Logistic Regression with a Neural Network mindset

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.

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 [153…

```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage


%matplotlib inline
```

2 - Overview of the Problem set

Let's get more familiar with the dataset. Load the data by running the following code.

In [83]:

```
f = h5py.File('train_catvnoncat.h5', 'r')
```

In [84]:

```
list(f.keys())
```

Out[84]:

```
['list_classes', 'train_set_x', 'train_set_y']
```

In [85]:

```
###list(f.keys()) lists out all the datasets present in the h5 file. Since it is a h5 file,it contains multi
train_set_x_orig=np.array(f["train_set_x"][:])
#f["train_set_x"][:] means we are taking all the values of the train_set_x dataset from f file and storing i
#train_set_x contains the training set features
train_set_y=np.array(f["train_set_y"][:])
#same for train_y_orig. train_y_orig contains the training set label, X means features and y means label
```

In [86]:

```
f1 = h5py.File('test_catvnoncat.h5', 'r')
```

In [87]:

```
f1 = h5py.File('test_catvnoncat.h5', 'r')
```

In [88]:

```
test_set_x_orig=np.array(f1["test_set_x"][:])
#test_x_orig contains the test set features
test_set_y=np.array(f1["test_set_y"][:])
#test_y_orig contains the test set label
```

In [89]:

```
test_set_y.shape
```

Out[89]:

```
(50,)
```

In [90]:

```
train_set_y.shape
```

Out[90]:

```
(209,)
```

In [91]:

```
test_set_y = test_set_y.reshape(1,50)
```

In [92]:

```
train_set_y = train_set_y.reshape(1,209)
```

In [93]:

```
train_set_x_orig.shape
```

Out[93]:

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

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]`.

train_set_x_orig is a numpy-array of shape (m_train, num_px, num_px, 3).

In [96]:

```
m_train = train_set_x_orig.shape[0]
m_test =  test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]


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)
```

2.1 - Reshaping of the Problem set

We need to 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.

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
```

What is Flattening ?

Flattening array means converting a multidimensional array into a 1D array.

We can use reshape(-1) to do this.

Method 1

We want rows of the new matrix to be num_px $*$ num_px $*$ 3 which is x[1] *x[2]* 3 the column should their respective columns in m_train which is x[0]

By doing the math and say rows is 64*643* =12288 and column is 209. The python code is X.reshape(12288,209).T

In [97]:

```
train_set_x_orig = train_set_x_orig.reshape(12288,209)
train_set_x_orig.shape
```

Out[97]:

```
(12288, 209)
```

Method 2

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

removing transpose it will be X_flatten = X.reshape(-1, X.shape[0])

where -1 is the unspecified element of new the shape of x

since x is originally of shape (m_train, num_px, num_px, 3) since already used x.shape[0] aka m_train the remainng elements in teh new shape should be num_px *num_px* 3 to make sure x can be reshaped

so instead of specifying num_px *num_px* 3, we specify it as -1

In [98]:

```
train_set_x_orig = train_set_x_orig.reshape(-1, train_set_x_orig.shape[0])
train_set_x_orig.shape
```

Out[98]:

```
(209, 12288)
```

In [99]:

```
train_set_x_orig = train_set_x_orig.T
train_set_x_orig.shape
```

Out[99]:

```
(12288, 209)
```

When we shape the matrix we have the id matrix (we put -1 for id) to have this dimensions (train_set_x_orig.shape[0], -1) # it is the new shape of the matrix of the matrix

In [106…

```
# Reshape the training and test examples


train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0], -1)
test_set_x_flatten =  test_set_x_orig.reshape(test_set_x_orig.shape[0], -1).T


print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

```
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]
```

Why divide by 255 ?

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 [107…

```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
```

3 - General Architecture of the learning algorithm

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:

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

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

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

In [108…

```
def sigmoid(z):
    """
    Compute the sigmoid of z

    Arguments:
    z -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(z)
    """


    s = 1/(1+np.exp(-z))


    return s
```

4.2 - Initializing parameters

Implement parameter initialization in the cell below. You have to initialize w as a vector of zeros.

In [110…

```
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)
    """

    w = np.zeros((dim, 1))
    b = 0


    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

In [111…

```
dim = 2
w, b = initialize_with_zeros(dim)
print ("w = " + str(w))
print ("b = " + str(b))
```

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

In [ ]:

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.

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

Forward Propagation:

- You get X
- You compute $A = \sigma(w^T X + b) = (a^{(1)}, a^{(2)}, \ldots, a^{(m-1)}, a^{(m)})$
- 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 [112…

```
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)

    A = sigmoid(np.dot(w.T, X) + b)                                    # compute activation
    cost = -1./m* np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))                                   # compute cost


    # BACKWARD PROPAGATION (TO FIND GRAD)

    dw = 1./m*np.dot(X, (A-Y).T)
    db = 1./m*np.sum(A-Y)


    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 [113…

```
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 = 0.001455578136784208
cost = 5.801545319394553
```

In [ ]:

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.

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

In [114…

```
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 (num_px * num_px * 3, 1)
    b -- bias, a scalar
    X -- data of shape (num_px * num_px * 3, number of examples)
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat), of shape (1, number of examples)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- True to print the loss every 100 steps

    Returns:
    params -- dictionary containing the weights w and bias b
    grads -- dictionary containing the gradients of the weights and bias with respect to the cost function
    costs -- list of all the costs computed during the optimization, this will be used to plot the learning

    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 (≈ 1-4 lines of code)

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


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

        # update rule (≈ 2 lines of code)

        w = w - learning_rate * dw
        b = b - learning_rate * db


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

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

    params = {"w": w,
              "b": b}

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

    return params, grads, costs
```

In [115…

```
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 = 1.9253598300845747
dw = [[0.67752042]
 [1.41625495]]
db = 0.21919450454067652
```

In [ ]:

In [ ]:

Prediction

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 <= 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 [116…

```
# 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

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


    for i in range(A.shape[1]):

        # Convert probabilities A[0,i] to actual predictions p[0,i]

        if A[0, i] > 0.5:
            Y_prediction[0, i] = 1
        else:
             Y_prediction[0, i] = 0


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

    return Y_prediction
```

In [117…

```
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.]]
```

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.

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()
```

Run the following cell to train your model.

In [157…

```
def model(train_x, train_y_orig, test_x, test_y_orig, num_iterations = 3000, learning_rate = 0.6, print_cost
    w=np.zeros((train_x.shape[0],1))
    b=0
    parameters, grads, costs = optimize(w, b, train_x, train_y_orig, num_iterations, learning_rate, print_co

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]

    # Predict test/train set examples
    Y_prediction_test = predict(w, b, test_x)
    Y_prediction_train = predict(w, b, train_x)


    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - train_y_orig)) * 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - test_y_orig)) * 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
```

In [158…

```
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations = 5000, learning_rate = 0.8, prin
```

```
<ipython-input-112-f5a34df051a0>:27: RuntimeWarning: divide by zero encountered in log
  cost = -1./m* np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))                                   # compute cost
<ipython-input-112-f5a34df051a0>:27: RuntimeWarning: invalid value encountered in multiply
  cost = -1./m* np.sum(Y*np.log(A) + (1-Y)*np.log(1-A))                                   # compute cost
<ipython-input-108-8089dbe7e556>:15: RuntimeWarning: overflow encountered in exp
  s = 1/(1+np.exp(-z))
train accuracy: 99.99876382512535 %
test accuracy: 72.01052229722973 %
```

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

7 - Test with your own image

In [ ]:

```
Shown in the attached second file.

I was unable to load packages in the computer so I had done that using an online jupyter notebook.
```

In [ ]:

```
## START CODE HERE ## (PUT YOUR IMAGE NAME)
my_image = "download.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))
image = image/255.
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.squee
```

Bibliography:

- 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