Logistic+Regression+with+a+Neural+Network+mindset+v4

January 27, 2018

1 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.

1.1 1 - Packages

First, let's run the cell below to import all the packages that you will need during this assignment. - numpy is the fundamental package for scientific computing with Python. - h5py is a common package to interact with a dataset that is stored on an H5 file. - matplotlib is a famous library to plot graphs in Python. - PIL and scipy are used here to test your model with your own picture at the end.

```
In [3]: 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
```

```
/opt/conda/lib/python3.5/site-packages/matplotlib/font_manager.py:273: UserWarning: Matplotlib i
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a moment.')
/opt/conda/lib/python3.5/site-packages/matplotlib/font_manager.py:273: UserWarning: Matplotlib i
  warnings.warn('Matplotlib is building the font cache using fc-list. This may take a moment.')
```

1.2 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 [4]: # Loading the data (cat/non-cat)
        train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes = load_dataset()
```

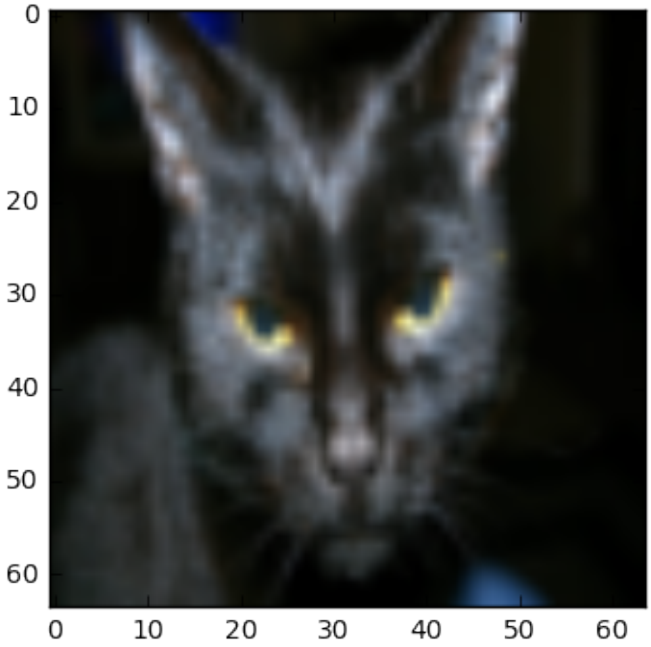
We added "_orig" at the end of image datasets (train and test) because we are going to preprocess them. After preprocessing, we will end up with train_set_x and test_set_x (the labels train_set_y and test_set_y don't need any preprocessing).

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 [5]: np.squeeze?
```

```
In [6]: # 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 = [1], it's a '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 [7]: print(train_set_x_orig.shape)
```

```
(209, 64, 64, 3)
```

```
In [8]: ### START CODE HERE ### ( 3 lines of code)
        m_train = train_set_x_orig.shape[0]
        m_test = test_set_x_orig.shape[0]
        num_px = train_set_x_orig.shape[1]
        ### END CODE HERE ###

        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_train = 209
Number of testing examples: m_test = 50
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:

```
<td>**m_train**</td>
<td> 209 </td>

<td>**m_test**</td>
<td> 50 </td>

<td>**num_px**</td>
<td> 64 </td>
```

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 [9]: flattenBroBro = np.reshape(train_set_x_orig, (train_set_x_orig.shape[0], -1))
        flattenNoNo = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1).T
        print(flattenBroBro.shape)
        print(flattenNoNo.shape)
```

```
(209, 12288)
(12288, 209)
```

```
In [10]: # Reshape the training and test examples

         ### START CODE HERE ### ( 2 lines of code)
         train_set_x_flatten = np.reshape(train_set_x_orig, (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:

```
<td>**train_set_x_flatten shape**</td>
<td> (12288, 209)</td>

<td>**train_set_y shape**</td>
<td>(1, 209)</td>

<td>**test_set_x_flatten shape**</td>
<td>(12288, 50)</td>
```

```
<td>**test_set_y shape**</td>
<td>(1, 50)</td>
```

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 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 substract 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 [11]: 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 size (num_px * num_px * 3, 1) - "Standardize" the data

1.3 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 $x^{(i)}$:

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

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

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

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) - Analyse the results and conclude

1.4 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()`.

1.4.1 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 $sigmoid(w^T x + b) = \frac{1}{1+e^{-(w^T x+b)}}$ to make predictions. Use np.exp().

```
In [12]: # 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 [13]: print ("sigmoid([0, 2]) = " + str(sigmoid(np.array([0,2]))))
         np.zeros?
```

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

Expected Output:

```
<td>**sigmoid([0, 2])**</td>
<td> [ 0.5         0.88079708]</td>
```

1.4.2 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 [14]: # 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 t

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

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

Expected Output:

```
<tr>
    <td>  ** w **  </td>
    <td> [[ 0.]
```

[0.]]

```
</tr>
<tr>
    <td>  ** b **  </td>
    <td> 0 </td>
</tr>
```

For image inputs, w will be of shape (num_px $\times$ num_px $\times$ 3, 1).

1.4.3 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 $A = \sigma(w^T X + b) = (a^{(0)}, a^{(1)}, ..., a^{(m-1)}, a^{(m)})$ - 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:

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

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

In [16]:
```
# 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

    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 activatio
    cost = -np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))/m                                  #
    ### END CODE HERE ###

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

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

    grads = {"dw": dw,
```