Problem Statement: Implement the Multi Layer Perceptron Learning algorithm.

Introduction: Perceptron Learning algorithm is used for supervised learning for binary classification. In this problem, a Multi layer perceptron learning algorithm was used to perform classification of a given dataset. Multi Layer Perceptron means more than one hidden layer is used for classification. Here the back propagation algorithm is used for supervised learning. Using this back propagation algorithm the weights at each node of each layer and the bias value at each layer is updated. The main target is to minimize the cost function utilizing the back propagation algorithm.

Dataset Description: For this classification, a manually generated numpy array dataset was use. The Number of training Examples was 500, Number of testing examples was 100, Each data was of shape (64, 64, 3). Original training data shape was (500, 64, 64, 3), training label shape (1, 500). Original testing data shape was (100, 64, 64, 3), testing label shape (1, 100).

Data Preprocessing: The dataset was flattened. The new shape for training data was (12288, 500) and shape of testing data was (12288, 100). Then the dataset was used to perform the classification.

Methodology: A 5 layer model including input and output layer was used to perform the classification. The number of nodes in each hidden layer were 4, 3, and 2 respectively. A learning rate of 0.0075 was used initially and the model was trained for 2500 iterations.

Code:

```
import time
#import tensorflow as tf
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'

%load_ext autoreload
%autoreload 2

np.random.seed(1)


def parameters_initialization(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[l - 1]) * 0.01
        parameters['b' + str(l)] = np.zeros((layer_dims[l], 1))

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

    return parameters


def forward(A, W, b):

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

    assert(Z.shape == (W.shape[0], A.shape[1]))          #checks the shape whether they are same or not
    cache = (A, W, b)

    return Z, cache
```

```
def sigmoid(Z):

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

    return A, cache

def relu(Z):

    A = np.maximum(0,Z)

    assert(A.shape == Z.shape)

    cache = Z
    return A, cache

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

    if activation == "sigmoid":
        Z, linear_cache = forward(A_prev, W, b)
        A, activation_cache = sigmoid(Z)

    elif activation == "relu":
        Z, linear_cache = 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):
```

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

    for l in range(1, L):
        A_prev = A
        A, cache = forwardActivation(A_prev, parameters['W' + str(l)], parameters['b' + str(l)], activation='relu')   #[LINEAR -> RELU]*(L-1)
        caches.append(cache)
    AL, cache = forwardActivation(A, parameters['W' + str(L)], parameters['b' + str(L)], activation='sigmoid')   #LINEAR -> SIGMOID
    caches.append(cache)

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

    return AL, caches

def costFunction(AL, Y):

    m = Y.shape[1]
    cost = (-1 / m) * np.sum(np.multiply(Y, np.log(AL)) + np.multiply(1 - Y, np.log(1 - AL)))

    cost = np.squeeze(cost)      # To make sure the cost's shape is as expected. (converts [[10]] into 10).
    assert(cost.shape == ())

    return cost


def backward(dZ, cache):

    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 relu_backward(dA, cache):

    Z = cache
    dZ = np.array(dA, copy=True)
    dZ[Z <= 0] = 0

    assert (dZ.shape == Z.shape)

    return dZ

def sigmoid_backward(dA, cache):

    Z = cache

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

    assert (dZ.shape == Z.shape)

    return dZ

def L_model_backward(AL, Y, caches):
    grads = {}
    L = len(caches) # the number of layers
```

```
    m = AL.shape[1]
    Y = Y.reshape(AL.shape)
    dAL = dAL = - (np.divide(Y, AL) - np.divide(1 - Y, 1 - AL))

    current_cache = caches[-1]
    grads["dA" + str(L)], grads["dW" + str(L)], grads["db" + str(L)] = backward(sigmoid_backward(dAL, current_cache[1]), current_cache[0])

    for l in reversed(range(L-1)):
        current_cache = caches[l]
        dA_prev_temp, dW_temp, db_temp = backward(sigmoid_backward(dAL, current_cache[1]), current_cache[0])
        grads["dA" + str(l + 1)] = dA_prev_temp
        grads["dW" + str(l + 1)] = dW_temp
        grads["db" + str(l + 1)] = db_temp

    return grads


def updateParameters(parameters, grads, learning_rate):

    L = len(parameters) // 2 # number of layers in the neural network. Divided by 2 as two parameters per layer(Weight & Bias)

    for l in range(L):
        parameters["W" + str(l + 1)] = parameters["W" + str(l + 1)] - learning_rate * grads["dW" + str(l + 1)]
        parameters["b" + str(l + 1)] = parameters["b" + str(l + 1)] - learning_rate * grads["db" + str(l + 1)]

    return parameters


train_x_orig = np.random.randint(0,255,(500, 64, 64, 3))
train_y = np.random.randint(0,1,(1, 500))
test_x_orig = np.random.randint(0,255,(100, 64, 64, 3))
test_y = np.random.randint(0,1,(1, 100))
```

```
# 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))

# 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

train_x = train_x_flatten/255.          # Standardize data to have feature values between 0 and 1.
test_x = test_x_flatten/255.

print ("train_x's shape: " + str(train_x.shape))
print ("test_x's shape: " + str(test_x.shape))

layers_dims = [12288, 4, 3, 2, 1]

def predict(X, y, parameters):

    m = X.shape[1]   #Takes the total number of data
    n = len(parameters) // 2 # number of layers in the neural network. Divided by 2 as two parameters per layer(Weight & Bias)
    p = np.zeros((1,m))   #initializing variable to store the prediction

    probas, caches = L_model_forward(X, parameters)         #calculates the value of non-linear activation
```

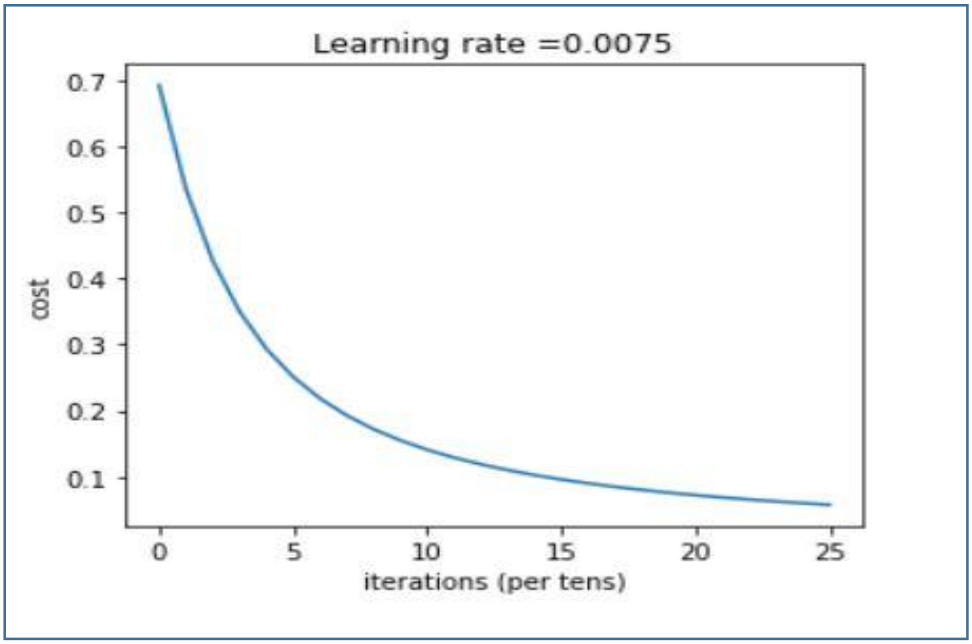
```
    for i in range(0, probas.shape[1]):
        if probas[0,i] > 0.5:
            p[0,i] = 1
        else:
            p[0,i] = 0

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

    return np.sum((p == y)/m)



def L_layer_model(X, Y, layers_dims, learning_rate = 0.0075, num_iterations = 3000, print_cost=False):#lr was 0.009
    np.random.seed(1)
    pred_trains=[]
    pred_tests=[]
    costs = []                                           # Store cost value
    parameters = parameters_initialization(layers_dims)

    # Loop (gradient descent)
    for i in range(0, num_iterations+1):
        AL, caches = L_model_forward(X, parameters)                 #Forward propagation
        cost = costFunction(AL, Y)                               # Compute cost
        grads = L_model_backward(AL, Y, caches)                  # Backward propagation

        pred_train = predict(train_x, train_y, parameters)
        pred_trains.append(pred_train)

        pred_test = predict(test_x, test_y, parameters)
        pred_tests.append(pred_test)

        parameters = updateParameters(parameters, grads, learning_rate)       # Update parameters
```

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


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

    plt.plot(np.squeeze(pred_trains))
    plt.plot(np.squeeze(pred_tests))
    plt.ylabel('Accuracy')
    plt.xlabel('number of iterations')
    #plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    plt.plot(np.squeeze(pred_trains))
    #plt.plot(np.squeeze(pred_tests))
    plt.ylabel('training accuracy')
    plt.xlabel('number of iterations')
    #plt.title("Learning rate =" + str(learning_rate))
    plt.show()

    #plt.plot(np.squeeze(pred_trains))
    plt.plot(np.squeeze(pred_tests))
    plt.ylabel('testing accuracy')
    plt.xlabel('number of iterations')
    #plt.title("Learning rate =" + str(learning_rate))
```

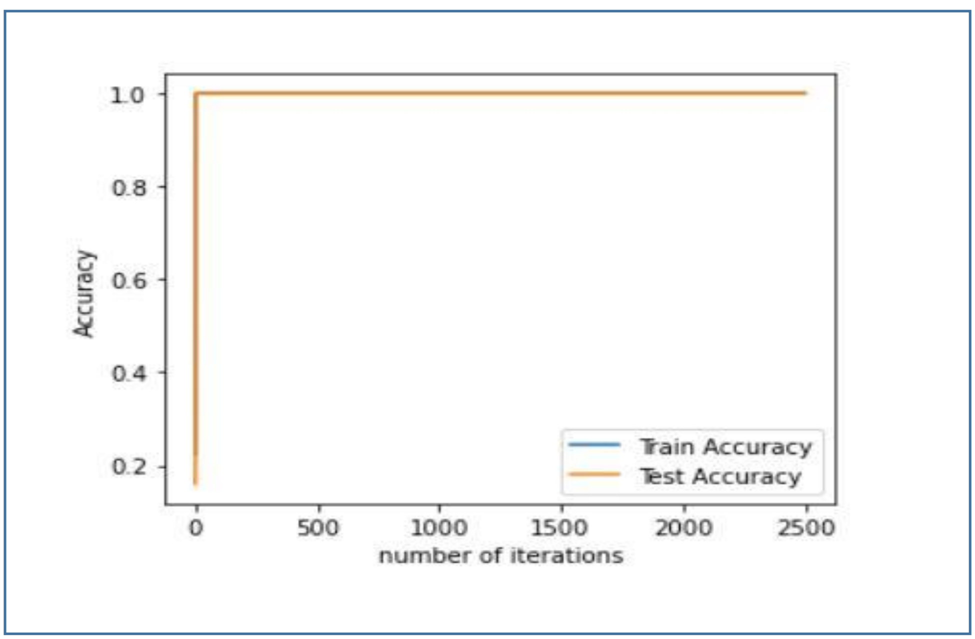
```
    plt.show()

    return parameters


parameters = L_layer_model(train_x, train_y, layers_dims, num_iterations = 2500, print_cost = True)

def predict(X, y, parameters):

    m = X.shape[1]   #Takes the total number of data
    n = len(parameters) // 2 # number of layers in the neural network. Divided by 2 as two parameters per layer(Weight & Bias)
    p = np.zeros((1,m))   #initializing variable to store the prediction

    probas, caches = L_model_forward(X, parameters)          #calculates the value of non-linear activation

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

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

    return p

pred_train = predict(train_x, train_y, parameters)

pred_test = predict(test_x, test_y, parameters)
```

Experimental Result:

Cost Curve:



Accuracy Curve:



Conclusion: Because of the manually generated numpy array dataset the result of this classification was not satisfactory. The main drawback of this code is that, the loss for training and testing was not demonstrated.