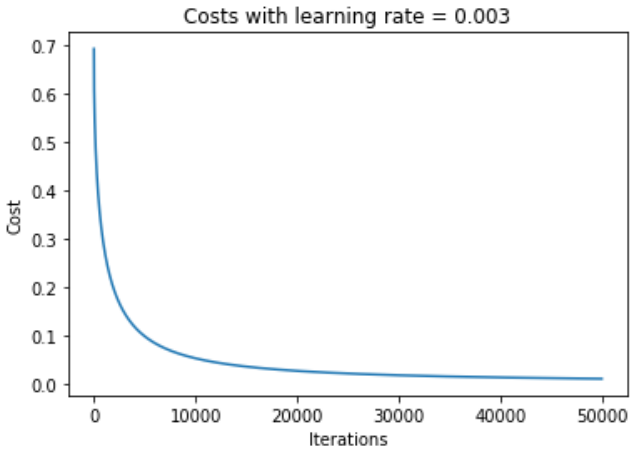
Assignment1: Single Neuron Classifier

Jack Rome H00219766

September 26, 2018

1 Report (Desired Mark - 5/5)

The dataset is imported, and the shape is recorded so that it can be converted into a 1xN vector. This shape is denoted as nx in my code. A neuron class is set up with the input matrix, output matrix and learning rate being the initial parameters. This is to make it easier to set up future the next assignment with more nodes. The weights and bias are initialized to zero so that the initial results lie in the centre of the sigmoid graph. The input data is normalized by dividing by 255 as that is the maximum pixel value. The package numpy is imported and used to control all matrices and vectors without using loops as I have vectored my code. The node is trained in the main method under a while loop that breaks when the last recorded loss is very close to zero or if the iterations are too high. I have included the results recorded using matplotlib for learning rate = 0.003. This turned out to be the optimal rate during testing. A high rate tended to break the cross-entropy loss function and so rates above 0.1 are not viable for my code. The loss after around 50000 iterations near 0.01. It was very close to convergence.



Errors still exist when applying the neuron on the testing set. A predection is considered to be incorrect if the error is above 0.5 and this occurs around 15 times out of 50. This is may be due to the training set being too small and so the neuron cannot correctly predict pictures that are too different from what has been trained on. I had had cases to catch exceptions within the loss function before I vectorized the code. These would skip the loss function and return zero if y == yhat or return a large number when the difference in y and yhat is 1. This is to handle nan values and infinites. These cases were removed when the vectors were utilized, however, using np.where would possibly be a way to handle these exceptions when working with vectors and would allow larger learning rates to be used.

2 Appendix

The program is written in python. I have changed the parameters of the h5py import to look in the current directory for the data sets. As long as the files are in the same directory then my code is simple to execute.

Code copy:

```
Created on 28 Aug 2018
'''
@author: Jack
'''
import numpy as np
```

```
import matplotlib.pyplot as plt
import h5py

class Neuron():
    def __init__(self, input_X, output_Y, learning_rate):
        self.nx = np.prod(input_X.shape[1:]) #vector length
        self.input_X = vectorise(input_X)
        self.input_X = np.divide(self.input_X, 255) #normalise
        self.output_Y = output_Y
        self.m = input_X.shape[0] #numper of samples
        #weights
        self.weights = np.zeros(self.nx)
        self.y_hat = np.zeros(self.m)
        self.bias = 0
        self.alpha = learning_rate
        self.z = np.zeros(self.m)
        #later variables
        self.loss = 0
        self.dw = np.zeros(self.nx) #weight change of size nx
        self.db = 0
        self.dz = np.zeros(self.m)


    def grad_calc(self): #i = index
        self.dz = np.subtract(self.y_hat,self.output_Y[0])
        self.dw = np.dot(self.dz, self.input_X)
        self.db = np.sum(self.dz)
        return

    def loss_accumulation(self):
        self.loss /= self.m
        self.dw = np.divide(self.dw, self.m)
        self.db /= self.m
        return

    def learningRules(self):
        self.weights = np.subtract(self.weights, np.multiply(self.alpha,self.dw))
        self.bias = self.bias - self.alpha*self.db
        return

    def forward_prob(self):
        self.z = np.add(np.dot(self.weights, np.transpose(self.input_X)),self.bias)
        self.y_hat = SigV(self.z)
        self.loss = np.sum(cross_entropy_Loss(self.output_Y[0], self.y_hat))
        self.loss = np.nan_to_num(self.loss) #replace nan values with 0

    def backward_prob(self):
        self.grad_calc()
        self.loss_accumulation()
        self.learningRules()

def cross_entropy_Loss(y, a):
   # loss is performed on entire vector inputs using numpy
   return -np.add(np.multiply(y,np.log(a)),np.multiply(np.subtract(1,y), np.log(np.subtract(1,a))))


def SigV(matrix): #return sigmoid matrix
    return np.divide(1,1+np.exp(-matrix))

def Sigmoid(z): #return sigmoid of value
```

```
    return 1/(1+np.exp(-z))

def vectorise(matrix): #input matrix, get (1,nx) vector
    n = np.prod(matrix.shape[1:])
    return np.reshape(matrix, (matrix.shape[0], n))

if __name__ == '__main__':
    # Loading the dataset
    train_dataset = h5py.File('trainCats.h5', "r")
    trainSetX = np.array(train_dataset["train_set_x"][:]) # your train set features
    trainSetY = np.array(train_dataset["train_set_y"][:]) # your train set labels
    trainSetY = trainSetY.reshape((1, trainSetY.shape[0]))

    # Loading the test set
    test_dataset = h5py.File('testCats.h5', "r")
    testSetX = np.array(test_dataset["test_set_x"][:]) # your test set features
    testSetY = np.array(test_dataset["test_set_y"][:]) # your test set labels
    testSetY = testSetY.reshape((1, testSetY.shape[0]))

    classes = np.array(test_dataset["list_classes"][:]) # the list of classes
    m = trainSetX.shape[0] #size of dataset
    #initalize node
    alpha = 0.003 #low for smoother cost descent, high for rapid but quick regression
    neuron = Neuron(trainSetX, trainSetY, alpha)
    J = [] #list of losses for graph
    j = 0 #iteration count
    Loss = 1 #dummy value

    #Training
    while(Loss >1e-3 and j < 25000): #very close to zero gradient or taken too long
        j = j + 1
        neuron.forward_prob()
        neuron.backward_prob()
        loss = neuron.loss
        J.append(loss)
    print(str(j) + " iterations")

    #create cost graph
    plt.plot(np.asanyarray(J))
    plt.title("Costs with learning rate = " + str(alpha))
    plt.ylabel("Cost")
    plt.xlabel("Iterations")
    plt.show()# costValues array with the costs for each iteration

    #Testing
    Error = np.zeros(testSetX.shape[0])
    testX = vectorise(testSetX)
    testX = np.divide(testX, 255) #normalise
    y_hat = np.zeros(testSetX.shape[0])
    y_hat = np.add(np.dot(neuron.weights, np.transpose(testX)), neuron.bias)
    y_hat = SigV(y_hat)
    Error = np.absolute(np.subtract(y_hat, testSetY[0]))

    #Plot test error graph
    plt.plot(np.asanyarray(Error), "ro")
    plt.title("Error in test data. Learning Rate = " + str(alpha))
    plt.ylabel("Differnce in Yhat and Y")
    plt.xlabel("Test Number")
    plt.show()
```