

# Logistic Regression with a Neural Network mindset

Welcome to your first (required) programming assignment! You will build a logistic regression classifier to recognize cats. This assignment will step you through how to do this with a Neural Network mindset, and so will also hone your intuitions about deep learning.

## Instructions:

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

## 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](http://www.numpy.org) ([www.numpy.org](http://www.numpy.org)) is the fundamental package for scientific computing with Python.
- [h5py](http://www.h5py.org) (<http://www.h5py.org>) is a common package to interact with a dataset that is stored on an H5 file.
- [matplotlib](http://matplotlib.org) (<http://matplotlib.org>) is a famous library to plot graphs in Python.
- [PIL](http://www.pythonware.com/products/pil/) (<http://www.pythonware.com/products/pil/>) and [scipy](https://www.scipy.org/) (<https://www.scipy.org/>) are used here to test your model with your own picture at the end.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy 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  $m_{\text{train}}$  images labeled as cat ( $y=1$ ) or non-cat ( $y=0$ )
- a test set of  $m_{\text{test}}$  images labeled as cat or non-cat
- each image is of shape  $(\text{num\_px}, \text{num\_px}, 3)$  where 3 is for the 3 channels (RGB). Thus, each image is square ( $\text{height} = \text{num\_px}$ ) and ( $\text{width} = \text{num\_px}$ ).

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.

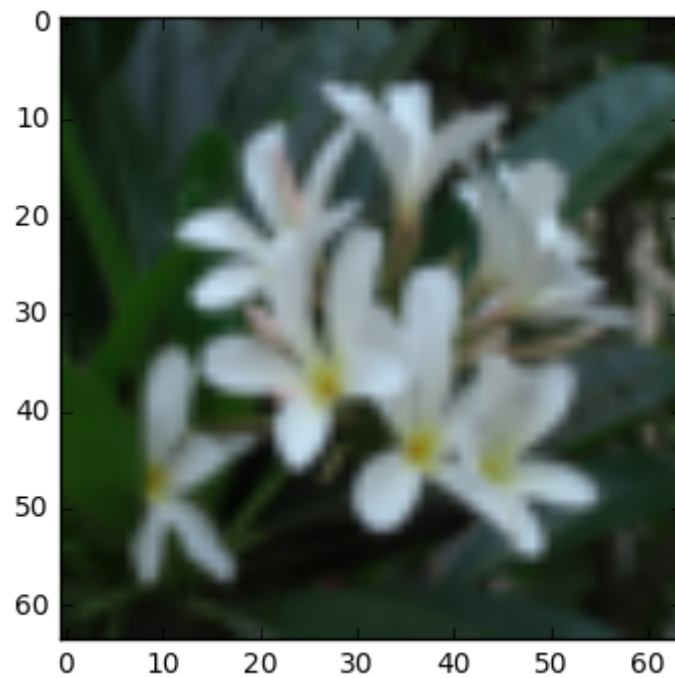
```
In [2]: # Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()
```

We added "\_orig" at the end of image datasets (train and test) because we are going to preprocess them. After preprocessing, we will end up with `train_set_x` and `test_set_x` (the labels `train_set_y` and `test_set_y` don't need any preprocessing).

Each line of your `train_set_x_orig` and `test_set_x_orig` is an array representing an image. You can visualize an example by running the following code. Feel free also to change the index value and re-run to see other images.

```
In [3]: # Example of a picture
index = 28
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 = [0], it's a 'non-cat' picture.



Many software bugs in deep learning come from having matrix/vector dimensions that don't fit. If you can keep your matrix/vector dimensions straight you will go a long way toward eliminating many bugs.

**Exercise:** 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]: test_set_x_orig[0].shape
```

```
Out[4]: (64, 64, 3)
```

```
In [5]: ### START CODE HERE ### (~ 3 lines of code)
m_train = len(train_set_x_orig)
m_test = test_set_x_orig
num_px = 64
### END CODE HERE ###

print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

Number of training examples:  $m_{\text{train}} = 209$

Number of testing examples:  $m_{\text{test}} = [[ [158 \ 104 \ 83]$

$[161 \ 106 \ 85]$

$[162 \ 107 \ 84]$

$\dots,$

$[170 \ 120 \ 100]$

$[167 \ 121 \ 103]$

$[172 \ 127 \ 109]]$

$[ [158 \ 103 \ 82]$

$[160 \ 104 \ 82]$

$[162 \ 105 \ 83]$

$\dots,$

$[169 \ 118 \ 99]$

$[164 \ 117 \ 98]$

$[168 \ 121 \ 104]]$

$[ [158 \ 104 \ 82]$

$[161 \ 105 \ 82]$

$[162 \ 105 \ 83]$

$\dots,$

$[173 \ 123 \ 102]$

$[169 \ 122 \ 104]$

$[168 \ 122 \ 104]]$

$\dots,$

$[ [102 \ 68 \ 52]$

$[ 44 \ 31 \ 23]$

$[ 28 \ 23 \ 20]$

$\dots,$

$[163 \ 118 \ 102]$

$[163 \ 120 \ 103]$

$[166 \ 121 \ 105]]$

$[ [ 99 \ 67 \ 51]$

$[ 38 \ 28 \ 22]$

$[ 30 \ 26 \ 23]$

$\dots,$

$[161 \ 117 \ 100]$

$[164 \ 121 \ 104]$

$[168 \ 123 \ 106]]$

```

[[127 95 72]
 [ 39 29 22]
 [ 30 25 22]
 ...,
 [165 122 105]
 [169 126 109]
 [173 128 110]]]

```

```

[[[115 110 111]
 [137 129 129]
 [155 146 145]
 ...,
 [159 156 157]
 [141 141 145]
 [121 122 127]]]

```

```

[[123 118 120]
 [143 136 136]
 [159 153 150]
 ...,
 [167 164 165]
 [151 151 154]
 [130 133 137]]]

```

```

[[135 130 130]
 [150 145 141]
 [164 159 153]
 ...,
 [173 174 172]
 [160 162 162]
 [141 144 148]]]

```

```

...,
[[197 196 205]
 [208 209 214]
 [215 216 220]
 ...,
 [222 224 229]
 [205 208 215]
 [185 189 197]]]

```

```
[[190 192 199]
 [203 205 210]
 [213 214 218]
 ...,
 [217 220 225]
 [198 202 209]
 [178 182 191]]
```

```
[[183 186 193]
 [197 199 205]
 [208 210 214]
 ...,
 [212 215 220]
 [192 196 203]
 [171 176 186]]]
```

```
[[[255 253 254]
 [255 253 254]
 [255 253 254]
 ...,
 [197 178 118]
 [195 177 116]
 [192 176 115]]]
```

```
[[[255 253 254]
 [255 253 254]
 [255 253 254]
 ...,
 [197 178 120]
 [195 176 118]
 [193 174 118]]]
```

```
[[[255 253 254]
 [255 253 254]
 [255 253 254]
 ...,
 [197 177 121]
 [194 174 121]
 [190 171 122]]]
```



```

...,
[[ 94  95  97]
 [102 104 108]
 [115 113 115]

...,
[ 68  67  91]
[ 71  55  80]
[ 64  51  82]]

[[ 68  71  69]
 [ 75  79  80]
 [ 82  91  91]

...,
[122 103 109]
[114  84  99]
[ 97  72 100]]

[[ 67  86  89]
 [ 62  83  90]
 [ 47  73  82]

...,
[147 120 125]
[142 110 118]
[133 101 121]]]

...,
[[[ 41  47  84]
 [ 72  78 112]
 [103 102 124]

...,
[125 112 119]
[107 101 117]
[ 76  79 109]]

[[ 93  96 119]
 [112 109 124]
 [130 122 136]

...,
[141 121 118]
[130 112 114]
[122 109 117]]

```

```
[ [119 114 123]
  [133 124 128]
  [164 149 145]
  ...,
  [187 160 135]
  [157 132 120]
  [136 115 109]]
```

```
...,
[ [233 223 207]
  [235 226 218]
  [242 236 235]
  ...,
  [164 170 196]
  [189 185 196]
  [190 182 188]]
```

```
[ [247 250 248]
  [236 238 232]
  [219 215 219]
  ...,
  [118 101 117]
  [153 134 154]
  [174 146 156]]
```

```
[ [253 254 254]
  [246 250 249]
  [226 233 234]
  ...,
  [154 111 93]
  [176 132 104]
  [183 141 116]]]
```

```
[ [ [ 18 18 16]
    [ 35 36 31]
    [ 54 57 52]
    ...,
    [138 140 118]
    [139 141 124]
    [130 129 114]]
```

```
[ [ 21  21  18]
  [ 44  45  39]
  [ 68  74  66]
  ...,
  [128 131 109]
  [116 117  98]
  [ 95  93  76]]
```

```
[ [ 30  31  26]
  [ 54  58  52]
  [ 99 110 104]
  ...,
  [127 129 108]
  [113 114  94]
  [ 99  99  82]]
```

```
...,
[ [178 178 163]
  [161 163 146]
  [170 174 163]
  ...,
  [112 108  85]
  [149 150 131]
  [180 185 172]]
```

```
[ [182 183 167]
  [172 175 162]
  [170 173 162]
  ...,
  [ 88  90  77]
  [122 115  89]
  [157 154 131]]
```

```
[ [188 190 179]
  [173 175 163]
  [160 162 146]
  ...,
  [ 48  50  44]
  [103 103  89]
  [144 137 108]]]
```

```
[[[133 163 75]
   [ 98 120 44]
   [108 132 47]
   ...,
   [ 96 138 97]
   [ 96 146 109]
   [ 81 132 101]]]
```

```
[[[ 79 102 39]
   [ 83 112 33]
   [ 76 106 26]
   ...,
   [ 99 149 76]
   [ 71 119 62]
   [ 58 106 72]]]
```

```
[[[ 35 53 27]
   [ 54 74 46]
   [ 35 55 17]
   ...,
   [110 157 122]
   [ 72 119 89]
   [ 81 131 108]]]
```

```
...,
[[[ 76 108 79]
   [ 70 110 83]
   [ 70 105 84]
   ...,
   [ 25 49 21]
   [ 26 53 24]
   [ 27 54 28]]]
```

```
[[[109 120 97]
   [ 86 116 86]
   [ 71 105 79]
   ...,
   [ 17 46 20]
   [ 23 49 28]
   [ 19 33 18]]]
```

```

[[ 97 117 100]
 [ 70 108 75]
 [ 75 99 69]
 ...,
 [ 8 33 12]
 [ 13 35 18]
 [ 5 22 5]]]
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (209, 64, 64, 3)
train_set_y shape: (1, 209)
test_set_x shape: (50, 64, 64, 3)
test_set_y shape: (1, 50)

```

**Expected Output for m\_train, m\_test and num\_px:**

<b>**m_train**</b>	209
<b>**m_test**</b>	50
<b>**num_px**</b>	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]: train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T.shape
```

```
Out[6]: (12288, 209)
```

```
In [7]: # 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 ####

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]))

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]
```

**Expected Output:**

<b>**train_set_x_flatten shape**</b>	(12288, 209)
<b>**train_set_y shape**</b>	(1, 209)
<b>**test_set_x_flatten shape**</b>	(12288, 50)
<b>**test_set_y shape**</b>	(1, 50)
<b>**sanity check after reshaping**</b>	[17 31 56 22 33]

To represent color images, the red, green and blue channels (RGB) must be specified for each pixel, and so the pixel value is actually a vector of three numbers ranging from 0 to 255.

One common preprocessing step in machine learning is to center and standardize your dataset, meaning that you subtract the mean of the whole numpy array from each example, and then divide each example by the standard deviation of the whole numpy array. But for picture datasets, it is simpler and more convenient and works almost as well to just divide every row of the dataset by 255 (the maximum value of a pixel channel).

Let's standardize our dataset.

```
In [8]: train_set_x = train_set_x_flatten/255.  
       test_set_x = test_set_x_flatten/255.
```

**What you need to remember:**

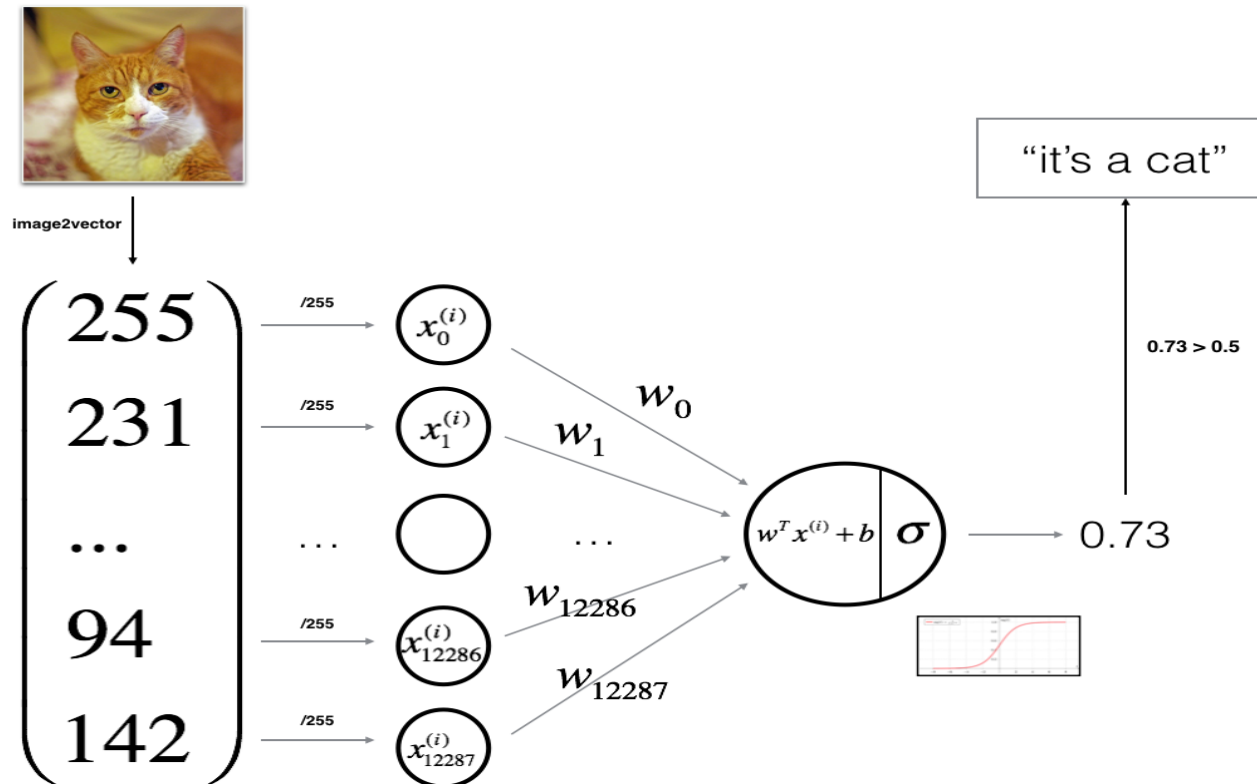
Common steps for pre-processing a new dataset are:

- Figure out the dimensions and shapes of the problem ( $m_{\text{train}}$ ,  $m_{\text{test}}$ ,  $\text{num\_px}$ , ...)
- Reshape the datasets such that each example is now a vector of size ( $\text{num\_px} * \text{num\_px} * 3, 1$ )
- "Standardize" the data

### 3 - General Architecture of the learning algorithm

It's time to design a simple algorithm to distinguish cat images from non-cat images.

You will build a Logistic Regression, using a Neural Network mindset. The following Figure explains why **Logistic Regression is actually a very simple Neural Network!**



**Mathematical expression of the algorithm:**

For one example  $x^{(i)}$ :

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

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

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



The cost is then computed by summing over all training examples:

$$J = \frac{1}{m} \sum_{i=1}^m \mathcal{L}(a^{(i)}, y^{(i)}) \quad (6)$$

**Key steps:** In this exercise, you will carry out the following steps:

- Initialize the parameters of the model
- Learn the parameters for the model by minimizing the cost
- Use the learned parameters to make predictions (on the test set)
- Analyse the results and conclude

## 4 - Building the parts of our algorithm

The main steps for building a Neural Network are:

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

You often build 1-3 separately and integrate them into one function we call `model()`.

### 4.1 - Helper functions

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

```
In [9]: # 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
```

```
In [10]: print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))

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

**Expected Output:**

**sigmoid([0, 2])**	[ 0.5 0.88079708]
---------------------	-------------------

## 4.2 - Initializing parameters

**Exercise:** Implement parameter initialization in the cell below. You have to initialize  $w$  as a vector of zeros. If you don't know what numpy function to use, look up `np.zeros()` in the Numpy library's documentation.

```
In [11]: # 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
```

```
In [12]: dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))

w = [[ 0.]
      [ 0.]]
b = 0
```

**Expected Output:**

<b>** w **</b>	[[ 0.] [ 0.]]
<b>** b **</b>	0

For image inputs, w will be of shape (num\_px × num\_px × 3, 1).

**4.3 - Forward and Backward propagation**

Now that your parameters are initialized, you can do the "forward" and "backward" propagation steps for learning the parameters.

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

**Hints:**

Forward Propagation:

- You get X
- You compute  $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \dots, a^{(m-1)}, a^{(m)})$
- You calculate the cost function:  $J = -\frac{1}{m} \sum_{i=1}^m y^{(i)} \log(a^{(i)}) + (1 - y^{(i)}) \log(1 - a^{(i)})$

Here are the two formulas you will be using:

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

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

```
In [13]: # 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)
### START CODE HERE ### (≈ 2 lines of code)
A = sigmoid(np.dot(w.T,X)+b) # compute activation
cost = -np.sum(Y*np.log(A)+(1-Y)*np.log(1-A))/m # compute cost
### END CODE HERE ###

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

assert(dw.shape == w.shape)
assert(db.dtype == float)
cost = np.squeeze(cost)
assert(cost.shape == ())

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

return grads, cost

```

```
In [14]: 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 = [[ 0.99845601]
      [ 2.39507239]]
db = 0.00145557813678
cost = 5.80154531939
```

## 4.4 - Optimization

- You have initialized your parameters.
- You are also able to compute a cost function and its gradient.
- Now, you want to update the parameters using gradient descent.

**Exercise:** Write down the optimization function. The goal is to learn  $w$  and  $b$  by minimizing the cost function  $J$ . For a parameter  $\theta$ , the update rule is  $\theta = \theta - \alpha d\theta$ , where  $\alpha$  is the learning rate.

**Expected Output:**

<b>** dw **</b>	[[ 0.99845601] [ 2.39507239]]
<b>** db **</b>	0.00145557813678
<b>** cost **</b>	5.801545319394553

```
In [15]: propagate(w, b, X, Y)

Out[15]: ({'db': 0.001455578136784208, 'dw': array([[ 0.99845601],
          [ 2.39507239]])}, 5.8015453193945534)
```



In [16]: *# GRADED FUNCTION: optimize*

```
def 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  $(\text{num\_px} * \text{num\_px} * 3, 1)$*

*$b$  -- bias, a scalar*

*$X$  -- data of shape  $(\text{num\_px} * \text{num\_px} * 3, \text{number of examples})$*

*$Y$  -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape  $(1, \text{number of examples})$*

*$\text{num\_iterations}$  -- number of iterations of the optimization loop*

*$\text{learning\_rate}$  -- learning rate of the gradient descent update rule*

*$\text{print\_cost}$  -- True to print the loss every 100 steps*

*Returns:*

*$\text{params}$  -- dictionary containing the weights  $w$  and bias  $b$*

*$\text{grads}$  -- dictionary containing the gradients of the weights and bias with respect to the cost function*

*$\text{costs}$  -- 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 steps and iterate through them:*

*1) Calculate the cost and the gradient for the current parameters. Use `propagate()`.*

*2) Update the parameters using gradient descent rule for  $w$  and  $b$ .*

*"""*

`costs = []`

`for i in range(num_iterations):`

*# Cost and gradient calculation ( $\approx$  1-4 lines of code)*

*### START CODE HERE ###*

`grads, cost = propagate(w, b, X, Y)`

*### END CODE HERE ###*

*# Retrieve derivatives from grads*

`dw = grads["dw"]`

`db = grads["db"]`

*# update rule ( $\approx$  2 lines of code)*

*### START CODE HERE ###*

`w = w - learning_rate * dw`

`b = b - learning_rate * db`

*### END CODE HERE ###*

```

# Record the costs
if i % 100 == 0:
    costs.append(cost)

# Print the cost every 100 training iterations
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 [17]: `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 = 0.219194504541

```

### Expected Output:

<b>**w**</b>	[[ 0.19033591] [ 0.12259159]]
<b>**b**</b>	1.92535983008
<b>**dw**</b>	[[ 0.67752042] [ 1.41625495]]
<b>**db**</b>	0.219194504541

**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 are two steps to computing predictions:

1. Calculate  $\hat{Y} = A = \sigma(w^T X + b)$
2. Convert the entries of  $a$  into 0 (if activation  $\leq 0.5$ ) or 1 (if activation  $> 0.5$ ), stores the predictions in a vector `Y_prediction`. If you wish, you can use an `if/else` statement in a `for` loop (though there is also a way to vectorize this).

In [18]: *# 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)

    Returns:
    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
    ### 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]=1
        else:
            Y_prediction[0][i]=0
        pass
        ### END CODE HERE ###

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

    return Y_prediction
```

```
In [19]: w = np.array([[0.1124579],[0.23106775]])
b = -0.3
X = np.array([[1.,-1.1,-3.2],[1.2,2.,0.1]])
print ("predictions = " + str(predict(w, b, X)))

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

**Expected Output:**

<b>**predictions**</b>	[[ 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\_test 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 [20]: *# GRADED FUNCTION: model*

```
def 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 function 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 -- Set to true to print the cost every 100 iterations

    Returns:
    d -- dictionary containing information about the 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 ###

    # 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  
  
return d
```

Run the following cell to train your model.



```
In [21]: 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 %
```

### Expected Output:

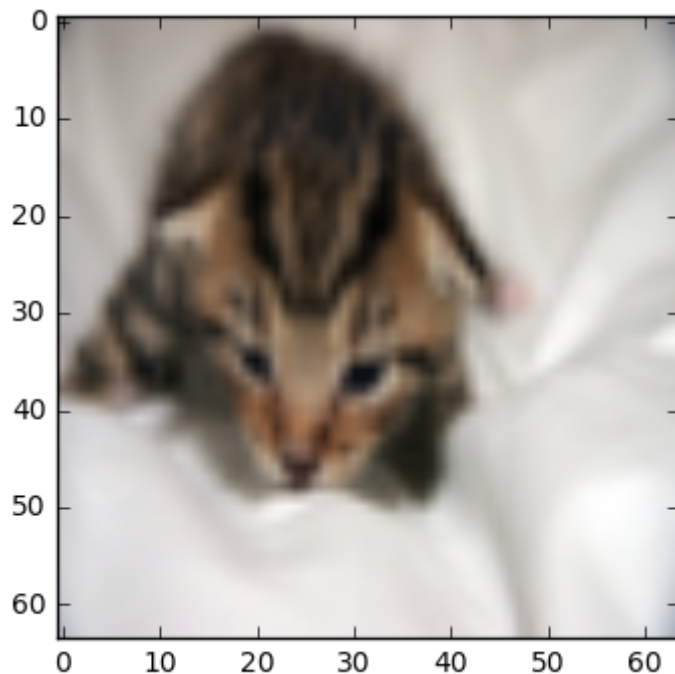
<b>**Cost after iteration 0 **</b>	0.693147
<b>⋮</b>	<b>⋮</b>
<b>**Train Accuracy**</b>	99.04306220095694 %
<b>**Test Accuracy**</b>	70.0 %

**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 we used and that logistic regression is a linear classifier. But no worries, you'll build an even better classifier next week!

Also, you see that the model is clearly overfitting the training data. Later in this specialization you will learn how to reduce overfitting, for example by using regularization. Using the code below (and changing the `index` variable) you can look at predictions on pictures of the test set.

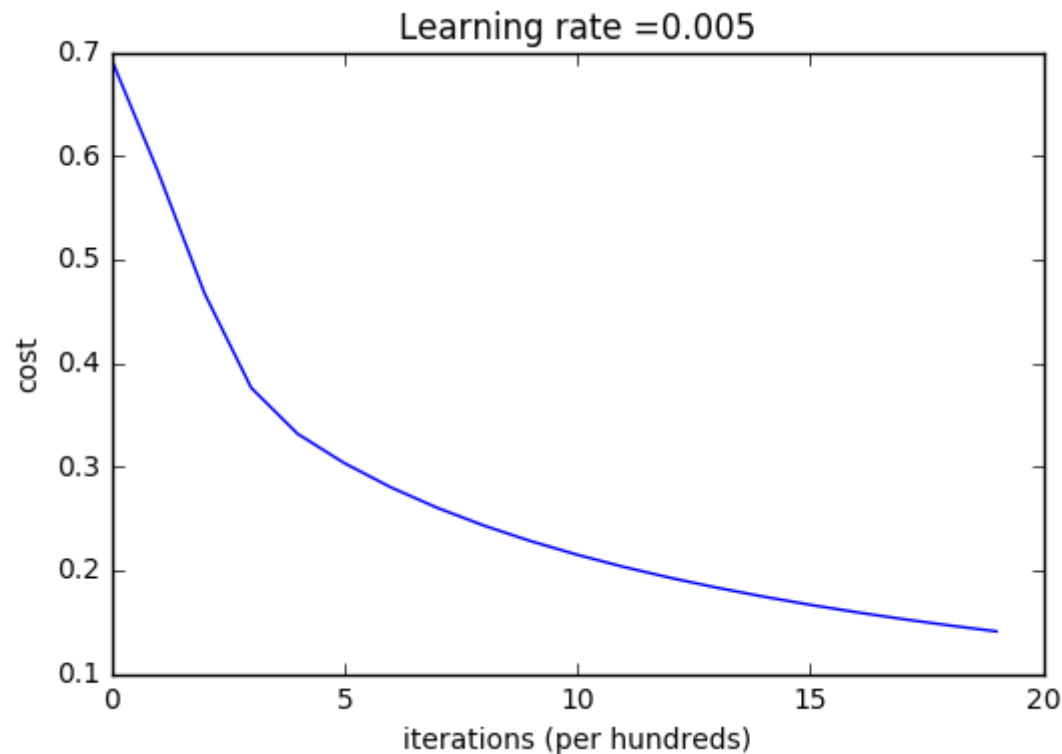
```
In [22]: # Example of a picture that was wrongly classified.
index = 1
plt.imshow(test_set_x[:,index].reshape((num_px, num_px, 3)))
print ("y = " + str(test_set_y[0,index]) + ", you predicted that it is a \"" + classes[d["Y_prediction_test"][0,index]
].decode("utf-8") + "\" picture.")
```

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



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

```
In [23]: # 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. You might see that the training set accuracy goes up, but the test set accuracy goes down. This is called overfitting.

## 6 - Further analysis (optional/ungraded exercise)

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

### Choice of learning rate

**Reminder:** In order for Gradient Descent to work you 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. That's why it is crucial to use a well-tuned learning rate.

Let's compare the learning curve of our model with several choices of learning rates. Run the cell below. This should take about 1 minute. Feel free also to try different values than the three we have initialized the `learning_rates` variable to contain, and see what happens.

```
In [24]: 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 (hundreds)')

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

learning rate is: 0.01  
train accuracy: 99.52153110047847 %  
test accuracy: 68.0 %

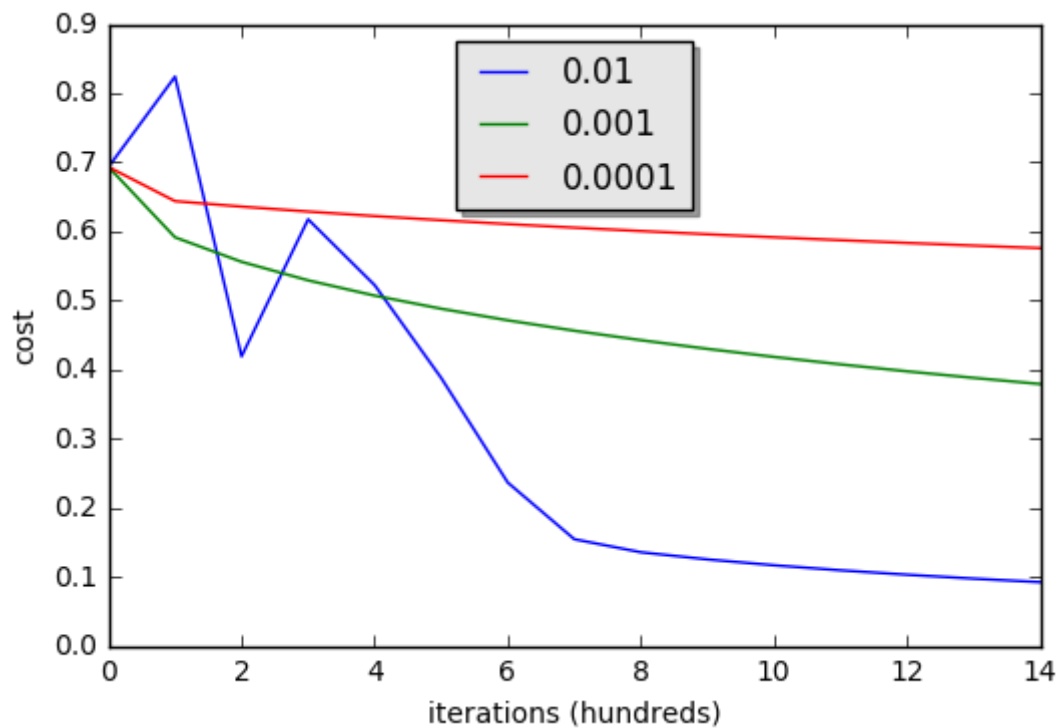
-----

learning rate is: 0.001  
train accuracy: 88.99521531100478 %  
test accuracy: 64.0 %

-----

learning rate is: 0.0001  
train accuracy: 68.42105263157895 %  
test accuracy: 36.0 %

-----



**Interpretation:**

- Different learning rates give different costs and thus different predictions results.
- If the learning rate is too large (0.01), the cost may oscillate up and down. It may even diverge (though in this example, using 0.01 still eventually ends up at a good value for the cost).
- A lower cost doesn't mean a better model. You have to check if there is possibly overfitting. It happens when the training accuracy is a lot higher than the test accuracy.
- In deep learning, we usually recommend that you:
  - Choose the learning rate that better minimizes the cost function.
  - If your model overfits, use other techniques to reduce overfitting. (We'll talk about this in later videos.)

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

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

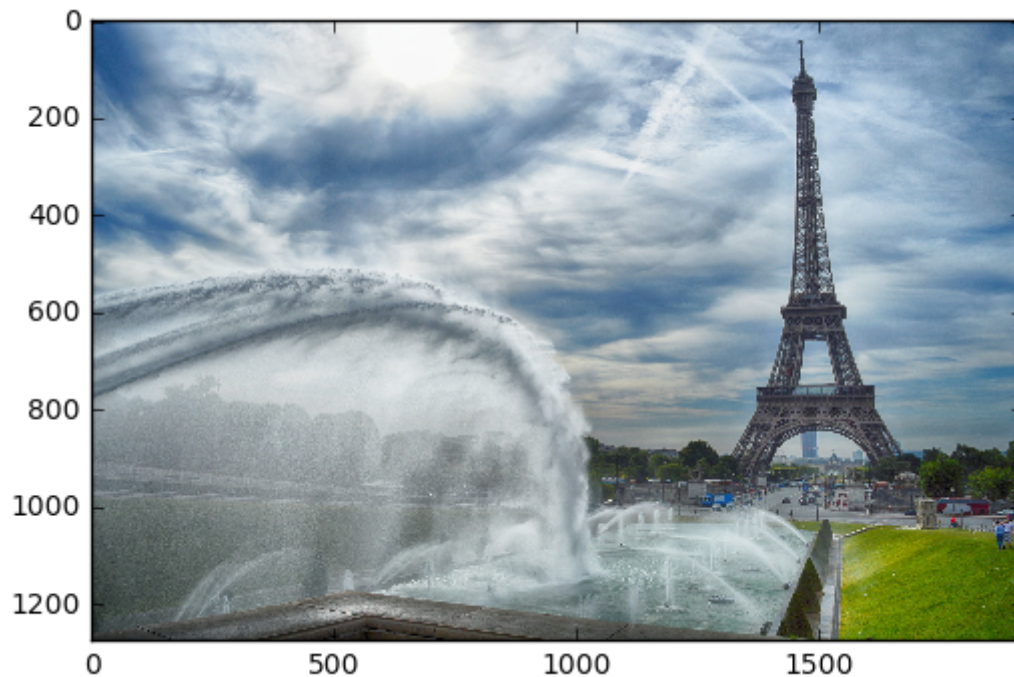
1. Click on "File" in the upper bar of this notebook, then click "Open" to go on your Coursera Hub.
2. Add your image to this Jupyter Notebook's directory, in the "images" folder
3. Change your image's name in the following code
4. Run the code and check if the algorithm is right (1 = cat, 0 = non-cat)!

```
In [25]: ## 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))
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.squeeze(my_predicted_image)),].decode("utf-8") + " picture.")
```

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





**What to remember from this assignment:**

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

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

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

**Bibliography:**

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