hw2

April 2, 2020

1 Basic Instructions

1. Enter your Name and UID in the provided space.
2. Do the assignment in the notebook itself
3. you are free to use Google Colab

Name: Aditya Khopkar
UID: 116911627

In the first part, you will implement all the functions required to build a two layer neural network. In the next part, you will use these functions for image and text classification. Provide your code at the appropriate placeholders.

1.1 1. Packages

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

1.2 2. Layer Initialization

Exercise: Create and initialize the parameters of the 2-layer neural network. Use random initialization for the weight matrices and zero initialization for the biases.

[ ]:
```
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)*0.01
    b1 = np.zeros((n_h,1))
    W2 = np.random.randn(n_y,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
```

[ ]:
```
parameters = initialize_parameters(3,2,1)
print("W1 = " + str(parameters["W1"]))
print("b1 = " + str(parameters["b1"]))
print("W2 = " + str(parameters["W2"]))
print("b2 = " + str(parameters["b2"]))
```

```
W1 = [[ 0.01624345 -0.00611756 -0.00528172]
 [-0.01072969  0.00865408 -0.02301539]]
b1 = [[0.]
 [0.]]
W2 = [[ 0.01744812 -0.00761207]]
b2 = [[0.]]
```

Expected output:

W1

[[0.01624345 -0.00611756 -0.00528172] [-0.01072969 0.00865408 -0.02301539]]

b1

[[0.] [0.]]

W2

[[0.01744812 -0.00761207]]

b2

[[0.]]

1.3 3. Forward Propagation

Now that you have initialized your parameters, you will do the forward propagation module. You will start by implementing some basic functions that you will use later when implementing the model. You will complete three functions in this order:

- LINEAR
- LINEAR -> ACTIVATION where ACTIVATION will be either ReLU or Sigmoid.

The linear module computes the following equation:

$$Z = WA + b \tag{4}$$

1.3.1 3.1 Exercise: 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 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
```

[ ]:
```python
np.random.seed(1)

A = np.random.randn(3,2)
W = np.random.randn(1,3)
b = np.random.randn(1,1)

Z, linear_cache = linear_forward(A, W, b)
print("Z = " + str(Z))
```

```
Z = [[ 3.26295337 -1.23429987]]
```

Expected output:

Z

[[3.26295337 -1.23429987]]

1.3.2 3.2 - Linear-Activation Forward

In this notebook, you will use two activation functions:

- **Sigmoid**: $\sigma(Z) = \sigma(WA + b) = \frac{1}{1+e^{-(WA+b)}}$. Write the code for the `sigmoid` function. This function returns **two** items: the activation value "`a`" and a "`cache`" that contains "`Z`" (it's what we will feed in to the corresponding backward function). To use it you could just call:

```
A, activation_cache = sigmoid(Z)
```

- **ReLU**: The mathematical formula for ReLu is $A = RELU(Z) = max(0, Z)$. Write the code for the `relu` function. This function returns **two** items: the activation value "`A`" and a "`cache`" that contains "`Z`" (it's what we will feed in to the corresponding backward function). To use it you could just call: "' python A, activation_cache = relu(Z)

Exercise: - Implement the activation functions - Build the linear activation part of forward propagation. Mathematical relation is: $A = g(Z) = g(WA_{prev} + b)$

[ ]:
```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, useful during backpropagation
    """
    ### START CODE HERE ### ( 2 line of code)
    A = 1/(1+np.exp(-Z))
    cache = Z

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

```
    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 --  returns Z, useful during backpropagation
    """

    ### START CODE HERE ### ( 2 line of code)
    A = np.maximum(0,Z)
    cache = Z

    ### END CODE HERE ###

    assert(A.shape == Z.shape)
    return A, cache
```

[ ]:
```
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
    """

    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)
        #print('Z',Z)
        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 = relu(Z)

        ### END CODE HERE ###

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

[ ]:
```
np.random.seed(2)
A_prev = np.random.randn(3,2)
W = np.random.randn(1,3)
b = np.random.randn(1,1)

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation␣
 ↪= "sigmoid")
print("With sigmoid: A = " + str(A))
A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activation␣
 ↪= "relu")
print("With ReLU: A = " + str(A))
```

```
With sigmoid: A = [[0.96890023 0.11013289]]
With ReLU: A = [[3.43896131 0.        ]]
```

Expected output:

With sigmoid: A

[[0.96890023 0.11013289]]

With ReLU: A

[[3.43896131 0.]]

1.4 4 - Loss function

Now you will implement forward and backward propagation. You need to compute the loss, because you want to check if your model is actually learning.

Exercise: Compute the cross-entropy loss J, using the following formula:

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

[ ]:
```
# GRADED FUNCTION: compute_loss

def compute_loss(A, Y):
    """
    Implement the loss function defined by equation (7).

    Arguments:
    A -- 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:
    loss -- cross-entropy loss
    """

    m = Y.shape[1]

    # Compute loss from aL and y.
    ### START CODE HERE ### ( 1 lines of code)
    loss = (-1/m)*np.sum((Y*np.log(A)) + (1-Y)*(np.log(1-A)))
    ### END CODE HERE ###

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

    return loss
```

[ ]:
```
Y = np.asarray([[1, 1, 1]])
A = np.array([[.8,.9,0.4]])

print("loss = " + str(compute_loss(A, Y)))
```

```
loss = 0.41493159961539694
```

Expected Output:

loss

0.41493159961539694

1.5 5 - Backward propagation module

Just like with forward propagation, you will implement helper functions for backpropagation. Remember that back propagation is used to calculate the gradient of the loss function with respect to the parameters.

Now, similar to forward propagation, you are going to build the backward propagation in two steps: - LINEAR backward - LINEAR -> ACTIVATION backward where ACTIVATION computes the derivative of either the ReLU or sigmoid activation

1.5.1 5.1 - Linear backward

[ ]:
```
# GRADED 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 loss 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 loss with respect to the activation (of the␣
↪previous layer l-1), same shape as A_prev
    dW -- Gradient of the loss with respect to W (current layer l), same shape␣
↪as W
    db -- Gradient of the loss 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)
    dA_prev = np.dot(W.T,dZ)
    dW = np.dot(dZ,A_prev.T)
    db = np.sum(dZ,axis=1,keepdims=True)
    ### 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
```

[ ]:
```
np.random.seed(1)
dZ = np.random.randn(1,2)
A = np.random.randn(3,2)
W = np.random.randn(1,3)
b = np.random.randn(1,1)
linear_cache = (A, W, b)

dA_prev, dW, db = linear_backward(dZ, linear_cache)
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

```
dA_prev = [[ 0.51822968 -0.19517421]
 [-0.40506361  0.15255393]
 [ 2.37496825 -0.89445391]]
dW = [[-0.2015379   2.81370193  3.2998501 ]]
db = [[1.01258895]]
```

Expected Output:

dA_prev

[[0.51822968 -0.19517421] [-0.40506361 0.15255393] [2.37496825 -0.89445391]]

dW

[[-0.2015379 2.81370193 3.2998501]]

db

[[1.01258895]]

1.5.2 5.2 - Linear Activation backward

Next, you will create a function that merges the two helper functions: `linear_backward` and the backward step for the activation `linear_activation_backward`.

Before implementing `linear_activation_backward`, you need to implement two backward functions for each activations: - `sigmoid_backward`: Implements the backward propagation for SIGMOID unit. You can call it as follows:

```
dZ = sigmoid_backward(dA, activation_cache)
```

- `relu_backward`: Implements the backward propagation for RELU unit. You can call it as follows:

```
dZ = relu_backward(dA, activation_cache)
```

If $g(.)$ is the activation function, `sigmoid_backward` and `relu_backward` compute

$$dZ^{[l]} = dA^{[l]} * g'(Z^{[l]}) \tag{11}$$

.

Exercise: - Implement the backward functions for the relu and sigmoid activation layer. - Implement the backpropagation for the *LINEAR->ACTIVATION* layer.

[ ]:
```
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 loss with respect to Z
    """

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

    ### START CODE HERE ### ( 1 line of code)
    dZ[Z<=0] = 0
    ### END CODE HERE ###

    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 loss with respect to Z
    """

    Z = cache

    ### START CODE HERE ### ( 2 line of code)
    sig = 1/(1+np.exp(-Z))
    dZ = dA * sig[0]*(1-sig[0])
    ### END CODE HERE ###

    assert (dZ.shape == Z.shape)
```

```
    return dZ
```

[ ]:
```
# GRADED 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 loss with respect to the activation (of the␣
 ↪previous layer l-1), same shape as A_prev
    dW -- Gradient of the loss with respect to W (current layer l), same shape␣
 ↪as W
    db -- Gradient of the loss 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
```

[ ]:
```
np.random.seed(2)
dA = np.random.randn(1,2)
A = np.random.randn(3,2)
W = np.random.randn(1,3)
b = np.random.randn(1,1)
```

```
Z = np.random.randn(1,2)
linear_cache = (A, W, b)
activation_cache = Z
linear_activation_cache = (linear_cache, activation_cache)

dA_prev, dW, db = linear_activation_backward(dA, linear_activation_cache,␣
 ↪activation = "sigmoid")
print ("sigmoid:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db) + "\n")

dA_prev, dW, db = linear_activation_backward(dA, linear_activation_cache,␣
 ↪activation = "relu")
print ("relu:")
print ("dA_prev = "+ str(dA_prev))
print ("dW = " + str(dW))
print ("db = " + str(db))
```

```
sigmoid:
dA_prev = [[ 0.11017994  0.01105339]
 [ 0.09466817  0.00949723]
 [-0.05743092 -0.00576154]]
dW = [[ 0.20533573  0.19557101 -0.03936168]]
db = [[-0.11459244]]

relu:
dA_prev = [[ 0.44090989  0.        ]
 [ 0.37883606  0.        ]
 [-0.2298228   0.        ]]
dW = [[ 0.89027649  0.74742835 -0.20957978]]
db = [[-0.41675785]]
```

Expected output with sigmoid:

dA_prev

    <td >[[0.11017994 0.01105339]

[0.09466817 0.00949723] [-0.05743092 -0.00576154]]

dW

    <td > [[0.20533573 0.19557101 -0.03936168]] </td>

db

    <td > [[-0.11459244]] </td>

Expected output with relu:

dA_prev

<td > [[0.44090989 0.]

[0.37883606 0.] [-0.2298228 0.]]

dW

<td > [[0.89027649 0.74742835 -0.20957978]] </td>

db

<td > [[-0.41675785]] </td>

1.5.3 6 - Update Parameters

In this section you will update the parameters of the model, using gradient descent:

$$W^{[1]} = W^{[1]} - \alpha \ dW^{[1]} \tag{16}$$
$$b^{[1]} = b^{[1]} - \alpha \ db^{[1]} \tag{17}$$
$$W^{[2]} = W^{[2]} - \alpha \ dW^{[2} \tag{16}$$
$$b^{[2]} = b^{[2]} - \alpha \ db^{[2]} \tag{17}$$

where α is the learning rate. After computing the updated parameters, store them in the parameters dictionary.

Exercise: Implement `update_parameters()` to update your parameters using gradient descent.

Instructions: Update parameters using gradient descent.

```
[ ]: # GRADED 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)] = ...
    """
    # Update rule for each parameter. Use a for loop.
    ### START CODE HERE ### ( 4 lines of code)
    for key in parameters:
        parameters[key] = parameters[key] - (learning_rate * grads["d" +␣
 ↪str(key)])
```

```
    ### END CODE HERE ###
    return parameters
```

[ ]:
```
np.random.seed(2)
W1 = np.random.randn(3,4)
b1 = np.random.randn(3,1)
W2 = np.random.randn(1,3)
b2 = np.random.randn(1,1)
parameters = {"W1": W1,
              "b1": b1,
              "W2": W2,
              "b2": b2}
np.random.seed(3)
dW1 = np.random.randn(3,4)
db1 = np.random.randn(3,1)
dW2 = np.random.randn(1,3)
db2 = np.random.randn(1,1)
grads = {"dW1": dW1,
         "db1": db1,
         "dW2": dW2,
         "db2": db2}
parameters = update_parameters(parameters, grads, 0.1)

print ("W1 = "+ str(parameters["W1"]))
print ("b1 = "+ str(parameters["b1"]))
print ("W2 = "+ str(parameters["W2"]))
print ("b2 = "+ str(parameters["b2"]))
```

```
W1 = [[-0.59562069 -0.09991781 -2.14584584  1.82662008]
 [-1.76569676 -0.80627147  0.51115557 -1.18258802]
 [-1.0535704  -0.86128581  0.68284052  2.20374577]]
b1 = [[-0.04659241]
 [-1.28888275]
 [ 0.53405496]]
W2 = [[-0.55569196  0.0354055   1.32964895]]
b2 = [[-0.84610769]]
```

Expected Output:

W1

```
<td > [[-0.59562069 -0.09991781 -2.14584584  1.82662008]
```

[-1.76569676 -0.80627147 0.51115557 -1.18258802] [-1.0535704 -0.86128581 0.68284052 2.20374577]]

b1

```
<td > [[-0.04659241]
```

[-1.28888275] [0.53405496]]

W2

<td > [[-0.55569196 0.0354055   1.32964895]]</td>

b2

<td > [[-0.84610769]] </td>

1.6 7 - Conclusion

Congrats on implementing all the functions required for building a deep neural network!

We know it was a long assignment but going forward it will only get better. The next part of the assignment is easier.

2 Part 2:

In the next part you will put all these together to build a two-layer neural networks for image classification.

[55]:
```
%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)
```

```
The autoreload extension is already loaded. To reload it, use:
  %reload_ext autoreload
```

3 Dataset

Problem Statement: You are given a dataset ("data/train_catvnoncat.h5", "data/test_catvnoncat.h5") containing: - a training set of m_train images labelled as cat (1) or non-cat (0) - a test set of m_test images labelled as cat and non-cat - each image is of shape (num_px, num_px, 3) where 3 is for the 3 channels (RGB).

Let's get more familiar with the dataset. Load the data by completing the function and run the cell below.

[56]:
```
def load_data(train_file, test_file):
    # Load the training data
    train_dataset = h5py.File(train_file,'r')

    # Separate features(x) and labels(y) for training set
    train_set_x_orig = np.array(train_dataset["train_set_x"])
    train_set_y_orig = np.array(train_dataset["train_set_y"])
```

```
    # Load the test data
    test_dataset = h5py.File(test_file,'r')

    # Separate features(x) and labels(y) for training set
    test_set_x_orig = np.array(test_dataset["test_set_x"])
    test_set_y_orig = np.array(test_dataset["test_set_y"])

    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
```

[57]:
```
train_file="train_catvnoncat.h5"
test_file="test_catvnoncat.h5"
train_x_orig, train_y, test_x_orig, test_y, classes = load_data(train_file,␣
↪test_file)
```

The following code will show you an image in the dataset. Feel free to change the index and re-run the cell multiple times to see other images.

[58]:
```
# Example of a picture
index = 10
plt.imshow(train_x_orig[index])
print ("y = " + str(train_y[0,index]) + ". It's a " + classes[train_y[0,index]].
↪decode("utf-8") +  " picture.")
```

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

output_46_1.png

[59]:
```
# 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))
```

```
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)
```

As usual, you reshape and standardize the images before feeding them to the network.

Figure 1: Image to vector conversion.

[60]:
```
# 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))
```

```
train_x's shape: (12288, 209)
test_x's shape: (12288, 50)
```

3.1 3 - Architecture of your model

Now that you are familiar with the dataset, it is time to build a deep neural network to distinguish cat images from non-cat images.

3.1.1 2-layer neural network

Figure 2: 2-layer neural network. The model can be summarized as: ***INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT***.

Detailed Architecture of figure 2: - 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 $W^{[1]}$ of size $(n^{[1]},12288)$. - You then add a bias term and take its relu to get the following vector: $[a_0^{[1]},a_1^{[1]},...,a_{n^{[1]}-1}^{[1]}]^T$. - You multiply the resulting vector by $W^{[2]}$ and add your intercept (bias). - Finally, you take the sigmoid of the result. If it is greater than 0.5, you classify it to be a cat.

3.1.2 General methodology

As usual you will follow the Deep Learning methodology to build the model: 1. Initialize parameters / Define hyperparameters 2. Loop for num_iterations: a. Forward propagation b. Compute loss function c. Backward propagation d. Update parameters (using parameters, and grads from backprop) 4. Use trained parameters to predict labels

Let's now implement those the model!

Question: Use the helper functions you have implemented in the previous assignment to build a 2-layer neural network with the following structure: *LINEAR -> RELU -> LINEAR -> SIGMOID*. The functions you may need and their inputs are:

```
def initialize_parameters(n_x, n_h, n_y):
    ...
    return parameters
def linear_activation_forward(A_prev, W, b, activation):
    ...
    return A, cache
def compute_loss(AL, Y):
    ...
    return loss
def linear_activation_backward(dA, cache, activation):
    ...
    return dA_prev, dW, db
def update_parameters(parameters, grads, learning_rate):
    ...
    return parameters
```

[61]:
```
### 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)
```

[62]:
```
def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations =␣
 ↪3000, print_loss=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_loss -- If set to True, this will print the loss every 100 iterations
```

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

    np.random.seed(1)
    grads = {}
    losses = []                              # to keep track of the loss
    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")
       # print('cache2',cache2)
        ### END CODE HERE ###

        # Compute loss
        ### START CODE HERE ### ( 1 line of code)
        loss = compute_loss(A2,Y)
        ### END CODE HERE ###

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

        # 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
        ### START CODE HERE ### ( 4 lines of code)
        grads['dW1'] = dW1
        grads['dW2'] = dW2
        grads['db1'] = db1
        grads['db2'] = db2

        ### END CODE HERE ###


        # 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 loss every 100 training example
        if print_loss and i % 100 == 0:
            print("Loss after iteration {}: {}".format(i, np.squeeze(loss)))
        if print_loss and i % 100 == 0:
            losses.append(loss)

    # plot the loss

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

    return parameters
```

[63]:
```
parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y),␣
↪learning_rate = 0.025, num_iterations = 10000, print_loss=True)
```

```
Loss after iteration 0: 0.693049735659989
Loss after iteration 100: 0.5896239828131958
Loss after iteration 200: 0.5104045081853634
```

```
Loss after iteration 300: 0.44010076134072124
Loss after iteration 400: 0.4142307053862242
Loss after iteration 500: 0.4144198776785578
Loss after iteration 600: 0.3631206146390945
Loss after iteration 700: 0.3287496600095329
Loss after iteration 800: 0.19231482814248474
Loss after iteration 900: 0.20528038966389023
Loss after iteration 1000: 0.08604547767959055
Loss after iteration 1100: 0.15233564932198757
Loss after iteration 1200: 0.05610234090379547
Loss after iteration 1300: 0.34137890569008583
Loss after iteration 1400: 0.10852166706188232
Loss after iteration 1500: 0.059543035153596595
Loss after iteration 1600: 0.04382966432663086
Loss after iteration 1700: 0.03310824486274162
Loss after iteration 1800: 0.02685958028787537
Loss after iteration 1900: 0.022642878640190975
Loss after iteration 2000: 0.019708692804953486
Loss after iteration 2100: 0.017547557526126502
Loss after iteration 2200: 0.015874771004955466
Loss after iteration 2300: 0.014585890568333934
Loss after iteration 2400: 0.01356583433840927
Loss after iteration 2500: 0.012705736529662205
Loss after iteration 2600: 0.012005706568514062
Loss after iteration 2700: 0.011416987059154726
Loss after iteration 2800: 0.010912738305409897
Loss after iteration 2900: 0.010472049587622384
Loss after iteration 3000: 0.010089230465936816
Loss after iteration 3100: 0.009746963924199412
Loss after iteration 3200: 0.009445560522553732
Loss after iteration 3300: 0.009172776329559077
Loss after iteration 3400: 0.008926997835135496
Loss after iteration 3500: 0.008701018367307333
Loss after iteration 3600: 0.008493345866759915
Loss after iteration 3700: 0.008303785821994984
Loss after iteration 3800: 0.00812842140242095l
Loss after iteration 3900: 0.00796283333605266
Loss after iteration 4000: 0.007809302919607043
Loss after iteration 4100: 0.007667371197590984
Loss after iteration 4200: 0.007532298054554675
Loss after iteration 4300: 0.007403939681548437
Loss after iteration 4400: 0.007284247656322006
Loss after iteration 4500: 0.007168731353928581
Loss after iteration 4600: 0.007061439845461443
Loss after iteration 4700: 0.0069599292667934425
Loss after iteration 4800: 0.0068570433865402
Loss after iteration 4900: 0.006762639606378089
Loss after iteration 5000: 0.00667080934783059
```

```
Loss after iteration 5100: 0.0065914661823120784
Loss after iteration 5200: 0.006499658710387131
Loss after iteration 5300: 0.006417934134241922
Loss after iteration 5400: 0.006339871992446925
Loss after iteration 5500: 0.006262867361938575
Loss after iteration 5600: 0.006189910328303968
Loss after iteration 5700: 0.006121032558817848
Loss after iteration 5800: 0.006049629750466431
Loss after iteration 5900: 0.005983108292764657
Loss after iteration 6000: 0.005917785467969577
Loss after iteration 6100: 0.005854582784445553
Loss after iteration 6200: 0.005793293445791697
Loss after iteration 6300: 0.005732983634946663
Loss after iteration 6400: 0.005674865260826775
Loss after iteration 6500: 0.005620536287697322
Loss after iteration 6600: 0.005562212500813223
Loss after iteration 6700: 0.0055082122874803505
Loss after iteration 6800: 0.005454448192454006
Loss after iteration 6900: 0.00540298361741481
Loss after iteration 7000: 0.005352000137141972
Loss after iteration 7100: 0.0053022171975148375
Loss after iteration 7200: 0.0052535905727747086
Loss after iteration 7300: 0.0052064517547342724
Loss after iteration 7400: 0.005159262624106201
Loss after iteration 7500: 0.005113579422128931
Loss after iteration 7600: 0.0050757918618601515
Loss after iteration 7700: 0.005024598938382136
Loss after iteration 7800: 0.004984982634194094
Loss after iteration 7900: 0.004938934586809853
Loss after iteration 8000: 0.004898931401908163
Loss after iteration 8100: 0.004856496250764368
Loss after iteration 8200: 0.004817198186105093
Loss after iteration 8300: 0.004776647200442619
Loss after iteration 8400: 0.004739011757369382
Loss after iteration 8500: 0.0046994146656003455
Loss after iteration 8600: 0.004662224237979721
Loss after iteration 8700: 0.004624963085844035
Loss after iteration 8800: 0.004590470045664712
Loss after iteration 8900: 0.004552760884905265
Loss after iteration 9000: 0.004520114233587442
Loss after iteration 9100: 0.004482671311224544
Loss after iteration 9200: 0.004455792232496872
Loss after iteration 9300: 0.004414851547906239
Loss after iteration 9400: 0.004381444662538033
Loss after iteration 9500: 0.004349027286774829
Loss after iteration 9600: 0.0043166581896012055
Loss after iteration 9700: 0.0042850116852920905
Loss after iteration 9800: 0.00425347275495452
```

```
Loss after iteration 9900: 0.004222949677735184
```

output_54_1.png

Expected Output:

Loss after iteration 0

0.6930497356599888

Loss after iteration 100

0.6464320953428849

...

...

Loss after iteration 2400

0.048554785628770206

Good thing you built a vectorized implementation! Otherwise it might have taken 10 times longer to train this.

Now, you can use the trained parameters to classify images from the dataset.

Exercise: - Implement the forward function - Implement the predict function below to make prediction on test_images

[64]:
```python
def two_layer_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

    # Implement LINEAR -> RELU. Add "cache" to the "caches" list.
    ### START CODE HERE ### (approx. 3 line of code)
    W1,b1 = parameters["W1"], parameters["b1"]
    A1,cache1 = linear_activation_forward(A,W1,b1,"relu")
    caches.append(cache1)


    ### END CODE HERE ###

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    ### START CODE HERE ### (approx. 3 line of code)
    W2,b2 = parameters["W2"], parameters["b2"]
    A2,cache2 = linear_activation_forward(A1,W2,b2,"sigmoid")
    caches.append(cache2)


    ### END CODE HERE ###

    assert(A2.shape == (1,X.shape[1]))

    return A2, caches
```

[65]:
```
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
    ### START CODE HERE ### ( 1 lines of code)
    probas, caches = two_layer_forward(X,parameters)
    ### END CODE HERE ###

    # convert probas to 0/1 predictions
```

```
    for i in range(0, probas.shape[1]):
        ### START CODE HERE ### ( 4 lines of code)
        prob = probas[0]
        if prob[i] > 0.5:
          p[0][i] = 1
        else:
          p[0][i] = 0


        ### END CODE HERE ###

    print("Accuracy: "  + str(np.sum((p == y)/m)))

    return p
```

[66]:
```
predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.9999999999999998
```

[67]:
```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.76
```

Exercise: Identify the hyperparameters in the model and For each hyperparameter - Briefly explain its role - Explore a range of values and describe their impact on (a) training loss and (b) test accuracy - Report the best hyperparameter value found.

Note: Provide your results and explanations in the report for this question.

****ANSWER-****

Hyperparameters: The Hyperparameters are defined as the parameters which needs tuning for a perfect fit to obtain. The hyperparameters in the model are as follows: \ 1. Learning Rate: The learning rate is defined as the rate at which the learning occurs in the model i.e., the rate at which the model proceeds towards convergence. 2. Epochs: Number of epochs is the number of iterations requred for the model to reach to optimum convergence.

Observation: We experiment with a range of values - \ Learning Rate: 0.0075, Epochs = 2500, n_h = 7 ; Training Loss = 0.048554785628770185, Test Accuracy = 0.7000000000000001 \ Learning Rate: 0.0075, Epochs = 5000, n_h = 7 ; Training Loss = 0.008719375589196448, Test Accuracy = 0.7200000000000001 \ Learning Rate: 0.01, Epochs = 2500, n_h = 7 ; Training Loss = 0.026822467408873438, Test Accuracy = 0.7 \ Learning Rate: 0.01, Epochs = 5000, n_h = 7 ; Training Loss = 0.005256576439731829, Test Accuracy = 0.7200000000000001 \ Learning Rate: 0.015, Epochs = 5000, n_h = 7 ; Training Loss = 0.0025284842579255083, Test Accuracy = 0.72 \ Learning Rate: 0.02, Epochs = 8000, n_h = 7 ; Training Loss = 0.0008280839239902898, Test Accuracy = 0.7 \ Learning Rate: 0.025, Epochs = 8000, n_h = 7 ; Training Loss = 0.004938934586809853, Test Accuracy = 0.76 \ Learning Rate: 0.025, Epochs = 10000, n_h = 7 ; Training Loss = 0.004222949677735184, Test Accuracy = 0.76 \ Learning Rate: 0.0275, Epochs = 10000, n_h = 7 ; Training Loss =

0.00038902457219370143, Test Accuracy = 0.72 \ Learning Rate: 0.03, Epochs = 10000, n_h = 7 ; Training Loss = 0.0003897854060466247, Test Accuracy = 0.76 \

Conclusion: It was observed that while some overfitted the data, some didnt converge well. It was thus concluded based on the observation that the best hyper-parameters chosen in this case for me, would be when: \ Learning Rate = 0.025, \ Epochs = 10000, \

The output for this case could be seen above **

3.2 Results Analysis

First, let's take a look at some images the 2-layer model labeled incorrectly. This will show a few mislabeled images.

[ ]:
```
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"))
```

[ ]:
```
print_mislabeled_images(classes, test_x, test_y, predictions_test)
```

output_65_0.png

Exercise: Identify a few types of images that tends to perform poorly on the model

Answer: \ It can be observed that the images having the cat image with some rotation and in a different colorspace, affects the training of the model. Also, the images which are generally containning shadows or too many subjects, result in poor training. A solution to this could be to increase the training set by including various image transformations (such as, rotation, color rectification outputs) to facilitate appropriate learning in the first case, and include all the data points which give misread predictions in the dataset. **

Now, lets use the same architecture to predict sentiment of movie reviews. In this section, most of the implementation is already provided. The exercises are mainly to understand what the workflow is when handling the text data.

```
[ ]: import re
```

4 Dataset

Problem Statement: You are given a dataset ("train_imdb.txt", "test_imdb.txt") containing: - a training set of m_train reviews - a test set of m_test reviews - the labels for the training examples are such that the first 50% belong to class 1 (positive) and the rest 50% of the data belong to class 0(negative)

Let's get more familiar with the dataset. Load the data by completing the function and run the cell below.

```
[ ]: def load_data(train_file, test_file):
         train_dataset = []
         test_dataset = []

         # Read the training dataset file line by line
         for line in open(train_file, 'r'):
             train_dataset.append(line.strip())

         for line in open(test_file, 'r'):
             test_dataset.append(line.strip())
         return train_dataset, test_dataset
```

```
[ ]: train_file = "train_imdb.txt"
     test_file = "test_imdb.txt"
     train_dataset, test_dataset = load_data(train_file, test_file)
```

```
[ ]: # This is just how the data is organized. The first 50% data is positive and the␣
      ↪rest 50% is negative for both train and test splits.
     y = [1 if i < len(train_dataset)*0.5 else 0 for i in range(len(train_dataset))]
```

As usual, lets check our dataset

```
[ ]: # Example of a review
     index = 10
     print(train_dataset[index])
     print ("y = " + str(y[index]))
```

```
I liked the film. Some of the action scenes were very interesting, tense and
well done. I especially liked the opening scene which had a semi truck in it. A
very tense action scene that seemed well done.<br /><br />Some of the
transitional scenes were filmed in interesting ways such as time lapse
photography, unusual colors, or interesting angles. Also the film is funny is
several parts. I also liked how the evil guy was portrayed too. I'd give the
film an 8 out of 10.
y = 1
```

[ ]:
```
# Explore your dataset
m_train = len(train_dataset)
m_test = len(test_dataset)

print ("Number of training examples: " + str(m_train))
print ("Number of testing examples: " + str(m_test))
```

```
Number of training examples: 1001
Number of testing examples: 201
```

4.1 Pre-Processing

From the example review, you can see that the raw data is really noisy! This is generally the case with the text data. Hence, Preprocessing the raw input and cleaning the text is essential. Please run the code snippet provided below.

Exercise: Explain what pattern the model is trying to capture using re.compile in your report.

Answer:

re.compile is used to remove all the symbols and spaces and breakspaces to include just the words and complete sentences in the training of the data. ***

[ ]:
```
REPLACE_NO_SPACE = re.compile("(\.)|(\;)|(\:)|(\!)|(\')|(\?
 ↪)|(\,)|(\")|(\()|(\))|(\[)|(\])|(\d+)")
REPLACE_WITH_SPACE = re.compile("(<br\s*/><br\s*/>)|(\-)|(\/)")
NO_SPACE = ""
SPACE = " "

def preprocess_reviews(reviews):

    reviews = [REPLACE_NO_SPACE.sub(NO_SPACE, line.lower()) for line in reviews]
    reviews = [REPLACE_WITH_SPACE.sub(SPACE, line) for line in reviews]

    return reviews

train_dataset_clean = preprocess_reviews(train_dataset)
test_dataset_clean = preprocess_reviews(test_dataset)
```

[ ]:
```
# Example of a clean review
index = 10
print(train_dataset_clean[index])
print ("y = " + str(y[index]))
```

```
i liked the film some of the action scenes were very interesting tense and well
done i especially liked the opening scene which had a semi truck in it a very
tense action scene that seemed well done some of the transitional scenes were
filmed in interesting ways such as time lapse photography unusual colors or
interesting angles also the film is funny is several parts i also liked how the
evil guy was portrayed too id give the film an  out of
y = 1
```

4.2 Vectorization

Now lets create a feature vector for our reviews based on a simple bag of words model. So, given an input text, we need to create a numerical vector which is simply the vector of word counts for each word of the vocabulary. Run the code below to get the feature representation.

[ ]:
```
from sklearn.feature_extraction.text import CountVectorizer

cv = CountVectorizer(binary=True, stop_words="english", max_features=2000)
cv.fit(train_dataset_clean)
X = cv.transform(train_dataset_clean)
X_test = cv.transform(test_dataset_clean)
```

CountVectorizer provides a sparse feature representation by default which is reasonable because only some words occur in individual example. However, for training neural network models, we generally use a dense representation vector.

[ ]:
```
X = np.array(X.todense()).astype(float)
X_test = np.array(X_test.todense()).astype(float)
y = np.array(y)
```

4.3 Model

[ ]:
```
from sklearn.metrics import accuracy_score
from sklearn.model_selection import train_test_split

X_train, X_val, y_train, y_val = train_test_split(
    X, y, train_size = 0.80
)
```

[ ]:
```
# This is just to correct the shape of the arrays as required by the␣
 ↪two_layer_model
X_train = X_train.T
X_val = X_val.T
```

```
y_train = y_train.reshape(1,-1)
y_val = y_val.reshape(1,-1)
```

[ ]:
```
### CONSTANTS DEFINING THE MODEL ####
n_x = X_train.shape[0]
n_h = 200
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

We will use the same two layer model that you completed in the previous section for training.

[ ]:
```
parameters = two_layer_model(X_train, y_train, layers_dims = (n_x, n_h,␣
↪n_y),learning_rate = 0.05, num_iterations = 3000, print_loss=True)
```

```
Loss after iteration 0: 0.6930794161691755
Loss after iteration 100: 0.6865694636725932
Loss after iteration 200: 0.6537467829024916
Loss after iteration 300: 0.5136378883276818
Loss after iteration 400: 0.32019564345849943
Loss after iteration 500: 0.20172484809309096
Loss after iteration 600: 0.1348199762191766
Loss after iteration 700: 0.09484013090636753
Loss after iteration 800: 0.06958636258548555
Loss after iteration 900: 0.05289162645593645
Loss after iteration 1000: 0.04148639996801364
Loss after iteration 1100: 0.03346401453332644
Loss after iteration 1200: 0.027650556748986144
Loss after iteration 1300: 0.023313400642090584
Loss after iteration 1400: 0.019991151062835836
Loss after iteration 1500: 0.017387623809518922
Loss after iteration 1600: 0.015306676209391544
Loss after iteration 1700: 0.013614662051177473
Loss after iteration 1800: 0.012218088972129238
Loss after iteration 1900: 0.011050224936784292
Loss after iteration 2000: 0.010062214655025914
Loss after iteration 2100: 0.009217663387552104
Loss after iteration 2200: 0.008489102814755763
Loss after iteration 2300: 0.007855373178672895
Loss after iteration 2400: 0.007300004618643866
Loss after iteration 2500: 0.006810030942075309
Loss after iteration 2600: 0.006375066459761163
Loss after iteration 2700: 0.005986796501139827
Loss after iteration 2800: 0.005638467870710704
Loss after iteration 2900: 0.005324491339104593
```

output_93_1.png

4.4 Predict the review for our movies!

[ ]:
```
predictions_train = predict(X_train, y_train, parameters)
```

```
Accuracy: 0.9999999999999998
```

[ ]:
```
predictions_val = predict(X_val, y_val, parameters)
```

```
Accuracy: 0.8507462686567162
```

4.5 Results Analysis

Let's take a look at some examples the 2-layer model labeled incorrectly

[ ]:
```
def print_mislabeled_reviews(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_reviews = len(mislabeled_indices[0])
    for i in range(num_reviews):
        index = mislabeled_indices[1][i]

        print((" ").join(cv.inverse_transform(X[index])[0]))
        print("Prediction: " + str(int(p[0,index])) + " \n Class: " + str(y[0,index]))
```

[ ]:
```
print_mislabeled_reviews(X_val.T, y_val, predictions_val)
```

```
actors attempt beauty believable big bit charismatic claims definitely delivery
did didnt disappointing disaster entertained fact film fine group job line
looked lost miss offensive performance playing plays plot project recommend rent
scenes screen seen strong talent wish writing
Prediction: 0
```

Class: 1
acting add annoying bad change character dicaprio did director does eyes film filmmakers films glad going good great half hand hardly impressive just kate learned lesson love mean million movie opinion oscar performance possible really romance romantic second ship shouldnt single sit stories story sure talented think thinking time times titanic try watching win wonderful wont worst
Prediction: 0
Class: 1
anna appearance away bad better bible big black blue book boys build capture cat catch chaos charles child city comic connected cops cult deal didnt doesnt earth edge exactly extreme far favorite fictional fine finished followed fond form fun gang gas genius giant god going good got government green guy guys half happening happens hard havent having heroes hey hospital include japanese join just kind know known like liked line looks lot make match monster movie mysterious naked named names new nice oh order paid past people place places police power problem project puts quickly red remains right seconds seeing series set sexy soon sorts special starts story taking thanks thats theres theyre thing tom unfortunately use using villains violence want whos woman world yeah year youd
Prediction: 0
Class: 1
able action add admit adult ahead anna bit black boy characters cinematography come deserves disappointed disappointment does dont doubt downright elements end entirely expect expecting fear film forget genre girls guys hopes imagination instead intense knew leave lesbian level like little looking lot love managed memorable mid movie ok performances play pleasure prepared read real realized really received romance school secret shot shown soon stars story straight sudden teenagers theyre think time times trying unexpected watching way white women wont wrong years youll
Prediction: 0
Class: 1
actor actually adults bringing calls cast child children doing era eyes famous focus fun given guess guy history host interesting john julia kenneth kind king like martin movie natural news park police provided question really say seeing short shouldnt simply smith sort story thought true version voice voices woman work worth wouldnt young
Prediction: 0
Class: 1
based got involved movie moving mystery oscar review script slow star
Prediction: 1
Class: 0
actually ahead better box budget burning character charlie come damn development did didnt dont enjoy eye fake far film flick forced genre good guinea guts hand hear heard horror hours interested just know like listen looks lot low making men minutes movie naturally offer painful pretty really recommend say scene scenes second seen set sharp short simply snuff story think thought throwing told torture trying ultimately unless various watching ways went woman worst
Prediction: 0

Class: 1
cause early effort government heavy past people problems production propaganda short spending sudden time truly using war window
Prediction: 1
Class: 0
ability acting actor ages anti better cheap cinema complete confused day decent direction disappointed disappointment does dont dull dvd explanation film finally finding flat gate good got great guess intriguing kind lead like liked love main meet mouth movie performance plot poor premise remind required result rip saw say store story tell thought took tv wanted way week written
Prediction: 1
Class: 0
action adventure adventures bad camp character characters check crew decided doc elements familiar fan fans feel feeling film good hero heroes im james jones just know long lot major minutes movie movies music number ones promise provided really resulting savage say seeing somewhat spirit star thats theres throw time trying unfortunate way
Prediction: 0
Class: 1
better body cast central cinema come coming comments company computer decides die entertaining exist failed fall fan film films genius gets getting going great hard hey highly hollywood house idea interested judging latest lesson lessons like make making man member money movie movies near potential premise puts review reviews role scenes seen set shock soon stupid taken takes type unless unrelated wish wow writer writing yes zero
Prediction: 1
Class: 0
admit almighty attempt big bruce carrey cast cheesy comedy dont end enjoyable fan feel funny gets gone havent help hilarious ill im jim just know let light like movie movies music note poor positive really rest reviews saying seen shows somewhat start steve thinking want writers youre
Prediction: 0
Class: 1
action age body brain building certainly computer crazy damme daughter dead entertaining especially fan fi fights folks genius goes going goldberg good government guess hes humor just keeps king lame later latest like manages mean named new original particularly perfect power pretty pro reason run sci sequel shoot site snake soldiers sort step super takes thriller train usual van war white working wrong year years youre
Prediction: 1
Class: 0
acting animals best better die dont entire episode episodes funny good horrible ice just killing know life like movie obviously plot pro problem really remember right scene scenes season second series shocking suspense think torture turns victims watch women wonderful worst
Prediction: 0
Class: 1
bunch doesnt feel got laugh laughed left like loud make masterpiece movie ok

purpose smile times viewer worth
Prediction: 0
 Class: 1
acting away beautifully biggest burt came character drinking fact failure fast fell general help hoping job movie movies night notice played promising real right screen single state thats walk way
Prediction: 1
 Class: 0
charlie dont eye fake film final harder hot im know like look looks real said say scene scenes sure tell thing truth
Prediction: 0
 Class: 1
actually ago american begin begins big bring buy century circumstances couple does doesnt effects emotional flicks follows happen highly home house husband impact john life like man masterpiece mysterious old outside plot recommended simple special story strange supposedly things turn unknown went woman world
Prediction: 0
 Class: 1
absolutely add bad best better boat book brought cases classic clear cliché close course critics deserves didnt disappointed exactly excitement family felt field film finally giving grew hard hear hero heroes home ill im instead know latest like line mind missing musical names nature non offensive old particularly past poor professional race real reality reviewer ridiculous right rock sadly said scene scenes score sense shot shots shows smile sound spot starting supposed taking talking theres theyve thrill time took town versions water wonderful years yes
Prediction: 1
 Class: 0
ability able accident action actresses actually aspect away bad believe better bit blood brothers cause cgi charlie crap crime cut days deal death disturbing does doesnt dont effects especially eyes fact fake favorite film films footage forget funny happens hope horror im instead leaving like look lot make makers making marry money movie movies overall people plot point porn probably pull rape rating real saying says scene scenes seen series shocking snuff sound stand stars sucked sucks super supposed sure talent talking thing thinking time tried visual visuals want wanted wasnt watch wouldnt
Prediction: 0
 Class: 1
achieve acting approach art artistic background box brief cast cheap cinema close come cons considered consists contemporary country dealing deals deserves director fact fan fit good half hard history hope hot huge job just knows like make manage masterpiece meant media members money movie naked near office ones opinion perfect perfectly provide purpose real roles short single small success talent theatrical thing time touching tried usual waiting women word work
Prediction: 1
 Class: 0
acting actors admit annoying arent art bad ball beginning best better big billy bits book calling camera case character characters cinematic come coming cons

crouse cusack david definitely dialogue did didnt direct directed does doesnt dont early end ending entertaining expecting extremely far feel film filmed films flat forth free fun game games gets getting girl going good guy half help heres hes hour house ill im inner involved isnt james john just keeps lesson let level like lindsay line lines little look looked lose mamet mantegna mark maybe mean men middle mind minutes moves movie narration nature new ones opening pick play precious pretty problem quality questions read reading real realize really result ring roll room scene second shes sort sound sounds speaking standard start stick story strange stuff supposed theatre theyre things true want wants watch way weird whats words work wouldnt write

Prediction: 0

Class: 1

accept ago army away bad begins body bucks budget chase comes couple dolph door energy especially exist explained feel feeling fight fighting films flash flick follow forward goes good happens hell human idea ideas involved isnt just key lacks like long looks low lukas make man master member merely middle movie movies needless new order place plays potential previous satan say scene scenes secret sense sort stars story study sucks supposed sure takes theres thrown time underground wish wont years york youll

Prediction: 1

Class: 0

absolutely acted art audience bad bar beginning came chinese come coming comments course deep didnt director doing drawn end ending entertaining essentially experience faces fact fantastic far feel festival film final following forget fresh fun gonna government half happy hard hidden hollywood hour hours im immediately incredibly intelligent intriguing judging just land late life likable long looked lot loved make making match meaning mention natural new number pain painful point post probably problem promising reading really reason reviews right russian said saw say sense sharp simply society sounds spent started state talking thank theatre thought time took try utter utterly view want wanted warned way week whats whatsoever words working years yes

Prediction: 1

Class: 0

actually ago bad better book church course does enjoyable familiar film forgotten forward good hadnt heard hour instantly job know laid let long minute minutes missed mr nearly overall quick read really school second sense short simply story tales thats thing time trilogy watched worked write years

Prediction: 0

Class: 1

actors ask blood cares character conclusion content crap crew damn day disturbing effort ends episode exception family fan fate gets going gore great gross hopes horror hour imagine lot mindless new performances pointless producers production reason season sense series shock stories story tend thinking utter values violence work worse

Prediction: 1

Class: 0

absolutely acting bits casting cheap close come comments completely couldve did

```
direction edge end film gone humor intense literally little loved mediocre movie
number perfect phone points read rest ring scary script second spot story thrill
time years
Prediction: 0
 Class: 1
ann cause characters comedy compared computer connection considered day days did
dont end entertainment ex fight film films flight future george given going got
hand having help hes home human including instead isnt issue just kids kill
killed know lesson life like live losing lost love make making man matter maybe
meets money necessary people pictures plan plays plot prior project rich school
sets shows stars step street streets stupid technology tender theyre things
think thrown treasure used using vote wall wants war wasnt woman work world
written years young
Prediction: 0
 Class: 1
actors alive based childhood documentary got kill know man mission monster movie
people personality played rate real scenes set seven turn used women work
Prediction: 0
 Class: 1
film forgotten late little long makers money movie night present subtle time
todays tv
Prediction: 0
 Class: 1
```

Exercise: Provide explanation as to why these examples were misclassified below.

Type your answer here

The training was done based on a simple bag of words used and the rating mentioned out of 10. However, the training doesnt work well for ambiguous words or words which cant be categorized based on their features. This is because we use sparse features and to fully and completely categorize dense feature vector is preferred.

```
[ ]:
```