Week 4 Report

Zhang Heartbright

July 16, 2018

Contents

1 Part 1: Build The Functions

Abstract

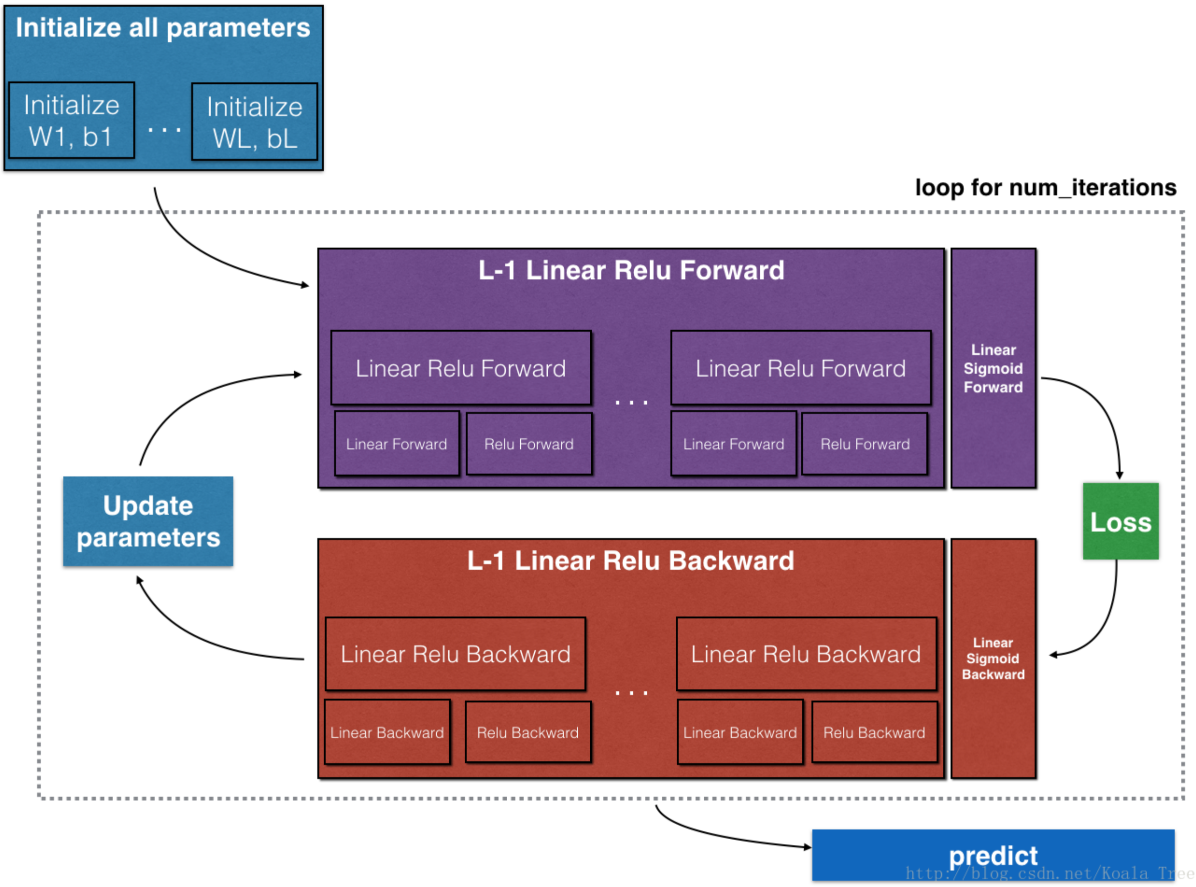
In this section, we will build the functions to be used for the neural network.

1.1 1-Packages

```
import numpy as np
import h5py
import matplotlib.pyplot as plt
from testCases_v3 import *
from dnn_utils_v2 import sigmoid, sigmoid_backward, relu,
                        relu_backward
```

1.2 2-Outline Of Assignment

- Initialize the parameters for a two-layer network or for an L-layer neural network.
- Implement the forward propagation module (shown in purple in the figure below).
- Complete the LINEAR part of a layers forward propagation step (resulting in $Z^{[l]}$)).
- Use the ACTIVATION function (relu/sigmoid) provided.
- Combine the previous two steps into a new [LINEAR→ACTIVATION] forward function.
- Stack the [LINEAR→RELU] forward function L-1 time (for layers 1 through L-1) and add a [LINEAR→SIGMOID] at the end (for the final layer L). This gives you a new L_model_forward function.
- Compute the loss and implement the backward propagation module (denoted in red in the figure below)
- Complete the LINEAR part of a layers backward propagation step.
- ACTIVATE function (relu_backward/sigmoid_backward) is given and combine the previous two steps into a new [LINEAR-ACTIVATION] backward function.
- Stack [LINEAR→RELU] backward L-1 times and add [LINEAR→SIGMOID] backward in a new L_model_backward function



Initialize all parameters
Initialize W1, b1
Initialize WL, bL
loop for num_iterations
L-1 Linear Relu Forward
Linear Relu Forward
Linear Forward
Relu Forward
Linear Relu Forward
Linear Forward
Relu Forward
Linear Sigmoid Forward
Update parameters
Loss
L-1 Linear Relu Backward
Linear Relu Backward
Linear Backward
Relu Backward
Linear Relu Backward
Linear Backward
Relu Backward
Linear Sigmoid Backward
predict
http://blog.csdn.net/Koala_Tree

1.3 3-Initialization

1.3.1 Two Layer Neural Network

- The models structure is: LINEAR → RELU → LINEAR → SIGMOID.
- Use random initialization for the weight matrices. Use np.random.randn(shape)*0.01 with the correct shape.
- Use zero initialization for the biases. Use np.zeros(shape).

```
def initialize_parameters(n_x, n_h, n_y):
##--                                                                  --##
"""
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---###
W1=np.random.randn(n_h,n_x)*0.01
b1=np.zeros((n_h,1))
W2=np.random.randn(n_y,n_h)*0.01
b2=np.zeros((n_y,1))

###---debug---###
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))
###---debug_end---###
parameters={
  "W1":W1,
  "b1":b1,
  "W2":W2,
  "b2":b2
}
return parameters
```

1.3.2 L Layer Neural Network

The initialization for a deeper L-layer neural network is more complicated because there are many more weight matrices and bias vectors.

When computing the initialization, we should make sure that our demensions math in each layer

	Shape of W	**Shape of b**	**Activation**	**Shape of Activation**
Layer 1	$(n^{[1]}, 12288)$	$(n^{[1]}, 1)$	$Z^{[1]} = W^{[1]}X + b^{[1]}$	$(n^{[1]}, 209)$
Layer 2	$(n^{[2]}, n^{[1]})$	$(n^{[2]}, 1)$	$Z^{[2]} = W^{[2]}A^{[1]} + b^{[2]}$	$(n^{[2]}, 209)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
Layer L-1	$(n^{[L-1]}, n^{[L-2]})$	$(n^{[L-1]}, 1)$	$Z^{[L-1]} = W^{[L-1]}A^{[L-2]} + b^{[L-1]}$	$(n^{[L-1]}, 209)$
Layer L	$(n^{[L]}, n^{[L-1]})$	$(n^{[L]}, 1)$	$Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[L]}$	$(n^{[L]}, 209)$

Remember that when we compute $WX + b$ in python, it carries out broadcasting. For example, if:

$$W = \begin{bmatrix} j & k & l \\ m & n & o \\ p & q & r \end{bmatrix} \quad X = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \quad b = \begin{bmatrix} s \\ t \\ u \end{bmatrix} \tag{2}$$

Then $WX + b$ will be:

$$WX + b = \begin{bmatrix} (ja + kd + lg) + s & (jb + ke + lh) + s & (jc + kf + li) + s \\ (ma + nd + og) + t & (mb + ne + oh) + t & (mc + nf + oi) + t \\ (pa + qd + rg) + u & (pb + qe + rh) + u & (pc + qf + ri) + u \end{bmatrix} \tag{3}$$

- Use random initialization for the weight matrices.
- Use zeros initialization for the biases.
- We will store $n^{[l]}$, the number of units in different layers, in a variable layer_dims.
- The following is the implementation for L=1, which can implement the general cases.

```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)

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
```

1.4 4-Forward Propogation

1.4.1 Linear Forward

- LINEAR
- LINEAR $\rightarrow$ ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.
- (LINEAR - RELU) (L-1) $\rightarrow$ LINEAR $\rightarrow$ SIGMOID (whole model)

$Z^{[L]} = W^{[L]} A^{[L-1]}$
where $A^{[0]} = X$

```python
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---###
Z=np.dot(W,A)+b
###---end---###
assert(Z.shape==(W.shape[0],A.shape[1]))
cache=(A,W,b)

return Z,cache
```

1.4.2 Linear-Activation Forward

The functions to be used:

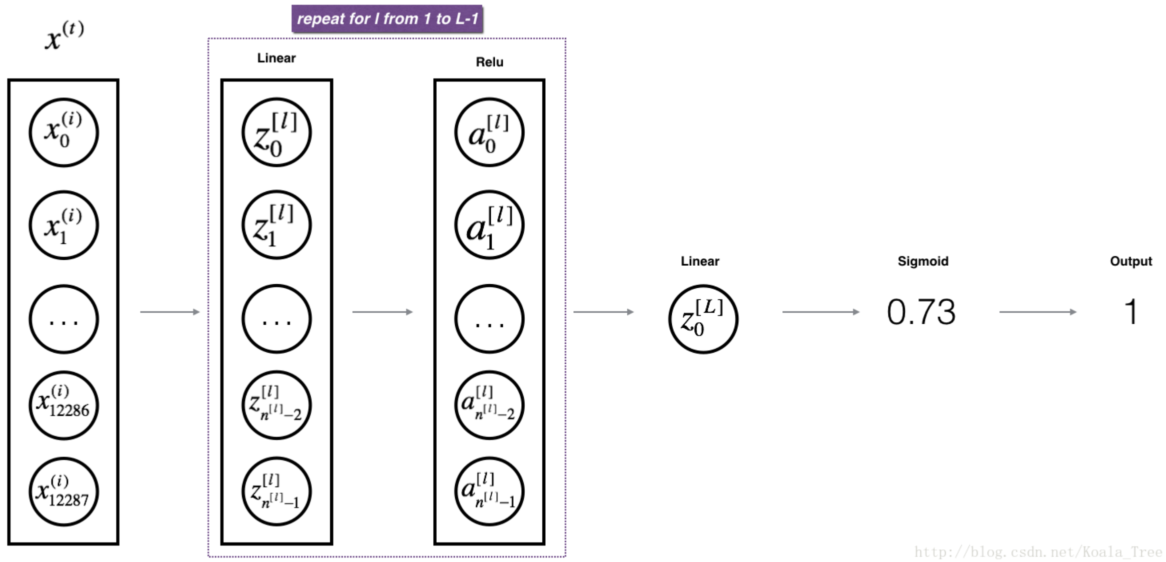
- Sigmoid: $\delta(Z) = \delta(WA + b) = \frac{1}{1+e^{-WA+b}}$
- Relu: $A = RELU(Z) = max(0, Z)$

```
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 dictionary containing "linear_cache" and "activation_cache";
stored for computing the backward pass efficiently

"""
###---start---###
if activation =="sigmoid":#
Z,linear_cache=linear_forward(A_prev,W,b)#linear_cache    A_prev, W, b  Z=WX+b    X   W   b
A,activation_cache=sigmoid(Z)#cache is Z

elif activation =="relu":
Z,linear_cache=linear_forward(A_prev,W,b)
A,activation_cache=relu(Z)
###---end---###
assert(A.shape==(W.shape[0],A_prev.shape[1]))
cache=(linear_cache,activation_cache)
return A,cache
```

1.4.3 L-Layer Model Forward

The function to be used:

- $A^{[L]} = \delta(Z) = \delta(W^{[l]}A^{[L-1]} + b^{[L]})$



```python
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_relu_forward() (there are L-1 of them, indexed from 0 to L-2)
the cache of linear_sigmoid_forward() (there is one, indexed L-1)
"""

caches = []
A = X
L = len(parameters) // 2 # number of layers in the neural network
#                 w   b       parameters        2
# Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
for l in range(1, L):#        L   L  -1
A_prev = A
###---start---###
A,cache=linear_activation_forward(A_prev,W=parameters['W'+str(l)],b=parameters['b'+str(l)],activation="relu")
caches.append(cache)
###---end---###

# Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
###---start---### (    2 lines of code)
AL,cache=linear_activation_forward(A,W=parameters['W'+str(L)],b=parameters['b'+str(L)],activation="sigmoid")
caches.append(cache)
###---end---###

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

return AL, caches
```

1.5 5-Cost Function

The cost function is used to check if the model is really learning

$$-\frac{1}{m}\sum_{i=1}^{m}(y^{(i)}\log\left(a^{[L](i)}\right)+(1-y^{(i)})\log\left(1-a^{[L](i)}\right)) \tag{7}$$

```
    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---### (    1 lines of code)
    cost=-np.sum(np.multiply(np.log(AL),Y)+np.multiply(np.log(1-AL),1-Y))/m
    ###---end---###

    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
```

1.6 6-Backward Propogation

We use a picture to illustrate it:

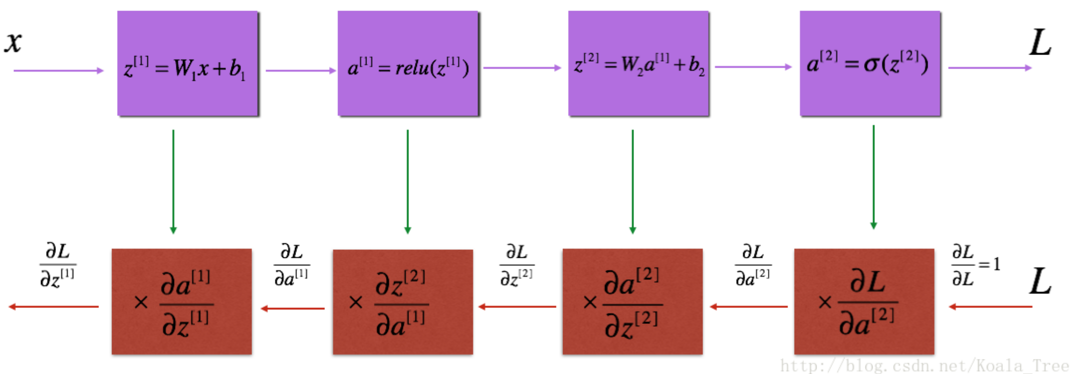


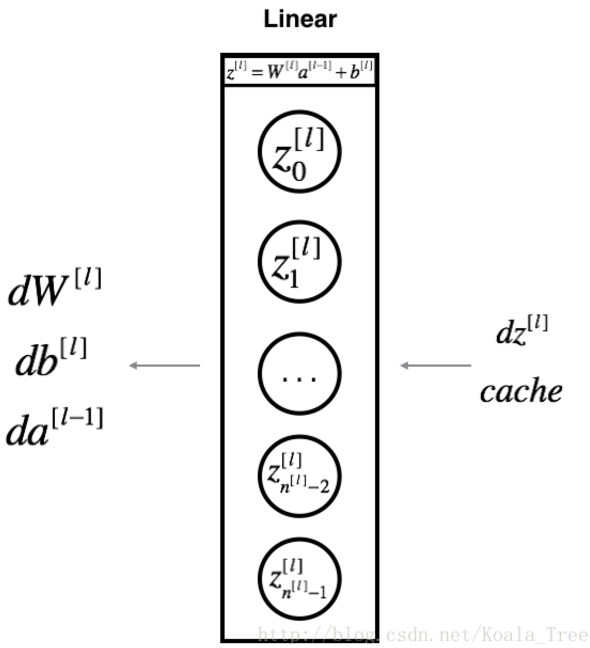
Figure 3 : Forward and Backward propagation for *LINEAR->RELU->LINEAR->SIGMOID*
The purple blocks represent the forward propagation, and the red blocks represent the backward propagation.

$$\frac{d\mathcal{L}(a^{[2]},y)}{dz^{[1]}} = \frac{d\mathcal{L}(a^{[2]},y)}{da^{[2]}}\frac{da^{[2]}}{dz^{[2]}}\frac{dz^{[2]}}{da^{[1]}}\frac{da^{[1]}}{dz^{[1]}} \tag{8}$$

In order to calculate the gradient $dW^{[1]} = \frac{\partial L}{\partial W^{[1]}}$, you use the previous chain rule and you do $dW^{[1]} = dz^{[1]} \times \frac{\partial z^{[1]}}{\partial W^{[1]}}$. During the backpropagation, at each step you multiply your current gradient by the gradient corresponding to the specific layer to get the gradient you wanted. Equivalently, in order to calculate the gradient $db^{[1]} = \frac{\partial L}{\partial b^{[1]}}$, you use the previous chain rule and you do $db^{[1]} = dz^{[1]} \times \frac{\partial z^{[1]}}{\partial b^{[1]}}$. This is why we talk about **backpropagation**. !-->
Now, similar to forward propagation, you are going to build the backward propagation in three steps:
- LINEAR backward
- LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation
- [LINEAR -> RELU] $\times$ (L-1) -> LINEAR -> SIGMOID backward (whole model)

1.6.1 Linear backward

- For layer l, the linear part is: $Z^{[L]} = W^{[L]}A^{[L-1]} + b^{[l]}$



To compute dW,db,da :

$$dW^{[l]} = \frac{\partial \mathcal{L}}{\partial W^{[l]}} = \frac{1}{m} dZ^{[l]} A^{[l-1]T} \tag{8}$$

$$db^{[l]} = \frac{\partial \mathcal{L}}{\partial b^{[l]}} = \frac{1}{m} \sum_{i=1}^{m} dZ^{[l](i)} \tag{9}$$

$$dA^{[l-1]} = \frac{\partial \mathcal{L}}{\partial A^{[l-1]}} = W^{[l]T} dZ^{[l]} \tag{10}$$

```python
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---###
    dW=np.dot(dZ,A_prev.T)/m
    db=np.sum(dZ,axis=1,keepdims=True)# a x i s              ,
                                      k e e p d i m s
    dA_prev=np.dot(W.T,dZ)
    ###---end---###

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

    return dA_prev, dW, db
```

1.6.2 Linear-Activation backward

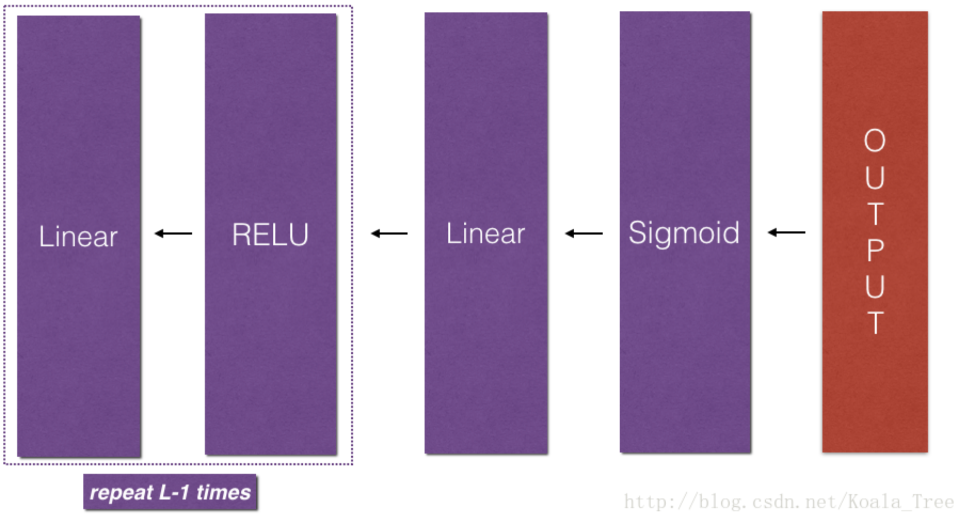
In this section we create a function that merges the two helper functions: linear_backward and the backward step for the activation linear_activation_backward.

And the codes are as followings:

```
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---###
dZ=relu_backward(dA,activation_cache)
dA_prev,dW,db=linear_backward(dZ,linear_cache)
###---end---###

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

return dA_prev, dW, db
```

1.6.3 L-Model Backward

- Now we can implement the backward function for the whole network.
- On each step, we use the cached values for layer l to backpropagate through layer l



```
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
Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

# Initializing the backpropagation
###---start---### (1 line of code)
dAL=-Y/AL +(1-Y)/(1-AL)
###---end---###

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

for l in reversed(range(L-1)):
# lth layer: (RELU -> LINEAR) gradients.
# Inputs: "grads["dA" + str(l + 2)], caches".
# Outputs: "grads["dA" + str(l + 1)] , grads["dW" + str(l + 1)] ,
                                   grads["db" + str(l + 1)]

###---start---### (approx. 5 lines)
print(l)
current_cache=caches[l]
dA_prev_t,dW_t,db_t=linear_activation_backward(dA=grads["dA"+str(l+2)]
                                   ,cache=current_cache,activation="relu"
                                   )
grads["dA"+str(l+1)]=dA_prev_t
grads["dW"+str(l+1)]=dW_t
grads["db"+str(l+1)]=db_t
###---end---###

return grads
```

1.6.4 Update Parameters

In this section we use grad-decents to update the parameters.
Functions are as follows:

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

2 Part 2: Build A Two Layer Neural Network

Abstract

This time, we can use the models above to make a 2-layer neural network


```
#this is a 2-layer network
# And we use the codes we have done before
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_v2 import *
"""                    """
np.random.seed(2)
train_x_orign,train_y,test_x_orign,test_y,classes=load_data()
#print("train_x_orign shape: "+ str(train_x_orign.shape))
train_x_flatten=train_x_orign.reshape(train_x_orign.shape[0],-1).T
#after flatten:(12288,209)
test_x_flatten=test_x_orign.reshape(test_x_orign.shape[0],-1).T
#after flatten:(12288,209)
###---standarlize---###
train_x=train_x_flatten/255.
test_x=test_x_flatten/255.

###---                          ---####
n_x = 12288 # num_px * num_px * 3
n_h = 7     #           7
n_y = 1     #
layers_dims = (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
"""

#############################
###- - - - - - - - - - - -###
###- - B I G S T A R T - -###
###- - - - - - - - - - - -###
#############################
```

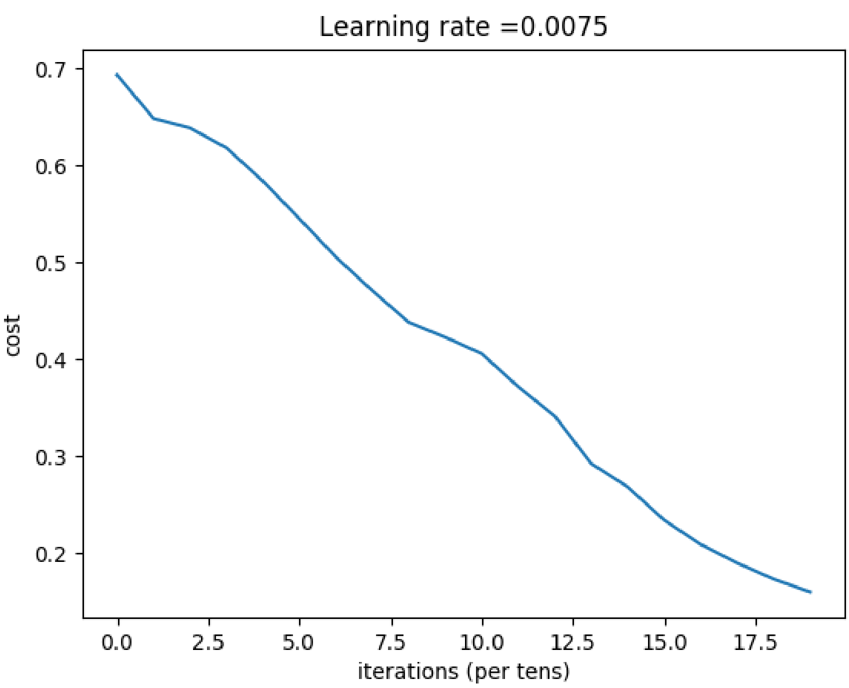
```
###--- 2                ---###
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):
"""
Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGMOID.

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---### (     1 line of code)
parameters = initialize_parameters(n_x, n_h, n_y)
###---end---###

# 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". Output: "A1, cache1, A2, cache2".
###---start---### (     2 lines of code)
# A1, cache1 = linear_activation_forward(X, W1, b1, activation="relu")
# A2, cache2 = linear_activation_forward(A1, W2, b2, activation="sigmoid")
A1,cache1=linear_activation_forward(A_prev=X,W=W1,b=b1,activation="relu")
A2,cache2=linear_activation_forward(A_prev=A1,W=W2,b=b2,activation="sigmoid")
###---end---###

#
###---start---### (     1 line of code)
#cost = compute_cost(A2, Y)
```

```
cost=compute_cost(AL=A2,Y=Y)
###---end---###

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

# Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1,
                                dW2, db2; also dA0 (not used), dW1,
                                db1".
#
###---start---### (    2 lines of code)
# dA1, dW2, db2 = linear_activation_backward(dA2, cache2, activation="
                                sigmoid")
# dA0, dW1, db1 = linear_activation_backward(dA1, cache1, activation="
                                relu")
dA1,dW2,db2=linear_activation_backward(dA=dA2,cache=cache2,activation=
                                "sigmoid")
dA0,dW1,db1=linear_activation_backward(dA=dA1,cache=cache1,activation=
                                "relu")
###---end---###

# 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---### (approx. 1 line of code)
#parameters = update_parameters(parameters, grads, learning_rate)
parameters=update_parameters(parameters,grads,learning_rate)
###---end---###

# 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)
    parameters=two_layer_model(train_x,train_y,learning_rate=0.0075,
                                    layers_dims=(n_x,n_h,n_y),
                                    num_iterations=2000,print_cost=True)
    predictions_train = predict(train_x, train_y, parameters)
    predictions_test = predict(test_x, test_y, parameters)
```

part of the results:

```
    Cost after iteration 0: 0.693049735659989
    Cost after iteration 100: 0.6482760951508908
    Cost after iteration 200: 0.638429336340072
    Cost after iteration 300: 0.6178888768868509
    Cost after iteration 400: 0.5836516666225391
    Cost after iteration 500: 0.5449045395510413
    Cost after iteration 600: 0.5054427467790368
    Cost after iteration 700: 0.47074581506515784
    Cost after iteration 800: 0.4381328874628742
    Cost after iteration 900: 0.42283406420844005
    Cost after iteration 1000: 0.405879611011641
    Cost after iteration 1100: 0.3717984866548225
    Cost after iteration 1200: 0.3416123497167632
    Cost after iteration 1300: 0.2926623715470592
    Cost after iteration 1400: 0.2686173406673315
    Cost after iteration 1500: 0.2344362805323333
    Cost after iteration 1600: 0.20941723048656552
    Cost after iteration 1700: 0.19027892302796698
    Cost after iteration 1800: 0.1736818636718253
    Cost after iteration 1900: 0.160404331880999
```



the analysis:

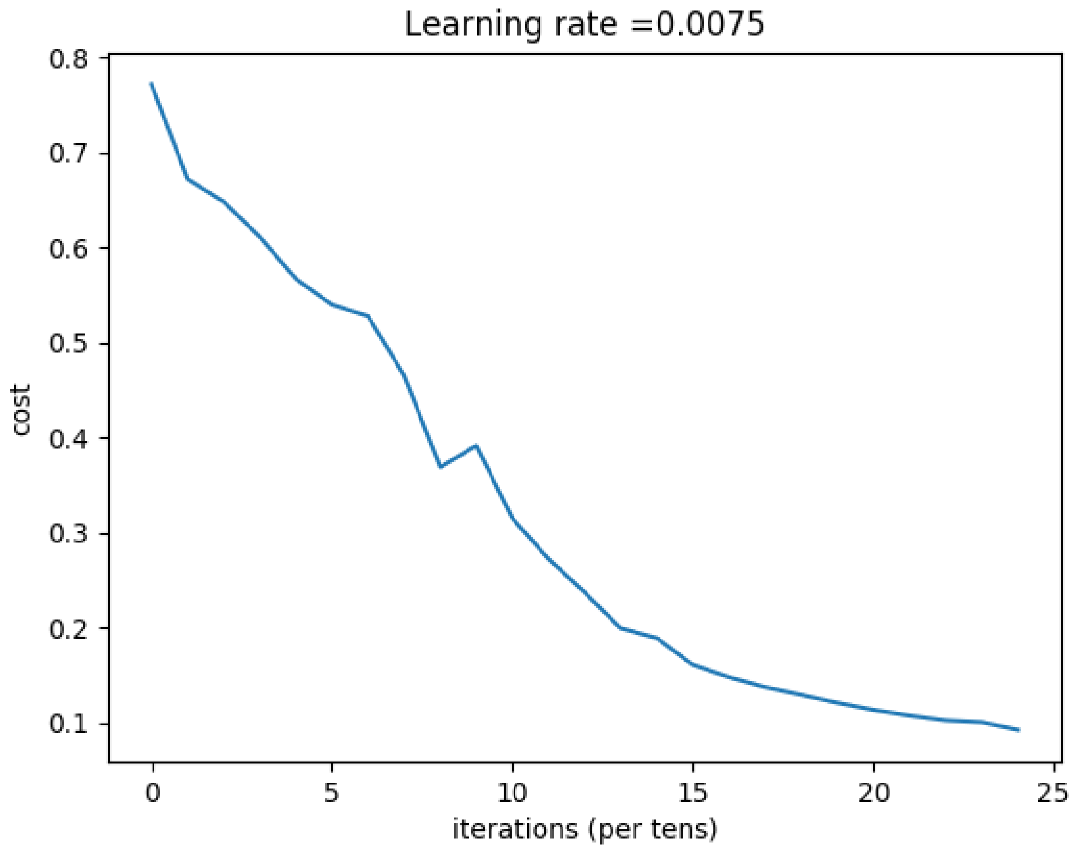
3 Part 3: Build A L-layer Neural Network

No explain, just similiar as before.

```
#this is a 5-layer network
# And we use the codes we have done before
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_v2 import *

np.random.seed(1)

train_x_orign,train_y,test_x_orign,test_y,classes=load_data()

train_x_flatten=train_x_orign.reshape(train_x_orign.shape[0],-1).T
#print(format(train_x_flatten.shape))# (12288,209)
test_x_flatten=test_x_orign.reshape(test_x_orign.shape[0],-1).T
#print(format(test_x_flatten.shape))#(12288,50)
###---standarlize---###
train_x=train_x_flatten/255
test_x=test_x_flatten/255

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

#############################
###- - - - - - - - - - - -###
###- - B I G S T A R T - -###
###- - - - - - - - - - - -###
#############################

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
```

```python
    # Parameters initialization.
    ###---start---###
    # parameters = initialize_parameters_deep(layers_dims)
    parameters=initialize_parameters_deep(layers_dims)
    ###---end---###

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

    # Forward propagation: [LINEAR -> RELU]*(L-1) -> LINEAR -> SIGMOID.
    ###---start---### (    1 line of code)
    # AL, caches = L_model_forward(X, parameters)
    A_last,caches=L_model_forward(X=X,parameters=parameters)
    ###---end---###

    # Compute cost.
    ###---start---### (    1 line of code)
    # cost = compute_cost(AL, Y)
    cost=compute_cost(AL=A_last,Y=Y)
    ###---end---###

    # Backward propagation.
    ###---start---### (    1 line of code)
    # grads = L_model_backward(AL, Y, caches)
    grads=L_model_backward(AL=A_last,Y=Y,caches=caches)
    ###---end---###

    # Update parameters.
    ###---start---### (    1 line of code)
    # parameters = update_parameters(parameters, grads, learning_rate)
    parameters=update_parameters(parameters=parameters,learning_rate=
                                  learning_rate,grads=grads)
    ###---end---###

    # 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=L_layer_model(train_x,train_y,layers_dims,learning_rate=0.
                              0075,num_iterations=2500,print_cost=
                              True)

    predict(train_x,train_y,parameters)
    predict(test_x,test_y,parameters)
```

result analysis:


Learning rate =0.0075
cost
iterations (per tens)
0.8
0.7
0.6
0.5
0.4
0.3
0.2
0.1
0
5
10
15
20
25