



# MACHINE LEARNING LABORATORY

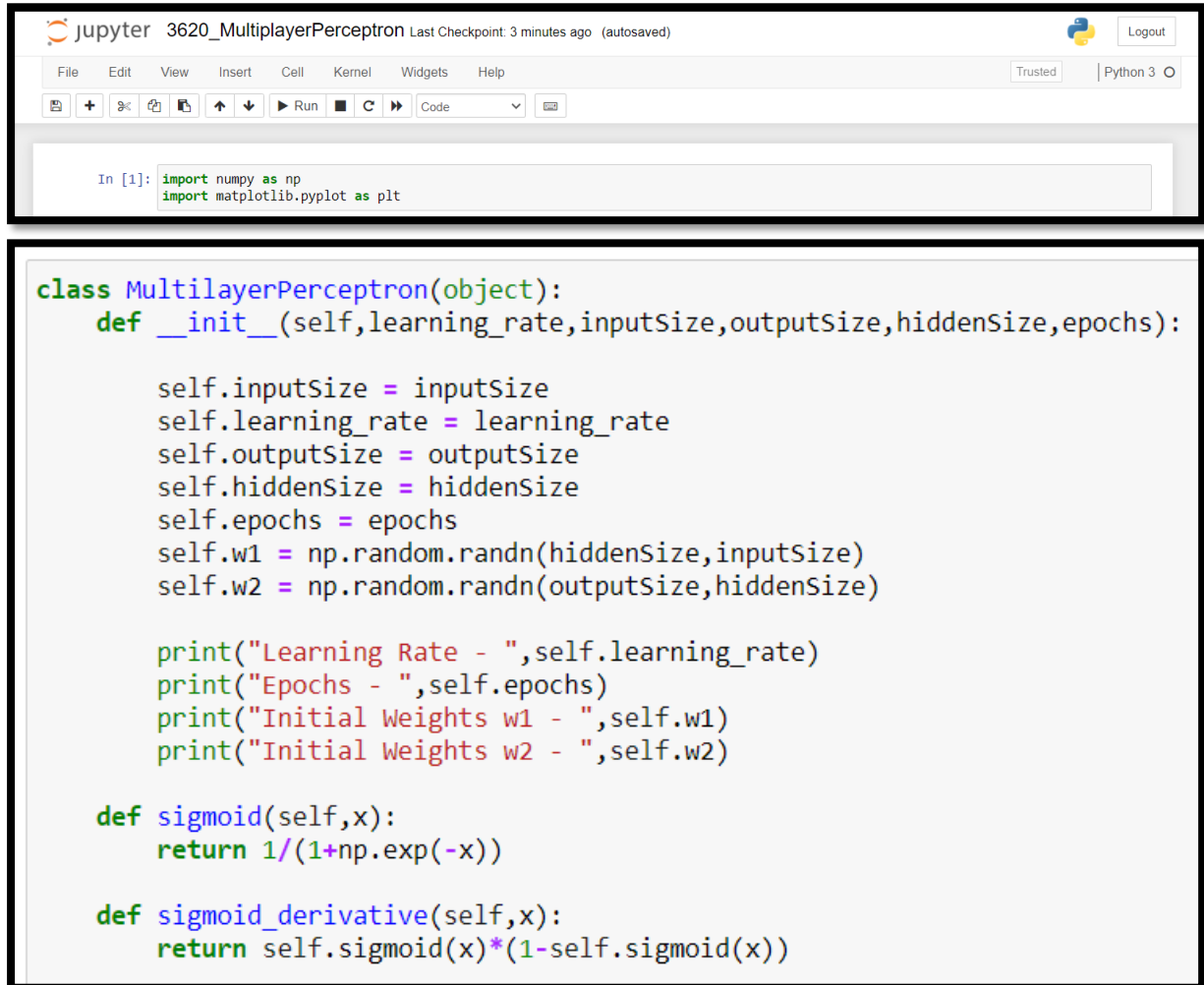
**LAB 06**  
**VEDANTH SUBRAMANIAM**

**CSE RUSA 'Q' BATCH | 2018103620**

## Implementation and study of Multilayer Perceptron

Implement the multilayer perceptron, and train it to perform the logical functions NAND, NOR and XOR functions?

### Code



The image shows a Jupyter Notebook interface with the title '3620\_MultiplayerPerceptron'. The top bar includes a 'Logout' button and a 'Python 3' environment selector. The menu bar contains 'File', 'Edit', 'View', 'Insert', 'Cell', 'Kernel', 'Widgets', and 'Help'. Below the menu is a toolbar with icons for file operations, running, and code execution. The notebook contains two code cells. The first cell is a simple import statement. The second cell defines a 'MultilayerPerceptron' class with an initialization method and two helper methods for the sigmoid function.

```
In [1]: import numpy as np
import matplotlib.pyplot as plt

class MultilayerPerceptron(object):
    def __init__(self, learning_rate, inputSize, outputSize, hiddenSize, epochs):

        self.inputSize = inputSize
        self.learning_rate = learning_rate
        self.outputSize = outputSize
        self.hiddenSize = hiddenSize
        self.epochs = epochs
        self.w1 = np.random.randn(hiddenSize, inputSize)
        self.w2 = np.random.randn(outputSize, hiddenSize)

        print("Learning Rate - ", self.learning_rate)
        print("Epochs - ", self.epochs)
        print("Initial Weights w1 - ", self.w1)
        print("Initial Weights w2 - ", self.w2)

    def sigmoid(self, x):
        return 1/(1+np.exp(-x))

    def sigmoid_derivative(self, x):
        return self.sigmoid(x)*(1-self.sigmoid(x))
```

```

def train(self,x,y):
    w1 = self.w1
    w2 = self.w2
    learning_rate = self.learning_rate
    epochs = self.epochs
    loss = []
    for iter in range(epochs):
        #FORWARD
        z1 = np.dot(x,w1.T)
        a1 = self.sigmoid(z1)
        z2 = np.dot(a1,w2.T)
        a2 = self.sigmoid(z2)

        #LOSS
        error = np.abs(a2 - y)
        loss.append(error.sum())

        #BACKPROPAGATION
        delta2 = (a2 - y)* self.sigmoid_derivative(a2)
        delta1 = np.dot(delta2,w2) * (a1 * (1-a1))
        w2 -= learning_rate * (np.dot(a1.T,delta2)).T
        w1 -= learning_rate * (np.dot(x.T,delta1)).T

    self.w1 = w1
    self.w2 = w2

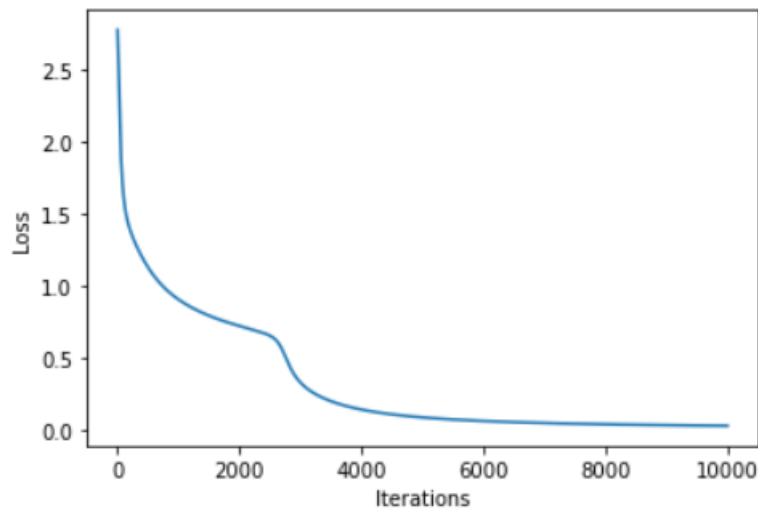
    plt.plot(loss)
    plt.xlabel("Iterations")
    plt.ylabel("Loss")
    plt.show()

```

```
def test(self,x):  
  
    #Loading saved weights  
    w1 = self.w1  
    w2 = self.w2  
  
    z1 = np.dot(x,w1.T)  
    a1 = self.sigmoid(z1)  
    z2 = np.dot(a1,w2.T)  
    a2 = self.sigmoid(z2)  
  
    prediction = (a2 > 0.5) * 1.0  
    print(prediction)
```

```
#NAND  
x = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])  
y = np.array([1,1,1,0])  
mlp = MultilayerPerceptron(0.1,3,1,4,10000)  
y = y.reshape(4,1)  
mlp.train(x,y)  
mlp.test(x)
```

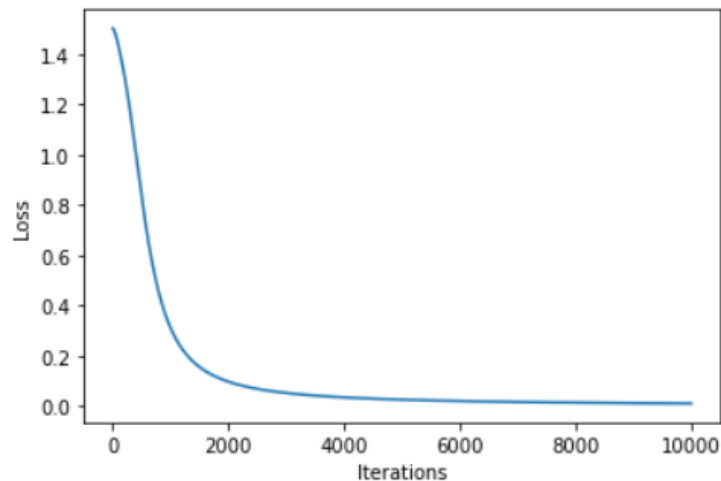
```
Learning Rate - 0.1
Epochs - 10000
Initial Weights w1 - [[-0.61757817 -0.70907714 0.2209179 ]
 [-0.35867307 0.61824806 -0.48205085]
 [ 0.52259746 -0.85085144 1.03807048]
 [-1.38511206 -0.17826028 -0.33993698]]
Initial Weights w2 - [[-0.41104798 -0.16125149 -1.49264939 -1.68962639]]
```



```
[[1.]
 [1.]
 [1.]
 [0.]]
```

```
#NOR
x = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
y = np.array([1,0,0,0])
mlp = MultilayerPerceptron(0.1,3,1,4,10000)
y = y.reshape(4,1)
mlp.train(x,y)
mlp.test(x)
```

```
Learning Rate - 0.1
Epochs - 10000
Initial Weights w1 - [[-0.86889984 -0.36853244 -0.8761646 ]
 [-2.26180796  1.41054331  1.92944358]
 [-1.28841289  1.79437463  0.1160148 ]
 [ 0.26888062 -0.34941096  0.31957689]]
Initial Weights w2 - [[ 0.30945852 -0.11381153 -1.00719691 -0.8000238 ]]
```

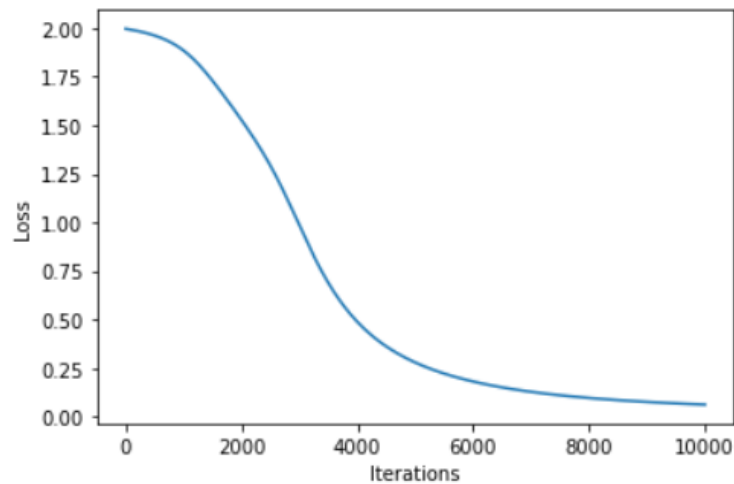


```
[[1.]
 [0.]
 [0.]
 [0.]]
```

*#XOR*

```
x = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
y = np.array([0,1,1,0])
mlp = MultilayerPerceptron(0.1,3,1,4,10000)
y = y.reshape(4,1)
mlp.train(x,y)
mlp.test(x)
```

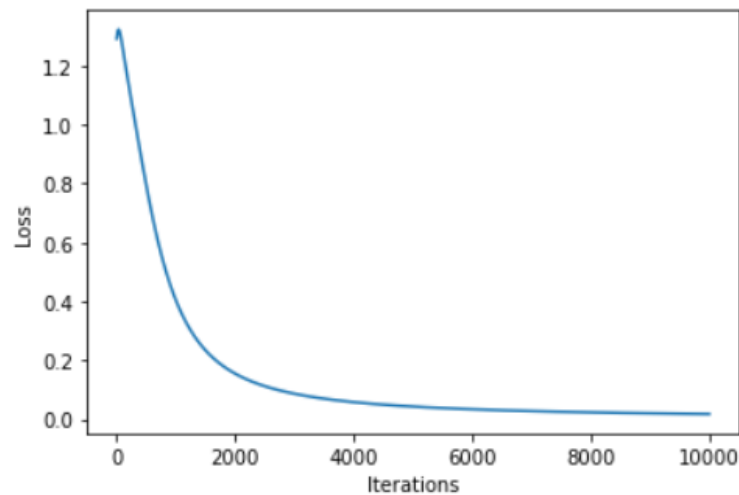
```
Learning Rate - 0.1
Epochs - 10000
Initial Weights w1 - [[-2.0039926  0.40796301 -0.21135248]
 [ 0.38269629 -0.11127041  0.15182741]
 [ 0.39885895  1.02590702 -0.00354507]
 [-0.55999923 -0.27853725  0.80784836]]
Initial Weights w2 - [[-0.40496146 -1.7096751  1.14031412  0.59100466]]
```



```
[[0.]
 [1.]
 [1.]
 [0.]]
```

```
#AND
x = np.array([[0,0,1],[0,1,1],[1,0,1],[1,1,1]])
y = np.array([0,0,0,1])
mlp = MultilayerPerceptron(0.1,3,1,4,10000)
y = y.reshape(4,1)
mlp.train(x,y)
mlp.test(x)
```

```
Learning Rate - 0.1
Epochs - 10000
Initial Weights w1 - [[ 1.18262999e+00  1.76790208e-01  1.31930942e+00]
 [ 1.25202656e+00 -1.42396177e+00 -1.68262066e-03]
 [ 7.91925650e-01  1.71589162e-01 -3.07411607e-01]
 [-6.18565208e-01 -1.90365516e+00  1.35290108e+00]]
Initial Weights w2 - [[ 0.03298256 -1.86206819 -0.63614443 -0.94891708]]
```



```
[[0.]
 [0.]
 [0.]
 [1.]]
```

## Inference

Here we have implemented Multilayer Perceptron Algorithm to perform the logical functions NAND, NOR, AND and XOR. We have plotted graphs and printed out the predicted outputs.



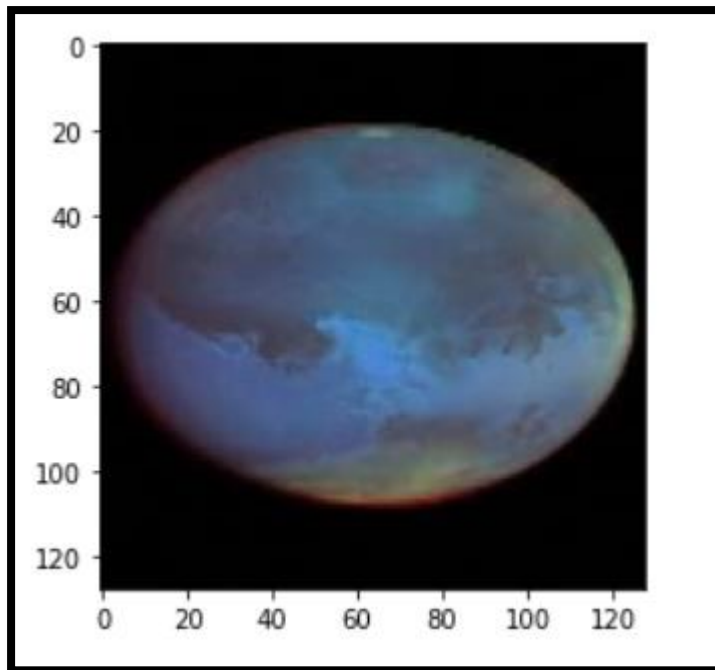
## Use Multilayer Perceptron for classification

### Code

```
import numpy as np
import pandas as pd
import os, cv2, random, warnings
from sklearn.utils import shuffle
from sklearn.model_selection import train_test_split
import matplotlib.pyplot as plt
warnings.filterwarnings('ignore')
```

```
x_train_org, y_train_org = [], []
for i in range(18):
    try:
        img = cv2.imread("./mars_input/train/earth/earth_"+str(i+1)+".jpg")
        img = cv2.resize(img, (128, 128))
        img_flip = cv2.flip(img, 1)
        x_train_org.append(img)
        y_train_org.append(0)
        x_train_org.append(img_flip)
        y_train_org.append(0)
    except Exception as e:
        print(e)
        pass
```

```
i=0
for i in range(18):
    try:
        img = cv2.imread("./mars_input/train/mars/telescope_"+str(i+1)+".jpg")
        img = cv2.resize(img, (128, 128))
        plt.imshow(img)
        img_flip = cv2.flip(img, 1)
        x_train_org.append(img)
        y_train_org.append(1)
        x_train_org.append(img_flip)
        y_train_org.append(1)
    except Exception as e:
        print(e)
        pass
```



```
x_train_org = np.array(x_train_org)
x_train_org.shape
```

```
(72, 128, 128, 3)
```

```
y_train_org = np.array(y_train_org)
y_train_org.shape
```

```
(72,)
```

```
#-1 means numpy will automatically identify the that particular dimension
y_train_org = y_train_org.reshape(-1, 1)
y_train_org.shape
```

```
(72, 1)
```

```
x_train_org, y_train_org = shuffle(x_train_org, y_train_org, random_state = 0)
```

```
m_train = y_train_org.shape[1]
num_px = x_train_org.shape[1]
```

```
#Splitting train and test
```

```
x_train_org, x_test_org, y_train_org, y_test_org = train_test_split(x_train_org,y_train_org,test_size=0.2)
```

```
x_train_org.shape, x_test_org.shape, y_train_org.shape, y_test_org.shape
```

```
((57, 128, 128, 3), (15, 128, 128, 3), (57, 1), (15, 1))
```

```
y_train = y_train_org
y_test = y_test_org
```

```
: #Flattening the RGB image  
x_train_flatten = x_train_org.reshape(x_train_org.shape[0],-1).T  
x_test_flatten = x_test_org.reshape(x_test_org.shape[0],-1).T  
x_train_flatten.shape, x_test_flatten.shape
```

```
: ((49152, 57), (49152, 15))
```

```
: #Normalizing the images  
x_train = x_train_flatten / 255  
x_test = x_test_flatten / 255
```

```
: def sigmoid(x):  
    return 1 / (1 + np.exp(-x))
```

```
: def initialize_with_zeros(dim):  
    w = np.zeros((dim,1))  
    b = 0  
    #print("Initialize w",w.shape)  
    return w,b
```

```
def propogate(w,b,X,Y):  
    #Forward Propogation  
    m = X.shape[1]  
    #print("w",w.shape)  
    #print("X",X.shape)  
    Z = np.dot(w.T,X) + b  
    A = sigmoid(Z)  
    cost = (-1 / m) * np.sum(Y * np.log(A) + (1 - Y) * np.log(1 - A))  
  
    #Backward Prop  
    dw = np.dot(X,(A-Y).T) / m  
    db = np.sum(A-Y)/m  
    grads = {"dw":dw,"db":db}  
    return grads, cost
```

```

def optimize(w,b,X,Y,num_iterations,learning_rate,print_cost=False):
    costs = []
    for i in range(num_iterations):
        grads, cost = propogate(w,b,X,Y)
        dw = grads['dw']
        db = grads['db']
        #print("Optimize w",w.shape)
        #print("Optimize b",b.shape)
        #print("Optimize dw",dw.shape)
        #print("Optimize db",db.shape)
        w = w - learning_rate*dw
        b = b - learning_rate*db
        if i % 100 == 0:
            costs.append(cost)
        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

```

```

def predict(w,b,X):
    m = X.shape[1]
    Y_pred = np.zeros((1,m))
    Z = np.dot(w.T,X) + b
    A = sigmoid(Z)

    for i in range(A.shape[1]):
        if A[0,i] > 0.5:
            Y_pred[0,i] = 1
        else:
            Y_pred[0,i] = 0
    return Y_pred

```

```

def model(X_train,Y_train,X_test,Y_test,num_iterations = 2000, learning_rate = 0.5, print_cost = False):
    w, b = initialize_with_zeros(X_train.shape[0])
    parameters, grads, costs = optimize(w, b, X_train, Y_train, num_iterations, learning_rate, print_cost)
    w = parameters["w"]
    b = parameters["b"]
    Y_prediction_test = predict(w, b, X_test)
    Y_prediction_train = predict(w, b, X_train)
    print("train accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_train - Y_train)) * 100))
    print("test accuracy: {} %".format(100 - np.mean(np.abs(Y_prediction_test - Y_test)) * 100))
    data = {"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 data

```

```
: data = model(x_train, y_train, x_test, y_test, num_iterations = 2000, learning_rate = 0.005, print_cost = True)

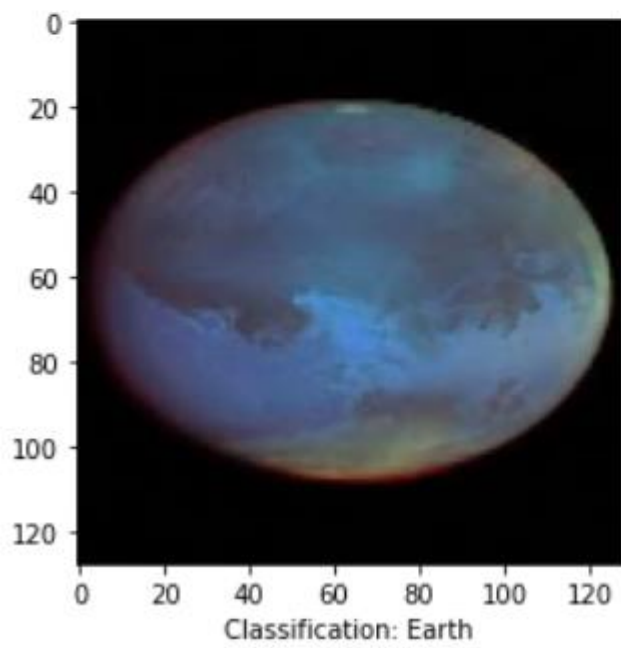
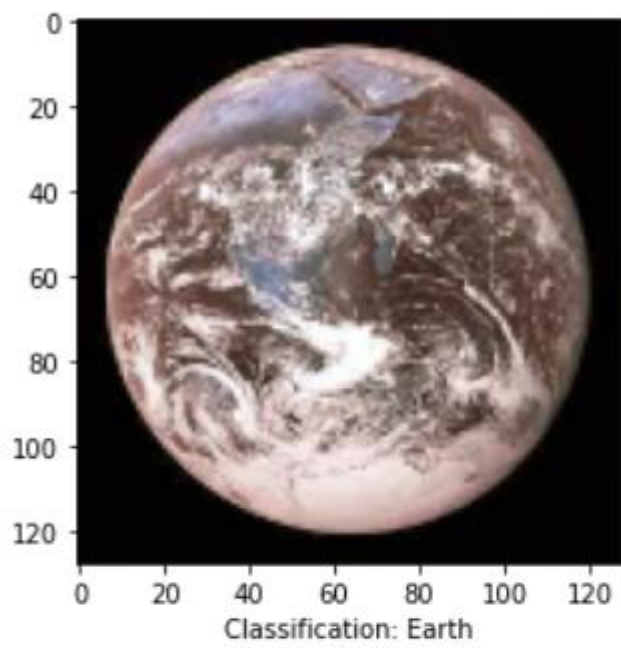
Cost after iteration 0: 39.509389
Cost after iteration 100: 0.118073
Cost after iteration 200: 0.067260
Cost after iteration 300: 0.047100
Cost after iteration 400: 0.036270
Cost after iteration 500: 0.029505
Cost after iteration 600: 0.024876
Cost after iteration 700: 0.021508
Cost after iteration 800: 0.018947
Cost after iteration 900: 0.016933
Cost after iteration 1000: 0.015308
Cost after iteration 1100: 0.013969
Cost after iteration 1200: 0.012846
Cost after iteration 1300: 0.011892
Cost after iteration 1400: 0.011069
Cost after iteration 1500: 0.010354
Cost after iteration 1600: 0.009726
Cost after iteration 1700: 0.009170
Cost after iteration 1800: 0.008675
Cost after iteration 1900: 0.008230
train accuracy: 43.85964912280702 %
test accuracy: 73.33333333333333 %
```

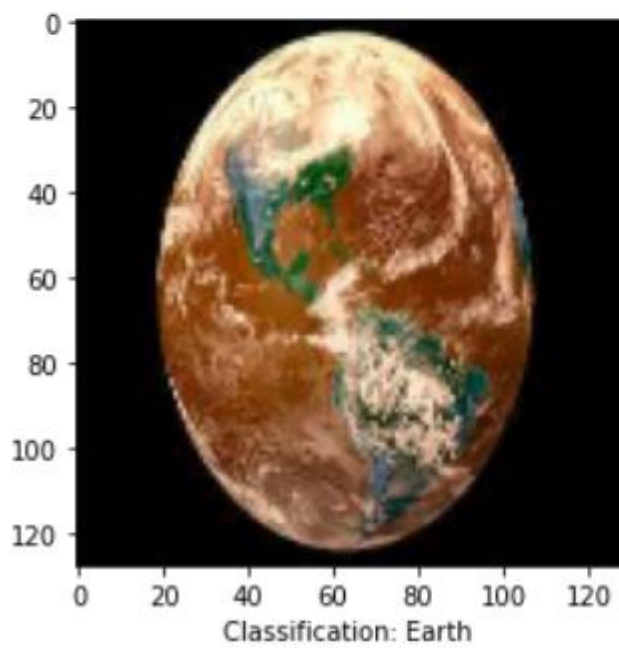
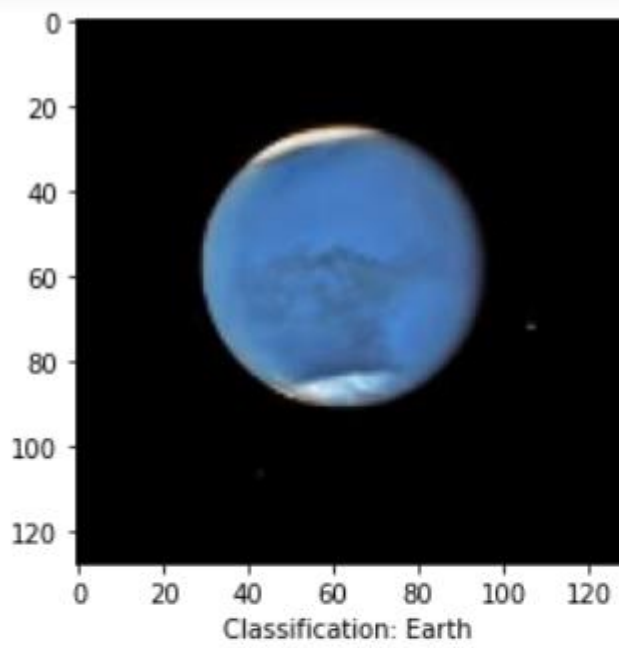
```
costs = data['costs']
y_pred = data['Y_prediction_train']
print(y_pred.shape)
```

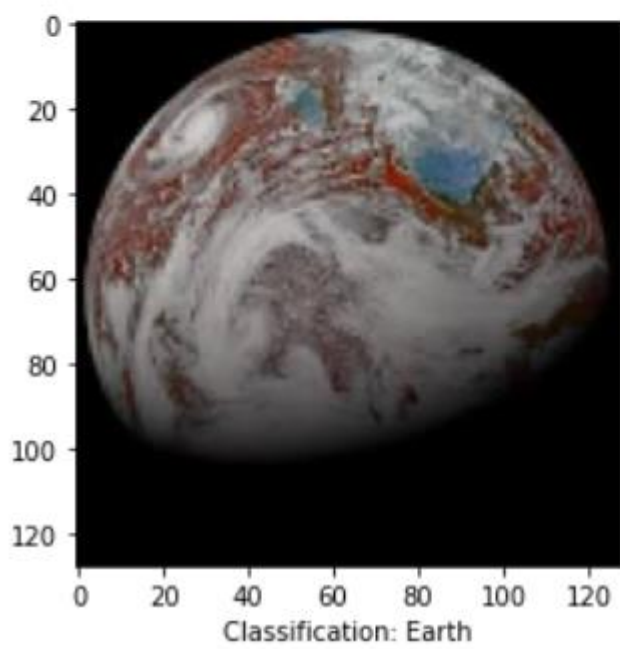
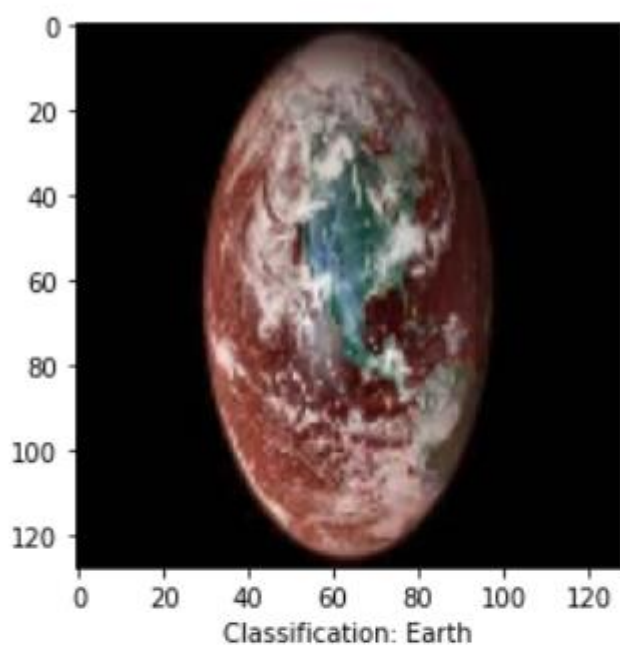
(1, 57)

```
for i in range(x_test_org.shape[0]):
    plt.imshow(x_test_org[i])

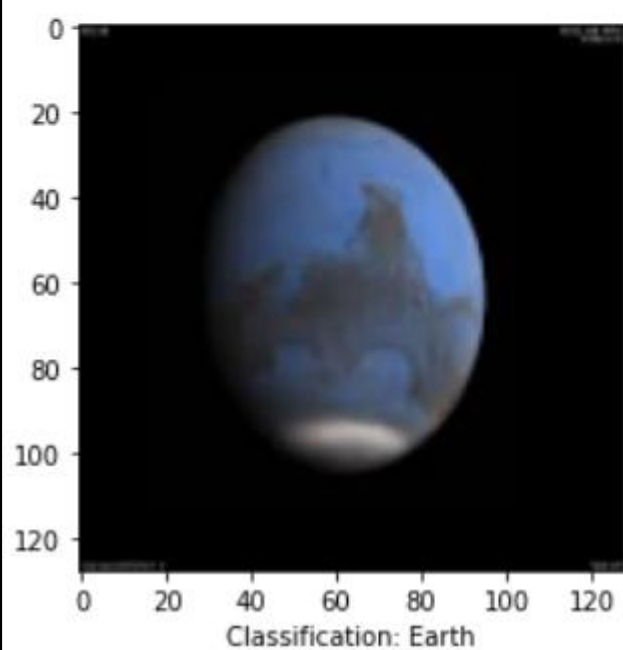
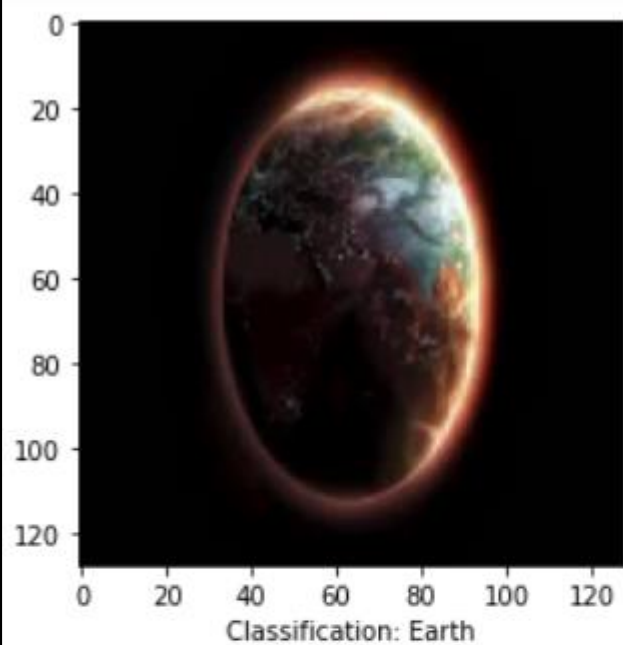
    if y_pred[0,1] == 0:
        plt.xlabel('Classification: {}'.format("Earth"))
    else:
        plt.xlabel('Classification: {}'.format("Mars"))
plt.show()
```

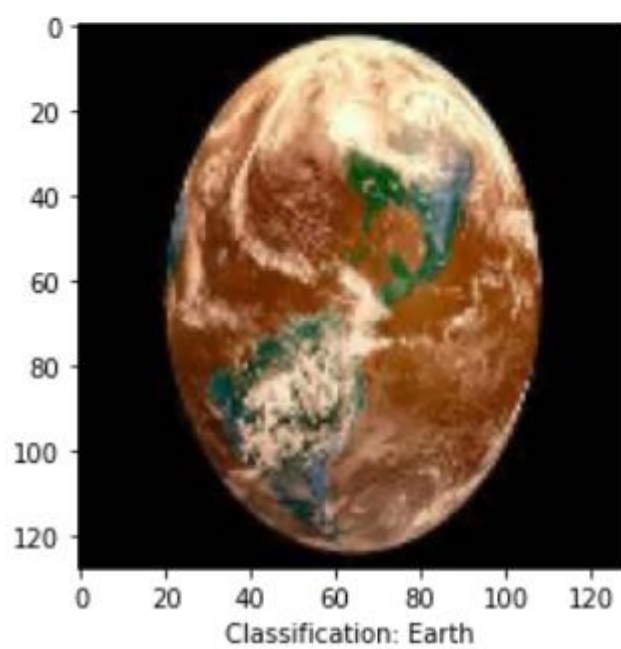
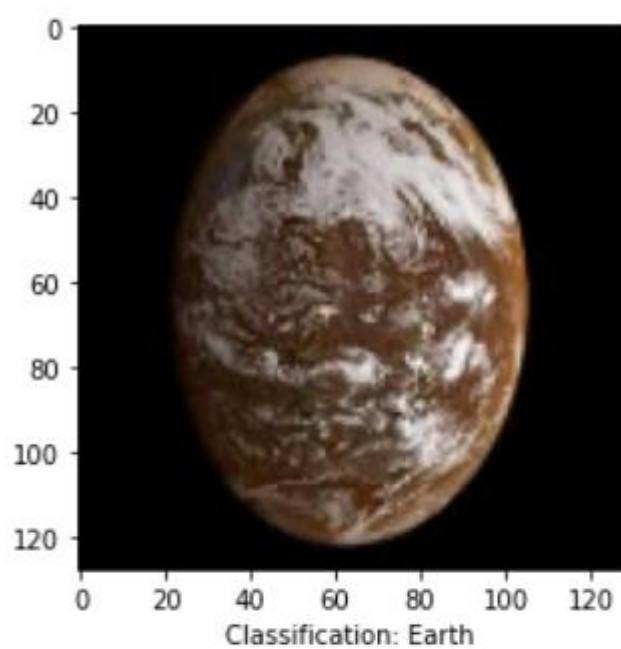


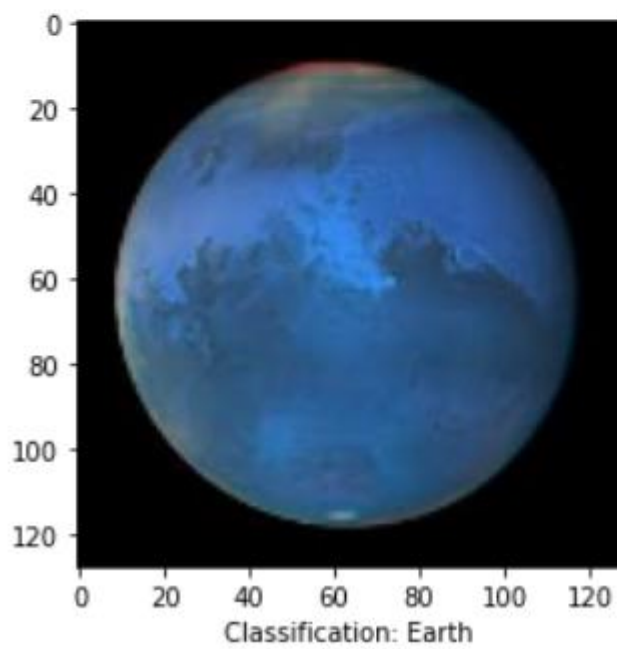
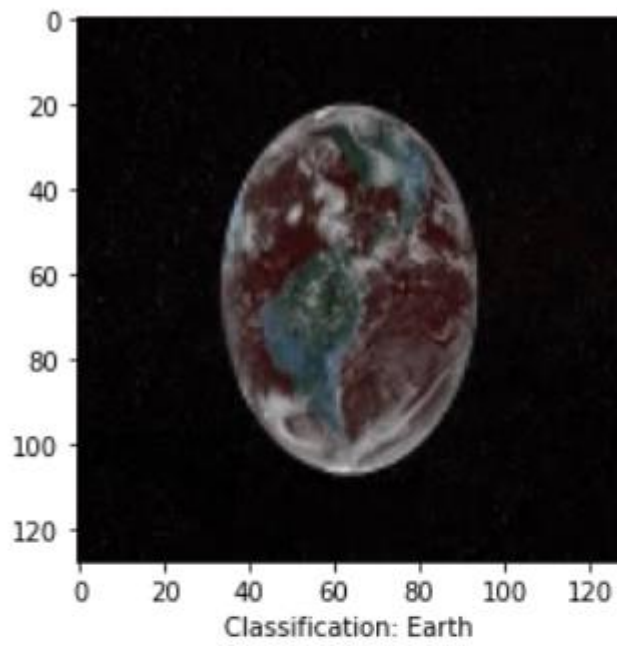


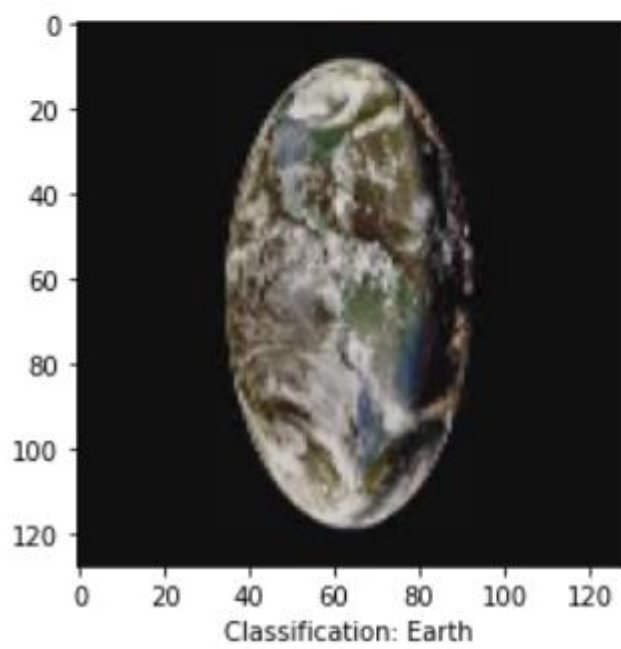
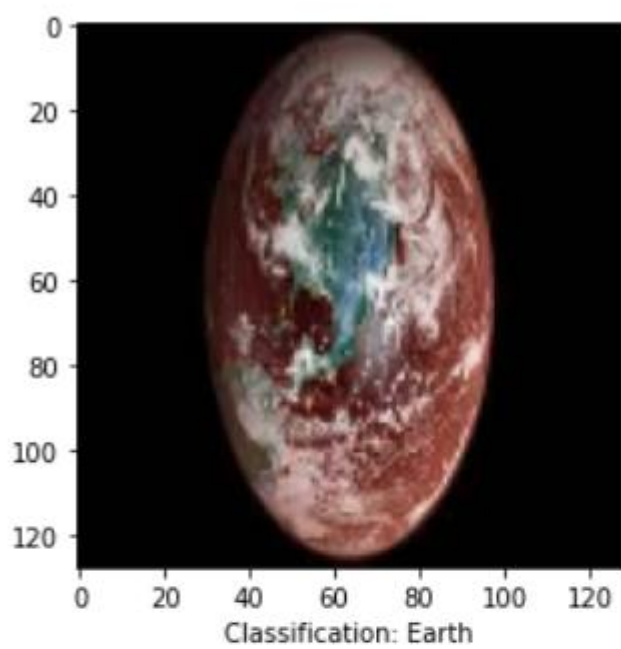


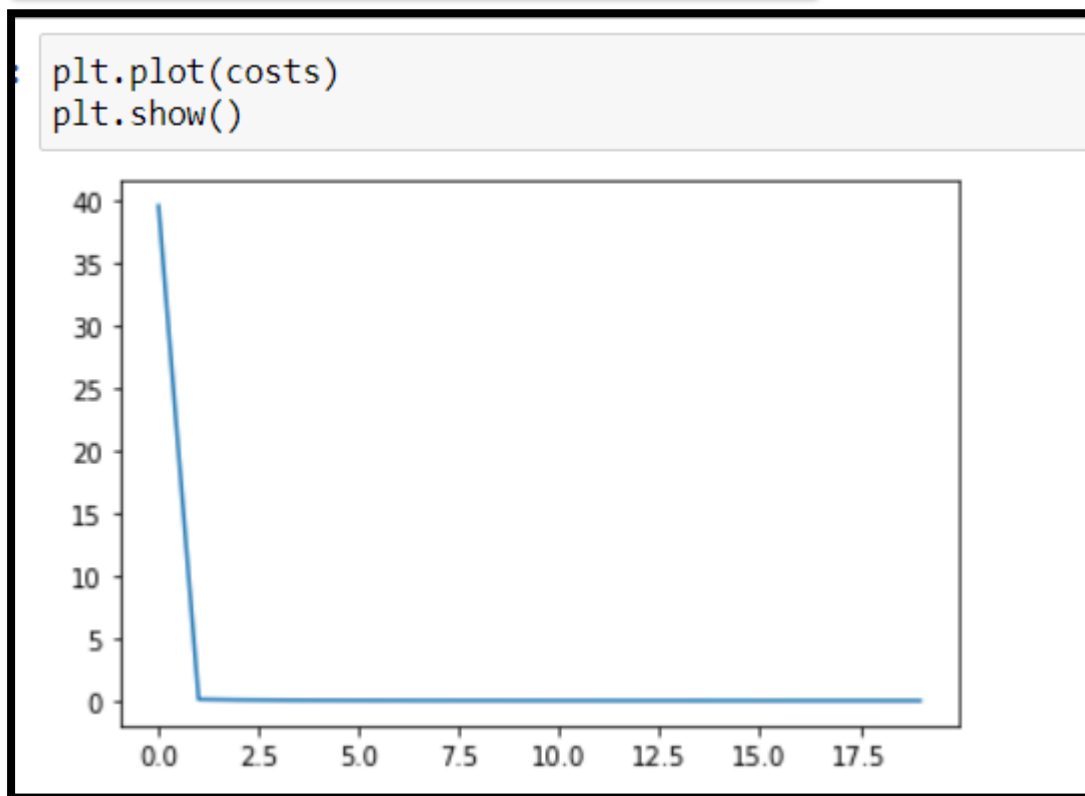
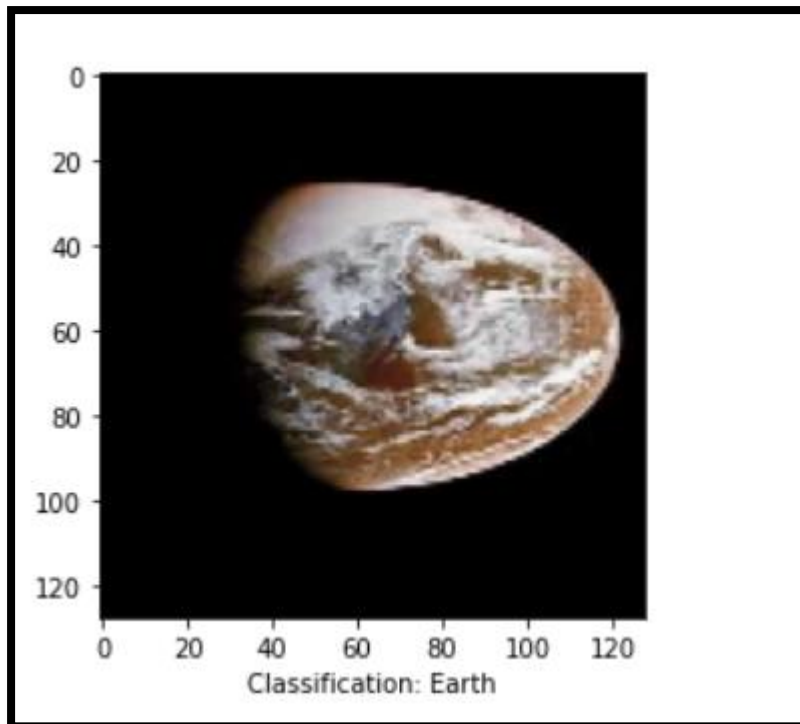












### Inference

Here, we are trying to classify earth and mars using Multilayer Perceptron. We get an training accuracy of 43% and testing accuracy of 73%.

## SPOT

Modify multi-layer perceptron you already implemented for Mars image classification with different activation functions

### a) Sigmoid

#### Activation Function

```
def sigmoid(x):  
    return 1 / (1 + np.exp(-x))
```

### b) Tanh

#### Activation Function

```
def tanh(x):  
    return (math.tanh(x))
```

```
data = model(x_train, y_train, x_test, y_test, num_iterations = 2000, learning_rate = 0.005, print_cost = True)  
  
Cost after iteration 0: 39.509389  
Cost after iteration 100: 0.123604  
Cost after iteration 200: 0.070354  
Cost after iteration 300: 0.049271  
Cost after iteration 400: 0.037948  
Cost after iteration 500: 0.030875  
Cost after iteration 600: 0.026035  
Cost after iteration 700: 0.022512  
Cost after iteration 800: 0.019833  
Cost after iteration 900: 0.017727  
Cost after iteration 1000: 0.016027  
Cost after iteration 1100: 0.014626  
Cost after iteration 1200: 0.013451  
Cost after iteration 1300: 0.012452  
Cost after iteration 1400: 0.011591  
Cost after iteration 1500: 0.010843  
Cost after iteration 1600: 0.010185  
Cost after iteration 1700: 0.009603  
Cost after iteration 1800: 0.009085  
Cost after iteration 1900: 0.008619  
train accuracy: 50.877192982456144 %  
test accuracy: 46.666666666666664 %
```

## Inference

Here we are using TanH activation function instead of sigmoid. We can see the change in training and testing accuracy because of activation function change.

### c) ReLu

#### Activation Function

```
def ReLu(x):  
    if x>0:  
        return x  
    else:  
        return 0
```

```
data = model(x_train, y_train, x_test, y_test, num_iterations = 2000, learning_rate = 0.005, print_cost = True)  
  
Cost after iteration 0: 38.816242  
Cost after iteration 100: 0.000000  
Cost after iteration 200: 0.000000  
Cost after iteration 300: 0.000000  
Cost after iteration 400: 0.000000  
Cost after iteration 500: 0.000000  
Cost after iteration 600: 0.000000  
Cost after iteration 700: 0.000000  
Cost after iteration 800: 0.000000  
Cost after iteration 900: 0.000000  
Cost after iteration 1000: 0.000000  
Cost after iteration 1100: 0.000000  
Cost after iteration 1200: 0.000000  
Cost after iteration 1300: 0.000000  
Cost after iteration 1400: 0.000000  
Cost after iteration 1500: 0.000000  
Cost after iteration 1600: 0.000000  
Cost after iteration 1700: 0.000000  
Cost after iteration 1800: 0.000000  
Cost after iteration 1900: 0.000000  
train accuracy: 51.785714285714285 %  
test accuracy: 35.71428571428571 %
```

#### Inference

Here we are using ReLu activation function instead of sigmoid. We can see the change in training and testing accuracy because of activation function change.