Chapter 7

Multiple-Layer Neural Networks

7.1 Learning Objectives

The previous chapter addressed the basic concepts of neural networks with one hidden layer. These concepts form a foundation of deep learning machines. In this chapter, we will extend these concepts to general multiple-layer neural networks, called *deep neural networks*, which consist of many hidden layers. Multiple-layer neural networks or deep learning are often used to recognize complicated patterns, such as face recognition and nature language process, such as speech recognition, language translation, and speech-text transcription. To make it concrete, we use an example, image classification, to demonstrate a deep NN development which is close to practical applications. Upon the completion of this chapter, one should be able to

- Understand and use ReLU activation function.
- To initialize parameters of neural network.
- Apply gradient descent to multiple-layer neural network training.
- Gain some programming skills, such as image reading and display, accuracy computing.
- Develop neural networks with any size from the scratch.

7.2 Representation of Multiple-Layer Neural Networks

7.2.1 Architecture

A deep neural network consists of multiple hidden layers with multiple units in each layer. For instance, a fully connected 4-hidden-layer neural network with 4 units in the output layer is shown in Fig.1. The input x has 8 features. There are 9 units in each hidden layer. Please note that different hidden layers may have different numbers of units. The network delivers 4 outputs.

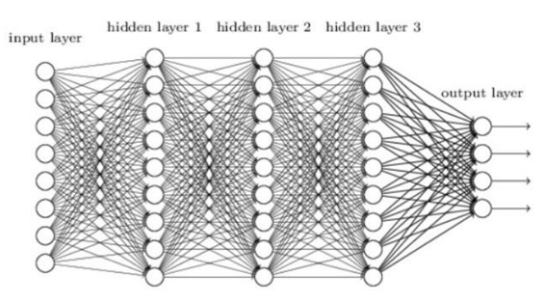


Fig.1 multiple layer neural network

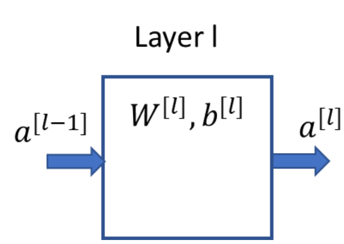


Fig. 2 diagram of layer *l*.

Now let's focus on one particular layer. Let *L* be the total number of layers (not including the input layer, input layer is called layer [0], just input data). Thus, layer *l*, *l=1,2,..,L-1*, is hidden layer, and layer *L* is the output layer. In general, layer *l* can be defined by weight matrix $W^{[l]}$ and bias vector $b^{[l]}$, and an activation function $g^{[l]}(\,)$ for each unit. The input and output of layer *l* are denoted by vectors $a^{[l-1]}$ and $a^{[l]}$, respectively. The block diagram representation of layer *l* is shown in Fig.2. The shapes of $W^{[l]}$ and $b^{[l]}$ are defined by the number of units ($n^{[l]}$ and $n^{[l-1]}$) in layer *l* and layer *l-1*.

Now let's zoom in layer *l* with parameters $W^{[l]}$ and $b^{[l]}$. The shape of $W^{[l]}$ is $(n^{[l]}, n^{[l-1]})$, and shape of $b^{[l]}$ is $(n^{[l]}, 1)$. The *ith* row in $W^{[l]}$ and $b^{[l]}$ is responsible for the input of the *ith* unit in the layer. The relationship between the input and output in layer *l* can be described as a forward propagation

$$z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]} \tag{1.a}$$

Check shape consistence for (1.a): $(n^{[l]}, 1)$=$(n^{[l]}, n^{[l-1]}) \times (n^{[l-1]}, 1) + (n^{[l-1]}, 1)$

$$a^{[l]} = g^{[l]}(z^{[l]}) \tag{1.b}$$

Check shape consistence for (1.b): $(n^{[l]}, 1)$=$g^{[l]}(n^{[l]}, 1)$. Please note that (1.a) and (1.b) is based on one data example.

7.2.2 Forward propagation and backward propagation

The data flow, shown in Fig.3, from the input layer to the output layer is referred as forward propagation, which involves computing of (1.a) and (1.b) for all layers. The purpose of calculating forward propagation is two-fold: 1) to deliver quantities needed for derivative computing during the training process; 2) to predict the output for an input *x* during the inference for new data examples. The backward propagation will provide all derivatives of cost function with respect to parameters, which are needed in the gradient descent algorithm. The gradient descent algorithm updates the parameters until the optimal parameters have been found according to a stopping criterion.

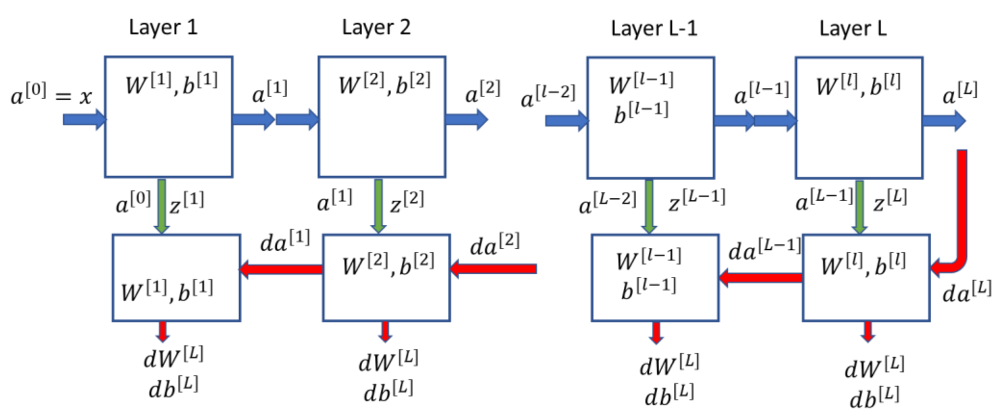


Fig.3 forward and backward block diagram

In Fig.3, the top row of boxes shows the forward propagation, i.e. the neural network, and the bottom row the backward propagation for derivative computation. In the previous chapter, we derived the equations in forward propagation and backward propagation as follows.

1) Forward propagation for layer *l*
   Input: $a^{[l-1]}$
   Output: $a^{[l]}$, $z^{[l]}$(saved to cache)

   For one data example *x*
   $a^{[0]} = x$
   $z^{[l]} = W^{[l]}a^{[l-1]} + b^{[l]}$
   $a^{[l]} = g^{[l]}(z^{[l]})$

   For m data examples (vectorized) (shape)
   $A^{[0]} = X$ (n_x, m)

   $Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$ $(n^{[l]}, m) = (n^{[l]}, n^{[l-1]}) \times (n^{[l-1]}, m) + (n^{[l-1]}, 1)$
   $A^{[l]} = g^{[l]}(Z^{[l]})$ $(n^{[l]}, m)$

2) Backward propagation for layer *l*
   Input: $da^{[l]}$(from forward propagation), $a^{[l-1]}, z^{[l]}$ (from cache)
   Output: $da^{[l-1]}, dW^{[l]}, db^{[l]}$

   For one data example *x* (shape)
   $da^{[L]} = -\frac{y}{a^{[L]}} + \frac{1-y}{1-a^{[L]}}$ (assume sigmoid function in the output layer)
   $da^{[L]} = \frac{dL(\hat{y},y)}{da^{[L]}} = -\frac{y}{a^{[L]}}$ (assume softmax in the output layer)
   $dz^{[L]} = a^{[L]} - y$ (for either sigmoid or softmax)

   $dz^{[l]} = da^{[l]} * g^{[l]'}(z^{[l]})$ $(n^{[l]}, 1) = (n^{[l]}, 1) * (n^{[l]}, 1)$
   $dW^{[l]} = dz^{[l]}{a^{[l-1]}}^T$ $(n^{[l]}, n^{[l-1]}) = (n^{[l]}, 1) \times (n^{[l-1]}, 1)^T$
   $db^{[l]} = dz^{[l]}$ $(n^{[l]}, 1) = (n^{[l]}, 1)$
   $da^{[l-1]} = {W^{[l]}}^T dz^{[l]}$ $(n^{[l-1]}, 1) = (n^{[l]}, n^{[l-1]})^T \times (n^{[l]}, 1)$

   For m data examples (vectorized)
   $dA^{[L]} = -\frac{Y}{a^{[L]}} + \frac{1-Y}{1-a^{[L]}}$ $(1, m)$ $for\ sigmoid\ function$ at layer L

   $dA^{[L]} = -\frac{Y}{a^{[L]}}$ $(n^{[L]}, m)$ for softmax function at layer L

   $dZ^{[L]} = A^{[L]} - Y$ ($(n^{[L]}, m)$ for either sigmoid or softmax)
   $dZ^{[l]} = dA^{[l]} * g^{[l]'}(Z^{[l]})$ $(n^{[l]}, m) = (n^{[l]}, m) * (n^{[l]}, m)$
   $dW^{[l]} = dZ^{[l]}{A^{[l-1]}}^T * (\frac{1}{m})$ $(n^{[l]}, n^{[l-1]}) = (n^{[l]}, m) \times (n^{[l-1]}, m)^T$
   $db^{[l]} = \left(\frac{1}{m}\right) np.sum(dZ^{[l]}, axis = 1, keepdims = True)$ $(n^{[l]}, 1) = (n^{[l]}, 1)$
   $dA^{[l-1]} = {W^{[l]}}^T dZ^{[l]}$ $(n^{[l-1]}, 1) = (n^{[l]}, n^{[l-1]})^T \times (n^{[l]}, 1)$

Attention should be paid to the differences when different activation functions (sigmoid for binary classification or softmax for multiple classification) are applied in the output layer. In the case of softmax, Y is encoded as one-hot code matrix with a shape of $(n^{[L]}, m)$. Although $dA^{[L]}$ has different equations for sigmoid and softmax functions, $dZ^{[L]} = A^{[L]} - Y$ is the same for both (see (6.35) and (6.56) for the reason).

7.3 L-layer Neural Network for Image Classification: A Tutorial

In this section, we will a neural network for image classification. In the dataset, some images include a cat while others don't. Our trained neural network will recognize whether an image has a cat in it. The image size is $(64 \times 64 \times 3)$, i.e., 64-by-64 pixels, and 3 values (for red, blue and green) at each pixel. An image is converted into a vector of 12288 elements as the input to the neural network. The hyperparameters and parameters of the neural network include:

- Learning rate: α
- The number of iterations for parameter update
- The number of layers, L.
- The number of units for each layer: $n^{[1]}, n^{[2]}, \dots, n^{[L]}$
- Choice of activation functions: ReLu for hidden layers, sigmoid for output layer
- Learned parameters: $W^{[1]}, b^{[1]}, W^{[2]}, b^{[2]}, \dots, W^{[L]}, b^{[L]}$

The architecture of the neural network is illustrated by Fig. 4, where ReLu is a widely used activation function for computer vision, defined by

$$ReLu(z) = \begin{cases} z & z \geq 0 \\ 0 & otherwise \end{cases} \qquad (2)$$

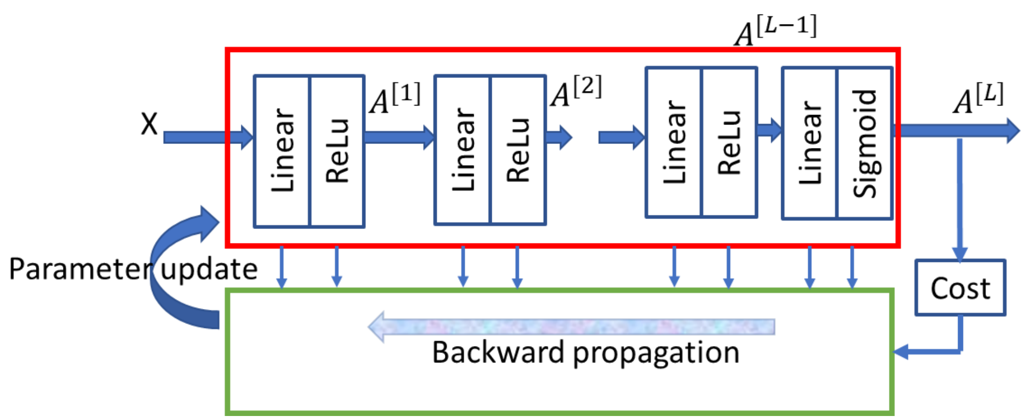


Fig.4 L-layer neural network with ReLu in hidden layers

The development process includes the following steps:

1) Initialize the parameters for an L-layer neural network.
2) Implement the forward propagation module (shown in the red box in Fig.4).

- Complete the LINEAR part of a layer's forward propagation step (resulting in Z[l]).
- Complete ACTIVATION function (relu/sigmoid).
- Combine the previous two steps into a LINEAR_ACTIVATION forward function.
- Stack the LINEAR_RELU block L-1 time (for layers 1 through L-1) and add a LINEAR_SIGMOID block at the end (for the final layer L). This results in a new L_model_forward function.

3) Compute the loss.
4) Implement the backward propagation module (denoted in the green box in Fig.4).
    - Complete the LINEAR part of a layer's backward propagation step.
    - Compute gradient of the ACTIVATE function (relu_backward/sigmoid_backward)
    - Combine the previous two steps into a LINEAR_ACTIVATION backward function.
    - Stack LINEAR_RELU backward block L-1 times and add LINEAR-_SIGMOID backward in a new L_model_backward function
5) Finally update the parameters.

7.3.1 Implementation of block functions

We develop the basic blocks/functions for the DNN model, which were mentioned above. All the functions are included in the file *dnn_utils.py*. To utilize these functions, a top-level file needs to import the package *dnn_utils* at the beginning.

1) Functions sigmoid(Z), relu(Z), relu_backward(dA,Z), and sigmoid_backward(dA,Z)
   sigmoid(Z), relu(Z): compute the value of the function, and return both function value and Z.
   relu_backward(dA,Z): compute $\frac{dJ}{dZ} = \frac{dJ}{dA}\frac{dA}{dZ}$ (denoted as: $dZ = dA\frac{dA}{dZ}$, where $\frac{dA}{dZ} = 1, Z > 0$; $\frac{dA}{dZ} = 0, Z < 0$), and return the derivative dZ.
   sigmoid_backward(dA,Z): compute $\frac{dJ}{dZ} = \frac{dJ}{dA}\frac{dA}{dZ}$ (denoted as: $dZ = dA\frac{dA}{dZ}$, where $\frac{dA}{dZ} = sigmoid(Z) * (1 - sigmoid(Z))$, and return the derivative dZ

2) Initializations of parameters
   initialize_parameters_deep(layer_dims):initialize the parameters for L-layer neural network. The input layer_dims is an array that defines the number of units in each layer.
   Both two functions output a dictionary that include parameters W and b
   Note: Scaling of the initial parameter is important. For the 2-layer nn, both /np.sqrt(n_x) and *0.01 work. But for L-layer nn, *0.01 does not work (demonstrated by the later image classification example). Thus, the parameters are scaled by / np.sqrt(layer_dims[l-1]), not *0.01.
```
parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])/ np.sqrt(layer_dims[l-1])
```

3) linear_forward(A,W,b)
$$Z^{[l]} = W^{[l]}A^{[l-1]} + b^{[l]}$$
   linear_forward(A, W, b): compute Z = np.dot(W,A)+b , return Z and cache=(A,W,b)

4) linear_activation_forward(A_prev, W, b, activation): combine the linear_forward(A,W,b) and signmoid(Z) or relu(Z) functions, and compute

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

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

return $A = A^{[l]}$, and cache $(A^{[l-1]}, W^{[l]}, b^{[l]}, Z^{[l]})$

5) L_model_forward(X, parameters):
Given training data X and parameters, L_model_forward(X,parameters) instantiates linear_activation_forward() L-1 times with activation=”relu” to build the L-1 hidden layers, and instantiate linear_activation_forward() once with activation=”sigmoid) to build the output layer. The function returns the output $A^{[L]}$ of the output layer, and caches all the intermediate results and parameters for all layers, (A_prev, W, b, Z).

6) Cost function
compute_cost(AL, Y): compute the cost

$$J(W,b) = \frac{1}{m}\sum_{i=1}^{m} L\left(\hat{y}^{(i)}, y^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\ln\left(\hat{y}^{(i)}\right) + (1-y^{(i)})\ln\left(1-\hat{y}^{(i)}\right)\right]$$

by

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

Note: the cost function is not a part of the DNN, but it outputs the cost for each iteration, and visualize the behavior of the algorithm.

7) linear_backward(dZ, cache):

$$dW^{[l]} = dZ^{[l]}A^{[l-1]^T} * (\tfrac{1}{m}) \qquad \left(n^{[l]}, n^{[l-1]}\right) = \left(n^{[l]}, m\right) \times \left(n^{[l-1]}, m\right)^T$$

$$db^{[l]} = \left(\tfrac{1}{m}\right) np.sum(dZ^{[l]}, axis = 1, keepdims = True) \qquad \left(n^{[l]}, 1\right) = \left(n^{[l]}, 1\right)$$

$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]} \qquad \left(n^{[l-1]}, 1\right) = \left(n^{[l]}, n^{[l-1]}\right)^T \times \left(n^{[l]}, 1\right)$$

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

8) linear_activation_backward(dA, cache, activation):

$$dZ^{[l]} = dA^{[l]} * g^{[l]'}\left(Z^{[l]}\right) \qquad \left(n^{[l]}, m\right) = \left(n^{[l]}, m\right) * \left(n^{[l]}, m\right)$$

$$dW^{[l]} = dZ^{[l]} A^{[l-1]^T} * (\frac{1}{m}) \qquad (n^{[l]}, n^{[l-1]}) = (n^{[l]}, m) \times (n^{[l-1]}, m)^T$$

$$db^{[l]} = \left(\frac{1}{m}\right) np.sum(dZ^{[l]}, axis = 1, keepdims = True) \qquad (n^{[l]}, 1) = (n^{[l]}, 1)$$

$$dA^{[l-1]} = W^{[l]^T} dZ^{[l]} \qquad (n^{[l-1]}, 1) = (n^{[l]}, n^{[l-1]})^T \times (n^{[l]}, 1)$$

Implement the backward propagation for the LINEAR plus 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

9) L_model_backward(AL, Y, caches): by instantiating linear_activation_backward(dA, cache, activation) L times, implement the backward propagation for the [LINEAR plus RELU] * (L-1) layers and LINEAR plus SIGMOID layer
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)] = ...

10) update_parameters(parameters, grads, learning_rate): Update parameters using gradient descent
Arguments:

parameters -- 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)] = ...

11) predict(X, y, parameters): This function is used to predict the results of a L-layer neural network.
Arguments:
X -- data set of examples you would like to label
parameters -- parameters of the trained model

Returns:
p -- predictions for the given dataset X

12) load_data(): read the images from *.h files into arrays.
13) print_mislabeled_images(classes, X, y, p): Plots images where predictions and truth were different.
X -- dataset y -- true labels p -- predictions

dnn_utils.py

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jun  4 14:26:51 2019
from my assignment notes: dnn_utils.py
@author: weido
"""

import numpy as np
import h5py
import matplotlib.pyplot as plt
#from testCases_v4 import *
#from dnn_utils_v2 import sigmoid, sigmoid_backward, relu, relu_backward

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
    """
```

```python

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache


def relu_backward(dA, Z):
    """
    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
    """

    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, Z):
    """
    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
    """

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

    assert (dZ.shape == Z.shape)

    return dZ


#%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

#%load_ext autoreload
#%autoreload 2

np.random.seed(1)


# FUNCTION: initialize_parameters

def initialize_parameters(n_x, n_h, n_y):
    """
```

```
    Argument:
    n_x -- size of the input layer
    n_h -- size of the hidden layer
    n_y -- size of the output layer

    Returns:
    parameters -- python dictionary containing your parameters:
                    W1 -- weight matrix of shape (n_h, n_x)
                    b1 -- bias vector of shape (n_h, 1)
                    W2 -- weight matrix of shape (n_y, n_h)
                    b2 -- bias vector of shape (n_y, 1)
    """

    np.random.seed(1)

    ### START CODE HERE ### (≈ 4 lines of code)
    W1 = np.random.randn(n_h,n_x)/np.sqrt(n_x)#*0.01
    b1 = np.zeros((n_h,1))
    W2 = np.random.randn(n_y,n_h)/np.sqrt(n_h)#*0.01
    b2 = np.zeros((n_y,1))
    ### END CODE HERE ###

    assert(W1.shape == (n_h, n_x))
    assert(b1.shape == (n_h, 1))
    assert(W2.shape == (n_y, n_h))
    assert(b2.shape == (n_y, 1))

    parameters = {"W1": W1,
                  "b1": b1,
                  "W2": W2,
                  "b2": b2}

    return parameters

# FUNCTION: initialize_parameters_deep

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(1)
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        ### START CODE HERE ### (≈ 2 lines of code)
        parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[l-1])/ np.sqrt(layer_dims[l-1]) #*0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l],1))
        ### END CODE HERE ###

```

```
        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

# FUNCTION: linear_forward

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 dictionary containing "A", "W" and "b" ; stored for computing the backward pass efficiently
    """

    ### START CODE HERE ### (≈ 1 line of code)
    Z = np.dot(W,A)+b
    ### END CODE HERE ###

    assert(Z.shape == (W.shape[0], A.shape[1]))
    cache = (A, W, b)

    return Z, cache

# FUNCTION: linear_activation_forward

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
```

```python
    cache -- a python dictionary containing "linear_cache" and "activation_cache";
             stored for computing the backward pass efficiently
    """

    if activation == "sigmoid":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### (≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)
        #A, activation_cache = A, activation_cache = sigmoid(Z)
        ### END CODE HERE ###

    elif activation == "relu":
        # Inputs: "A_prev, W, b". Outputs: "A, activation_cache".
        ### START CODE HERE ### (≈ 2 lines of code)
        Z, linear_cache = linear_forward(A_prev, W, b)
        #A, activation_cache = A, activation_cache = relu(Z)
        A, activation_cache = relu(Z)
        ### END CODE HERE ###

    assert (A.shape == (W.shape[0], A_prev.shape[1]))
    cache = (linear_cache, activation_cache)

    return A, cache

# FUNCTION: L_model_forward

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

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

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
        A_prev = A
        ### START CODE HERE ### (≈ 2 lines of code)
        A, cache = linear_activation_forward(A_prev,parameters['W'+str(l)],parameters['b'+str(l)], activation = "relu")
        caches.append(cache)
        ### END CODE HERE ###

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    ### START CODE HERE ### (≈ 2 lines of code)
```

```
    # A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation = "sigmoid")
    AL, cache = linear_activation_forward(A,parameters['W'+str(L)],parameters['b'+str(L)], activation = "sigmoid")
    caches.append(cache)
    ### END CODE HERE ###

    assert(AL.shape == (1,X.shape[1]))

    return AL, caches


# FUNCTION: compute_cost

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]

    # Compute loss from aL and y.
    ### START CODE HERE ### (≈ 1 lines of code)
    cost = (-1/m)*(np.dot(np.log(AL), Y.T)+np.dot(np.log(1-AL), (1-Y).T))
    ### END CODE HERE ###

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we expect (e.g. this turns [[17]] into 17).
    assert(cost.shape == ())

    return cost

# FUNCTION: linear_backward

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]

    ### START CODE HERE ### (≈ 3 lines of code)
    dW = (1/m)*np.dot(dZ, A_prev.T)
    db = (1/m)*np.sum(dZ, axis=1, keepdims = True)
    dA_prev = np.dot(W.T, dZ)
    ### END CODE HERE ###

    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)

    return dA_prev, dW, db

# FUNCTION: linear_activation_backward

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":
        ### START CODE HERE ### (≈ 2 lines of code)
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    elif activation == "sigmoid":
        ### START CODE HERE ### (≈ 2 lines of code)
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
        ### END CODE HERE ###

    return dA_prev, dW, db

# FUNCTION: L_model_backward
```

```

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
    ### START CODE HERE ### (1 line of code)
    dAL = -(np.divide(Y,AL)-np.divide(1-Y,1-AL))
    ### END CODE HERE ###

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "dAL, current_cache". Outputs: "grads["dAL-1"], grads["dWL"], grads["dbL"]
    ### START CODE HERE ### (approx. 2 lines)
    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = linear_activation_backward(dAL, current_cache,"sigmoid")
    ### END CODE HERE ###

    # Loop from l=L-2 to l=0
    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        # Inputs: "grads["dA" + str(l + 1)], current_cache". Outputs: "grads["dA" + str(l)] , grads["dW" + str(l + 1)] , grads["db" + str(l + 1)]
        ### START CODE HERE ### (approx. 5 lines)
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["dA"+str(l+1)], current_cache,"relu")
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp
        ### END CODE HERE ###

    return grads

# FUNCTION: update_parameters

def update_parameters(parameters, grads, learning_rate):
```

```
    """
    Update parameters using gradient descent

    Arguments:
    parameters -- 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)] = ...
    """

    L = len(parameters) // 2 # number of layers in the neural network

    # Update rule for each parameter. Use a for loop.
    ### START CODE HERE ### (≈ 3 lines of code)
    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)]
    ### END CODE HERE ###
    return parameters

def load_data():
    train_dataset = h5py.File('train_catvnoncat.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your train set features
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your train set labels

    test_dataset = h5py.File('test_catvnoncat.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test set features
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test set labels

    classes = np.array(test_dataset["list_classes"][:]) # 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

def predict(X, y, parameters):
    """
    This function is used to predict the results of a  L-layer neural network.

    Arguments:
    X -- data set of examples you would like to label
    parameters -- parameters of the trained model

    Returns:
    p -- predictions for the given dataset X
    """

    m = X.shape[1]
```

```
    n = len(parameters) // 2 # number of layers in the neural network
    p = np.zeros((1,m))

    # Forward propagation
    probas, caches = L_model_forward(X, parameters)


    # convert probas to 0/1 predictions
    for i in range(0, probas.shape[1]):
        if probas[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0

    #print results
    #print ("predictions: " + str(p))
    #print ("true labels: " + str(y))
    print("Accuracy: "  + str(np.sum((p == y)/m)))

    return p

def print_mislabeled_images(classes, X, y, p):
    """
    Plots images where predictions and truth were different.
    X -- dataset
    y -- true labels
    p -- predictions
    """
    a = p + y
    mislabeled_indices = np.asarray(np.where(a == 1))
    plt.rcParams['figure.figsize'] = (40.0, 40.0) # set default size of plots
    num_images = len(mislabeled_indices[0])
    for i in range(num_images):
        index = mislabeled_indices[1][i]

        plt.subplot(2, num_images, i + 1)
        plt.imshow(X[:,index].reshape(64,64,3), interpolation='nearest')
        plt.axis('off')
        plt.title("Prediction: " + classes[int(p[0,index])].decode("utf-8") + " \n Class: " + classes[y[0,index]].decode("utf-8"))
```

7.3.2 Package (h5py and pillow) and dataset preparation

Before we develop the python code for the DNN, we need to import the following packages:

- h5py is a common package to interact with a dataset that is stored on an H5 file.
- PIL and scipy are used here to test the DNN model with our own picture at the end.

1) Install h5py

(base) C:\Users\weido>**activate tensorflow_cpu**

(tensorflow_cpu) C:\Users\weido>**conda install h5py**
Collecting package metadata: done
Solving environment: done

Package Plan

environment location: C:\Users\weido\AppData\Local\Continuum\anaconda3\envs\tensorflow_cpu

added / updated specs:
- h5py

```
The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    h5py-2.9.0                 |   py36h5e291fa_0         969 KB
    hdf5-1.10.4                |       h7ebc959_0        19.2 MB
    intel-openmp-2019.4        |              245         1.7 MB
    mkl-2019.4                 |              245       157.5 MB
    mkl_fft-1.0.12             |   py36h14836fe_0         136 KB
    mkl_random-1.0.2           |   py36h343c172_0         318 KB
    numpy-1.16.4               |   py36h19fb1c0_0          49 KB
    numpy-base-1.16.4          |   py36hc3f5095_0         4.1 MB
    openssl-1.1.1c             |       he774522_1         5.7 MB
    pyreadline-2.1             |           py36_1         141 KB
    ------------------------------------------------------------
                                           Total:       189.9 MB

The following NEW packages will be INSTALLED:

  blas               pkgs/main/win-64::blas-1.0-mkl
  h5py               pkgs/main/win-64::h5py-2.9.0-py36h5e291fa_0
  hdf5               pkgs/main/win-64::hdf5-1.10.4-h7ebc959_0
  icc_rt             pkgs/main/win-64::icc_rt-2019.0.0-h0cc432a_1
  intel-openmp       pkgs/main/win-64::intel-openmp-2019.4-245
  mkl                pkgs/main/win-64::mkl-2019.4-245
  mkl_fft            pkgs/main/win-64::mkl_fft-1.0.12-py36h14836fe_0
  mkl_random         pkgs/main/win-64::mkl_random-1.0.2-py36h343c172_0
  numpy              pkgs/main/win-64::numpy-1.16.4-py36h19fb1c0_0
  numpy-base         pkgs/main/win-64::numpy-base-1.16.4-py36hc3f5095_0
  pyreadline         pkgs/main/win-64::pyreadline-2.1-py36_1

The following packages will be UPDATED:

  openssl                              1.1.1b-he774522_1 --> 1.1.1c-he774522_1


Proceed ([y]/n)? y

Downloading and Extracting Packages
h5py-2.9.0           | 969 KB    | ############################################################################ | 100%
mkl_fft-1.0.12       | 136 KB    | ############################################################################ | 100%
numpy-1.16.4         | 49 KB     | ############################################################################ | 100%
mkl-2019.4           | 157.5 MB  | ############################################################################ | 100%
intel-openmp-2019.4  | 1.7 MB    | ############################################################################ | 100%
numpy-base-1.16.4    | 4.1 MB    | ############################################################################ | 100%
openssl-1.1.1c       | 5.7 MB    | ############################################################################ | 100%
mkl_random-1.0.2     | 318 KB    | ############################################################################ | 100%
pyreadline-2.1       | 141 KB    | ############################################################################ | 100%
hdf5-1.10.4          | 19.2 MB   | ############################################################################ | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

2) Install PIL

```
(tensorflow_cpu) C:\Users\weido>conda install pillow
Collecting package metadata: done
Solving environment: done

## Package Plan ##

  environment location: C:\Users\weido\AppData\Local\Continuum\anaconda3\envs\tensorflow_cpu
```

```
  added / updated specs:
    - pillow


The following packages will be downloaded:

    package                    |            build
    ---------------------------|-----------------
    ca-certificates-2019.5.15  |                0         166 KB
    libtiff-4.0.10             |       hb898794_2         1.1 MB
    olefile-0.46               |           py36_0          49 KB
    pillow-6.0.0               |   py36hdc69c19_0         699 KB
    ------------------------------------------------------------
                                           Total:         1.9 MB

The following NEW packages will be INSTALLED:

  freetype           pkgs/main/win-64::freetype-2.9.1-ha9979f8_1
  libtiff            pkgs/main/win-64::libtiff-4.0.10-hb898794_2
  olefile            pkgs/main/win-64::olefile-0.46-py36_0
  pillow             pkgs/main/win-64::pillow-6.0.0-py36hdc69c19_0
  tk                 pkgs/main/win-64::tk-8.6.8-hfa6e2cd_0
  xz                 pkgs/main/win-64::xz-5.2.4-h2fa13f4_4
  zstd               pkgs/main/win-64::zstd-1.3.7-h508b16e_0

The following packages will be UPDATED:
  ca-certificates                               2019.1.23-0 --> 2019.5.15-0
Proceed ([y]/n)? y
Downloading and Extracting Packages
libtiff-4.0.10       | 1.1 MB    | #################################################################################### | 100%
pillow-6.0.0         | 699 KB    | #################################################################################### | 100%
olefile-0.46         | 49 KB     | #################################################################################### | 100%
ca-certificates-2019 | 166 KB    | #################################################################################### | 100%
Preparing transaction: done
Verifying transaction: done
Executing transaction: done
```

3) Download train_catvnoncat.h5 and test_catvnoncat.h5. These two files will be used for DNN training and testing.

7.3.3 Put all together (week4_assign.py)

Top-level file "week4_assign.py" is pasted below and followed by explanations.

```
# -*- coding: utf-8 -*-
"""
Created on Tue Jun  4 15:51:29 2019
week4_assign.py
@author: weido
"""

import time
import numpy as np
import h5py
import matplotlib.pyplot as plt
import scipy
from PIL import Image
from scipy import ndimage
#from dnn_app_utils_v3 import *
```

```
from dnn_utils import *

#%matplotlib inline
plt.rcParams['figure.figsize'] = (5.0, 4.0) # set default size of plots
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'

#%load_ext autoreload
#%autoreload 2

np.random.seed(1)

train_x_orig, train_y, test_x_orig, test_y, classes = load_data()
# Example of a picture
index = 10
plt.imshow(train_x_orig[index])
plt.show()
print ("y = " + str(train_y[0,index]) + ". It's a " + classes[train_y[0,index]].decode("utf-8") +  " picture.")

# Example of a picture
index = 11
plt.imshow(train_x_orig[index])
plt.show()
print ("y = " + str(train_y[0,index]) + ". It's a " + classes[train_y[0,index]].decode("utf-8") +  " picture.")

# Explore your dataset
m_train = train_x_orig.shape[0]
num_px = train_x_orig.shape[1]
m_test = test_x_orig.shape[0]

print ("Number of training examples: " + str(m_train))
print ("Number of testing examples: " + str(m_test))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)")
print ("train_x_orig shape: " + str(train_x_orig.shape))
print ("train_y shape: " + str(train_y.shape))
print ("test_x_orig shape: " + str(test_x_orig.shape))
print ("test_y shape: " + str(test_y.shape))
# Reshape the training and test examples
train_x_flatten = train_x_orig.reshape(train_x_orig.shape[0], -1).T   # The "-1" makes reshape flatten the remaining dimensions
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))

### CONSTANTS DEFINING THE MODEL ####
n_x = 12288     # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)



# GRADED FUNCTION: two_layer_model

```

```python
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):
    """
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

    Arguments:
    X -- input data, of shape (n_x, number of examples)
    Y -- true "label" vector (containing 0 if cat, 1 if non-cat), of shape (1, number of examples)
    layers_dims -- dimensions of the layers (n_x, n_h, n_y)
    num_iterations -- number of iterations of the optimization loop
    learning_rate -- learning rate of the gradient descent update rule
    print_cost -- If set to True, this will print the cost every 100 iterations

    Returns:
    parameters -- a dictionary containing W1, W2, b1, and b2
    """

    np.random.seed(1)
    grads = {}
    costs = []                              # to keep track of the cost
    m = X.shape[1]                           # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions you'd previously implemented
    ### START CODE HERE ### (≈ 1 line of code)
    parameters = initialize_parameters(n_x, n_h, n_y)
    ### END CODE HERE ###

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]

    # Loop (gradient descent)

    for i in range(0, num_iterations):

        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2".
        ### START CODE HERE ### (≈ 2 lines of code)
        A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")
        ### END CODE HERE ###

        # Compute cost
        ### START CODE HERE ### (≈ 1 line of code)
        cost = compute_cost(A2, Y)
        ### END CODE HERE ###

        # Initializing backward propagation
        dA2 = - (np.divide(Y, A2) - np.divide(1 - Y, 1 - A2))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1, dW2, db2; also dA0 (not used), dW1, db1".
        ### START CODE HERE ### (≈ 2 lines of code)
        dA1, dW2, db2 =  linear_activation_backward(dA2, cache2, "sigmoid")
        dA0, dW1, db1 =  linear_activation_backward(dA1, cache1, "relu")
```

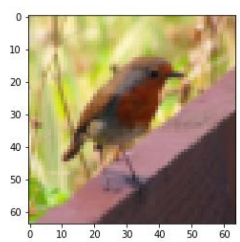
```
        ### END CODE HERE ###

        # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, grads['db2'] to db2
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2

        # Update parameters.
        ### START CODE HERE ### (approx. 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate)
        ### END CODE HERE ###

        # Retrieve W1, b1, W2, b2 from parameters
        W1 = parameters["W1"]
        b1 = parameters["b1"]
        W2 = parameters["W2"]
        b2 = parameters["b2"]

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

    # plot the cost

    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters


parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y), num_iterations = 2500, print_cost=True)

predictions_train = predict(train_x, train_y, parameters)

predictions_test = predict(test_x, test_y, parameters)

### CONSTANTS ###
layers_dims = [12288, 20, 7, 5, 1] #  4-layer model

# GRADED FUNCTION: L_layer_model

def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):#lr was 0.009
    """
    Implements a L-layer neural network: [LINEAR->RELU]*(L-1)->LINEAR->SIGMOID.

    Arguments:
    X -- data, numpy array of shape (number of examples, num_px * num_px * 3)
    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 initialization. (≈ 1 line of code)
    ### START CODE HERE ###
    parameters = initialize_parameters_deep(layers_dims)
    ### END CODE HERE ###

    # Loop (gradient descent)
    for i in range(0, num_iterations):

        # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
        ### START CODE HERE ### (≈ 1 line of code)
        AL, caches = L_model_forward(X, parameters)
        ### END CODE HERE ###

        # Compute cost.
        ### START CODE HERE ### (≈ 1 line of code)
        cost = compute_cost(AL, Y)
        ### END CODE HERE ###

        # Backward propagation.
        ### START CODE HERE ### (≈ 1 line of code)
        grads = L_model_backward(AL, Y, caches)
        ### END CODE HERE ###

        # Update parameters.
        ### START CODE HERE ### (≈ 1 line of code)
        parameters = update_parameters(parameters, grads, learning_rate)
        ### END CODE HERE ###

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

    # plot the cost
    plt.plot(np.squeeze(costs))
    plt.ylabel('cost')
    plt.xlabel('iterations (per tens)')
    plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    return parameters

parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations = 2500, print_cost = True)
```

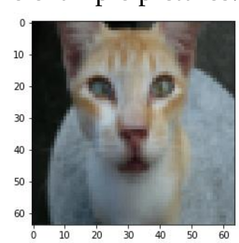
```
240.    pred_train = predict(train_x, train_y, parameters)
241.
242.    #Accuracy: 0.985645933014
243.
244.    pred_test = predict(test_x, test_y, parameters)
245.
246.    #Accuracy: 0.8
247.    print_mislabeled_images(classes, test_x, test_y, pred_test)
248.
249.    import PIL
250.
251.
252.    im=Image.open(r"cat13.jpg")
253.    im = im.resize((64,64), PIL.Image.ANTIALIAS)
254.    temp_file='sized.jpg'
255.    im.save(temp_file)
256.    pic = Image.open(temp_file)
257.    pix = np.array(pic.getdata()).reshape(pic.size[0]*pic.size[1]*3, 1)
258.    pix=pix/255.
259.
260.    my_label=[1]
261.    my_predicted_image = predict(pix, my_label, parameters)
262.
263.    plt.imshow(im)
264.    print ("y = " + str(np.squeeze(my_predicted_image)) + ", your L-
        layer model predicts a \"" + classes[int(np.squeeze(my_predicted_image)),].decod
        e("utf-8") +  "\" picture.")
```

Explanations:

1) Lines 7-16 import all packages.
2) Lines 26-40 load the dataset, and display some example pictures.



`y = 0. It's a non-cat picture.`



`y = 1. It's a cat picture.`

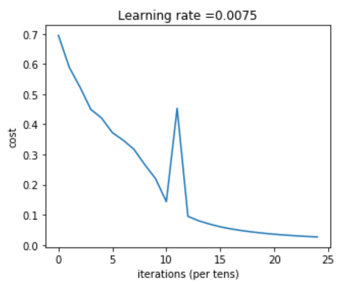
3) Lines 41-63 print the sizes and shapes of the data, and prepare inputs for neural network.

```
Number of training examples: 209
Number of testing examples: 50
Each image is of size: (64, 64, 3)
train_x_orig shape: (209, 64, 64, 3)
train_y shape: (1, 209)
test_x_orig shape: (50, 64, 64, 3)
test_y shape: (1, 50)
train_x's shape: (12288, 209)
test_x's shape: (12288, 50)
```

4) Lines 71-162 define a 2-layer neural network.
5) Lines 164-169 learn the 2-layer nn model, and predict based on training set and test set.

```
Cost after iteration 0: 0.6950464961800915
Cost after iteration 100: 0.5892596054583805
Cost after iteration 200: 0.5232609173622991
Cost after iteration 300: 0.4497686396221906
Cost after iteration 400: 0.42090021618838996
Cost after iteration 500: 0.37246403061745953
```

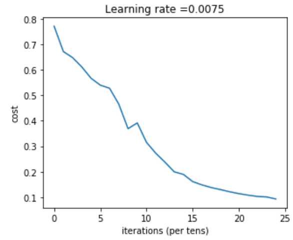
```
Cost after iteration 600: 0.3474205187020189
Cost after iteration 700: 0.31719191987370277
Cost after iteration 800: 0.266437743477466
Cost after iteration 900: 0.21991432807842576
Cost after iteration 1000: 0.1435789889362377
Cost after iteration 1100: 0.4530921262322144
Cost after iteration 1200: 0.09499357670093511
Cost after iteration 1300: 0.08014128076781371
Cost after iteration 1400: 0.0694023400553646
Cost after iteration 1500: 0.06021664023174591
Cost after iteration 1600: 0.05327415758001876
Cost after iteration 1700: 0.047629032620984335
Cost after iteration 1800: 0.04297588879436867
Cost after iteration 1900: 0.039036074365138215
Cost after iteration 2000: 0.03568313638049026
Cost after iteration 2100: 0.03291526373054676
Cost after iteration 2200: 0.03047219305912061
Cost after iteration 2300: 0.028387859212946117
Cost after iteration 2400: 0.02661521237277608
```




```
Accuracy: 0.9999999999999998
Accuracy: 0.74
```

6) Lines 174-236 define a L-layer nn model.
7) Lines 237-248 learn a particular L-layer nn model with L=4, and make predictions.
layers_dims = [12288, 20, 7, 5, 1] # 4-layer model

```
Cost after iteration 0: 0.771749
Cost after iteration 100: 0.672053
Cost after iteration 200: 0.648263
Cost after iteration 300: 0.611507
Cost after iteration 400: 0.567047
Cost after iteration 500: 0.540138
Cost after iteration 600: 0.527930
Cost after iteration 700: 0.465477
Cost after iteration 800: 0.369126
Cost after iteration 900: 0.391747
Cost after iteration 1000: 0.315187
Cost after iteration 1100: 0.272700
Cost after iteration 1200: 0.237419
Cost after iteration 1300: 0.199601
Cost after iteration 1400: 0.189263
Cost after iteration 1500: 0.161189
Cost after iteration 1600: 0.148214
Cost after iteration 1700: 0.137775
Cost after iteration 1800: 0.129740
Cost after iteration 1900: 0.121225
Cost after iteration 2000: 0.113821
Cost after iteration 2100: 0.107839
Cost after iteration 2200: 0.102855
Cost after iteration 2300: 0.100897
Cost after iteration 2400: 0.092878
```




```
Accuracy: 0.9856459330143539
Accuracy: 0.8

Miss-classified images (10 images) in test dataset:
```

8) Lines 248-264 test on an extra jpg image.

7.3.4 Results

In this section, we developed multiple-layer neural networks for image (cat or non-cat) classification. The training set includes 209 images and testing set has 50 images. Each image is `(64, 64, 3)`. A two-layer network with 7 units in the hidden layer achieves an accuracy of 0.74 on testing set, and a four-layer network with 20, 7, and 5 units in the three hidden layers gets an accuracy of 0.8.

Files for the project:

dnn_utils.py: functions
week4_assign.py: main program
train_catvnoncat.h5: training set
test_catvnoncat.h5: testing set

Summary

This chapter introduces ReLu activation function for image feature extraction, which is an important activation function in computer vision. We developed multiple-layer neural networks (any number of layers and any number units in each layer) from the scratch, using basic gradient descent algorithm.

References

[1] "Neural networks and deep learning" online course at www.coursera.org

Exercises

1. Develop a 5-layer neural network to classify the images in the dataset train_catvnoncat.h5 and test_catvnoncat.h5. The number of units in the four hidden layers are 32, 16, 8, 4, respectively. All hidden layers use ReLU activation function and output layer uses sigmoid function for binary classification (cat or non-cat).

   1) Plot the cost function versus iterations.

   2) Print the prediction accuracies on training set and testing set.

2. Try different sizes by assigning different values to *layers_dims* in week4_assign.py. Can you find a value for *layers_dims* such that the neural network outperforms the neural network with *layers_dims=[12288,20,7,5,1]*?

   Note: you can change both the length of *layers_dims* and the values in it.