VIETNAM NATIONAL UNIVERSITY, HO CHI MINH CITY
UNIVERSITY OF TECHNOLOGY
FACULTY OF COMPUTER SCIENCE AND ENGINEERING

MACHINE LEARNING

Assignment report

Gender Classification

Lecturer: - Nguyễn Đức Dũng
Student: - Đinh Hoàng Anh 1952553

HO CHI MINH CITY, NOVEMBER 2021

Contents

1 Problem description

1.1 Overview of gender classification

A gender classification system uses face of a person from a given image to tell the gender (male/female) of the given person. A successful gender classification approach can boost the performance of many other applications including face recognition and smart human-computer interface.

Usually facial images are used to extract features and then a classifier is applied to the extracted features to learn a gender recognizer. It is an active research topic in Computer Vision and Biometrics fields. The gender classification result is often a binary value, e.g., 1 or 0, representing either male or female. Gender recognition is essentially a two-class classification problem. Although other biometric traits could also be used for gender classification, such as gait, face-based approaches are still the most popular for gender discrimination.

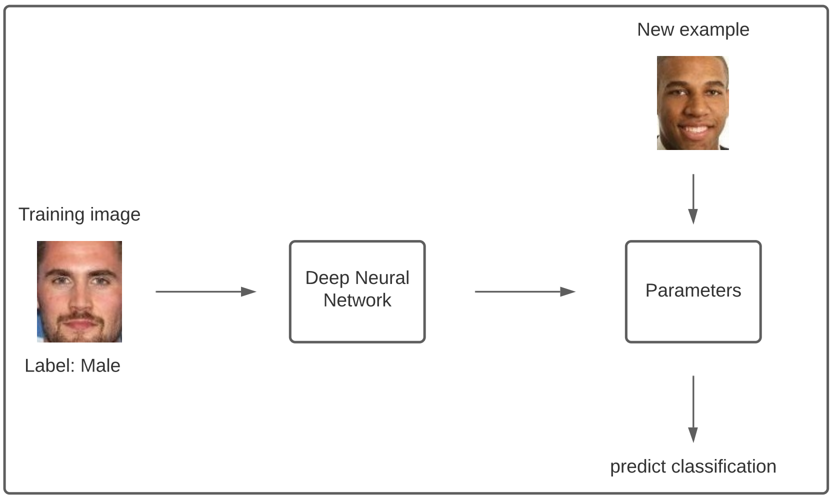

Figure 1: Overview of gender classification

1.2 Detailed description

The problem is describe as:

- Get an input image.
- Output the gender prediction: male or female.
- The problem become a binary classification.

2 Deep Neural Networks

2.1 Deep L-Layer Neural Network

In this assignment, I will use **Deep Neural Network** with 5 layer. The number of units in each layer is $n^{[1]} = 30, n^{[2]} = 20, n^{[3]} = 7, n^{[4]} = 5, n^{[5]} = 1$ respectively.

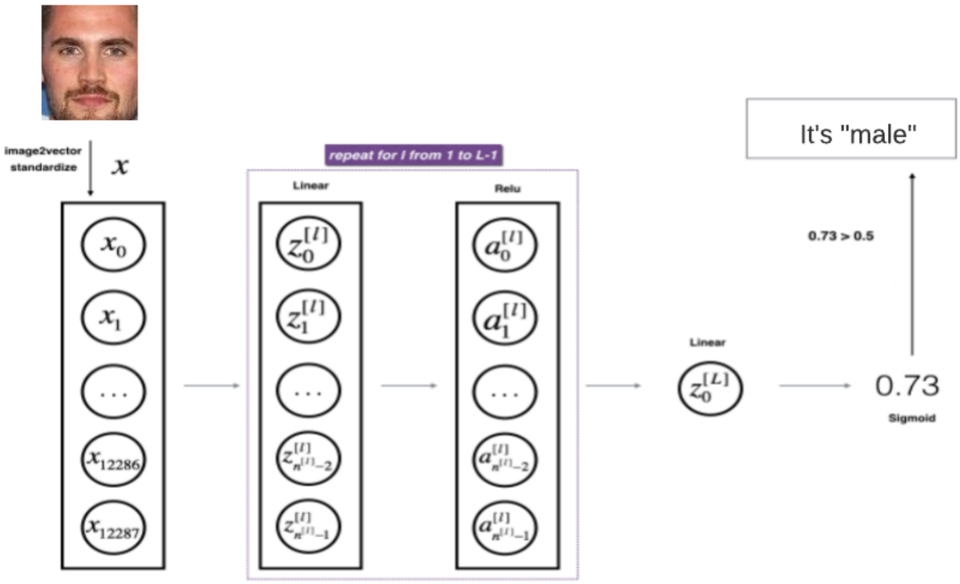

Figure 2: Deep neural network

- The input is a (64,64,3) image which is flattened to a vector of size (12288,1).
- The corresponding vector: $[x_0, x_1, ..., x_{12287}]^T$ is then multiplied by the weight matrix $W^{[1]}$ and then added the intercept $b^{[1]}$. The result is called the linear unit.
- Next, take the relu of the linear unit. This process could be repeated several times for each $(W^{[l]}, b^{[l]})$ depending on the model architecture.
- Finally, take the sigmoid of the final linear unit. If it is greater than 0.5, classify it as male, otherwise classify it as female.

2.2 Forward Propagation

- Input: $a^{[l-1]}$
- Output: $a^{[l]}, cache(z^{[l]})$

The linear forward propagation computed by the following equations:

$$z^{[l]} = w^{[l]}a^{[l-1]} + b[l]$$
$$a^{[l]} = g^{[l]}(z^{[l]})$$

With m examples, we have:

$$Z^{[l]} = W^{[l]}A^{[l-1]} + b[l]$$

with $A^{[0]} = X$

$$A^{[l]} = g^{[l]}(Z^{[l]})$$

2.3 Backrward Propagation

- Input: $da^{[l]}$
- Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

With 1 training sample, we compute as:

$$dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}} = da^{[l]} \times g'^{[l]}(Z^{[l]})$$

$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = dZ^{[l]}a^{[l-1]T}$$

$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = dZ^{[l]}$$

$$da^{[l-1]} = \frac{\partial \mathcal{J}}{\partial a^{[l-1]}} = W^{[l]T}dZ^{[l]}$$

With m samples, we compute as:

$$dZ^{[l]} = \frac{\partial \mathcal{L}}{\partial Z^{[l]}} = da^{[l]} \times g'^{[l]}(Z^{[l]})$$

$$dW^{[l]} = \frac{\partial \mathcal{J}}{\partial W^{[l]}} = \frac{1}{m}dZ^{[l]}A^{[l-1]T}$$

$$db^{[l]} = \frac{\partial \mathcal{J}}{\partial b^{[l]}} = \frac{1}{m}\sum_{n=1}^{m} dZ^{[l]}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{J}}{\partial A^{[l-1]}} = W^{[l]T}dZ^{[l]}$$

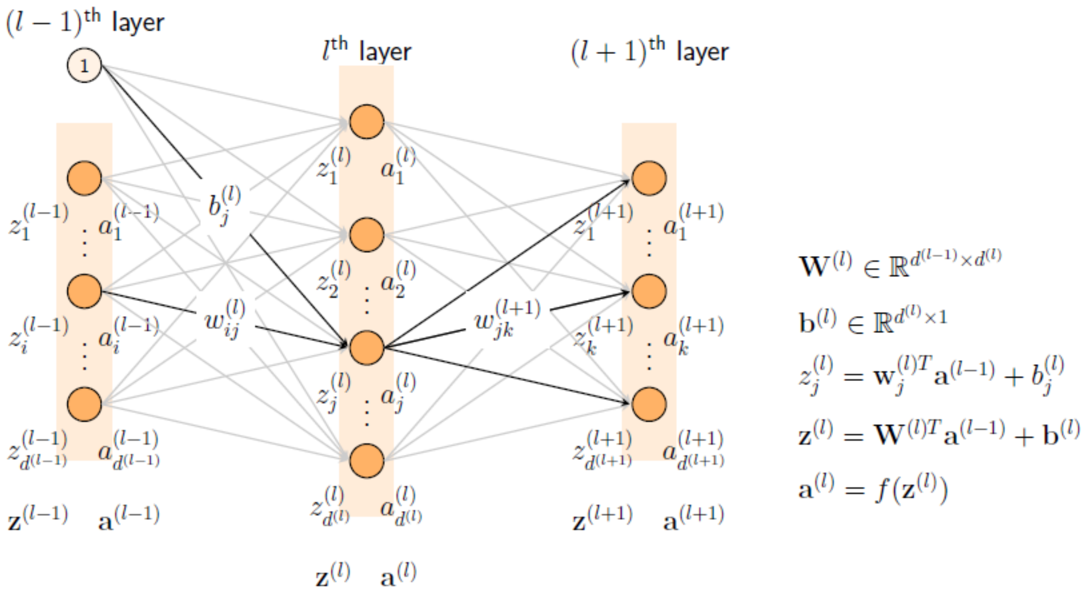

Figure 3: Backward propagation calculation

2.4 Update Paramters

After backward propagation, I will update parameters as the following equations (α is the learning rate):

$$W^{[l]} = W^{[l]} - \alpha dW^{[l]}$$
$$b^{[l]} = b^{[l]} - \alpha db^{[l]}$$

2.5 Cost Function

Compute the cross-entropy cost $\mathcal{L}$ using the following formula:

$$\mathcal{L} = -\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}log(a^{[L](i)}) + (1 - y^{(i)})log(1 - a^{[L](i)}))$$

3 Dataset

The dataset I use in this assignment is found on Kaggle.
https://www.kaggle.com/cashutosh/gender-classification-dataset
This dataset contain about 28500 images of each class (male and female). I just use 149 first images of each class for training and 60 last images for testing.

4 Implementation

4.1 Packages

First, import all the packages I will need during this assignment.

- **numpy** is the main package for scientific computing with Python.
- **matplotlib** is a library to plot graphs in Python.
- **cv2** is used for resizing the image to 64x64

```
import time
import scipy
from scipy import ndimage
import pathlib
import numpy as np
from numpy import asarray
import matplotlib.pyplot as plt
from PIL import Image
import random
import cv2
import os
```

4.2 Data Pre-processing

Because the size of images in this dataset is not consistent, I have to resize all of images into 64x64

```
image_path = 'C:/Users/Admin/PycharmProjects/ML_asm/dataset'
i = 0
for path in os.listdir(image_path):
  full_path = os.path.join(image_path, path)
  img = cv2.imread(full_path)
  res = cv2.resize(img, dsize=(64,64), interpolation=cv2.INTER_CUBIC)
  cv2.imwrite('Image' + str(i) + '.jpg', res)
  i+=1
```

Next, I will read all images and save into two lists (training list and testing list)

```
train_images = list()
test_images = list()

image_path = 'C:/Users/Admin/PycharmProjects/ML_asm/newMale'
for path in os.listdir(image_path):
    full_path = os.path.join(image_path, path)
    img = Image.open(full_path)
    train_images.append((img, 1))

image_path = 'C:/Users/Admin/PycharmProjects/ML_asm/newFemale'
for path in os.listdir(image_path):
    full_path = os.path.join(image_path, path)
    img = Image.open(full_path)
    train_images.append((img, 0))

image_path = 'C:/Users/Admin/PycharmProjects/ML_asm/newMale_validation'
for path in os.listdir(image_path):
    full_path = os.path.join(image_path, path)
```

```
    img = Image.open(full_path)
    test_images.append((img, 1))

image_path = 'C:/Users/Admin/PycharmProjects/ML_asm/newFemale_validation'
for path in os.listdir(image_path):
    full_path = os.path.join(image_path, path)
    img = Image.open(full_path)
    test_images.append((img, 0))

print(len(train_images))
random.shuffle(train_images)
random.shuffle(test_images)
```

Finally, convert images into array and modify the shape

```
def load_data():
    train_data = [asarray(pair[0]) for pair in train_images]
    test_data = [asarray(pair[0]) for pair in test_images]
    train_label = [pair[1] for pair in train_images]
    test_label = [pair[1] for pair in test_images]

    print(np.shape(train_data))

    train_set_x_orig = np.array(train_data) # your train set features
    train_set_y_orig = np.array(train_label) # your train set labels

    print(np.shape(train_set_x_orig))

    test_set_x_orig = np.array(test_data) # your test set features
    test_set_y_orig = np.array(test_label) # your test set labels

    classes = np.array((b'Female', b'Male')) # the list of classes

    train_set_y_orig = train_set_y_orig.reshape((1, train_set_y_orig.shape[0]))
    test_set_y_orig = test_set_y_orig.reshape((1, test_set_y_orig.shape[0]))

    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_orig,
    classes

train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T
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))
```

4.3 Initialization

- The model's structure is [LINEAR -> RELU] × (L-1) -> LINEAR -> SIGMOID. I.e., it has $L-1$ layers using a ReLU activation function followed by an output layer with a sigmoid activation function.
- I will store $n^{[l]}$, the number of units in different layers, in a variable $layer_{d}ims$

```python
def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each layer in
    our network

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", ..., "
    WL", "bL":
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
                    bl -- bias vector of shape (layer_dims[l], 1)
    """

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims) # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims[l-1])
 * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[l -
1]))
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters
```

4.4 Activation funcion

In this assignment, I will use two activation functions: **sigmoid** and **relu**.

```python
def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the
    backward pass efficiently
```

```
    """

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache


def relu_backward(dA, cache):
    """
    Implement the backward propagation for a single RELU unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache
    dZ = np.array(dA, copy=True) # just converting dz to a correct object.

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):
    """
    Implement the backward propagation for a single SIGMOID unit.

    Arguments:
    dA -- post-activation gradient, of any shape
    cache -- 'Z' where we store for computing backward propagation efficiently

    Returns:
    dZ -- Gradient of the cost with respect to Z
    """

    Z = cache

    s = 1/(1+np.exp(-Z))
    dZ = dA * s * (1-s)

    assert (dZ.shape == Z.shape)

    return dZ
```

4.5 Forward Propagation

First, I will build the linear part of forward propagation.

```
def linear_forward(A, W, b):
    """
    Implement the linear part of a layer's forward propagation.
```

```

    Arguments:
    A -- activations from previous layer (or input data): (size of previous layer,
     number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of
    previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)

    Returns:
    Z -- the input of the activation function, also called pre-activation
    parameter
    cache -- a python tuple containing "A", "W" and "b" ; stored for computing the
     backward pass efficiently
    """

    Z = np.dot(W, A) + b
    cache = (A, W, b)

    return Z, cache
```

Next, I will implement forward propagation with activation function.

```
def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of previous
    layer, number of examples)
    W -- weights matrix: numpy array of shape (size of current layer, size of
    previous layer)
    b -- bias vector, numpy array of shape (size of the current layer, 1)
    activation -- the activation to be used in this layer, stored as a text string
    : "sigmoid" or "relu"

    Returns:
    A -- the output of the activation function, also called the post-activation
    value
    cache -- a python tuple containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = relu(Z)

    cache = (linear_cache, activation_cache)

    return A, cache
```

Finally, the full version of forward propagation

```
def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID
    computation

    Arguments:
    X -- data, numpy array of shape (input size, number of examples)
```

```
    parameters -- output of initialize_parameters_deep()

    Returns:
    AL -- activation value from the output (last) layer
    caches -- list of caches containing:
                every cache of linear_activation_forward() (there are L of them, indexed from 0 to L-1)
    """

    caches = []
    A = X
    L = len(parameters) // 2     # number of layers in the neural network


    for l in range(1, L):
        A, cache = linear_activation_forward(A_prev, parameters["W" + str(l)], parameters["b" + str(l)], activation = "relu")

        caches.append(cache)


    AL, cache = linear_activation_forward(A, parameters["W"+ str(L)], parameters["b" + str(L)], activation = "sigmoid")
    caches.append(cache)

    return AL, caches
```

4.6 Cost Function

```
def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape (1, number of examples)
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat), shape (1, number of examples)

    Returns:
    cost -- cross-entropy cost
    """

    m = Y.shape[1]

    cost = -np.sum(np.dot(Y, np.log(AL).T) + np.dot(1 - Y, np.log(1 - AL).T)) / m

    cost = np.squeeze(cost)

    return cost
```

4.7 Backward Propagation

Backpropagation is used to calculate the gradient of the loss function with respect to the parameters.

```
def linear_backward(dZ, cache):
```

```
    """
    Implement the linear portion of backward propagation for a single layer (layer l)

    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current layer l)
    cache -- tuple of values (A_prev, W, b) coming from the forward propagation in the current layer

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = np.dot(dZ, A_prev.T) / m
    db = np.sum(dZ, axis = 1, keepdims = True) / m
    dA_prev = np.dot(W.T, dZ)

    return dA_prev, dW, db
```

Next, I will implement backward propagation with activation function.

```
def linear_activation_backward(dA, cache, activation):
    """
    Implement the backward propagation for the LINEAR->ACTIVATION layer.

    Arguments:
    dA -- post-activation gradient for current layer l
    cache -- tuple of values (linear_cache, activation_cache) we store for computing backward propagation efficiently
    activation -- the activation to be used in this layer, stored as a text string: "sigmoid" or "relu"

    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the previous layer l-1), same shape as A_prev
    dW -- Gradient of the cost with respect to W (current layer l), same shape as W
    db -- Gradient of the cost with respect to b (current layer l), same shape as b
    """
    linear_cache, activation_cache = cache

    if activation == "relu":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

Finally, the full version of backward propagation

```
def L_model_backward(AL, Y, caches):
```

```
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LINEAR -> SIGMOID group

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_forward())
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (it's caches[l], for l in range(L-1) i.e l = 0...L-2)
                the cache of linear_activation_forward() with "sigmoid" (it's caches[L-1])

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))


    current_cache = caches[L-1]
    dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dAL, current_cache, activation = "sigmoid")
    grads["dA" + str(L-1)] = dA_prev_temp
    grads["dW" + str(L)] = dW_temp
    grads["db" + str(L)] = db_temp


    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(dA_prev_temp, current_cache, activation = "relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads
```

4.8 Update Parameters

```
def update_parameters(params, grads, learning_rate):
    """
    Update parameters using gradient descent

    Arguments:
    params -- python dictionary containing your parameters
    grads -- python dictionary containing your gradients, output of L_model_backward

```

```
    Returns:
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """
    parameters = params.copy()
    L = len(parameters) // 2 # number of layers in the neural network
    print(grads)

    for l in range(L):
        parameters['W' + str(l+1)] = parameters['W' + str(l+1)] - learning_rate*grads["dW" + str(l+1)]
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_rate*grads["db" + str(l+1)]

    return parameters
```

4.9 Merge all together

```
def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):
    """
    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 = initialize_parameters_deep(layers_dims)

    # Loop (gradient descent)
    for i in range(0, num_iterations):
        AL, caches = L_model_forward(X, parameters)
        cost = compute_cost(AL, Y)
        grads = L_model_backward(AL, Y, caches)
        parameters = update_parameters(parameters, grads, learning_rate)

        # Print the cost every 100 iterations
        if print_cost and i % 100 == 0 or i == num_iterations - 1:
            print("Cost after iteration {}: {}".format(i, np.squeeze(cost)))
        if i % 100 == 0 or i == num_iterations:
            costs.append(cost)

    return parameters, costs
```

5 Train and test the model

I will train the model with 2500 epochs

```
layers_dims = [12288, 30, 20, 7, 5, 1] #  5-layer model
parameters, costs = L_layer_model(train_x, train_y, layers_dims, num_iterations =
    2500, print_cost = True)
```

The result displayed on screen:

```
Cost after iteration 0: 0.6954910520868507
Cost after iteration 100: 0.6363603599908855
Cost after iteration 200: 0.5951576846762471
Cost after iteration 300: 0.5356822577317358
Cost after iteration 400: 0.6437753935550212
Cost after iteration 500: 0.6005912827800174
Cost after iteration 600: 0.41064940228175373
Cost after iteration 700: 0.4713705382972464
Cost after iteration 800: 0.45382283383380806
Cost after iteration 900: 0.5756933350193408
Cost after iteration 1000: 0.43581980393225767
Cost after iteration 1100: 0.25193893442340576
Cost after iteration 1200: 0.6394347638282675
Cost after iteration 1300: 0.8704262146564183
Cost after iteration 1400: 0.6009575015151132
Cost after iteration 1500: 0.3736721004281588
Cost after iteration 1600: 0.21695661007327502
Cost after iteration 1700: 0.6208097949566367
Cost after iteration 1800: 0.08488470674478711
Cost after iteration 1900: 0.172775495132184
Cost after iteration 2000: 1.329117758604708
Cost after iteration 2100: 0.27716715465070896
Cost after iteration 2200: 0.056080448570820104
Cost after iteration 2300: 0.29480557061918305
Cost after iteration 2400: 0.014825415876161705
Cost after iteration 2499: 0.007212752044204772
```

```
pred_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.9999999999999998
```

```
pred_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.9333333333333333
```

6 Source code and reference

Source code I upload all the source code and dataset here:
https://github.com/dinhhoanganh2001/Gender-Classification
Notebook version:
https://colab.research.google.com/drive/1KoVMBaBXttQlQf2HOXYuwtExxD840Gen?usp=sharing

References

[1] Vũ Hữu Tiệp (2018), Machine Learning cơ bản.

[2] Neural Networks and Deep Learning (Course on Coursera)-Andrew Ng.

[3] B.A.Golomb, D.T.Lawrence, T.J.Sejnowski (1991), SEXNET: A Neural Network Identifies Sex from Human Faces.