886
Cost after iteration 1100: 0.233645
Cost after iteration 1200: 0.229316
Cost after iteration 1300: 0.247338
Cost after iteration 1400: 0.225882
Cost after iteration 1500: 0.220186
Cost after iteration 1600: 0.223319
Cost after iteration 1700: 0.216250
Cost after iteration 1800: 0.209341
Cost after iteration 1900: 0.217567
Cost after iteration 2000: 0.199108
```

```
Cost after iteration 2100: 0.210870
Cost after iteration 2200: 0.200406
Cost after iteration 2300: 0.190871
Cost after iteration 2400: 0.199382
```



Expected Output:

```
<tr>
    <td> **Cost after iteration 0**</td>
    <td> 0.771749 </td>
</tr>
<tr>
    <td> **Cost after iteration 100**</td>
    <td> 0.672053 </td>
</tr>
<tr>

    <td> **...**</td>
    <td> ... </td>
</tr>
<tr>
    <td> **Cost after iteration 2400**</td>
    <td> 0.092878 </td>
</tr>
```

```
In [29]: pred_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.9259962049335864
```

```
<tr>
<td>
**Train Accuracy**
</td>
<td>
0.985645933014
</td>
</tr>
```

```
In [30]: pred_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.9266211604095561
```

Expected Output:

```
<tr>
    <td> **Test Accuracy**</td>
    <td> 0.8 </td>
</tr>
```

Congrats! It seems that your 4-layer neural network has better performance (80%) than your 2-layer neural network (72%) on the same test set.

This is good performance for this task. Nice job!

Though in the next course on "Improving deep neural networks" you will learn how to obtain even higher accuracy by systematically searching for better hyperparameters (learning_rate, layers_dims, num_iterations, and others you'll also learn in the next course).

1.6 6) Results Analysis

First, let's take a look at some images the L-layer model labeled incorrectly. This will show a few mislabeled images.

```
In [19]: print_mislabeled_images(classes, test_x, test_y, pred_test)
```


A few types of images the model tends to do poorly on include: - Cat body in an unusual position - Cat appears against a background of a similar color - Unusual cat color and species - Camera Angle - Brightness of the picture - Scale variation (cat is very large or small in image)

References:

- for auto-reloading external module: http://stackoverflow.com/questions/1907993/autoreload-of-modules-in-ipython