**Deep Neural Network for Image Classification: Application**[**¶**](https://ukoudwsv.labs.coursera.org/notebooks/Week%204/Deep%20Neural%20Network%20Application:%20Image%20Classification/Deep%20Neural%20Network%20-%20Application%20v8.ipynb#Deep-Neural-Network-for-Image-Classification:-Application)

When you finish this, you will have finished the last programming assignment of Week 4, and also the last programming assignment of this course!

You will 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 - Packages**

Let's first import all the packages that you will need during this assignment.

* [numpy](https://www.numpy.org/) is the fundamental package for scientific computing with Python.
* [matplotlib](http://matplotlib.org/) is a library to plot graphs in Python.
* [h5py](http://www.h5py.org/) is a common package to interact with a dataset that is stored on an H5 file.
* [PIL](http://www.pythonware.com/products/pil/) and [scipy](https://www.scipy.org/) 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 [1]:

**import** time

**import** numpy **as** np

**import** h5py

**import** matplotlib.pyplot **as** plt

**import** scipy

**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)

**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 [2]:

train\_x\_orig, train\_y, test\_x\_orig, test\_y, classes = load\_data()

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 [9]:

*# Example of a picture*

**for** index **in** range(20):

plt.imshow(train\_x\_orig[index])

print ("y = " **+** str(train\_y[0,index]) **+** ". It's a " **+** classes[train\_y[0,index]].decode("utf-8") **+** " picture.")

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 1. It's a cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 1. It's a cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 1. It's a cat picture.

y = 0. It's a non-cat picture.

y = 1. It's a cat picture.

y = 1. It's a cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 0. It's a non-cat picture.

y = 1. It's a cat picture.

A cat that is looking at the camera

Description automatically generated

In [10]:

*# 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: 209

Number of testing examples: 50

Each image is of size: (64, 64, 3)

train\_x\_orig shape: (209, 64, 64, 3)

train\_y shape: (1, 209)

test\_x\_orig shape: (50, 64, 64, 3)

test\_y shape: (1, 50)

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 [11]:

*# Reshape the training and test examples*

train\_x\_flatten = train\_x\_orig.reshape(train\_x\_orig.shape[0], **-**1).T *# The "-1" makes reshape flatten the remaining dimensions*

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, 209)

test\_x's shape: (12288, 50)

12,28812,288 equals 64×64×364×64×3 which is the size of one reshaped image vector.

**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 LL.

Let's look at the two architectures.

**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)(12288,1).
* The corresponding vector: [x0,x1,...,x12287]T[x0,x1,...,x12287]T is then multiplied by the weight matrix W[1]W[1] of size (n[1],12288)(n[1],12288).
* You then add a bias term and take its relu to get the following vector: [a[1]0,a[1]1,...,a[1]n[1]−1]T[a0[1],a1[1],...,an[1]−1[1]]T.
* You then repeat the same process.
* You multiply the resulting vector by W[2]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.

**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]***××***(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: [x0,x1,...,x12287]T[x0,x1,...,x12287]T is then multiplied by the weight matrix W[1]W[1] and then you add the intercept b[1]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])(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.

**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!

**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 [12]:

*### CONSTANTS DEFINING THE MODEL ####*

n\_x = 12288 *# num\_px \* num\_px \* 3*

n\_h = 7

n\_y = 1

layers\_dims = (n\_x, n\_h, n\_y)

In [13]:

*# GRADED FUNCTION: two\_layer\_model*

​

**def** two\_layer\_model(X, Y, layers\_dims, learning\_rate = 0.0075, num\_iterations = 3000, print\_cost=**False**):

"""

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 1 if cat, 0 if non-cat), of shape (1, number of examples)

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 previously implemented*

*### 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):

​

*# Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2".*

*### 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; also dA0 (not used), dW1, db1".*

*### 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['db2'] to db2*

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 hundreds)')

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 [14]:

parameters = two\_layer\_model(train\_x, train\_y, layers\_dims = (n\_x, n\_h, n\_y), num\_iterations = 2500, print\_cost=**True**)

Cost after iteration 0: 0.6930497356599888

Cost after iteration 100: 0.6464320953428849

Cost after iteration 200: 0.6325140647912677

Cost after iteration 300: 0.6015024920354665

Cost after iteration 400: 0.5601966311605747

Cost after iteration 500: 0.5158304772764729

Cost after iteration 600: 0.47549013139433255

Cost after iteration 700: 0.43391631512257495

Cost after iteration 800: 0.400797753620389

Cost after iteration 900: 0.3580705011323798

Cost after iteration 1000: 0.3394281538366411

Cost after iteration 1100: 0.3052753636196264

Cost after iteration 1200: 0.2749137728213018

Cost after iteration 1300: 0.24681768210614854

Cost after iteration 1400: 0.19850735037466094

Cost after iteration 1500: 0.17448318112556666

Cost after iteration 1600: 0.17080762978096128

Cost after iteration 1700: 0.11306524562164724

Cost after iteration 1800: 0.09629426845937152

Cost after iteration 1900: 0.08342617959726856

Cost after iteration 2000: 0.07439078704319078

Cost after iteration 2100: 0.06630748132267927

Cost after iteration 2200: 0.05919329501038164

Cost after iteration 2300: 0.05336140348560553

Cost after iteration 2400: 0.048554785628770115

Chart, line chart

Description automatically generated

**Expected Output**:

|  |  |
| --- | --- |
| **Cost after iteration 0** | 0.6930497356599888 |
| **Cost after iteration 100** | 0.6464320953428849 |
| **...** | ... |
| **Cost after iteration 2400** | 0.048554785628770226 |

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 [15]:

predictions\_train = predict(train\_x, train\_y, parameters)

Accuracy: 1.0

**Expected Output**:

|  |  |
| --- | --- |
| **Accuracy** | 1.0 |

In [16]:

predictions\_test = predict(test\_x, test\_y, parameters)

Accuracy: 0.72

**Expected Output**:

|  |  |
| --- | --- |
| **Accuracy** | 0.72 |

**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 LL-layer model.

**5 - L-layer Neural Network**

**Question**: Use the helper functions you have implemented previously to build an LL-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 [17]:

*### CONSTANTS ###*

layers\_dims = [12288, 20, 7, 5, 1] *# 4-layer model*

In [19]:

*# GRADED FUNCTION: L\_layer\_model*

​

**def** L\_layer\_model(X, Y, layers\_dims, learning\_rate = 0.0075, num\_iterations = 3000, print\_cost=**False**):*#lr was 0.009*

"""

Implements a L-layer neural network: [LINEAR->RELU]\*(L-1)->LINEAR->SIGMOID.

Arguments:

X -- data, numpy array of shape (num\_px \* num\_px \* 3, number of examples)

Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of examples)

layers\_dims -- list containing the input size and each layer size, of length (number of layers + 1).

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 hundreds)')

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 [20]:

parameters = L\_layer\_model(train\_x, train\_y, layers\_dims, num\_iterations = 2500, print\_cost = **True**)

Cost after iteration 0: 0.771749

Cost after iteration 100: 0.672053

Cost after iteration 200: 0.648263

Cost after iteration 300: 0.611507

Cost after iteration 400: 0.567047

Cost after iteration 500: 0.540138

Cost after iteration 600: 0.527930

Cost after iteration 700: 0.465477

Cost after iteration 800: 0.369126

Cost after iteration 900: 0.391747

Cost after iteration 1000: 0.315187

Cost after iteration 1100: 0.272700

Cost after iteration 1200: 0.237419

Cost after iteration 1300: 0.199601

Cost after iteration 1400: 0.189263

Cost after iteration 1500: 0.161189

Cost after iteration 1600: 0.148214

Cost after iteration 1700: 0.137775

Cost after iteration 1800: 0.129740

Cost after iteration 1900: 0.121225

Cost after iteration 2000: 0.113821

Cost after iteration 2100: 0.107839

Cost after iteration 2200: 0.102855

Cost after iteration 2300: 0.100897

Cost after iteration 2400: 0.092878

Chart

Description automatically generated

**Expected Output**:

|  |  |
| --- | --- |
| **Cost after iteration 0** | 0.771749 |
| **Cost after iteration 100** | 0.672053 |
| **...** | ... |
| **Cost after iteration 2400** | 0.092878 |

In [21]:

pred\_train = predict(train\_x, train\_y, parameters)

Accuracy: 0.985645933014

|  |  |
| --- | --- |
| **Train Accuracy** | 0.985645933014 |

In [22]:

pred\_test = predict(test\_x, test\_y, parameters)

Accuracy: 0.8

**Expected Output**:

|  |  |
| --- | --- |
| **Test Accuracy** | 0.8 |

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!

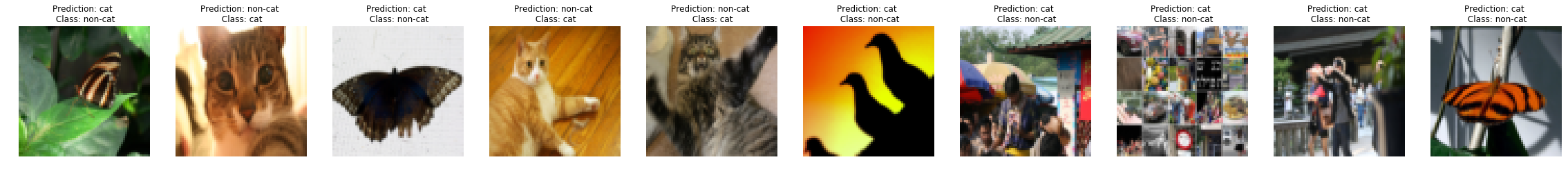
Though in the next course on "Improving deep neural networks" you will learn how to obtain even higher accuracy by systematically searching for better hyperparameters (learning\_rate, layers\_dims, num\_iterations, and others you'll also learn in the next course).

**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 [23]:

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)

**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 [24]:

*## START CODE HERE ##*

my\_image = "covid\_cat.jpg" *# change this to the name of your image file*

my\_label\_y = [1] *# the true class of your image (1 -> cat, 0 -> non-cat)*

*## END CODE HERE ##*

​

fname = "images/" **+** my\_image

image = np.array(ndimage.imread(fname, flatten=**False**))

my\_image = scipy.misc.imresize(image, size=(num\_px,num\_px)).reshape((num\_px**\***num\_px**\***3,1))

my\_image = my\_image**/**255.

my\_predicted\_image = predict(my\_image, my\_label\_y, parameters)

​

plt.imshow(image)

print ("y = " **+** str(np.squeeze(my\_predicted\_image)) **+** ", your L-layer model predicts a \"" **+** classes[int(np.squeeze(my\_predicted\_image)),].decode("utf-8") **+** "\" picture.")

Accuracy: 1.0

y = 1.0, your L-layer model predicts a "cat" picture.

A cat that is looking at the camera

Description automatically generated

**References**:

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