1.2.8 Code of Logistic Regression with a Neural Network

```python
#Logistic Regression with a Neural Network mindset

# -*- coding: utf-8 -*-

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

# Loading the data (cat/non-cat)
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, classes =
↪  load_dataset()


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

# Reshape the training and test examples
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

#"Standardize" the data
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.


# GRADED FUNCTION: sigmoid
def sigmoid(x):
    """
    Compute the sigmoid of x

    Arguments:
    x -- A scalar or numpy array of any size

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

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

    return s
```

```python
# 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)
    """

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

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

    return w, b


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

    # BACKWARD PROPAGATION (TO FIND GRAD)
    dw = np.dot(X,(A-Y).T)/m
    db = np.sum(A-Y)/m

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

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

    return grads, cost

# 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 (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 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 ( 1-4 lines of code)
        grads, cost = propagate(w, b, X, Y)

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

        # update rule
        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


# 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:A[0][i]=0
        else: A[0][i]=1
    Y_prediction=A

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

    return Y_prediction

#===============================================================================
# Merge all functions into a 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.
    """

    # 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)

    # 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


d = model(train_set_x, train_set_y, test_set_x, test_set_y,
 ↪  num_iterations = 2000, learning_rate = 0.005, print_cost = True)
```