PROJECT HW2

Load the dataset

In [ ]:

```
import os
import numpy as np
import cv2
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
import h5py

# Function to load images and labels in grayscale
def load_images_and_labels(folder):
    images = []
    labels = []

    for filename in os.listdir(folder):
        img_path = os.path.join(folder, filename)
        if filename.startswith('NORMAL'):
            labels.append('NORMAL')
        elif filename.startswith('VIRUS') or filename.startswith('BACTERIA')
            labels.append('PNEUMONIA')
        else:
            # Skip unrelated files
            continue

        if os.path.isfile(img_path):
            # Read image as grayscale
            img = cv2.imread(img_path, cv2.IMREAD_GRAYSCALE)
            if img is not None:
                images.append(img)

    return images, labels

# Load images and labels from the folder
folder_path = "/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/
images, labels = load_images_and_labels(folder_path)

# Convert labels to numerical values
label_encoder = LabelEncoder()
labels_encoded = label_encoder.fit_transform(labels)

# Preprocess images (resize, normalize, etc.)
def preprocess_images(images, size=(64, 64)):
    processed_images = []
    for img in images:
        # Resize images to the specified size
        resized_img = cv2.resize(img, size)

        # Normalize pixel values to be between 0 and 1
        normalized_img = resized_img / 255.0
```

```
        # Expand dimensions to retain consistency in shape for deep learning
        processed_images.append(np.expand_dims(normalized_img, axis=-1))
    return np.array(processed_images)

processed_images = preprocess_images(images)

# Split the data into training, cross-validation, and test sets
x_train, x_test, y_train, y_test = train_test_split(processed_images, labels
x_train, x_cv, y_train, y_cv = train_test_split(x_train, y_train, test_size=

# Function to save a single dataset to an HDF5 file
def save_dataset(h5_path, x_data, y_data, x_name, y_name):
    with h5py.File(h5_path, 'w') as h5f:
        # Create datasets for images and labels
        h5f.create_dataset(x_name, data=np.array(x_data, dtype='float32'))
        h5f.create_dataset(y_name, data=np.array(y_data, dtype='int64'))
        h5f.create_dataset('list_classes', data=np.array(['NORMAL', 'PNEUMON

# Specify paths for the training, testing, and cross-validation HDF5 files
train_h5_path = '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarte
test_h5_path = '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter
cv_h5_path = '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/D

# Save the training dataset
save_dataset(train_h5_path, x_train, y_train, 'x_train', 'y_train')

# Save the testing dataset
save_dataset(test_h5_path, x_test, y_test, 'x_test', 'y_test')

# Save the cross-validation dataset
save_dataset(cv_h5_path, x_cv, y_cv, 'x_cv', 'y_cv')

print("Training, testing, and cross-validation datasets have been saved to s
```

run everything below if u already have h5

In [ ]:

```
import h5py
from collections import Counter

# Function to inspect HDF5 file and check label distribution
def inspect_h5_distribution(file_path, label_dataset):
    # Open the HDF5 file in read mode
    with h5py.File(file_path, 'r') as h5f:
        # List all datasets and groups in the file
        print(f"\nContents of '{file_path}':")
        for key in h5f.keys():
            print(f" - {key}: {h5f[key].shape}, {h5f[key].dtype}")

        # Extract and decode the label dataset to calculate label distributi
        labels = h5f[label_dataset][:]
        label_counts = Counter(labels)

        # Map numeric labels to their actual names
        label_names = h5f['list_classes'][:].astype(str)
```

```
        distribution = {label_names[i]: count for i, count in label_counts.i

        print(f"Label Distribution ({label_dataset}):")
        for label, count in distribution.items():
            print(f" - {label}: {count} samples")

# Example usage: Provide paths to your HDF5 files
train_h5_path = '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarte
test_h5_path = '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter
cv_h5_path = '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/D

# Inspect each file and check the distribution of labels
inspect_h5_distribution(train_h5_path, 'y_train')
inspect_h5_distribution(test_h5_path, 'y_test')
inspect_h5_distribution(cv_h5_path, 'y_cv')
```

```
Contents of '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/De
ep Learning/Project/train_data2.h5':
 - list_classes: (2,), |S9
 - x_train: (3513, 64, 64, 1), float32
 - y_train: (3513,), int64
Label Distribution (y_train):
 - NORMAL: 951 samples
 - PNEUMONIA: 2562 samples

Contents of '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/De
ep Learning/Project/test_data2.h5':
 - list_classes: (2,), |S9
 - x_test: (1172, 64, 64, 1), float32
 - y_test: (1172,), int64
Label Distribution (y_test):
 - PNEUMONIA: 866 samples
 - NORMAL: 306 samples

Contents of '/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/De
ep Learning/Project/cv_data2.h5':
 - list_classes: (2,), |S9
 - x_cv: (1171, 64, 64, 1), float32
 - y_cv: (1171,), int64
Label Distribution (y_cv):
 - PNEUMONIA: 845 samples
 - NORMAL: 326 samples
```

In [ ]:

```
# find your current directory
import os
curDir = os.getcwd()
print(curDir)
```

```
/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/Deep Learning/P
roject
```

In [ ]:

```
# import
import numpy as np
import matplotlib.pyplot as plt
import h5py
```

```
import scipy
from PIL import Image
from scipy import ndimage
#from lr_utils import load_dataset

%matplotlib inline
```

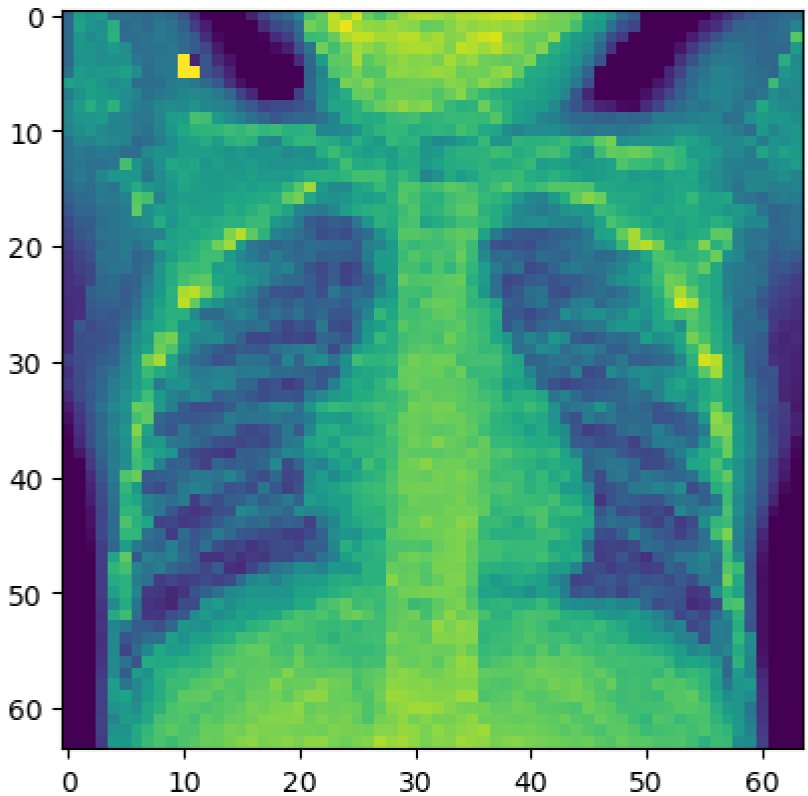
In [ ]:
```
import numpy as np
import h5py

def load_dataset():
    train_dataset = h5py.File('train_data2.h5', "r")
    train_set_x_orig = np.array(train_dataset["x_train"][:]) # your train se
    train_set_y_orig = np.array(train_dataset["y_train"][:]) # your train se

    test_dataset = h5py.File('test_data2.h5', "r")
    test_set_x_orig = np.array(test_dataset["x_test"][:]) # your test set fe
    test_set_y_orig = np.array(test_dataset["y_test"][:]) # your test set la

    cv_dataset = h5py.File('cv_data2.h5', "r")
    cv_set_x_orig = np.array(cv_dataset["x_cv"][:]) # your cv set features
    cv_set_y_orig = np.array(cv_dataset["y_cv"][:]) # your cv set labels

    classes = np.array(test_dataset["list_classes"][:]) # the list of classe

    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]))
    cv_set_y_orig = cv_set_y_orig.reshape((1, cv_set_y_orig.shape[0]))


    return train_set_x_orig, train_set_y_orig, test_set_x_orig, test_set_y_c
```

1) A logistic regression model (no hidden layers).

In [ ]:
```
# Loading the data
train_set_x_orig, train_set_y, test_set_x_orig, test_set_y, cv_set_x_orig, c
```

In [ ]:
```
# Example of a picture
index = np.random.randint(0, 299)
plt.imshow(train_set_x_orig[index])
print ("y = " + str(train_set_y[:, index]) + ", it's a '" + classes[np.squee
#Feel free also to change the index value and re-run to see other images.
```

```
y = [0], it's a 'NORMAL' picture.
```



In [ ]:
```
# Find the values below:
m_train = train_set_x_orig.shape[0]
m_test = test_set_x_orig.shape[0]
num_px = train_set_x_orig.shape[1]


print ("Number of training examples: m_train = " + str(m_train))
print ("Number of testing examples: m_test = " + str(m_test))
print ("Height/Width of each image: num_px = " + str(num_px))
print ("Each image is of size: (" + str(num_px) + ", " + str(num_px) + ", 3)
print ("train_set_x shape: " + str(train_set_x_orig.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
print ("test_set_x shape: " + str(test_set_x_orig.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
```

```
Number of training examples: m_train = 3513
Number of testing examples: m_test = 1172
Height/Width of each image: num_px = 64
Each image is of size: (64, 64, 3)
train_set_x shape: (3513, 64, 64, 1)
train_set_y shape: (1, 3513)
test_set_x shape: (1172, 64, 64, 1)
test_set_y shape: (1, 1172)
```

In [ ]:
```
# Reshape the training and test examples
train_set_x_flatten = train_set_x_orig.reshape(train_set_x_orig.shape[0],-1)
test_set_x_flatten = test_set_x_orig.reshape(test_set_x_orig.shape[0],-1).T
cv_set_x_flatten = cv_set_x_orig.reshape(cv_set_x_orig.shape[0],-1).T

print ("train_set_x_flatten shape: " + str(train_set_x_flatten.shape))
print ("train_set_y shape: " + str(train_set_y.shape))
```

```
print ("test_set_x_flatten shape: " + str(test_set_x_flatten.shape))
print ("test_set_y shape: " + str(test_set_y.shape))
print ("cv_set_x_flatten shape: " + str(cv_set_x_flatten.shape))
print ("cv_set_Y shape: " + str(cv_set_y.shape))
print ("sanity check after reshaping: " + str(train_set_x_flatten[0:5,0]))
```

```
train_set_x_flatten shape: (4096, 3513)
train_set_y shape: (1, 3513)
test_set_x_flatten shape: (4096, 1172)
test_set_y shape: (1, 1172)
cv_set_x_flatten shape: (4096, 1171)
cv_set_Y shape: (1, 1171)
sanity check after reshaping: [0.6039216  0.58431375 0.63529414 0.36862746
0.42352942]
```

In [ ]:
```
train_set_x = train_set_x_flatten/255.
test_set_x = test_set_x_flatten/255.
cv_set_x = cv_set_x_flatten/255.
```

In [ ]:
```
# This function creates a vector of zeros of shape (dim, 1) for w and initia
# dim -- size of the w vector we want (or number of parameters in this case)
# This function returns:
# w -- initialized vector of shape (dim, 1)
# b -- initialized scalar (corresponds to the bias)

def initialize_with_zeros(dim):
    w = np.zeros((dim,1))
    b = 0

    assert(w.shape == (dim, 1))
    assert(isinstance(b, float) or isinstance(b, int))

    return w, b
```

In [ ]:
```
# Helper function
def sigmoid(z):
    s = 1/(1+np.exp(-z))
    return s
```

In [ ]:
```
# w -- weights, a numpy array of size (num_px * num_px * 3, 1)
# b -- bias, a scalar
# X -- data of size (num_px * num_px * 3, number of examples)
# Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, n
# cost -- negative log-likelihood cost for logistic regression
# Dw -- gradient of the loss with respect to w, thus same shape as w
# Db -- gradient of the loss with respect to b, thus same shape as b
# the function returns the cost and gradients (Dw and Db)

def propagate(w, b, X, Y):

    m = X.shape[1]

    # FORWARD PROPAGATION (FROM X TO COST)
    A = sigmoid(np.dot(w.T,X)+b)
    cost = np.sum(-(Y*np.log(A)+(1-Y)*np.log(1-A)))/m
```

```python
    # BACKWARD PROPAGATION (TO FIND GRAD)
    Dw = 1/m * np.dot(X,(A-Y).T)
    Db = 1/m * np.sum(A-Y)

    assert(Dw.shape == w.shape)
    assert(Db.dtype == float)
    cost = np.squeeze(cost)
    assert(cost.shape == ())

    grads = {"Dw": Dw,
             "Db": Db}

    return grads, cost
```

In [ ]:

```python
# This function optimizes w and b by running a gradient descent algorithm
# w -- weights, a numpy array of size (num_px * num_px * 3, 1)
# b -- bias, a scalar
# X -- data of size (num_px * num_px * 3, number of examples)
# Y -- true "label" vector (containing 0 if non-cat, 1 if cat) of size (1, n

# num_iterations -- number of iterations of the optimization loop
# learning_rate -- learning rate of the gradient descent update rule
# print_cost -- True to print the loss every 100 steps

# this function returns:
# params -- dictionary containing the weights w and bias b
# grads -- dictionary containing the gradients of the weights and bias with
# costs -- list of all the costs computed during the optimization, this will


def optimize(w, b, X, Y, num_iterations, learning_rate, print_cost = False):

    costs = []

    for i in range(num_iterations):
        grads, cost = propagate(w, b, X, Y)


        # Retrieve derivatives from grads
        Dw = grads["Dw"]
        Db = grads["Db"]

        # update rule
        w = w-learning_rate*Dw
        b = b-learning_rate*Db

        # Record the costs
        if i % 100 == 0:
            costs.append(cost)

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

    params = {"w": w,
```

```
            "b": b}

    grads = {"Dw": Dw,
             "Db": Db}

    return params, grads, costs
```

In [ ]:

```
# Predict whether the label is 0 or 1 using learned logistic regression para
# w -- weights, a numpy array of size (num_px * num_px * 3, 1)
# b -- bias, a scalar
# X -- data of size (num_px * num_px * 3, number of examples)
# The function returns: Y_prediction -- a numpy array (vector) containing al
def predict(w, b, X):
    m = X.shape[1]
    Y_prediction = np.zeros((1,m))
    w = w.reshape(X.shape[0], 1)

    # Compute vector "A" predicting the probabilities of a cat being present
    A = sigmoid(np.dot(w.T,X)+b)

    for i in range(A.shape[1]):
        # Convert probabilities A[0,i] to actual predictions p[0,i]
        if (A[0,i] <= 0.5):
            Y_prediction[0,i] = 0
        else:
            Y_prediction[0,i] = 1
        # Y_prediction[0,i] = np.where(A[0,i]>0.5,1,0) ALTERNATIVE WAY



    assert(Y_prediction.shape == (1, m))

    return Y_prediction
```

In [ ]:

```
# This function builds the logistic regression model by calling the function
# X_train -- training set represented by a numpy array of shape (num_px * nu
# Y_train -- training labels represented by a numpy array (vector) of shape
# X_test -- test set represented by a numpy array of shape (num_px * num_px
# Y_test -- test labels represented by a numpy array (vector) of shape (1, m
# num_iterations -- hyperparameter representing the number of iterations to
# learning_rate -- hyperparameter representing the learning rate used in the
# print_cost -- Set to true to print the cost every 100 iterations

# This function returns d -- dictionary containing information about the mod

def model(X_train, Y_train, X_test, Y_test, num_iterations = 2000, learning_
    # initialize parameters with zeros
    w, b = initialize_with_zeros(X_train.shape[0])

    # Gradient descent
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iteratio

    # Retrieve parameters w and b from dictionary "parameters"
    w = parameters["w"]
    b = parameters["b"]
```

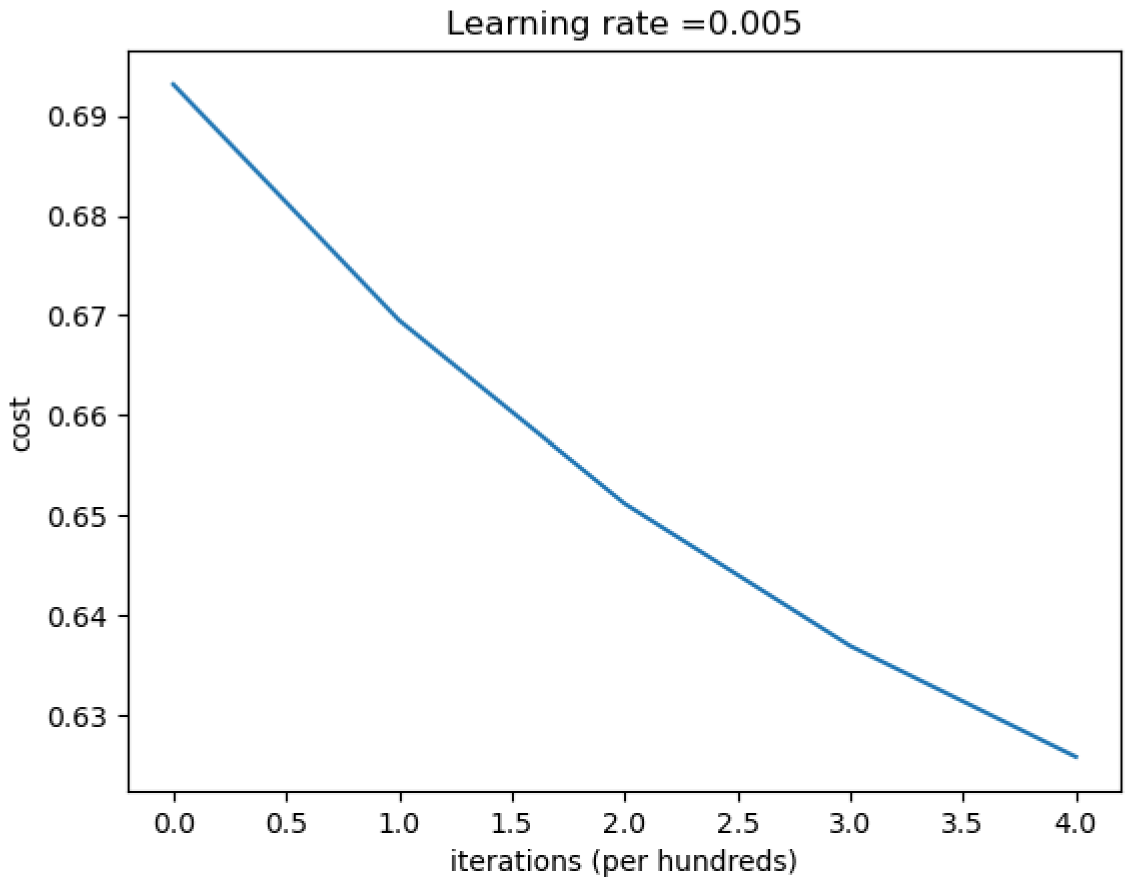
```python
    # Predict test/train set examples
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)


    # Print train/test Errors
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_tr
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_tes


    d = {"costs": costs,
         "Y_prediction_test": Y_prediction_test,
         "Y_prediction_train" : Y_prediction_train,
         "w" : w,
         "b" : b,
         "learning_rate" : learning_rate,
         "num_iterations": num_iterations}

    return d
```

TESTING ACCURACY

In [ ]:
```python
d = model(train_set_x, train_set_y, test_set_x, test_set_y, num_iterations =
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.669531
Cost after iteration 200: 0.651202
Cost after iteration 300: 0.636950
Cost after iteration 400: 0.625840
train accuracy: 72.92912040990606 %
test accuracy: 73.89078498293514 %
```

In [ ]:
```python
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```

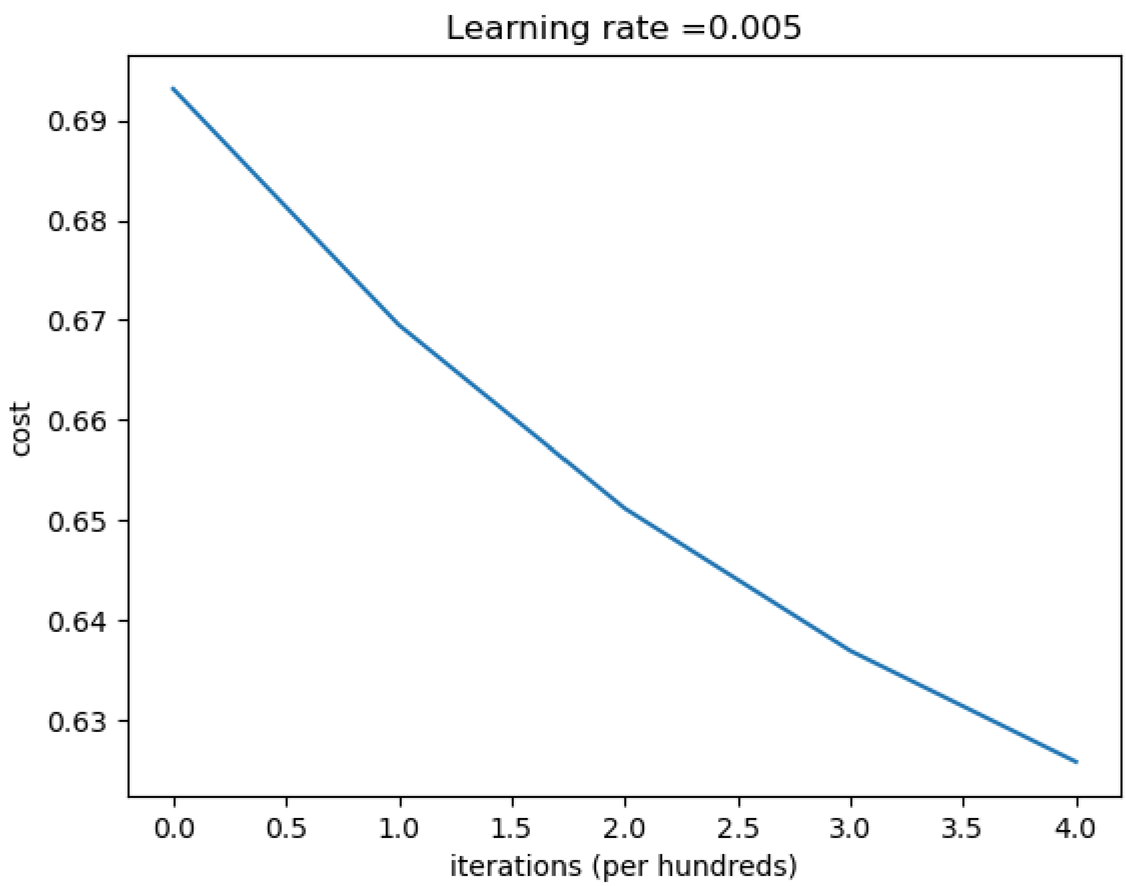


CV ACCURACY

In [ ]:
```
d = model(train_set_x, train_set_y, cv_set_x, cv_set_y, num_iterations = 500
```

```
Cost after iteration 0: 0.693147
Cost after iteration 100: 0.669531
Cost after iteration 200: 0.651202
Cost after iteration 300: 0.636950
Cost after iteration 400: 0.625840
train accuracy: 72.92912040990606 %
test accuracy: 72.16054654141759 %
```

In [ ]:
```
# Plot learning curve (with costs)
costs = np.squeeze(d['costs'])
plt.plot(costs)
plt.ylabel('cost')
plt.xlabel('iterations (per hundreds)')
plt.title("Learning rate =" + str(d["learning_rate"]))
plt.show()
```



2) A Neural Network with one hidden layer and four hidden units.

Use the ReLU activation function for the hidden layer and use the sigmoid activation function for the outcome layer.

In [ ]:
```
# Define the neural network structure (example: 2-layer neural network)
n_x = train_set_x.shape[0]  # num features
n_h = 4                     # num hidden units
n_y = 1                     # output layer

# Initialize parameters (simplified example)
def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(2)
    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))
    return {"W1": W1, "b1": b1, "W2": W2, "b2": b2}

parameters = initialize_parameters(n_x, n_h, n_y)
```

In [ ]:
```
import matplotlib.pyplot as plt
import numpy as np
import sklearn
```

```
import sklearn.datasets
import sklearn.linear_model

def plot_decision_boundary(model, X, y):
    # Set min and max values and give it some padding
    x_min, x_max = X[0, :].min() - 1, X[0, :].max() + 1
    y_min, y_max = X[1, :].min() - 1, X[1, :].max() + 1
    h = 0.01
    # Generate a grid of points with distance h between them
    xx, yy = np.meshgrid(np.arange(x_min, x_max, h), np.arange(y_min, y_max,
    # Predict the function value for the whole grid
    Z = model(np.c_[xx.ravel(), yy.ravel()])
    Z = Z.reshape(xx.shape)
    # Plot the contour and training examples
    plt.contourf(xx, yy, Z, cmap=plt.cm.Spectral)
    plt.ylabel('x2')
    plt.xlabel('x1')
    plt.scatter(X[0, :], X[1, :], c=y, cmap=plt.cm.Spectral)


def sigmoid(x):
    """
    Compute the sigmoid of x

    Arguments:
    x -- A scalar or numpy array of any size.

    Return:
    s -- sigmoid(x)
    """
    s = 1/(1+np.exp(-x))
    return s

def load_planar_dataset():
    np.random.seed(1)
    m = 400 # number of examples
    N = int(m/2) # number of points per class
    D = 2 # dimensionality
    X = np.zeros((m,D)) # data matrix where each row is a single example
    Y = np.zeros((m,1), dtype='uint8') # labels vector (0 for red, 1 for blu
    a = 4 # maximum ray of the flower

    for j in range(2):
        ix = range(N*j,N*(j+1))
        t = np.linspace(j*3.12,(j+1)*3.12,N) + np.random.randn(N)*0.2 # thet
        r = a*np.sin(4*t) + np.random.randn(N)*0.2 # radius
        X[ix] = np.c_[r*np.sin(t), r*np.cos(t)]
        Y[ix] = j

    X = X.T
    Y = Y.T

    return X, Y
```

In [ ]:

```
# Package imports
import numpy as np
```

```
import matplotlib.pyplot as plt
#from testCases_v2 import *
import sklearn
import sklearn.datasets
import sklearn.linear_model
#from planar_utils import plot_decision_boundary, sigmoid, load_planar_datas

%matplotlib inline

np.random.seed(1) # set a seed so that the results are consistent
```

In [ ]:

```
# X -- input dataset of shape (input size, number of examples)
# Y -- labels of shape (output size, number of examples)
# the function returns:
# n_x -- the size of the input layer
# n_h -- the size of the hidden layer
# n_y -- the size of the output layer

def layer_sizes(X, Y):

    n_x = X.shape[0] # size of input layer
    n_h = 4
    n_y = Y.shape[0] # size of output layer

    return (n_x, n_h, n_y)
```

In [ ]:

```
# n_x -- size of the input layer
# n_h -- size of the hidden layer
# n_y -- size of the output layer
# the function returns: params -- python dictionary containing your paramete
# 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)

def initialize_parameters(n_x, n_h, n_y):
    np.random.seed(2) # we set up a seed so that your output matches my valu

    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 [ ]:
```
# X -- input data of size (n_x, m)
# parameters -- python dictionary containing your parameters (output of init
# this function returns:
# A2 -- The sigmoid output of the second activation
# cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"


def forward_propagation(X, parameters):

    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]


    # Implement Forward Propagation to calculate A2 (probabilities)
    Z1 = np.dot(W1,X) + b1
    #A1 = np.tanh(Z1)
    A1 = np.maximum(0,Z1) # for RELU
    Z2 = np.dot(W2, A1) + b2
    A2 = sigmoid(Z2)


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

    cache = {"Z1": Z1,
             "A1": A1,
             "Z2": Z2,
             "A2": A2}

    return A2, cache
```

In [ ]:
```
# A2 -- The sigmoid output of the second activation, of shape (1, number of
# Y -- "true" labels vector of shape (1, number of examples)
# parameters -- python dictionary containing your parameters W1, b1, W2 and
# this function returns: cost

def compute_cost(A2, Y, parameters):

    m = Y.shape[1] # number of example

    # Compute the cost
    logprobs = np.multiply(np.log(A2),Y) + np.multiply(np.log(1-A2),(1-Y))
    cost = -(1/m) * np.sum(logprobs)

    cost = float(np.squeeze(cost))  # makes sure cost is the dimension we ex
                                    # E.g., turns [[17]] into 17
    assert(isinstance(cost, float))

    return cost
```

In [ ]:
```
# parameters -- python dictionary containing our parameters
# cache -- a dictionary containing "Z1", "A1", "Z2" and "A2".
# X -- input data of shape (2, number of examples)
```

```
# Y -- "true" labels vector of shape (1, number of examples)
# this function returns: grads -- python dictionary containing your gradient

def backward_propagation(parameters, cache, X, Y):

    m = X.shape[1]

    # First, retrieve W1 and W2 from the dictionary "parameters".
    W1 = parameters['W1']
    W2 = parameters['W2']

    # Retrieve also A1 and A2 from dictionary "cache".
    A1 = cache['A1']
    A2 = cache['A2']
    Z1 = cache['Z1']

    # Backward propagation: calculate dW1, db1, dW2, db2, corresponding to 6
    dZ2 = A2-Y
    dW2 = 1/m*np.dot(dZ2,A1.T)
    db2 = 1/m*np.sum(dZ2,axis=1,keepdims=True)
    #dZ1 = np.dot(W2.T,dZ2)*(1-np.power(A1,2))
    dZ1 = np.dot(W2.T,dZ2)*(Z1>0).astype(int)
    dW1 = 1/m*np.dot(dZ1,X.T)
    db1 = 1/m*np.sum(dZ1,axis=1,keepdims=True)

    grads = {"dW1": dW1,
             "db1": db1,
             "dW2": dW2,
             "db2": db2}

    return grads
```

In [ ]:

```
# This function updates parameters using the gradient descent update rule gi
# parameters -- python dictionary containing your parameters
# grads -- python dictionary containing your gradients
# This function returns python dictionary containing your updated parameters
def update_parameters(parameters, grads, learning_rate = 1.2):

    # Retrieve each parameter from the dictionary "parameters"
    W1 = parameters['W1']
    b1 = parameters['b1']
    W2 = parameters['W2']
    b2 = parameters['b2']

    # Retrieve each gradient from the dictionary "grads"
    dW1 = grads['dW1']
    db1 = grads['db1']
    dW2 = grads['dW2']
    db2 = grads['db2']

    # Update rule for each parameter
    W1 = W1 - learning_rate*dW1
    b1 = b1 - learning_rate*db1
    W2 = W2 - learning_rate*dW2
    b2 = b2 - learning_rate*db2
```

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

    return parameters
```

In [ ]:
```
# X -- dataset of shape (2, number of examples)
# Y -- labels of shape (1, number of examples)
# n_h -- size of the hidden layer
# num_iterations -- Number of iterations in gradient descent loop
# print_cost -- if True, print the cost every 1000 iterations
# this function returns parameters learnt by the model. They can then be use

def nn_model(X, Y, n_h, num_iterations = 10000, print_cost=False):

    np.random.seed(3)
    n_x = layer_sizes(X, Y)[0]
    n_y = layer_sizes(X, Y)[2]

    # Initialize parameters
    parameters = initialize_parameters(n_x, n_h, n_y)

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

        # Forward propagation. Inputs: "X, parameters". Outputs: "A2, cache"
        A2, cache = forward_propagation(X,parameters)

        # Cost function. Inputs: "A2, Y, parameters". Outputs: "cost".
        cost = compute_cost(A2,Y,parameters)

        # Backpropagation. Inputs: "parameters, cache, X, Y". Outputs: "grad
        grads = backward_propagation(parameters,cache,X,Y)

        # Gradient descent parameter update. Inputs: "parameters, grads". Ou
        parameters = update_parameters(parameters,grads)

        # Print the cost every 1000 iterations
        if print_cost and i % 1000 == 0:
            print ("Cost after iteration %i: %f" %(i, cost))

    return parameters
```

In [ ]:
```
# this function, using the learned parameters, predicts a class for each exa
# this function computes probabilities using forward propagation, and classi
# parameters -- python dictionary containing your parameters
# X -- input data of size (n_x, m)
# this function returns: predictions -- vector of predictions of our model (

def predict(parameters, X):

    A2, cache = forward_propagation(X, parameters)
    predictions = (A2>0.5)
```

```
    return predictions
```

TRAINING ACCURACY - 1 hidden layer 5 hidden units

In [ ]:
```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(train_set_x, train_set_y, n_h = 4, num_iterations = 50
# Print accuracy
predictions = predict(parameters, train_set_x)
print ('Accuracy: %d' % float((np.dot(train_set_y,predictions.T) + np.dot(1-
```

```
Cost after iteration 0: 0.693146
Cost after iteration 1000: 0.582105
Cost after iteration 2000: 0.583963
Cost after iteration 3000: 0.583963
Cost after iteration 4000: 0.583963
Accuracy: 72%
```

```
/var/folders/mw/bjc13fqn7l1b4fjjsfst7ry80000gn/T/ipykernel_6863/1972835148.p
y:5: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is
deprecated, and will error in future. Ensure you extract a single element fr
om your array before performing this operation. (Deprecated NumPy 1.25.)
  print ('Accuracy: %d' % float((np.dot(train_set_y,predictions.T) + np.dot
(1-train_set_y,1-predictions.T))/float(train_set_y.size)*100) + '%')
```

CV ACCURACY - 1 hidden layer 5 hidden units

In [ ]:
```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(train_set_x, train_set_y, n_h = 4, num_iterations = 50
# Print accuracy
predictions = predict(parameters, cv_set_x)
print ('Accuracy: %d' % float((np.dot(cv_set_y,predictions.T) + np.dot(1-cv_
```

```
Cost after iteration 0: 0.693146
Cost after iteration 1000: 0.582105
Cost after iteration 2000: 0.583963
Cost after iteration 3000: 0.583963
Cost after iteration 4000: 0.583963
Accuracy: 72%
```

```
/var/folders/mw/bjc13fqn7l1b4fjjsfst7ry80000gn/T/ipykernel_6863/4289306919.p
y:5: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is
deprecated, and will error in future. Ensure you extract a single element fr
om your array before performing this operation. (Deprecated NumPy 1.25.)
  print ('Accuracy: %d' % float((np.dot(cv_set_y,predictions.T) + np.dot(1-c
v_set_y,1-predictions.T))/float(cv_set_y.size)*100) + '%')
```

TESTING ACCURACY - 1 hidden layer 5 hidden units

In [ ]:
```
# Build a model with a n_h-dimensional hidden layer
parameters = nn_model(train_set_x, train_set_y, n_h = 4, num_iterations = 50
# Print accuracy
predictions = predict(parameters, test_set_x)
print ('Accuracy: %d' % float((np.dot(test_set_y,predictions.T) + np.dot(1-t
```

```
Cost after iteration 0: 0.693146
Cost after iteration 1000: 0.582105
Cost after iteration 2000: 0.583963
Cost after iteration 3000: 0.583963
Cost after iteration 4000: 0.583963
Accuracy: 73%
```

```
/var/folders/mw/bjc13fqn7l1b4fjjsfst7ry80000gn/T/ipykernel_6863/1434101315.p
y:5: DeprecationWarning: Conversion of an array with ndim > 0 to a scalar is
deprecated, and will error in future. Ensure you extract a single element fr
om your array before performing this operation. (Deprecated NumPy 1.25.)
  print ('Accuracy: %d' % float((np.dot(test_set_y,predictions.T) + np.dot(1
-test_set_y,1-predictions.T))/float(test_set_y.size)*100) + '%')
```

3) A Neural Network with two hidden layers.

The first hidden layer with seven hidden units, and the second hidden layer with four hidden units. Use the ReLU activation function for both hidden layers and use the sigmoid activation function for the outcome layer.

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
import h5py


def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the b
    """
```

```python
    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache


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 efficient

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

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

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, 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 efficient

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

    Z = cache

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

    assert (dZ.shape == Z.shape)

    return dZ


def load_data():
    train_dataset = h5py.File('datasets/train_catvnoncat.h5', "r")
    train_set_x_orig = np.array(train_dataset["train_set_x"][:]) # your trai
    train_set_y_orig = np.array(train_dataset["train_set_y"][:]) # your trai
```

```python
    test_dataset = h5py.File('datasets/test_catvnoncat.h5', "r")
    test_set_x_orig = np.array(test_dataset["test_set_x"][:]) # your test se
    test_set_y_orig = np.array(test_dataset["test_set_y"][:]) # your test se

    classes = np.array(test_dataset["list_classes"][:]) # the list of classe

    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_o


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


def initialize_parameters_deep(layer_dims):
    """
    Arguments:
    layer_dims -- python array (list) containing the dimensions of each laye

    Returns:
    parameters -- python dictionary containing your parameters "W1", "b1", .
                    Wl -- weight matrix of shape (layer_dims[l], layer_dims[
```

```
                bl -- bias vector of shape (layer_dims[l], 1)
    """

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

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l], layer_dims
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters

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
    W -- weights matrix: numpy array of shape (size of current layer, size c
    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 pa
    cache -- a python dictionary containing "A", "W" and "b" ; stored for co
    """

    Z = W.dot(A) + b

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

    return Z, cache

def linear_activation_forward(A_prev, W, b, activation):
    """
    Implement the forward propagation for the LINEAR->ACTIVATION layer

    Arguments:
    A_prev -- activations from previous layer (or input data): (size of prev
    W -- weights matrix: numpy array of shape (size of current layer, size c
    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

    Returns:
    A -- the output of the activation function, also called the post-activat
    cache -- a python dictionary containing "linear_cache" and "activation_c
             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

def L_model_forward(X, parameters):
    """
    Implement forward propagation for the [LINEAR->RELU]*(L-1)->LINEAR->SIGM

    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,
                the cache of linear_sigmoid_forward() (there is one, indexed
    """

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

    # Implement [LINEAR -> RELU]*(L-1). Add "cache" to the "caches" list.
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev, parameters['W' + str(l)
        caches.append(cache)

    # Implement LINEAR -> SIGMOID. Add "cache" to the "caches" list.
    AL, cache = linear_activation_forward(A, parameters['W' + str(L)], param
    caches.append(cache)

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

    return AL, caches

def compute_cost(AL, Y):
    """
    Implement the cost function defined by equation (7).

    Arguments:
    AL -- probability vector corresponding to your label predictions, shape
    Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat

    Returns:
    cost -- cross-entropy cost
```

```
    """
    
    m = Y.shape[1]

    # Compute loss from aL and y.
    cost = (1./m) * (-np.dot(Y,np.log(AL).T) - np.dot(1-Y, np.log(1-AL).T))
    
    cost = np.squeeze(cost)      # To make sure your cost's shape is what we
    assert(cost.shape == ())
    
    return cost

def linear_backward(dZ, cache):
    """
    Implement the linear portion of backward propagation for a single layer
    
    Arguments:
    dZ -- Gradient of the cost with respect to the linear output (of current
    cache -- tuple of values (A_prev, W, b) coming from the forward propagat
    
    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the p
    dW -- Gradient of the cost with respect to W (current layer l), same sha
    db -- Gradient of the cost with respect to b (current layer l), same sha
    """
    A_prev, W, b = cache
    m = A_prev.shape[1]

    dW = 1./m * np.dot(dZ,A_prev.T)
    db = 1./m * np.sum(dZ, axis = 1, keepdims = True)
    dA_prev = np.dot(W.T,dZ)
    
    assert (dA_prev.shape == A_prev.shape)
    assert (dW.shape == W.shape)
    assert (db.shape == b.shape)
    
    return dA_prev, dW, db

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 c
    activation -- the activation to be used in this layer, stored as a text
    
    Returns:
    dA_prev -- Gradient of the cost with respect to the activation (of the p
    dW -- Gradient of the cost with respect to W (current layer l), same sha
    db -- Gradient of the cost with respect to b (current layer l), same sha
    """
    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

def L_model_backward(AL, Y, caches):
    """
    Implement the backward propagation for the [LINEAR->RELU] * (L-1) -> LIN

    Arguments:
    AL -- probability vector, output of the forward propagation (L_model_for
    Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
    caches -- list of caches containing:
                every cache of linear_activation_forward() with "relu" (ther
                the cache of linear_activation_forward() with "sigmoid" (the

    Returns:
    grads -- A dictionary with the gradients
             grads["dA" + str(l)] = ...
             grads["dW" + str(l)] = ...
             grads["db" + str(l)] = ...
    """
    grads = {}
    L = len(caches) # the number of layers
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "AL, Y, caches". Outp
    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = lin

    for l in reversed(range(L-1)):
        # lth layer: (RELU -> LINEAR) gradients.
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = linear_activation_backward(grads["d
        grads["dA" + str(l)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads

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_

    Returns:
```

```
    parameters -- python dictionary containing your updated parameters
                  parameters["W" + str(l)] = ...
                  parameters["b" + str(l)] = ...
    """

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

    # Update rule for each parameter. Use a for loop.
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_r
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_r

    return parameters

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

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

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

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

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


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

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

    return p

def print_mislabeled_images(classes, X, y, p):
    """
    Plots images where predictions and truth were different.
    X -- dataset
    y -- true labels
    p -- predictions
    """
    a = p + y
```

```
    mislabeled_indices = np.asarray(np.where(a == 1))
    plt.rcParams['figure.figsize'] = (40.0, 40.0) # set default size of plot
    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")
```

In [ ]:

```
import numpy as np

def sigmoid(Z):
    """
    Implements the sigmoid activation in numpy

    Arguments:
    Z -- numpy array of any shape

    Returns:
    A -- output of sigmoid(z), same shape as Z
    cache -- returns Z as well, useful during backpropagation
    """

    A = 1/(1+np.exp(-Z))
    cache = Z

    return A, cache

def relu(Z):
    """
    Implement the RELU function.

    Arguments:
    Z -- Output of the linear layer, of any shape

    Returns:
    A -- Post-activation parameter, of the same shape as Z
    cache -- a python dictionary containing "A" ; stored for computing the b
    """

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache


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 efficient

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

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

    # When z <= 0, you should set dz to 0 as well.
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, 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 efficient

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

    Z = cache

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

    assert (dZ.shape == Z.shape)

    return dZ
```

In [ ]:

```
def initialize_parameters(n_x, n_h1, n_h2, n_y):
    np.random.seed(1)
    W1 = np.random.randn(n_h1, n_x) * 0.01
    b1 = np.zeros((n_h1, 1))
    W2 = np.random.randn(n_h2, n_h1) * 0.01
    b2 = np.zeros((n_h2, 1))
    W3 = np.random.randn(n_y, n_h2) * 0.01
    b3 = np.zeros((n_y, 1))

    parameters = {"W1": W1, "b1": b1, "W2": W2, "b2": b2, "W3": W3, "b3": b3
    return parameters
```

In [ ]:

```
# layer_dims -- python array (list) containing the dimensions of each layer
# this function returns parameters -- python dictionary containing your para
# Wl -- weight matrix of shape (layer_dims[l], layer_dims[l-1])
# bl -- bias vector of shape (layer_dims[l], 1)
```

```
def initialize_parameters_deep(layer_dims):

    np.random.seed(3)
    parameters = {}
    L = len(layer_dims)            # number of layers in the network

    for l in range(1, L):
        parameters['W' + str(l)] = np.random.randn(layer_dims[l],layer_dims[
        parameters['b' + str(l)] = np.zeros((layer_dims[l],1))


        assert(parameters['W' + str(l)].shape == (layer_dims[l], layer_dims[
        assert(parameters['b' + str(l)].shape == (layer_dims[l], 1))


    return parameters
```

In [ ]:

```
# linear_forward implements the linear part of a layer's forward propagation
# A -- activations from previous layer (or input data): (size of previous la
# W -- weights matrix: numpy array of shape (size of current layer, size of
# b -- bias vector, numpy array of shape (size of the current layer, 1)
# this function returns: Z -- the input of the activation function, also cal
# this function returns: cache -- a python tuple containing "A", "W" and "b"
def linear_forward(A, W, b):

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

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

    return Z, cache
```

In [ ]:

```
# A_prev -- activations from previous layer (or input data): (size of previo
# W -- weights matrix: numpy array of shape (size of current layer, size of
# 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

#this function returns: A -- the output of the activation function, also cal
#this function returns: cache -- a python tuple containing "linear_cache" an

def linear_activation_forward(A_prev, W, b, activation):

    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)
```

```python
    return A, cache
```

In [ ]:
```python
# L_model_forward implements forward propagation for the [LINEAR->RELU]*(L-1
# X -- data, numpy array of shape (input size, number of examples)
# parameters -- output of initialize_parameters_deep()

# this function returns: AL -- last post-activation value
# this function returns: caches -- list of caches containing: every cache of
# (there are L-1 of them, indexed from 0 to L-1)
def L_model_forward(X, parameters):

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

    # Implement [LINEAR -> RELU]*(L-1).
    for l in range(1, L):
        A_prev = A
        A, cache = linear_activation_forward(A_prev = A_prev, W = parameters
                                             activation='relu')
        caches.append(cache)                  # Add "cache" to the "caches"

    # Implement LINEAR -> SIGMOID.
    AL, cache = linear_activation_forward(A,parameters['W'+str(L)],parameter
    caches.append(cache)                      # Add "cache" to the "caches"

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

    return AL, caches
```

In [ ]:
```python
# AL -- probability vector corresponding to your label predictions, shape (1
# Y -- true "label" vector (for example: containing 0 if non-cat, 1 if cat),
# this function returns the cost

def compute_cost(AL, Y):

    m = Y.shape[1]

    # Compute loss from aL and y.
    cost = -1/m * np.sum(Y*np.log(AL)+(1-Y)*np.log(1-AL))

    cost = np.squeeze(cost)      # To make sure your cost's shape is what we
    assert(cost.shape == ())

    return cost
```

In [ ]:
```python
# linear_backward implements the linear portion of backward propagation for
# dZ -- Gradient of the cost with respect to the linear output (of current l
# cache -- tuple of values (A_prev, W, b) coming from the forward propagatio

# this function returns:
# dA_prev -- Gradient of the cost with respect to the activation (of the pre
# dW -- Gradient of the cost with respect to W (current layer l), same shape
# db -- Gradient of the cost with respect to b (current layer l), same shape
```

```python
def linear_backward(dZ, cache):

    A_prev, W, b = cache
    m = A_prev.shape[1]

    dA_prev = np.dot(W.T, dZ)
    dW = 1/m * np.dot(dZ,A_prev.T)
    db = 1/m * 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 [ ]:
```python
# linear_activation_backward implements the backward propagation for the LIN
# dA -- post-activation gradient for current layer l
# cache -- tuple of values (linear_cache, activation_cache) we store for com
# activation -- the activation to be used in this layer, stored as a text st
# this function returns:
# dA_prev -- Gradient of the cost with respect to the activation (of the pre
# dW -- Gradient of the cost with respect to W (current layer l), same shape
# db -- Gradient of the cost with respect to b (current layer l), same shape

def linear_activation_backward(dA, cache, activation):

    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 [ ]:
```python
# L_model_backward implement the backward propagation for the [LINEAR->RELU]
# AL -- probability vector, output of the forward propagation (L_model_forwa
# Y -- true "label" vector (containing 0 if non-cat, 1 if cat)
# caches -- list of caches containing:
# every cache of linear_activation_forward() with "relu" (it's caches[l], fo
# the cache of linear_activation_forward() with "sigmoid" (it's caches[L-1])

# this function returns: grads -- A dictionary with the gradients:
# grads["dA" + str(l)] = ...
# grads["dW" + str(l)] = ...
# grads["db" + str(l)] = ...
def L_model_backward(AL, Y, caches):

    grads = {}
    L = len(caches) # the number of layers
```

```
    m = AL.shape[1]
    Y = Y.reshape(AL.shape) # after this line, Y is the same shape as AL

    # Initializing the backpropagation
    dAL = -(np.divide(Y,AL)-np.divide(1-Y,1-AL))

    # Lth layer (SIGMOID -> LINEAR) gradients. Inputs: "dAL, current_cache".
    current_cache = caches[L-1]
    grads["dA" + str(L-1)], grads["dW" + str(L)], grads["db" + str(L)] = lir


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

    return grads
```

In [ ]:
```
# update_parameters updates parameters using gradient descent
# parameters -- python dictionary containing your parameters
# grads -- python dictionary containing your gradients, output of L_model_ba

# this function returns parameters -- python dictionary containing your upda
# parameters["W" + str(l)] = ...
# parameters["b" + str(l)] = ...

def update_parameters(parameters, grads, learning_rate):

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

    # Update rule for each parameter. Use a for loop.
    for l in range(L):
        parameters["W" + str(l+1)] = parameters["W" + str(l+1)] - learning_r
        parameters["b" + str(l+1)] = parameters["b" + str(l+1)] - learning_r

    return parameters
```

In [ ]:
```
# find your current directory
import os
curDir = os.getcwd()
print(curDir)
```

/Users/andrewgatchalian/Documents/UCI MSBA 24/Spring Quarter/Deep Learning/P
roject

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

```
from scipy import ndimage
#from dnn_app_utils_v3 import *

%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'

np.random.seed(1)
```

In [ ]:
```
### CONSTANTS DEFINING THE MODEL ####
n_x = 4096     # 64 * 64 * 1
n_h1 = 7
n_h2 = 4
n_y = 1
layers_dims = (n_x, n_h1, n_h2, n_y)

# hw 2 change this *****
```

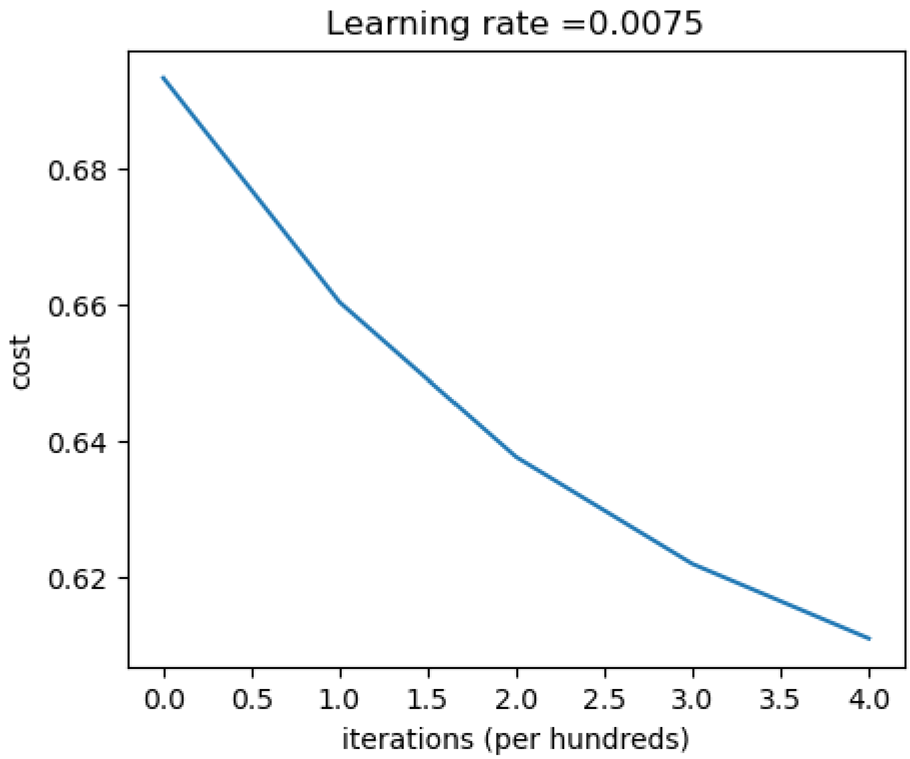
In [ ]:
```
# two_layer_model implements a two-layer neural network: LINEAR->RELU->LINEA
# X -- input data, of shape (n_x, number of examples)
# Y -- true "label" vector (containing 1 if cat, 0 if non-cat), of shape (1,
# layers_dims -- dimensions of the layers (n_x, n_h, n_y)
# num_iterations -- number of iterations of the optimization loop
# learning_rate -- learning rate of the gradient descent update rule
# print_cost -- If set to True, this will print the cost every 100 iteration
# this function returns: parameters -- a dictionary containing W1, W2, b1, a

def two_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iteration

    np.random.seed(1)
    grads = {}
    costs = []                              # to keep track of the cost
    m = X.shape[1]                           # number of examples
    (n_x, n_h1, n_h2, n_y) = layers_dims

    # Initialize parameters dictionary, by calling one of the functions you'
    parameters = initialize_parameters(n_x, n_h1, n_h2, n_y)

    # Get W1, b1, W2 and b2 from the dictionary parameters.
    W1 = parameters["W1"]
    b1 = parameters["b1"]
    W2 = parameters["W2"]
    b2 = parameters["b2"]
    W3 = parameters["W3"]
    b3 = parameters["b3"]

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

        # Forward propagation: LINEAR -> RELU -> LINEAR -> SIGMOID. Inputs:
        A1, cache1 = linear_activation_forward(X,W1,b1,activation='relu')
        A2, cache2 = linear_activation_forward(A1, W2, b2, activation='relu'
```

```python
        A3, cache3 = linear_activation_forward(A2, W3, b3, activation='sigmo

        # Compute cost
        cost = compute_cost(A3,Y)

        # Initializing backward propagation
        dA3 = - (np.divide(Y, A3) - np.divide(1 - Y, 1 - A3))

        # Backward propagation. Inputs: "dA2, cache2, cache1". Outputs: "dA1
        dA2, dW3, db3 = linear_activation_backward(dA3, cache3, activation='
        dA1, dW2, db2 = linear_activation_backward(dA2, cache2, activation='
        dA0, dW1, db1 = linear_activation_backward(dA1, cache1, activation='


        # Set grads['dWl'] to dW1, grads['db1'] to db1, grads['dW2'] to dW2,
        grads['dW1'] = dW1
        grads['db1'] = db1
        grads['dW2'] = dW2
        grads['db2'] = db2
        grads['dW3'] = dW3
        grads['db3'] = db3

        # 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"]
        W3 = parameters["W3"]
        b3 = parameters["b3"]

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

    # plot the cost

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

    return parameters
```

In [ ]:

```python
parameters = two_layer_model(train_set_x, train_set_y, layers_dims = (n_x, r
```

```
Cost after iteration 0: 0.6931471896629926
Cost after iteration 100: 0.660234171060722
Cost after iteration 200: 0.6375666986030382
Cost after iteration 300: 0.6218751110262329
Cost after iteration 400: 0.6109409235630995
```



TRAINING

```
In [ ]: predictions_train = predict(train_set_x, train_set_y, parameters)
```

```
Accuracy: 0.7292912040990607
```

TESTING

```
In [ ]: predictions_test = predict(test_set_x, test_set_y, parameters)
```

```
Accuracy: 0.7389078498293515
```

CROSS VALID

```
In [ ]: predictions_cv = predict(cv_set_x, cv_set_y, parameters)
```

```
Accuracy: 0.7216054654141759
```