

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: **Arpit Aggarwal**

UID: **116747189**

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. 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)

        ### START CODE HERE ### (~ 4 lines of code)

        W1 = np.random.randn(n_h, n_x) * 0.01
        b1 = np.zeros(shape=(n_h, 1))
        W2 = np.random.randn(n_y, n_h) * 0.01
        b2 = np.zeros(shape=(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
```

```
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:

$$Z = WA + b \quad (4)$$

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 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
```

```
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]: 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 = 1.0 / (1.0 + 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
```

```

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 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)
            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

```

```
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,
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.  ]]
```

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:

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


```
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.
    ### START CODE HERE ### (~ 1 lines of code)

    loss = (-1.0 / m) * np.sum((Y * np.log(A)) + ((1.0 - Y) * np.log(
    1.0 - 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
```

```
In [10]: Y = np.asarray([[1, 1, 1]])
A = np.array([[.8, .9, 0.4]])

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

loss = 0.414931599615
```

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 (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 p
    ropagation in the current layer

    Returns:
    dA_prev -- Gradient of the loss with respect to the activation (o
    f the previous layer l-1), same shape as A_prev
    dW -- Gradient of the loss with respect to W (current layer l), s
    ame shape as W
    db -- Gradient of the loss with respect to b (current layer l), s
    ame 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.array([np.sum(dZ, axis = 1)]).T

    ### 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
```

```
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

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

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.

    ### START CODE HERE ### (~ 1 line of code)

    dZ = dA * np.where(Z <= 0, 0, 1)

    ### 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)

    sigmoid_derivative = sigmoid(Z)[0] * (1.0 - sigmoid(Z)[0])
    dZ = dA * sigmoid_derivative

    ### END CODE HERE ###

    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 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

```

```
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, 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	[[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:

$$W^{[1]} = W^{[1]} - \alpha dW^{[1]} \quad (16)$$

$$b^{[1]} = b^{[1]} - \alpha db^{[1]} \quad (17)$$

$$W^{[2]} = W^{[2]} - \alpha dW^{[2]} \quad (16)$$

$$b^{[2]} = b^{[2]} - \alpha db^{[2]} \quad (17)$$

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_backward

    Returns:
    parameters -- python dictionary containing your updated parameters

    """
    # 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
```

```

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.

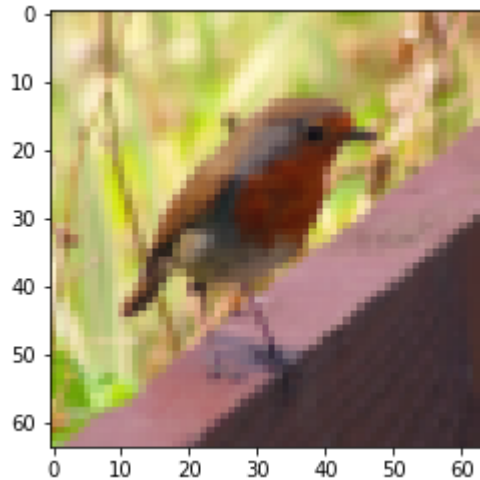
```
In [19]: def load_data(train_file, test_file):  
    # Load the training data  
    train_dataset = h5py.File(train_file, 'r')  
  
    # Separate features(x) and labels(y) for training set  
    train_set_x_orig = np.array(train_dataset['train_set_x'])  
    train_set_y_orig = np.array(train_dataset['train_set_y'])  
  
    # 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_orig, 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, 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.

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

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 [60]: ### CONSTANTS DEFINING THE MODEL ###
n_x = 12288      # num_px * num_px * 3
n_h = 16
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

```

In [61]: def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_ite
erations = 3000, print_loss=False):
    """
    Implements a two-layer neural network: LINEAR->RELU->LINEAR->SIGM
OID.

    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 rul
e
    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 l
oss
    m = X.shape[1]                                  # number of examples
    (n_x, n_h, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functio
ns 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. I
nputs: "X, W1, b1, W2, b2". Output: "A1, cache1, A2, cache2".
        ### START CODE HERE ### (~ 2 lines of code)

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

        ### END CODE HERE ###

        # Compute 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". Output
s: "dA1, dW2, db2; also dA0 (not used), dW1, db1".
    ### START CODE HERE ### (~ 2 lines of code)

    dA1, dW2, db2 = linear_activation_backward(dA2, cache2, "sigm
oid")
    dA0, dW1, db1 = linear_activation_backward(dA1, cache1, "rel
u")

    ### END CODE HERE ###

    # Set grads['dW1'] 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['db1'] = db1
    grads['dW2'] = dW2
    grads['db2'] = db2

    ### END CODE HERE ###

    # Update parameters.
    ### START CODE HERE ### (approx. 1 line of code)

    parameters = update_parameters(parameters, grads, learning_ra
te)

    ### 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')

```



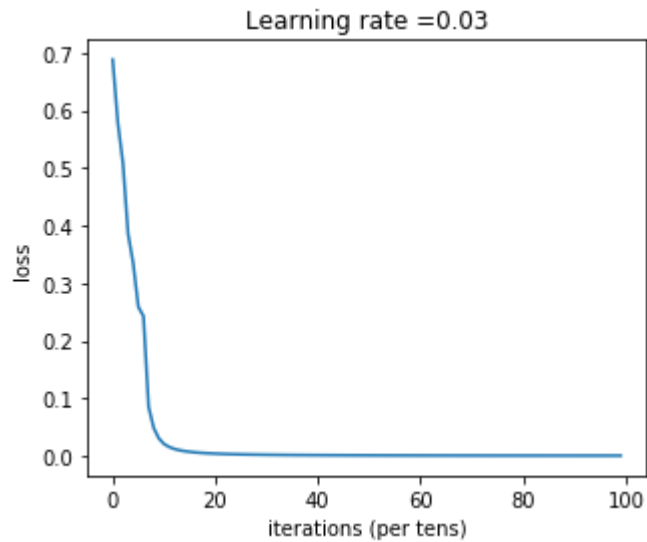
```
plt.xlabel('iterations (per tens)')  
plt.title("Learning rate =" + str(learning_rate))  
plt.show()
```

```
return parameters
```

```
In [62]: parameters = two_layer_model(train_x, train_y, layers_dims = (n_x, n_h, n_y), learning_rate=0.03, num_iterations = 10000, print_loss=True)
```

```
Loss after iteration 0: 0.688393523147
Loss after iteration 100: 0.579049134637
Loss after iteration 200: 0.508420918853
Loss after iteration 300: 0.385785570426
Loss after iteration 400: 0.334298174922
Loss after iteration 500: 0.258886131084
Loss after iteration 600: 0.242480920071
Loss after iteration 700: 0.085262963748
Loss after iteration 800: 0.0482942737336
Loss after iteration 900: 0.030316873026
Loss after iteration 1000: 0.0206189976738
Loss after iteration 1100: 0.0154346897346
Loss after iteration 1200: 0.0120604150315
Loss after iteration 1300: 0.00980617502059
Loss after iteration 1400: 0.00819703134039
Loss after iteration 1500: 0.00702193329309
Loss after iteration 1600: 0.00607945318277
Loss after iteration 1700: 0.00534481755265
Loss after iteration 1800: 0.00475985340071
Loss after iteration 1900: 0.00427200941256
Loss after iteration 2000: 0.0038745431193
Loss after iteration 2100: 0.00353729072171
Loss after iteration 2200: 0.00324225954921
Loss after iteration 2300: 0.00298923452383
Loss after iteration 2400: 0.0027730374875
Loss after iteration 2500: 0.00258249688328
Loss after iteration 2600: 0.00241219510226
Loss after iteration 2700: 0.00226132526151
Loss after iteration 2800: 0.00212889378751
Loss after iteration 2900: 0.00200644323403
Loss after iteration 3000: 0.00189828888188
Loss after iteration 3100: 0.00180083448423
Loss after iteration 3200: 0.0017106834854
Loss after iteration 3300: 0.00162971149333
Loss after iteration 3400: 0.00155565731518
Loss after iteration 3500: 0.00148657625629
Loss after iteration 3600: 0.0014232219759
Loss after iteration 3700: 0.00136597533629
Loss after iteration 3800: 0.00131100685538
Loss after iteration 3900: 0.00126052731698
Loss after iteration 4000: 0.00121380052901
Loss after iteration 4100: 0.00116996495597
Loss after iteration 4200: 0.00112972757602
Loss after iteration 4300: 0.00109132077069
Loss after iteration 4400: 0.00105504759078
Loss after iteration 4500: 0.00102139644714
Loss after iteration 4600: 0.000989002794319
Loss after iteration 4700: 0.000959243899863
Loss after iteration 4800: 0.00093027820525
Loss after iteration 4900: 0.000903701868289
Loss after iteration 5000: 0.000878213189419
Loss after iteration 5100: 0.0008538071834
Loss after iteration 5200: 0.000830925080664
Loss after iteration 5300: 0.000808883213712
Loss after iteration 5400: 0.00078771645784
Loss after iteration 5500: 0.000767924680549
Loss after iteration 5600: 0.000748877710796
```

```
Loss after iteration 5700: 0.000730977240722
Loss after iteration 5800: 0.000713383931703
Loss after iteration 5900: 0.000696763583711
Loss after iteration 6000: 0.000680944160894
Loss after iteration 6100: 0.000665736942768
Loss after iteration 6200: 0.000651290778289
Loss after iteration 6300: 0.000637122365836
Loss after iteration 6400: 0.000623561570686
Loss after iteration 6500: 0.000610654762376
Loss after iteration 6600: 0.000598141408121
Loss after iteration 6700: 0.000586258327153
Loss after iteration 6800: 0.000574721262969
Loss after iteration 6900: 0.00056364450166
Loss after iteration 7000: 0.000552924365765
Loss after iteration 7100: 0.000542325191918
Loss after iteration 7200: 0.000532353243035
Loss after iteration 7300: 0.000522633817397
Loss after iteration 7400: 0.000513476471429
Loss after iteration 7500: 0.000504413374686
Loss after iteration 7600: 0.000495649832349
Loss after iteration 7700: 0.000487058435733
Loss after iteration 7800: 0.000478830575937
Loss after iteration 7900: 0.000470858041239
Loss after iteration 8000: 0.000463253820976
Loss after iteration 8100: 0.000455730338756
Loss after iteration 8200: 0.000448495594325
Loss after iteration 8300: 0.000441493013111
Loss after iteration 8400: 0.000434534073784
Loss after iteration 8500: 0.000427870040696
Loss after iteration 8600: 0.000421531491197
Loss after iteration 8700: 0.000415125359062
Loss after iteration 8800: 0.000409026467465
Loss after iteration 8900: 0.000403138619622
Loss after iteration 9000: 0.000397335848205
Loss after iteration 9100: 0.000391700897989
Loss after iteration 9200: 0.000386209747614
Loss after iteration 9300: 0.000380846681169
Loss after iteration 9400: 0.000375643784842
Loss after iteration 9500: 0.000370567489968
Loss after iteration 9600: 0.000365672430362
Loss after iteration 9700: 0.000360804240652
Loss after iteration 9800: 0.000356113913313
Loss after iteration 9900: 0.00035153249996
```

**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 [63]: 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:
        A2 -- 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

```

```
In [64]: 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)

            if(probas[0][i] > 0.5):
                p[0][i] = 1
            else:
                p[0][i] = 0

            ### END CODE HERE ###

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

```
In [65]: predictions_train = predict(train_x, train_y, parameters)
```

Accuracy: 1.0

```
In [66]: predictions_test = predict(test_x, test_y, parameters)
```

Accuracy: 0.78

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.

Hyperparameters

The hyperparameters are:

1. Learning rate - It is used for updating the parameters of the neural network that is the weights and biases of the neural network. It controls the amount of update that needs to take place so that we are able to reach the minima of the loss function.
2. Epochs - It represents the number of times the network sees the data and adjusts its parameters for optimal learning.
3. Number of hidden neurons in the hidden layer - The number of neurons in the hidden layer where each neuron is learning some properties of the input data and able to establish a relationship between input and output.

Values of Hyperparameters tried:

1. Learning rate = 0.0075, Epochs = 2500, Hidden neurons = 7, Training loss = 0.0485, Testing accuracy: 72%
2. Learning rate = 0.0075, Epochs = 3000, Hidden neurons = 7, Training loss = 0.03, Testing accuracy: 72%
3. Learning rate = 0.01, Epochs = 8000, Hidden neurons = 7, Training loss = 0.0023, Testing accuracy: 72%
4. Learning rate = 0.01, Epochs = 10000, Hidden neurons = 7, Training loss = 0.001, Testing accuracy: 72%
5. Learning rate = 0.1, Epochs = 10000, Hidden neurons = 7, Training loss = 0.24, Testing accuracy: 62%
6. Learning rate = 0.001, Epochs = 10000, Hidden neurons = 7, Training loss = 0.16, Testing accuracy: 74%
7. Learning rate = 0.01, Epochs = 10000, Hidden neurons = 14, Training loss = 0.0015, Testing accuracy: 74%
8. Learning rate = 0.01, Epochs = 10000, Hidden neurons = 21, Training loss = 0.0014, Testing accuracy: 74%
9. Learning rate = 0.01, Epochs = 10000, Hidden neurons = 28, Training loss = 0.0014, Testing accuracy: 74%
10. Learning rate = 0.01, Epochs = 10000, Hidden neurons = 35, Training loss = 0.0014, Testing accuracy: 74%
11. Learning rate = 0.03, Epochs = 8000, Hidden neurons = 14, Training loss = 0.0004, Testing accuracy: 76%
12. Learning rate = 0.05, Epochs = 8000, Hidden neurons = 14, Training loss = 0.0002, Testing accuracy: 76%
13. Learning rate = 0.07, Epochs = 8000, Hidden neurons = 16, Training loss = 0.006, Testing accuracy: 72%
14. Learning rate = 0.05, Epochs = 8000, Hidden neurons = 16, Training loss = 0.0002, Testing accuracy: 74%
15. Learning rate = 0.05, Epochs = 10000, Hidden neurons = 16, Training loss = 0.001, Testing accuracy: 74%
16. Learning rate = 0.03, Epochs = 10000, Hidden neurons = 16, Training loss = 0.0001, Testing accuracy: 78%

Optimal hyperparameters found

1. Learning rate = 0.03
2. Epochs = 10000
3. Number of hidden neurons = 16

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 [67]: 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 [68]: 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

Answer The model performs poorly when the cat is at certain angle or rotated at some angle, which makes it classify it as a non-cat class.

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 [69]: 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 [70]: 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
```

```
In [71]: train_file = "data/train_imdb.txt"
          test_file = "data/test_imdb.txt"
          train_dataset, test_dataset = load_data(train_file, test_file)
```

```
In [72]: # 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, let's check our dataset

```
In [73]: # 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.

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 in several parts. I also liked how the evil guy was portrayed too. I'd give the film an 8 out of 10.
y = 1


```
In [76]: # 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 tens
e 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 t
he transitional scenes were filmed in interesting ways such as time l
apse photography unusual colors or interesting angles also the film i
s funny in several parts i also liked how 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 [77]: 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 [78]: X = np.array(X.todense()).astype(float)
X_test = np.array(X_test.todense()).astype(float)
y = np.array(y)
```

Model

```
In [79]: 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
)
```

/home/arpitdec5/.local/lib/python2.7/site-packages/sklearn/model_selection/_split.py:2178: FutureWarning: From version 0.21, test_size will always complement train_size unless both are specified.
FutureWarning)

```
In [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)
```

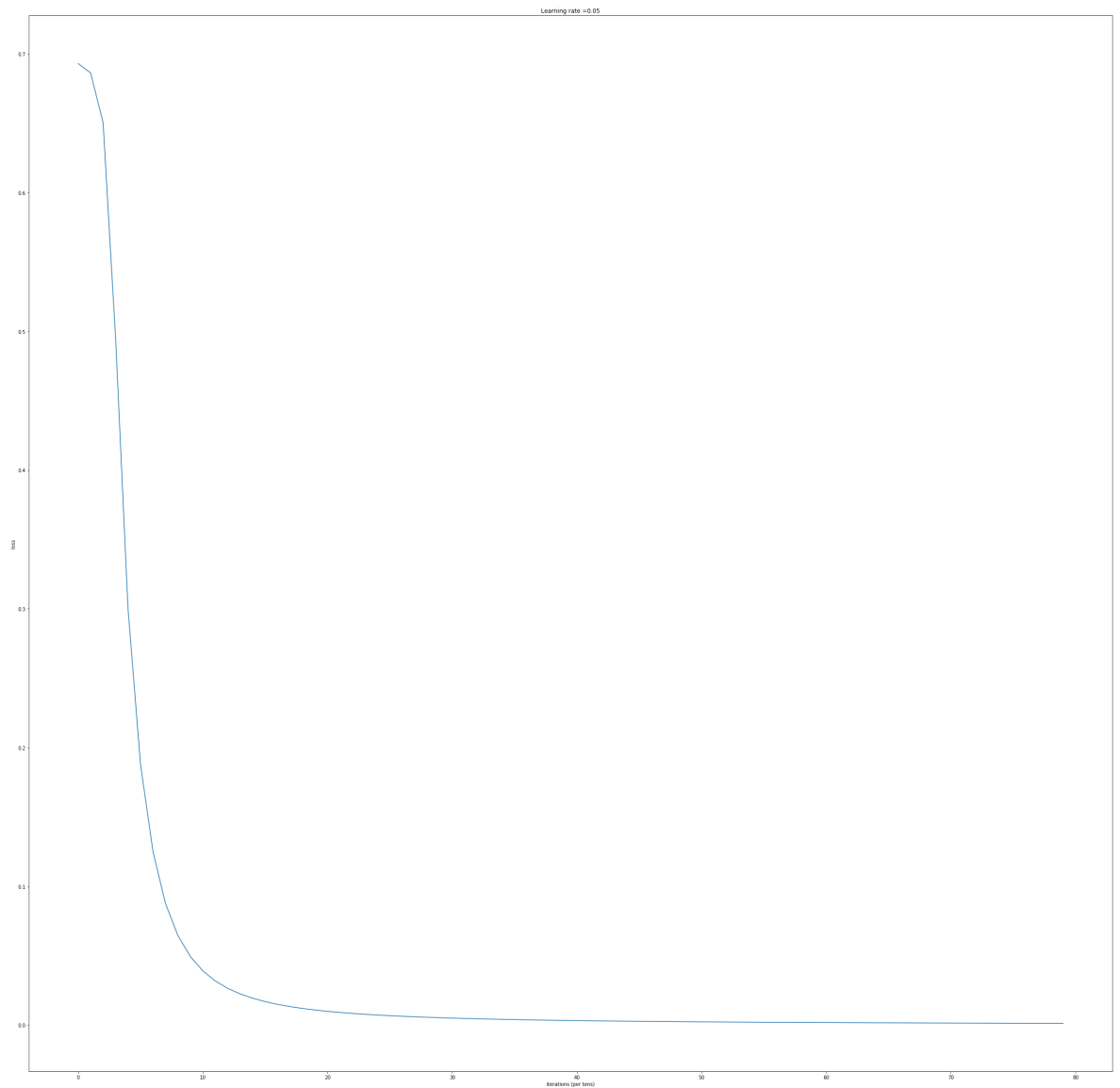
```
In [81]: ### CONSTANTS DEFINING THE MODEL ###
n_x = X_train.shape[0]
n_h = 200
n_y = 1
layers_dims = (n_x, n_h, n_y)
```

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

```
In [82]: parameters = two_layer_model(X_train, y_train, layers_dims = (n_x, n_h, n_y), learning_rate=0.05, num_iterations = 8000, print_loss=True)
```

```
Loss after iteration 0: 0.693103127068
Loss after iteration 100: 0.686325801758
Loss after iteration 200: 0.650424305473
Loss after iteration 300: 0.495118861505
Loss after iteration 400: 0.299957109486
Loss after iteration 500: 0.187918758462
Loss after iteration 600: 0.125078571109
Loss after iteration 700: 0.0877003817387
Loss after iteration 800: 0.0644730029248
Loss after iteration 900: 0.0493513725046
Loss after iteration 1000: 0.0390607670986
Loss after iteration 1100: 0.0317790029901
Loss after iteration 1200: 0.0264487274381
Loss after iteration 1300: 0.0224316559687
Loss after iteration 1400: 0.0193279528324
Loss after iteration 1500: 0.0168782083021
Loss after iteration 1600: 0.0149081139477
Loss after iteration 1700: 0.0132978073588
Loss after iteration 1800: 0.0119627052869
Loss after iteration 1900: 0.0108419352978
Loss after iteration 2000: 0.00989049719261
Loss after iteration 2100: 0.00907470205842
Loss after iteration 2200: 0.00836901672905
Loss after iteration 2300: 0.00775369690982
Loss after iteration 2400: 0.00721331418577
Loss after iteration 2500: 0.00673564318385
Loss after iteration 2600: 0.00631088725029
Loss after iteration 2700: 0.00593120039393
Loss after iteration 2800: 0.00559008290887
Loss after iteration 2900: 0.00528224353747
Loss after iteration 3000: 0.00500324893594
Loss after iteration 3100: 0.00474940633412
Loss after iteration 3200: 0.00451760447274
Loss after iteration 3300: 0.00430520997845
Loss after iteration 3400: 0.00410999304016
Loss after iteration 3500: 0.00393004471345
Loss after iteration 3600: 0.00376371494495
Loss after iteration 3700: 0.00360960510452
Loss after iteration 3800: 0.0034664531896
Loss after iteration 3900: 0.00333317845034
Loss after iteration 4000: 0.003208833456
Loss after iteration 4100: 0.00309258657312
Loss after iteration 4200: 0.00298370026242
Loss after iteration 4300: 0.00288152935292
Loss after iteration 4400: 0.00278549301041
Loss after iteration 4500: 0.00269507982634
Loss after iteration 4600: 0.00260982849242
Loss after iteration 4700: 0.0025293284677
Loss after iteration 4800: 0.00245321155267
Loss after iteration 4900: 0.00238114173526
Loss after iteration 5000: 0.00231281837098
Loss after iteration 5100: 0.00224796345618
Loss after iteration 5200: 0.00218633118886
Loss after iteration 5300: 0.00212769757703
Loss after iteration 5400: 0.00207185503525
Loss after iteration 5500: 0.00201861749274
Loss after iteration 5600: 0.0019678140272
```

```
Loss after iteration 5700: 0.00191928627225
Loss after iteration 5800: 0.00187289051559
Loss after iteration 5900: 0.00182849569683
Loss after iteration 6000: 0.0017859783184
Loss after iteration 6100: 0.00174522741013
Loss after iteration 6200: 0.00170614013283
Loss after iteration 6300: 0.00166861878011
Loss after iteration 6400: 0.00163257491662
Loss after iteration 6500: 0.00159792537434
Loss after iteration 6600: 0.00156459418368
Loss after iteration 6700: 0.00153251121141
Loss after iteration 6800: 0.00150160924088
Loss after iteration 6900: 0.00147182919778
Loss after iteration 7000: 0.00144311098702
Loss after iteration 7100: 0.0014154018356
Loss after iteration 7200: 0.00138865002942
Loss after iteration 7300: 0.00136280818019
Loss after iteration 7400: 0.0013378330865
Loss after iteration 7500: 0.0013136824285
Loss after iteration 7600: 0.00129031827686
Loss after iteration 7700: 0.00126770352539
Loss after iteration 7800: 0.00124580447418
Loss after iteration 7900: 0.00122458847703
```

Predict the review for our movies!

```
In [83]: predictions_train = predict(X_train, y_train, parameters)
```

Accuracy: 1.0

```
In [84]: predictions_val = predict(X_val, y_val, parameters)
```

Accuracy: 0.830845771144

Results Analysis

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

```
In [85]: 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]))
```

```
In [86]: print_mislabeled_reviews(X_val.T, y_val, predictions_val)
```

actually attractive budget characters considering dont end film fun good happens hollywood huge insane ive jesse just let lot mix performance pleasure plenty quite really roll running short time totally turned twists watch watched wont worse

Prediction: 0

Class: 1

acting actress added beautiful blood boring career cast color compared days earlier films golden great hunter just leading like man mark memories particularly performance photography picked pleasant quickly rate saw second shown story supporting terrible think time turned unbelievable

Prediction: 1

Class: 0

animal dont hours life love planet script series video wait

Prediction: 1

Class: 0

acting away bad cgi chose cool crazy cut disappointment exactly far felt goes good got imdb ive like lot make makes mind movie movies people plane plot poor real really reason recommend right score seen sense snakes start stay stuff stupid taste thats thinking title types unbelievable way worst wouldnt

Prediction: 0

Class: 1

according actors amazed bar come conclusion conversations decent director does doesnt expect experience feel feeling feelings film funny getting good half hear heard hour like listen lives living losing loss lot majority make makes minutes movie movies music ok people probably shame start starts stop story tell think vote walk wanted watch watching witness world worth

Prediction: 1

Class: 0

actors art audience comedy didnt director dog drama enjoyed festival film fine friends house including involving job lake looks marriage married movie needed place saw school scott stars subject takes thought treat wants way whos wife wonderful

Prediction: 1

Class: 0

actually amazed better big black brothers cheap comedy constantly content detective dont earth face fact famous film films funny genuine high hit horror huge just laugh look mention mid minutes place release screaming shot situation time today week works wrong yes

Prediction: 0

Class: 1

action actors adults age aged angles appears avoid away camera close core definitely different direction director disappointment dont entertainment example excellent expect expected famous forward going good got guess hard heroes impressive issues just kids kind late like liked long love mind movie music needs nice nicely okay older overall particular play presence problem pull quality really release remember romantic scenes seeing sequences slowly small songs star stick sure things time took touch touching trying ups various watching woman

Prediction: 1

Class: 0

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 sin

gle sit stories story sure talented think thinking time times titanic
try watching win wonderful wont worst

Prediction: 0

Class: 1

academy actors average best bit blown career changing character chara
cters completely connected considered cruise crying deep didnt doesnt
dont easy emotions enjoy enjoyable episode events fair far film finis
h flaws forced friends good great hate help high honestly hours im ju
st life like little long love meaningful movie music pace people perf
ormance play playing plot pointless police poor quality relationships
seemingly self share significant start stories stuff tells thats ther
es think thought tom total turns unique wasnt watch whilst wife worse
written wrong

Prediction: 1

Class: 0

able action actually arent better caught chair cheesy count credit de
cide deserves does dont episode episodes excuse explain fact fi follo
w going high hold hour idea ideas im impressive isnt long makes matte
r minutes movie movies nearly pretty probably quality really remains
sci series sg special stargate stick stolen storyline surprised talki
ng thats theres think try tv worse youll

Prediction: 0

Class: 1

acting attempt attempts camera cinematography complete did director d
oes effective entertainment experience extremely fact film films girl
s goes good henry high imagery including innocent intelligent interes
ting later lesbian long make nature nudity number paris school sex sl
owly stories story study sweet todays trip try went

Prediction: 0

Class: 1

bad changing china chinese contemporary culture deeper development di
rector entire era example excellent face film girls good meaning movi
e old opera post problem sense spiritual supposed traditional used

Prediction: 0

Class: 1

absolutely acting actors audrey bad bit brief build change characters
conclusion cute didnt different director dvd event exactly explanatio
n fate fight film films fine future great guy guys happens helps hes
instead isnt just like little main movie odd pace plus random reason
reminds role scene score seemingly shows small sort starring stuff su
ccess theme theres things think trying understand ways writers

Prediction: 0

Class: 1

branagh brought car chance children dangerous day discover drunk fath
er finds friends having home house later lawyer life local lost meeti
ng morning named night party seemingly sent situation spends successf
ul trial trouble trying walk willing woman

Prediction: 1

Class: 0

bit body clichés couldve dead decent didnt dont enjoy enjoyable enter
taining film gave getting going good horrible just little minutes mov
ie nice oh past people pretty rating sit started starts stuff think w
asnt

Prediction: 0

Class: 1

acted acting ann attempt attempts ball bar best buy cast characters c
harming cinematic classic design development direction drama dramatic

editing effective effort engaging entertainment excitement favorite g
em girl good held historical ideas interesting known like line little
look looking magical makes marry memorable memory mentioned money mus
ic musical obvious painful perspective photography pleasure points po
or poorly problem production provide review role roles self sets show
ing spoilers star story storytelling supporting tale tell throw turn
tv ugly value version want wants women wonderful work world writing w
ritten

Prediction: 1

Class: 0

accurate actors actual actually amazing bad best better big bother br
eak budget came cameron capture care certainly cover critics day dece
nt dialogue dicaprio did didnt dont effects enjoyed face film films f
ine form general going good guess guys heard hearing hit huge includi
ng isnt ive james job just lets like love make maybe million mind mon
ey mouth movie needed obviously people personally public real really
romance say screen seeing seen ship sinking special star started stor
y success sucks sweet talk talking thank theaters thing think thought
time times titanic took touch twice type wasnt winslet word work wors
t worth wouldnt writers years

Prediction: 0

Class: 1

age ball big blood camp characters daughter didnt double driving effe
ct film flick free good hell id im job just know like looking looks m
ain movie movies nudity ones place played problem released season see
n story sucked sure takes theres thing vision women young

Prediction: 1

Class: 0

film forgotten late little long makers money movie night present subt
le time todays tv

Prediction: 0

Class: 1

apparently away brilliant budget cinema compare convincing did didnt
downright film flaws gave high horrific horror knows like likes lines
look low make meant non original provoking really review reviewers sa
ys score sense shock sick slightly supposed thats thoroughly thought
trust value versions viewers visuals want wasnt way week

Prediction: 0

Class: 1

apparent audience better book boy certain certainly clear complete da
rk doesnt effort feels felt film good holes inspired intriguing like
makes mystery nature portrayal premise probably production recommend
tension thing true truly williams woman written

Prediction: 0

Class: 1

add average baby best better big buying cast cinema come comedy compa
red did doesnt dont dull emotional end ending expect face fails favor
ite funny girls good great hilarious hit isnt knew laugh laughs leave
looks loud martin mean movie movies predictable recommend renting sav
e scenes smile soon started steve supporting sure team thats times to
tally tries try wasnt world year

Prediction: 1

Class: 0

adult atmosphere bad boy brother camera cast characters christopher c
lose despite didnt director drama experience family film given matter
memorable movie natural nearly necessary older people picture project
really robert sad shooting situation small story subject talent treat

ment unfortunately worth young

Prediction: 1

Class: 0

actress attractive ball classic comedy comic count dreadful face good hands hopefully humor killing level likes little look low meant michael movies non odd paid pick picture play running screenplay sea shots speaking stars stuff watch

Prediction: 1

Class: 0

arent big bit boss car changed chases did didnt does dumb earlier enjoy enjoyed episode exactly excuse face fact fan far flick folks funny general goes great half hearing huge humor im job johnny jokes just kids lame lee like little lot makes material movie people perfect pilot plot probably really reason remake remember says scary scott screen small smile sorry source stays streets time tv versions wasnt william wont work

Prediction: 0

Class: 1

acting actually admit bad brought came damn didnt downright especially figured friends funny given good got guy hilarious holds home horror ive kinda know laugh like mom movie movies overall pretty promise released say scary seen store sucks supposed used video wait watch way weird work yeah yes

Prediction: 1

Class: 0

actors based break career chance circumstances creates deeply directed doesnt feels film garbage good help ill jones just killing like make performances place script second shouldnt story talented telling theyre told wonder world worth

Prediction: 1

Class: 0

accident acting actually anti away bad beginning best better bit bring car character comes completely confusing create created decent device dialogue didnt die director effect elements episode expecting extreme face far feel felt film forth fresh future genres ghost gives goes going good gore grace great harder hero hes hope horrible horror idea im images issue john key later left let like little live look looks main make man master nice opinion particularly plot prior protagonist read relationship remind revenge review reviewer revolves right said saving say season second seen sense solid somewhat spectacular state story strong suspense takes talking think time times turn twists viewer wants watch watching way wife wish wrong

Prediction: 0

Class: 1

ago based best big blood close copy ending enjoy fight funny glad got guess hey hurt im jesus kill killer like looks lot man memory mention movie scenes screen seen seven shes shown slasher spoil story surely suspense thats theres times ups video wearing whos wont wood woods work years

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 il

l 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 op
ening pick play precious pretty problem quality questions read readin
g real realize really result ring roll room scene second shes sort so
und sounds speaking standard start stick story strange stuff supposed
theatre theyre things true want wants watch way weird whats words wor
k wouldnt write

Prediction: 0

Class: 1

begin believable believe bit carter casting character characters chos
e didnt doesnt episode especially ex felt film guy hand hes hit innoc
ence like likely line little looks lot lovely make maybe means mind m
iss moment mother moved movie mr needed notes offer old older origina
l passion people play played playing problems project required right
run scenes similar sister slow stories suppose taste thought town tv
typical version war william wise woman women years young younger

Prediction: 1

Class: 0

angels apparent appear appearance author baby bad beautiful began bel
ieve better bloody book break came case chamberlains come completely
conclusion course cult cut day did didnt difficult dingo dont dr evid
ence evil fact far finding following going good hand hardly heard hig
hly human important intended ive james john just knowing later lie li
ked lindy meryl movie music obviously people perfect photography pick
ed point possible presented press public question quite reason releas
ed sam sorts spoilers story taken technical think time told trial und
erstand wasnt wild worked wrong

Prediction: 0

Class: 1

away better bit bought boy carried case comes complete day deserves d
iscoverers doggie dont drama dvd ending expect fact film final funny ge
ts girl good humor instead involving just kid kind like little lookin
g lot man manage minutes missing money needs nice obviously old overa
ll pass performer picture problems quality really rent rest sentiment
al son soon spending story street suspense thought treatment tv usual
ly wang way

Prediction: 0

Class: 1

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

Type your answer here

The main aim of the model is to predict the sentiment. As each word is taken individually, the model is failing to learn the sentiment of the current word from previous words and is failing to learn from the sentence as a whole.

In []: