Two Layer Neural Network

Name: **Rama Prashanth**
UID: **116428941**

In the first part, all the functions required to build a two layer neural networkare implemented. In the next part, these functions are used for image and text classification.

1. Packages

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
import h5py
import scipy
from PIL import Image
from scipy import ndimage
```

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.

In [2]:
```
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)

    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))

    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
```

In [3]:
```
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.]]

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:

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

3.1 Exercise: Build the linear part of forward propagation.

In [4]:
```
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 la
yer, 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 para
meter
    cache -- a python dictionary containing "A", "W" and "b" ; stored for comp
uting the backward pass efficiently
    """

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

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

    return Z, cache
```

In [5]:
```
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]]

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: $A = g(Z) = g(WA_{prev} + b)$

In [6]:
```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
    """

    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 --  returns Z, useful during backpropagation
    """

    A = np.maximum(0, Z)
    cache = Z

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

In [7]:
```
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 previo
us 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 st
ring: "sigmoid" or "relu"

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

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

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

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

    return A, cache
```

In [8]:
```
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, activatio
n = "sigmoid")
print("With sigmoid: A = " + str(A))

A, linear_activation_cache = linear_activation_forward(A_prev, W, b, activatio
n = "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.]]

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:

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

In [9]:
```
# 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.
    loss = (-1./m)* np.sum(Y * np.log(A) + (1. - Y) * np.log(1. - A))

    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
```

In [10]:
```
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

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

5.1 - Linear backward

In [11]:
```
# GRADED FUNCTION: linear_backward

def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer (l
ayer l)

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

    Returns:
    dA_prev -- Gradient of the loss with respect to the activation (of the pre
vious 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]

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

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

    return dA_prev, dW, db
```

In [12]:
```
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]]

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.

In [13]:

```
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.

    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

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

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

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

    Z = cache

    dZ = dA * sigmoid(Z)[0] * (1. - sigmoid(Z)[0])

    assert (dZ.shape == Z.shape)

    return dZ
```

In [14]:

```
# 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 com
puting backward propagation efficiently
    activation -- the activation to be used in this layer, stored as a text st
ring: "sigmoid" or "relu"

    Returns:
    dA_prev -- Gradient of the loss with respect to the activation (of the pre
vious 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":
        dZ = relu_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)
    elif activation == "sigmoid":
        dZ = sigmoid_backward(dA, activation_cache)
        dA_prev, dW, db = linear_backward(dZ, linear_cache)

    return dA_prev, dW, db
```

In [15]:
```
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, acti
vation = "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, acti
vation = "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	[[0.11017994 0.01105339] [0.09466817 0.00949723] [-0.05743092 -0.00576154]]
dW	[[0.20533573 0.19557101 -0.03936168]]
db	[[-0.11459244]]

Expected output with relu:

dA_prev	[[0.44090989 0.] [0.37883606 0.] [-0.2298228 0.]]
dW	[[0.89027649 0.74742835 -0.20957978]]
db	[[-0.41675785]]

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.

In [16]:
```
# 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_ba
ckward

    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.
    for key in parameters:
        parameters[key] = parameters[key] - learning_rate * grads["d"+key]

    return parameters
```

In [17]:
```
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	[[-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]]

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.

Part 2:

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

In [18]:
```
%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)
```

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.

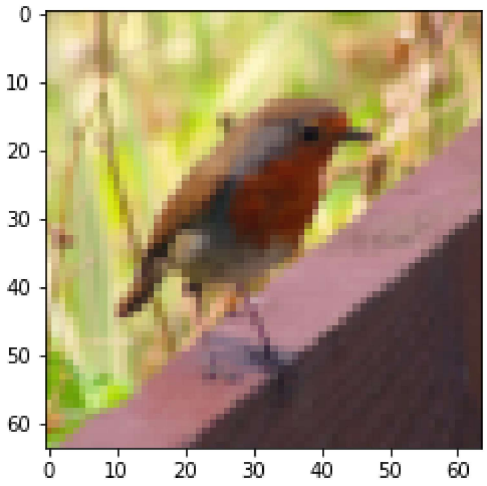
In [19]:
```
def load_data(train_file, test_file):
    # Load the training data
    train_dataset = h5py.File(train_file, 'r')

    # 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_ori
g, classes
```

In [20]:
```
train_file="data/train_catvnoncat.h5"
test_file="data/test_catvnoncat.h5"
train_x_orig, train_y, test_x_orig, test_y, classes = load_data(train_file, te
st_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.

In [21]:
```
# 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.
```



In [22]:
```
# 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.

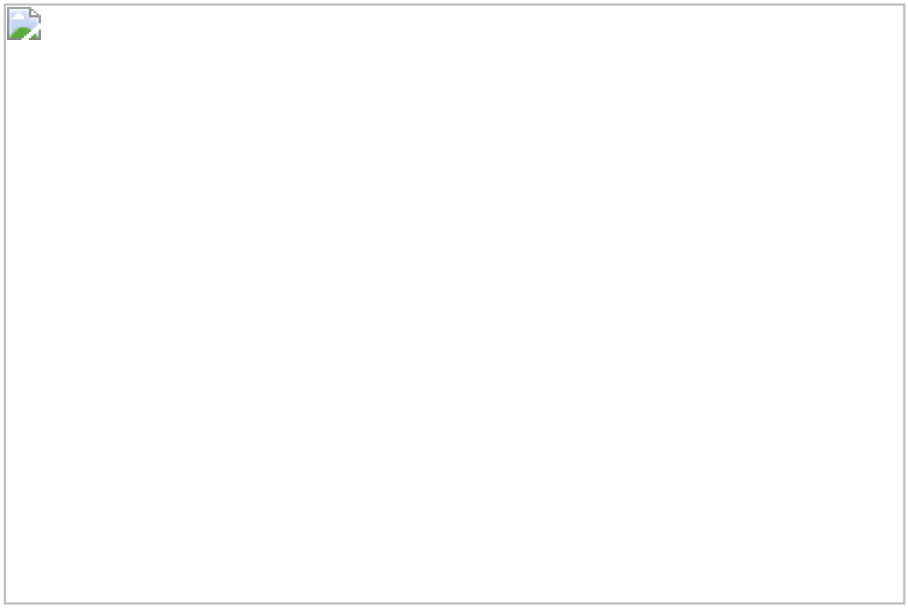


Figure 1: Image to vector conversion.

In [23]:
```
# 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 - 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.

2-layer neural network

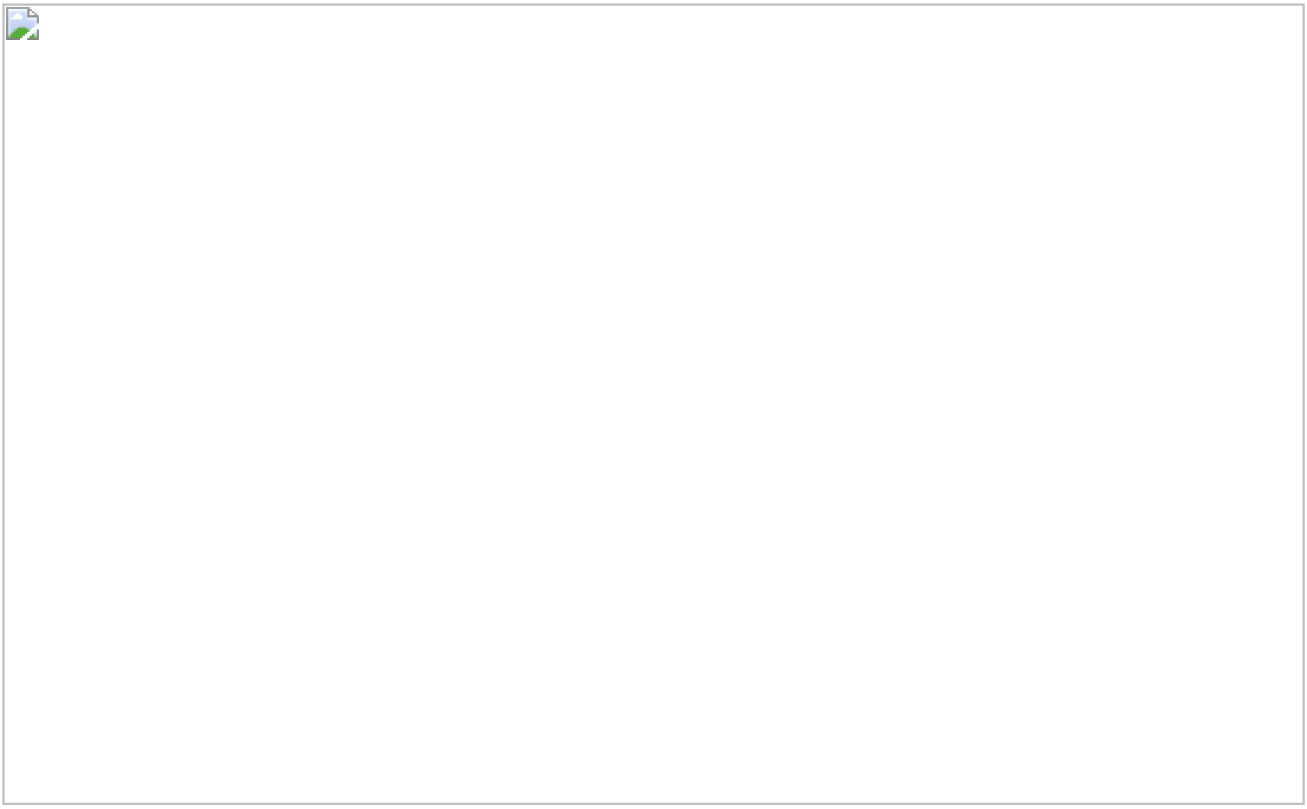


Figure 2: 2-layer neural network.
The model can be summarized as: ***INPUT -> LINEAR -> RELU -> LINEAR -> SIGMOID -> OUTPUT***.

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: $[x_0, x_1, \ldots, x_{12287}]^T$ 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: $[a_0^{[1]}, a_1^{[1]}, \ldots, a_{n^{[1]}-1}^{[1]}]^T$.
- 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.

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
```

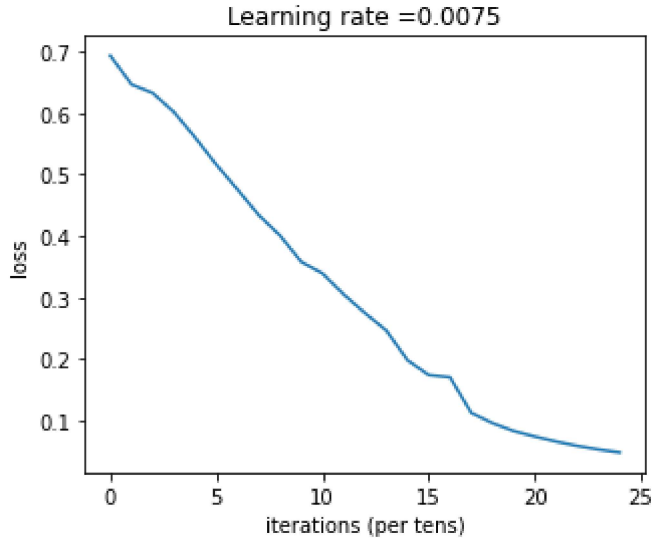
In [24]:
```
### CONSTANTS DEFINING THE MODEL ####
n_x = 12288     # num_px * num_px * 3
n_h = 7
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

In [25]:
```
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 iteration
s

    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
    parameters = initialize_parameters(n_x, n_h, n_y)

    # 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".
        A1, cache1 = linear_activation_forward(X, W1, b1, "relu")
        A2, cache2 = linear_activation_forward(A1, W2, b2, "sigmoid")

        # Compute loss
        loss = compute_loss(A2, Y)

        # 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".
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, "sigmoid")
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "relu")

        # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2, g
```

```
rads['db2'] to db2
        grads["dW1"] = dW1
        grads["db1"] = db1
        grads["dW2"] = dW2
        grads["db2"] = db2

        # Update parameters.
        parameters = update_parameters(parameters, grads, learning_rate)

        # 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
```

In [26]:
```
parameters = two_layer_model(train_x, train_y, layers_dims=(n_x, n_h, n_y), le
arning_rate=0.0075, num_iterations=2500, print_loss=True)
```

```
Loss after iteration 0: 0.693049735659989
Loss after iteration 100: 0.6464320953428849
Loss after iteration 200: 0.6325140647912677
Loss after iteration 300: 0.6015024920354665
Loss after iteration 400: 0.5601966311605747
Loss after iteration 500: 0.5158304772764729
Loss after iteration 600: 0.47549013139433255
Loss after iteration 700: 0.4339163151225749
Loss after iteration 800: 0.40079775362038894
Loss after iteration 900: 0.3580705011323798
Loss after iteration 1000: 0.3394281538366412
Loss after iteration 1100: 0.30527536361962637
Loss after iteration 1200: 0.27491377282130197
Loss after iteration 1300: 0.2468176821061484
Loss after iteration 1400: 0.19850735037466075
Loss after iteration 1500: 0.17448318112556627
Loss after iteration 1600: 0.17080762978096214
Loss after iteration 1700: 0.11306524562164723
Loss after iteration 1800: 0.09629426845937147
Loss after iteration 1900: 0.08342617959726863
Loss after iteration 2000: 0.0743907870431908
Loss after iteration 2100: 0.06630748132267932
Loss after iteration 2200: 0.05919329501038168
Loss after iteration 2300: 0.05336140348560555
Loss after iteration 2400: 0.04855478562877016
```



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

In [27]:
```
def two_layer_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGMOI
D 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, i
ndexed 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.
    A1, cache1 = linear_activation_forward(A, parameters["W1"], parameters["b
1"], "relu")
    caches.append(cache1)

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    A2, cache2 = linear_activation_forward(A1, parameters["W2"], parameters["b
2"], "sigmoid")
    caches.append(cache2)

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

    return A2, caches
```

In [28]:
```
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 =  two_layer_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("Accuracy: "  + str(np.sum((p == y)/m)))

    return p
```

In [29]:
```
predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.9999999999999998
```

In [30]:
```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.72
```

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.

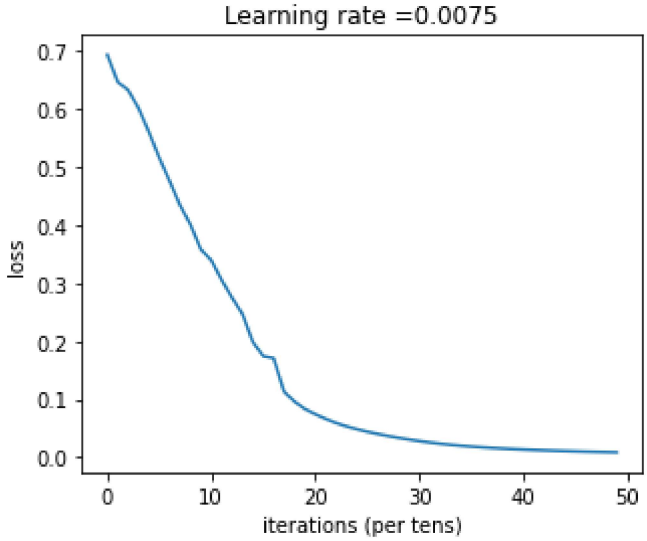
Report: The hyperparameters are variable that define the network structure and determine how the network is trained. The hyperparameters in the model are

- Learning rate : how quickly the network updates its parameters and proceeds towards convergence
- Number of iterations : number of times required for the model to reach convergence
- Hidden layers: layers between input layer and output layer

Experimenting with learning rate and number of iterations.

In [31]:
```
# (2) Learning Rate=0.0075, Num of Iterations=5000
parameters = two_layer_model(train_x, train_y, layers_dims=(n_x, n_h, n_y), le
arning_rate=0.0075, num_iterations=5000, print_loss=True)
```

```
Loss after iteration 0: 0.693049735659989
Loss after iteration 100: 0.6464320953428849
Loss after iteration 200: 0.6325140647912677
Loss after iteration 300: 0.6015024920354665
Loss after iteration 400: 0.5601966311605747
Loss after iteration 500: 0.5158304772764729
Loss after iteration 600: 0.47549013139433255
Loss after iteration 700: 0.4339163151225749
Loss after iteration 800: 0.40079775362038894
Loss after iteration 900: 0.3580705011323798
Loss after iteration 1000: 0.3394281538366412
Loss after iteration 1100: 0.30527536361962637
Loss after iteration 1200: 0.27491377282130197
Loss after iteration 1300: 0.2468176821061484
Loss after iteration 1400: 0.19850735037466075
Loss after iteration 1500: 0.17448318112556627
Loss after iteration 1600: 0.17080762978096214
Loss after iteration 1700: 0.11306524562164723
Loss after iteration 1800: 0.09629426845937147
Loss after iteration 1900: 0.08342617959726863
Loss after iteration 2000: 0.0743907870431908
Loss after iteration 2100: 0.06630748132267932
Loss after iteration 2200: 0.05919329501038168
Loss after iteration 2300: 0.05336140348560555
Loss after iteration 2400: 0.04855478562877016
Loss after iteration 2500: 0.044140596925487816
Loss after iteration 2600: 0.04034564500416593
Loss after iteration 2700: 0.03684121989478244
Loss after iteration 2800: 0.03366039892711179
Loss after iteration 2900: 0.030755596957824188
Loss after iteration 3000: 0.02809327869807166
Loss after iteration 3100: 0.025674470269470822
Loss after iteration 3200: 0.023538074095158668
Loss after iteration 3300: 0.02168811896962982
Loss after iteration 3400: 0.020061044752325806
Loss after iteration 3500: 0.018644251943883104
Loss after iteration 3600: 0.017356469767388413
Loss after iteration 3700: 0.016228603173680904
Loss after iteration 3800: 0.015225616153016694
Loss after iteration 3900: 0.014324063646356827
Loss after iteration 4000: 0.013498961092799482
Loss after iteration 4100: 0.01276132403035379
Loss after iteration 4200: 0.012091852624978146
Loss after iteration 4300: 0.011478737394770587
Loss after iteration 4400: 0.010918760508625105
Loss after iteration 4500: 0.010413951760046204
Loss after iteration 4600: 0.009932878969321586
Loss after iteration 4700: 0.009504412005290725
Loss after iteration 4800: 0.009095456333863114
Loss after iteration 4900: 0.008719375589196448
```



In [32]:
```
predictions_train = predict(train_x, train_y, parameters)
```

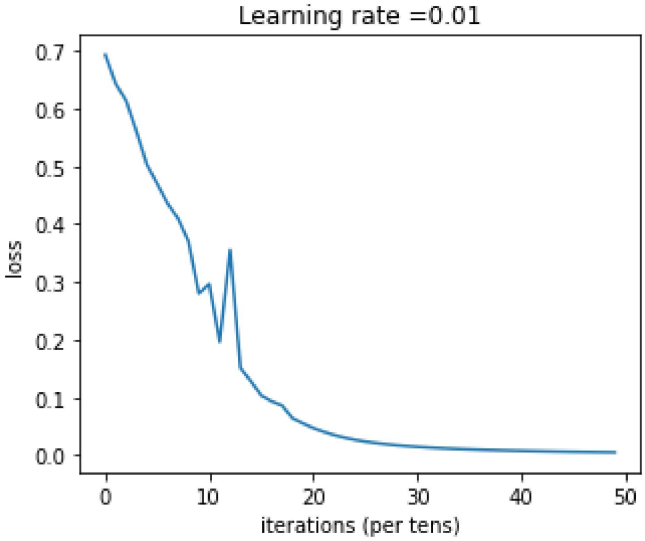
```
Accuracy: 0.9999999999999998
```

In [33]:
```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.7000000000000001
```

In [34]:
```
# (3) Learning Rate=0.01, Num of Iterations=5000
parameters = two_layer_model(train_x, train_y, layers_dims=(n_x, n_h, n_y), le
arning_rate=0.01, num_iterations=5000, print_loss=True)
```

```
Loss after iteration 0: 0.693049735659989
Loss after iteration 100: 0.6429673956634753
Loss after iteration 200: 0.6139330587122018
Loss after iteration 300: 0.5603909238662338
Loss after iteration 400: 0.5030563524628464
Loss after iteration 500: 0.4698443892342186
Loss after iteration 600: 0.4355621445123247
Loss after iteration 700: 0.40986977972092914
Loss after iteration 800: 0.3703324586925941
Loss after iteration 900: 0.2801254981389029
Loss after iteration 1000: 0.29654878488824177
Loss after iteration 1100: 0.1962538798352096
Loss after iteration 1200: 0.3552933244720939
Loss after iteration 1300: 0.15117073173355464
Loss after iteration 1400: 0.12822053502639688
Loss after iteration 1500: 0.10367329243705152
Loss after iteration 1600: 0.0933708195636492
Loss after iteration 1700: 0.08626919003765143
Loss after iteration 1800: 0.06430869400356842
Loss after iteration 1900: 0.055478150497232766
Loss after iteration 2000: 0.047190983560532064
Loss after iteration 2100: 0.0407888157746009
Loss after iteration 2200: 0.035118265397113144
Loss after iteration 2300: 0.030593822214670767
Loss after iteration 2400: 0.026822467408873063
Loss after iteration 2500: 0.0238284545558569
Loss after iteration 2600: 0.021330408382063865
Loss after iteration 2700: 0.01930887459058962
Loss after iteration 2800: 0.01753237676956266
Loss after iteration 2900: 0.016067905653536475
Loss after iteration 3000: 0.014764121454653441
Loss after iteration 3100: 0.013646291220035267
Loss after iteration 3200: 0.012654450027284657
Loss after iteration 3300: 0.011806015097269803
Loss after iteration 3400: 0.011034643583725072
Loss after iteration 3500: 0.010340844565270083
Loss after iteration 3600: 0.00971914786371022
Loss after iteration 3700: 0.009164971265751137
Loss after iteration 3800: 0.008660260155068595
Loss after iteration 3900: 0.008207315196935938
Loss after iteration 4000: 0.007790573008407104
Loss after iteration 4100: 0.0074074276469468835
Loss after iteration 4200: 0.007060193295864894
Loss after iteration 4300: 0.00673851065564536
Loss after iteration 4400: 0.006444637698926288
Loss after iteration 4500: 0.006174233714063698
Loss after iteration 4600: 0.0059145462197435356
Loss after iteration 4700: 0.005679218292481574
Loss after iteration 4800: 0.005461708455767702
Loss after iteration 4900: 0.005256576439731791
```



In [35]:
```
predictions_train = predict(train_x, train_y, parameters)
```

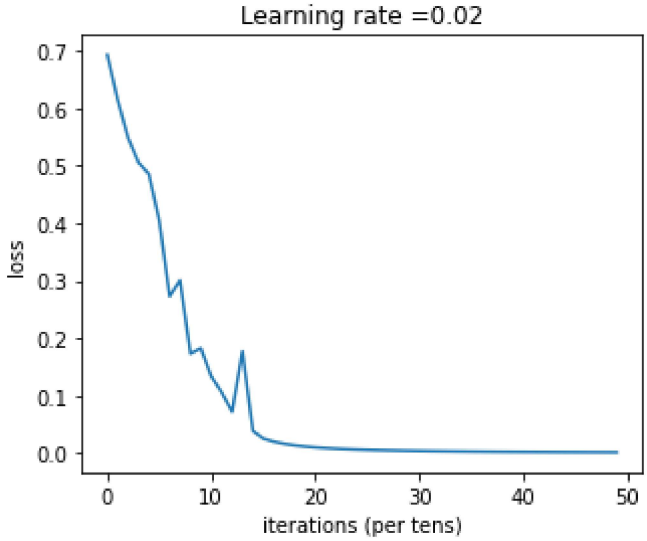
```
Accuracy: 0.9999999999999998
```

In [36]:
```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.72
```

In [37]:
```
# (4) Learning Rate=0.02, Num of Iterations=5000
parameters = two_layer_model(train_x, train_y, layers_dims=(n_x, n_h, n_y), le
arning_rate=0.02, num_iterations=5000, print_loss=True)
```

```
Loss after iteration 0: 0.693049735659989
Loss after iteration 100: 0.6145315364650429
Loss after iteration 200: 0.5488114210847703
Loss after iteration 300: 0.5058825075279194
Loss after iteration 400: 0.48569516783868544
Loss after iteration 500: 0.4048243720249345
Loss after iteration 600: 0.2732678832221182
Loss after iteration 700: 0.30073012628030377
Loss after iteration 800: 0.17329057983694227
Loss after iteration 900: 0.18291049276671917
Loss after iteration 1000: 0.13385565447873146
Loss after iteration 1100: 0.10595747519535126
Loss after iteration 1200: 0.07228866697883789
Loss after iteration 1300: 0.1774616196145582
Loss after iteration 1400: 0.03901303125966093
Loss after iteration 1500: 0.025658622447240285
Loss after iteration 1600: 0.020268587112739076
Loss after iteration 1700: 0.01658701283403051
Loss after iteration 1800: 0.013873288580346277
Loss after iteration 1900: 0.011852797250082169
Loss after iteration 2000: 0.010283563597833627
Loss after iteration 2100: 0.009046491031838081
Loss after iteration 2200: 0.008040529548383312
Loss after iteration 2300: 0.007217103637145896
Loss after iteration 2400: 0.0065308080369808445
Loss after iteration 2500: 0.005955423929960772
Loss after iteration 2600: 0.005459157042031566
Loss after iteration 2700: 0.00503285739045402
Loss after iteration 2800: 0.004665000262795303
Loss after iteration 2900: 0.004339751946752393
Loss after iteration 3000: 0.004052358640926915
Loss after iteration 3100: 0.003798328956699487
Loss after iteration 3200: 0.003572018920715597
Loss after iteration 3300: 0.0033683868740758544
Loss after iteration 3400: 0.0031863213814758086
Loss after iteration 3500: 0.0030190071770082585
Loss after iteration 3600: 0.0028689000144911533
Loss after iteration 3700: 0.0027311713324818434
Loss after iteration 3800: 0.002604080677192843
Loss after iteration 3900: 0.0024878767350114753
Loss after iteration 4000: 0.0023806450436634524
Loss after iteration 4100: 0.002281625587414716
Loss after iteration 4200: 0.0021898845637845625
Loss after iteration 4300: 0.002104693882466579
Loss after iteration 4400: 0.0020251062894580877
Loss after iteration 4500: 0.0019511019666554094
Loss after iteration 4600: 0.0018820271668535129
Loss after iteration 4700: 0.0018172600151084033
Loss after iteration 4800: 0.0017559686415460972
Loss after iteration 4900: 0.0016985912700763796
```



In [38]:
```
predictions_train = predict(train_x, train_y, parameters)
```

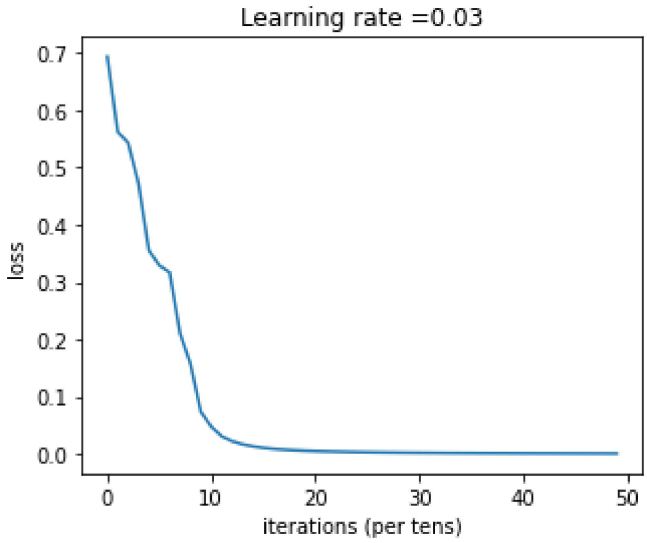
```
Accuracy: 0.9999999999999998
```

In [39]:
```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.7
```

In [40]:
```
# (5) Learning Rate=0.03, Num of Iterations=5000
parameters = two_layer_model(train_x, train_y, layers_dims=(n_x, n_h, n_y), le
arning_rate=0.03, num_iterations=5000, print_loss=True)
```

```
Loss after iteration 0: 0.693049735659989
Loss after iteration 100: 0.5619848087105088
Loss after iteration 200: 0.5431597368032648
Loss after iteration 300: 0.47223340373422207
Loss after iteration 400: 0.3553175029359639
Loss after iteration 500: 0.3291699923862487
Loss after iteration 600: 0.3167907370451563
Loss after iteration 700: 0.21048766105925854
Loss after iteration 800: 0.15751188890039813
Loss after iteration 900: 0.07383913770234714
Loss after iteration 1000: 0.047895499158278655
Loss after iteration 1100: 0.030910824116957638
Loss after iteration 1200: 0.022444957086082202
Loss after iteration 1300: 0.0170272751444631
Loss after iteration 1400: 0.013428523507592131
Loss after iteration 1500: 0.010840950614114398
Loss after iteration 1600: 0.009019915149393577
Loss after iteration 1700: 0.007685289774679797
Loss after iteration 1800: 0.006643699254909727
Loss after iteration 1900: 0.005817278179735696
Loss after iteration 2000: 0.0051552460757748764
Loss after iteration 2100: 0.00461026565098895
Loss after iteration 2200: 0.00416646358476657
Loss after iteration 2300: 0.0037888043511532204
Loss after iteration 2400: 0.0034706323758052168
Loss after iteration 2500: 0.003201149808743347
Loss after iteration 2600: 0.0029604802069483492
Loss after iteration 2700: 0.002752366769889475
Loss after iteration 2800: 0.0025723183963547334
Loss after iteration 2900: 0.002409791899734691
Loss after iteration 3000: 0.0022647641308828834
Loss after iteration 3100: 0.002135939723516692
Loss after iteration 3200: 0.002019747638574863
Loss after iteration 3300: 0.0019131195674080082
Loss after iteration 3400: 0.001817059673380523
Loss after iteration 3500: 0.0017305010343198004
Loss after iteration 3600: 0.0016492656953203421
Loss after iteration 3700: 0.0015756585116335172
Loss after iteration 3800: 0.0015076313749282366
Loss after iteration 3900: 0.0014454444290738124
Loss after iteration 4000: 0.00138758667165876 4
Loss after iteration 4100: 0.0013341968002012004
Loss after iteration 4200: 0.0012833184207657348
Loss after iteration 4300: 0.0012357507802573165
Loss after iteration 4400: 0.001192348910006889
Loss after iteration 4500: 0.0011508304542241243
Loss after iteration 4600: 0.001112334504061115
Loss after iteration 4700: 0.0010757090794808613
Loss after iteration 4800: 0.0010419903560546326
Loss after iteration 4900: 0.0010094956594547317
```



In [41]:
```
predictions_train = predict(train_x, train_y, parameters)
```

```
Accuracy: 0.9999999999999998
```

In [42]:
```
predictions_test = predict(test_x, test_y, parameters)
```

```
Accuracy: 0.74
```

The range of values experimented are as follows:

- learning_rate=0.0075, num_iterations=2500, n_h=7 -> training_loss=0.04855478562877016, test_accuracy=0.72
- learning_rate=0.0075 , num_iterations=5000, n_h=7 -> training_loss=0.008719375589196448, test_accuracy=0.7000000000000001
- learning_rate=0.01 , num_iterations=5000, n_h=7 -> training_loss=0.005256576439731791, test_accuracy=0.72
- learning_rate=0.02 , num_iterations=5000, n_h=7 -> training_loss=0.0016985912700763796, test_accuracy=0.7
- learning_rate=0.03 , num_iterations=5000, n_h=7 -> training_loss=0.0008597927494921494, test_accuracy=0.74

It can be seen that some range of valus overfitted and did not converge well. Also, increase in num of iterations validation accuracy starts decreasing even when training accuracy is increasing(overfitting).

Hence, optimal hyperparameters found are

- learning_rate=0.03
- num_iterations=5000

Results Analysis

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

In [43]:
```
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"))
```

In [44]:
```
print_mislabeled_images(classes, test_x, test_y, predictions_test)
```

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

Report: Images with cat but with background contrast similar to cat, different orientation, color, occlusion, shadow and multiple objects perform poorly for the model.

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.

In [45]:
```
import re
```

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.

In [46]:
```
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', encoding='utf8'):
        train_dataset.append(line.strip())

    for line in open(test_file, 'r', encoding='utf8'):
        test_dataset.append(line.strip())
    return train_dataset, test_dataset
```

In [47]:
```
train_file = "data/train_imdb.txt"
test_file = "data/test_imdb.txt"
train_dataset, test_dataset = load_data(train_file, test_file)
```

In [48]:
```
# This is just how the data is organized. The first 50% data is positive and t
he 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

In [49]:
```
# 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 trans
itional scenes were filmed in interesting ways such as time lapse photograph
y, unusual colors, or interesting angles. Also the film is funny is several p
arts. I also liked how the evil guy was portrayed too. I'd give the film an 8
out of 10.
y = 1
```

In [50]:
```
# 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
```

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.

In [51]:
```
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 review
s]
    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)
```

Report: re.compile() combines the regular expression pattern of special characters and converts to lowercase as well as replaces html tags like
with space.

In [52]:
```
# 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 we
ll done i especially liked the opening scene which had a semi truck in it a v
ery 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 ho
w the evil guy was portrayed too id give the film an  out of
y = 1
```

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.

In [53]:
```
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.

In [54]:
```
X = np.array(X.todense()).astype(float)
X_test = np.array(X_test.todense()).astype(float)
y = np.array(y)
```

Model

In [55]:
```
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
)
```

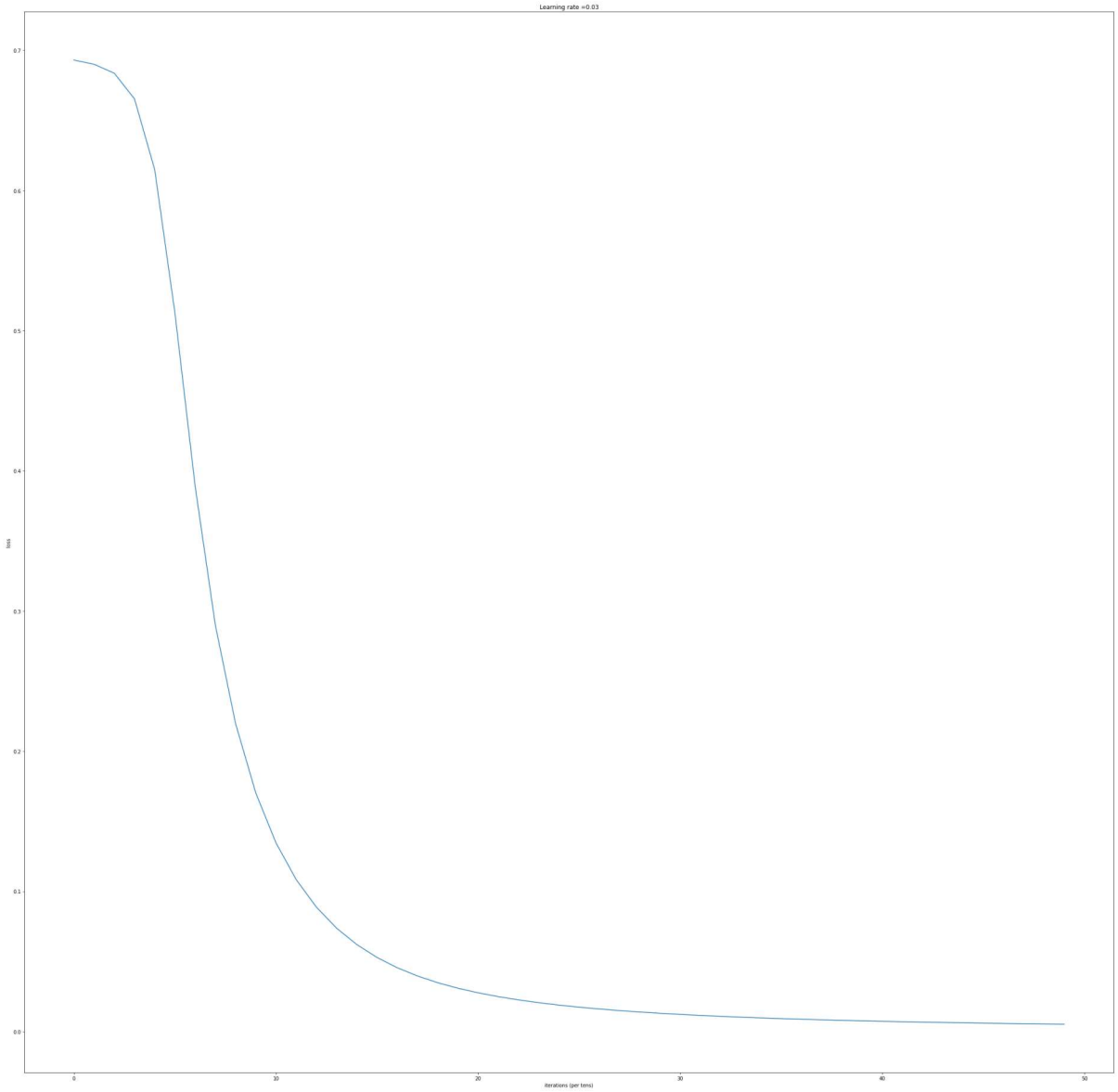
In [56]:
```
# This is just to correct the shape of the arrays as required by the two_layer
_model
X_train = X_train.T
X_val = X_val.T
y_train = y_train.reshape(1,-1)
y_val = y_val.reshape(1,-1)
```

In [57]:
```
### 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.

In [58]:
```
parameters = two_layer_model(X_train, y_train, layers_dims = (n_x, n_h, n_y),
learning_rate=0.03, num_iterations = 5000, print_loss=True)
```

```
Loss after iteration 0: 0.6930794161691755
Loss after iteration 100: 0.690075283281354
Loss after iteration 200: 0.6836840756228045
Loss after iteration 300: 0.6653641194544201
Loss after iteration 400: 0.6148463070680608
Loss after iteration 500: 0.5126071783395092
Loss after iteration 600: 0.38910289982819984
Loss after iteration 700: 0.2899036595756565
Loss after iteration 800: 0.2197087297385778
Loss after iteration 900: 0.1702128866142007
Loss after iteration 1000: 0.13458860287328744
Loss after iteration 1100: 0.10840031648734254
Loss after iteration 1200: 0.08875018516384307
Loss after iteration 1300: 0.07372246908489294
Loss after iteration 1400: 0.06203960510737666
Loss after iteration 1500: 0.05284103992236403
Loss after iteration 1600: 0.045517970450541234
Loss after iteration 1700: 0.039627494381771516
Loss after iteration 1800: 0.03483889488454306
Loss after iteration 1900: 0.030904356252072645
Loss after iteration 2000: 0.027636201093741618
Loss after iteration 2100: 0.024893615940999765
Loss after iteration 2200: 0.022569792344292107
Loss after iteration 2300: 0.020582795790961273
Loss after iteration 2400: 0.018869977950618805
Loss after iteration 2500: 0.01738242408329424
Loss after iteration 2600: 0.01608164912050358
Loss after iteration 2700: 0.014936962335099812
Loss after iteration 2800: 0.013923799601830567
Loss after iteration 2900: 0.013022194119360586
Loss after iteration 3000: 0.012215861307964758
Loss after iteration 3100: 0.011491467368566209
Loss after iteration 3200: 0.010837918593851921
Loss after iteration 3300: 0.010245911808327932
Loss after iteration 3400: 0.009707655722612259
Loss after iteration 3500: 0.009216585079138034
Loss after iteration 3600: 0.008767129796081317
Loss after iteration 3700: 0.008354535408948358
Loss after iteration 3800: 0.007974661593940245
Loss after iteration 3900: 0.007623968332480399
Loss after iteration 4000: 0.007299444450412163
Loss after iteration 4100: 0.006998398961735845
Loss after iteration 4200: 0.006718494925978342
Loss after iteration 4300: 0.006457701005977532
Loss after iteration 4400: 0.006214222756802738
Loss after iteration 4500: 0.005986498859914995
Loss after iteration 4600: 0.0057731282457380816
Loss after iteration 4700: 0.005572850821745077
Loss after iteration 4800: 0.005384565576589134
Loss after iteration 4900: 0.005207270375136098
```



Predict the review for our movies!

In [59]:
```
predictions_train = predict(X_train, y_train, parameters)
```

```
Accuracy: 0.9999999999999998
```

In [60]:
```
predictions_val = predict(X_val, y_val, parameters)
```

```
Accuracy: 0.8507462686567162
```

Results Analysis

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

In [61]:
```
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])) + " \nClass: " + str(y[0,i
ndex]) + "\n")
```

In [62]:
```
print_mislabeled_reviews(X_val.T, y_val, predictions_val)
```

actors attempt beauty believable big bit charismatic claims definitely delive
ry did didnt disappointing disaster entertained fact film fine group job line
looked lost miss offensive performance playing plays plot project recommend r
ent 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 rea
lly romance romantic second ship shouldnt single sit stories story sure talen
ted 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 doesn
t earth edge exactly extreme far favorite fictional fine finished followed fo
nd form fun gang gas genius giant god going good got government green guy guy
s half happening happens hard havent having heroes hey hospital include japan
ese join just kind know known like liked line looks lot make match monster mo
vie mysterious naked named names new nice oh order paid past people place pla
ces police power problem project puts quickly red remains right seconds seein
g series set sexy soon sorts special starts story taking thanks thats theres
theyre thing tom unfortunately use using villains violence want whos woman wo
rld yeah year youd
Prediction: 0
Class: 1

able action add admit adult ahead anna bit black boy characters cinematograph
y come deserves disappointed disappointment does dont doubt downright element
s end entirely expect expecting fear film forget genre girls guys hopes imagi
nation 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 st
raight sudden teenagers theyre think time times trying unexpected watching wa
y white women wont wrong years youll
Prediction: 0
Class: 1

actor actually adults bringing calls cast child children doing era eyes famou
s 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 voic
es 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 developm
ent did didnt dont enjoy eye fake far film flick forced genre good guinea gut
s 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 thr

owing 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 fas t fell general help hoping job movie movies night notice played promising rea l 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 c lose course critics deserves didnt disappointed exactly excitement family fel t field film finally giving grew hard hear hero heroes home ill im instead kn ow latest like line mind missing musical names nature non offensive old parti cularly past poor professional race real reality reviewer ridiculous right ro ck sadly said scene scenes score sense shot shots shows smile sound spot star ting supposed taking talking theres theyve thrill time took town versions wat er wonderful years yes
Prediction: 1
Class: 0

ability able accident action actresses actually aspect away bad believe bette r bit blood brothers cause cgi charlie crap crime cut days deal death disturb ing does doesnt dont effects especially eyes fact fake favorite film films fo otage forget funny happens hope horror im instead leaving like look lot make makers making marry money movie movies overall people plot point porn probabl y pull rape rating real saying says scene scenes seen series shocking snuff s ound stand stars sucked sucks super supposed sure talent talking thing thinki ng time tried visual visuals want wanted wasnt watch wouldnt
Prediction: 0
Class: 1

achieve acting approach art artistic background box brief cast cheap cinema c lose come cons considered consists contemporary country dealing deals deserve s director fact fan fit good half hard history hope hot huge job just knows l ike 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 bil

ly bits book calling camera case character characters cinematic come coming c
ons crouse cusack david definitely dialogue did didnt direct directed does do
esnt dont early end ending entertaining expecting extremely far feel film fil
med films flat forth free fun game games gets getting girl going good guy hal
f 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 manteg
na mark maybe mean men middle mind minutes moves movie narration nature new o
nes 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 thi
ngs 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 do
or energy especially exist explained feel feeling fight fighting films flash
flick follow forward goes good happens hell human idea ideas involved isnt ju
st key lacks like long looks low lukas make man master member merely middle m
ovie movies needless new order place plays potential previous satan say scene
scenes secret sense sort stars story study sucks supposed sure takes theres t
hrown time underground wish wont years york youll
Prediction: 1
Class: 0

absolutely acted art audience bad bar beginning came chinese come coming comm
ents course deep didnt director doing drawn end ending entertaining essential
ly experience faces fact fantastic far feel festival film final following for
get fresh fun gonna government half happy hard hidden hollywood hour hours im
immediately incredibly intelligent intriguing judging just land late life lik
able long looked lot loved make making match meaning mention natural new numb
er pain painful point post probably problem promising reading really reason r
eviews right russian said saw say sense sharp simply society sounds spent sta
rted state talking thank theatre thought time took try utter utterly view wan
t 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 forgo
tten forward good hadnt heard hour instantly job know laid let long minute mi
nutes missed mr nearly overall quick read really school second sense short si
mply 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 distur
bing effort ends episode exception family fan fate gets going gore great gros
s 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 d
id direction edge end film gone humor intense literally little loved mediocre
movie number perfect phone points read rest ring scary script second spot sto
ry 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 goi
ng 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 matte
r maybe meets money necessary people pictures plan plays plot prior project r
ich school sets shows stars step street streets stupid technology tender they
re things think thrown treasure used using vote wall wants war wasnt woman wo
rk world written years young
Prediction: 0
Class: 1

actors alive based childhood documentary got kill know man mission monster mo
vie 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.

Report: The classification solely relies on the number of words in positive and negative reviews. However, the context of phrases the words are used in determine the overall sentiment(For example, "not good"). Also, the relative ordering and context of the text adds more value to the sentiment.