**Problem no. 1** Suppose you are the CEO of a restaurant. Now you want to open some new outlets in different cities. Your restaurant already has trucks in various cities, and you have data for profits and populations from those cities. The file “dataset\_1\_one\_variable.txt” contains the dataset. The first column is the population of a city and the second column is the profit in that city. A negative value for profit indicates a loss.

Use linear regression with this data to select which city to expand to next. When solving the problem, you must comply with the following conditions.

* Use Gradient Descent method to optimize the regression line.
* Use Mean Squared Error (MSE) in the Cost function.
* Implement different functions to measure Cost and Gradient Descent.
* Show the Scatter plot of the training data and the regression line of the final model.
* Show the loss curve of the whole training.
* Predict values for population sizes of 20,000 and 26,000 on the final model.
* Use python (and Jupyter notebook) to solve the problem.
* Implement the linear regression from the scratch and do not use any build in function for linear regression.

#Connect to Google COlab and upload file/s

from google.colab import files

files.upload()

#Import Necessary Libraries

import numpy as np

from numpy import genfromtxt

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

#Load data

data=pd.read\_csv("p1.txt", header=None)

data.head()

#Visualiza the original data

plt.scatter(data[0],data[1])

plt.xlabel("Population of City (10,000s)")

plt.ylabel("Profit ($10,000")

plt.title("Profit Vs Population")

plt.tight\_layout()

Chart, scatter chart

Description automatically generated

#Split the dataset

X= data.iloc[:, :-1].values

y= data.iloc[:, -1].values

#Compute Cost function

def computeCost(X,y,theta):

    m=len(y)

    predictions=X.dot(theta)

    square\_err=(predictions - y)\*\*2

    return 1/(2\*m) \* np.sum(square\_err)

data\_n=data.values

m=len(data\_n[:,-1])

X=np.append(np.ones((m,1)),data\_n[:,0].reshape(m,1),axis=1)

y=data\_n[:,1].reshape(m,1)

theta=np.zeros((2,1))

computeCost(X,y,theta)

#Gradient Descent function

def gradientDescent(X,y,theta,alpha,num\_iters):

    m=len(y)

    J\_history=[]

    for i in range(num\_iters):

        predictions = X.dot(theta)

        error = np.dot(X.transpose(),(predictions -y))

        descent=alpha \* 1/m \* error

        theta-=descent

        J\_history.append(computeCost(X,y,theta))

    return theta, J\_history

theta,J\_history = gradientDescent(X,y,theta,0.01,1500)

print("h(x) ="+str(round(theta[0,0],2))+" + "+str(round(theta[1,0],2))+"x1")

#Loss curve Visualization

plt.plot(J\_history)

plt.xlabel("Iteration")

plt.ylabel("$J(\Theta)$")

plt.title("Cost function using Gradient Descent")

plt.tight\_layout()

Chart

Description automatically generated

#Regression line

plt.scatter(data[0],data[1])

x\_value=[x for x in range(25)]

y\_value=[y\*theta[1]+theta[0] for y in x\_value]

plt.plot(x\_value,y\_value,color="r")

plt.xlabel("Population of City (10,000s)")

plt.ylabel("Profit ($10,000")

plt.title("Profit vs Population")

plt.tight\_layout()

Chart, scatter chart

Description automatically generated

#Predict function

def predict(x,theta):

    predictions= np.dot(theta.transpose(),x)

    return predictions[0]

#Predicting

predict1=predict(np.array([1,3.5]),theta)\*10000

print("For population = 35,000, we predict a profit of $"+str(round(predict1,0)))

predict2=predict(np.array([1,70]),theta)\*10000

print("For population = 70,000, we predict a profit of $"+str(round(predict2,0)))

**Problem no. 2** The file “dataset\_2\_multiple\_variable.txt” contains house prices based on size and number of bedrooms. The first column is the sizes of the house, the second column is the number of bed rooms, and the third column is the price of the house. Use linear regression using this data to build a model to predict the price of the house for new values. When solving the problem, you must comply with the following conditions.

* Use Gradient Descent method to optimize the regression line.
* Use Mean Squared Error (MSE) in the Cost function.
* Implement different functions to measure Cost and Gradient Descent.
* Show the Scatter plot of the training data and the regression line of the final model.
* Show the loss curve of the whole training.
* Estimate the price of a 1360 sq-ft, 3 bed rooms house.
* Use python (and Jupyter notebook) to solve the problem.
* Implement the linear regression from the scratch and do not use any build in function for linear regression.

#Connect to Google COlab and upload file/s

from google.colab import files

files.upload()

#Import Necessary Libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from mpl\_toolkits.mplot3d import Axes3D

# Create 2 subplot, 1 for each variable

fig, axes = plt.subplots(figsize=(12,4),nrows=1,ncols=2)

axes[0].scatter(data[0],data[2],color="b")

axes[0].set\_xlabel("Size (Square Feet)")

axes[0].set\_ylabel("Prices")

axes[0].set\_title("House prices against size of house")

axes[1].scatter(data[1],data[2],color="r")

axes[1].set\_xlabel("Number of bedroom")

axes[1].set\_ylabel("Prices")

axes[1].set\_title("House prices against number of bedroom")

# Enhance layout

plt.tight\_layout()

Chart, scatter chart

Description automatically generated

#Cost Function

def computeCost(X,y,theta):

    m=len(y)

    predictions=X.dot(theta)

    square\_err=(predictions - y)\*\*2

    return 1/(2\*m) \* np.sum(square\_err)

#Gradient Descent Function

def gradientDescent(X,y,theta,alpha,num\_iters):

    m=len(y)

    J\_history=[]

    for i in range(num\_iters):

        predictions = X.dot(theta)

        error = np.dot(X.transpose(),(predictions -y))

        descent=alpha \* 1/m \* error

        theta-=descent

        J\_history.append(computeCost(X,y,theta))

    return theta, J\_history

#Feature Normalization

def featureNormalization(X):

    mean=np.mean(X,axis=0)

    std=np.std(X,axis=0)

    X\_norm = (X - mean)/std

    return X\_norm , mean , std

data\_new=data.values

m2=len(data\_new[:,-1])

X2=data\_new[:,0:2].reshape(m2,2)

X2, mean\_X2, std\_X2 = featureNormalization(X2)

X2 = np.append(np.ones((m2,1)),X2,axis=1)

y2=data\_new[:,-1].reshape(m2,1)

theta2=np.zeros((3,1))

#Function Calling

theta2, J\_history2 = gradientDescent(X2,y2,theta2,0.01,1000)

print("h(x)= "+str(round(theta2[0,0],2))+" + "+str(round(theta2[1,0],2))+"x1 + "+str(round(theta2[2,0],2))+"x2")

#Loss curve visualization

plt.plot(J\_history2)

plt.xlabel("Iteration")

plt.ylabel("$J(\Theta)$")

plt.title("Cost function using Gradient Descent")

Chart

Description automatically generated

#Predict function

def predict(x,theta):

    predictions= np.dot(theta.transpose(),x)

    return predictions[0]

#Predicting

x\_sample = featureNormalization(np.array([1650,3]))[0]

x\_sample=np.append(np.ones(1),x\_sample)

predict3= predict(x\_sample,theta2)

print("For size of house = 1650, Number of bedroom = 3, we predict a house value of $"+str(round(predict3,0)))

**Problem no. 3** In this part of the exercise, you will build a logistic regression model to predict whether a student gets admitted into a university. Suppose that you are the administrator of a university department and you want to determine each applicant's chance of admission based on their results on two exams. You have historical data from previous applicants that you can use as a training set (“dataset\_prob\_3”) for logistic regression. For each training example, you have the applicant's scores on two exams and the admissions decision. Your task is to build a classification model that estimates an applicant's probability of admission based the scores from those two exams.

When solving the problem, you must comply with the following conditions.

* Use Gradient Descent method to optimize the regression line.
* Implement different functions to measure Cost and Gradient Descent.
* Show the Scatter plot of the training data and the decision boundary of the final model.
* Show the loss curve of the whole training.
* Use python (and Jupyter notebook) to solve the problem.
* Implement the logistic regression from the scratch and do not use any build in function for logistic
* regression.

#Conncet to Google colab

from google.colab import files

files.upload()

#Import Necessary Libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

#Load data

data= pd.read\_csv("p3.txt", header=None)

data.head()

#Split the dataset

x= data.iloc[:,:-1].values

y= data.iloc[:,-1].values

m,n= x.shape

#Filter out the data between admitted and non admitted

ad = data.loc[y == 1]

non\_ad= data.loc[y==0]

#Visualization of the original data

plt.scatter(ad.iloc[:, 0], ad.iloc[:, 1], s=10, label="Admitted")

plt.scatter(non\_ad.iloc[:, 0], non\_ad.iloc[:, 1], s=10, label="Not Admitted")

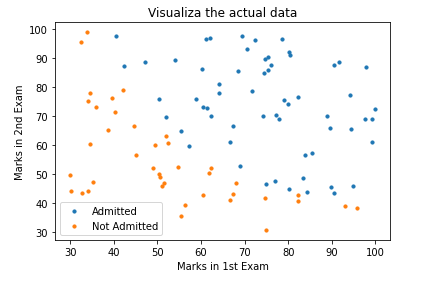
plt.xlabel("Marks in 1st Exam")

plt.ylabel("Marks in 2nd Exam")

plt.legend()

plt.title('Visualiza the actual data')

plt.show()



#Add a column of ones in X

z=np.ones(m)

z=z.reshape(m,1)

x=np.append(z,x,axis=1)

x.shape

#Add a column of zeros for theta

m,n= x.shape

theta= np.zeros(n)

theta= theta.reshape(n,1)

y= y.reshape(-1,1)

#Sigmoid function

def sigmoid(z):

  return 1/(1+np.exp(-z))

#Hypothesis function

def hypothesis(x, theta):

  return np.dot(x, theta)

#Cost function

def compute\_cost(theta, x, y):

  res= -y\*(np.log(sigmoid(hypothesis(x, theta))))- (1-y)\*(np.log(1-sigmoid(hypothesis(x, theta))))

  j= np.sum(res)/m

  return j

compute\_cost(theta, x, y)

#Define learning rate and iterations

iteration= 500000

alpha=0.1

#Gradient Descent function

cost\_history = []

def gradient\_descent(x, y, theta, alpha, iteration):

  for i in range(iteration):

    res= sigmoid(hypothesis(x, theta))-y

    grad= np.dot(x.transpose(), res)/m

    theta= theta- (alpha/m)\*grad

    cost= compute\_cost(theta, x, y)

    cost\_history.append(cost)

  return theta

theta= gradient\_descent(x, y, theta, alpha, iteration)

theta

compute\_cost(theta, x, y)

#Loss curve visualization

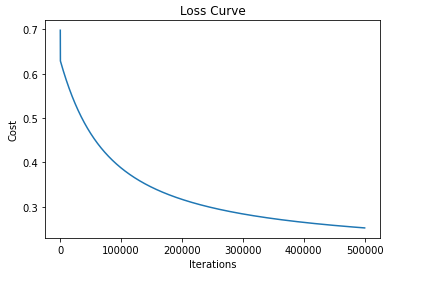
plt.plot(cost\_history)

plt.title('Loss Curve')

plt.xlabel('Iterations')

plt.ylabel('Cost')

plt.show()



#Regression line visualization

x\_values = [np.min(x[:, 1] ), np.max(x[:, 2] )]

a,b =x\_values

y\_min = - (theta[0] + np.dot(theta[1], a)) / theta[2]

y\_max = - (theta[0] + np.dot(theta[1], b)) / theta[2]

y\_values= y\_min, y\_max

plt.scatter(ad.iloc[:, 0], ad.iloc[:, 1], s=20, label="Admitted")

plt.scatter(non\_ad.iloc[:, 0], non\_ad.iloc[:, 1], s=20, label="Not Admitted")

plt.plot(x\_values, y\_values, label='Decision Boundary')

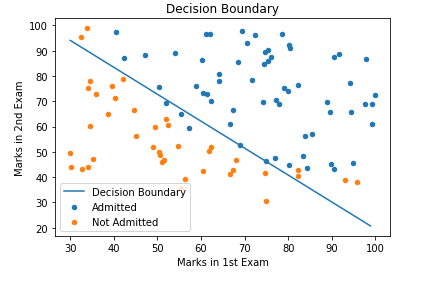
plt.xlabel('Marks in 1st Exam')

plt.ylabel('Marks in 2nd Exam')

plt.legend()

plt.title('Decision Boundary')

plt.show()



**Problem no. 4** In this part of the exercise, you will implement regularized logistic regression to predict whether microchipsfrom a fabrication plant passes quality assurance (QA). During QA, each microchip goes through varioustests to ensure it is functioning correctly. Suppose you are the product manager of the factory and you havethe test results for some microchips on two different tests. From these two tests, you would like to determinewhether the microchips should be accepted or rejected. To help you make the decision, you have a dataset(“dataset\_prob\_4”) of test results on past microchips, from which you can build a logistic regression model.When solving the problem, you must comply with the following conditions.

• Use Gradient Descent method to optimize the regression line.

• Implement different functions to measure Cost and Gradient Descent.

• Show the Scatter plot of the training data and the decision boundary of the final model.

• Show the loss curve of the whole training.

• Use python (and Jupyter notebook) to solve the problem.

• Implement the logistic regression from the scratch and do not use any build in function for logistic

regression.

#Connect to the google colab

from google.colab import files

files.upload()

#Import the necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

#Load Data

data= pd.read\_csv('p4.txt', header=None)

data.head()

#Split the dataset

x= data.iloc[:, :-1].values

y= data.iloc[:, -1].values

m,n= x.shape

#Filter out whether the chip is accepted or rejected

accepted = data.loc[y == 1]

rejected= data.loc[y==0]

#Original data visualization

plt.scatter(accepted.iloc[:, 0], accepted.iloc[:, 1], c='r', s=20, label="Accepted")

plt.scatter(rejected.iloc[:, 0], rejected.iloc[:, 1], c='b', s=20, label="Rejected")

plt.xlabel("Microchip test 1")

plt.ylabel("Microchip test 2")

plt.title('Visualiza the actual data')

plt.legend()

plt.show()

Chart, scatter chart

Description automatically generated

#Polynomial function

def mapping(x1,x2,degree):

    res = np.ones(len(x1)).reshape(len(x1),1)

    for i in range(1,degree+1):

        for j in range(i+1):

            terms= (x1\*\*(i-j) \* x2\*\*j).reshape(len(x1),1)

            res= np.hstack((res,terms))

    return res

x = mapping(x[:,0], x[:,1],6)

x.shape

#Sigmoid function

def sigmoid(z):

  return 1/(1+np.exp(-z))

#Cost function

def compute\_cost(theta, x, y ,Lambda):

    m=len(y)

    y=y[:,np.newaxis]

    h = sigmoid(x @ theta)

    error = (-y \* np.log(h)) - ((1-y)\*np.log(1-h))

    cost = 1/m \* sum(error)

    regCost= cost + Lambda/(2\*m) \* sum(theta\*\*2)

    # compute gradient

    j\_0= 1/m \* (x.transpose() @ (h - y))[0]

    j\_1 = 1/m \* (x.transpose() @ (h - y))[1:] + (Lambda/m)\* theta[1:]

    grad= np.vstack((j\_0[:,np.newaxis],j\_1))

    return regCost[0], grad

#Initialization of the parameter

initial\_theta = np.zeros((x.shape[1], 1))

Lambda = 1

alpha= 1

iteration= 800

cost, grad= compute\_cost(initial\_theta, x, y, Lambda)

cost

#Gradient Descent function

def gradientDescent(x,y,theta,alpha,num\_iters,Lambda):

    m=len(y)

    cost\_history= []

    for i in range(num\_iters):

        cost, grad = compute\_cost(theta,x,y,Lambda)

        theta = theta - (alpha \* grad)

        cost\_history.append(cost)

    return theta , cost\_history

theta, cost\_history = gradientDescent(x, y, initial\_theta, alpha, iteration, 0.2)

theta

#Loss curve visualization

plt.plot(cost\_history)

plt.xlabel("Iteration")

plt.ylabel("Cost")

plt.title("Cost function using Gradient Descent")

Chart

Description automatically generated

#Plotting function

def mapFeaturePlot(x1, x2, degree):

    res = np.ones(1)

    for i in range(1, degree+1):

        for j in range(i+1):

            terms= (x1\*\*(i-j) \* x2\*\*j)

            res= np.hstack((res,terms))

    return res

#Decesion boundary visualization

a= np.linspace(-1,1.5,50)

b= np.linspace(-1,1.5,50)

c= np.zeros((len(a),len(b)))

for i in range(len(a)):

    for j in range(len(b)):

        c[i,j] =mapFeaturePlot(a[i], b[j],6) @ theta

plt.contour(a, b, c.T, 0)

plt.scatter(accepted.iloc[:, 0], accepted.iloc[:, 1], c='r', s=20, label="Accepted")

plt.scatter(rejected.iloc[:, 0], rejected.iloc[:, 1], c='b', s=20, label="Rejected")

plt.xlabel("Microchip test 1")

plt.ylabel("Microchip test 2")

plt.title('Decision boundary')

plt.legend()

plt.show()

Chart

Description automatically generated

**Problem no. 5** In the first part of the exercise, you will extend your previous implementation of logistic regression usingthe above dataset and apply it to one-vs-all classification.

* When solving the problem, you must comply with the following conditions.
* Use Gradient Descent method to optimize the regression line.
* Implement different functions for Vectorized Logistic Regression, vectorized cost function and
* vectorized gradient.
* Show the loss curve of the whole training.
* Use python (and Jupyter notebook) to solve the problem.

#Connect to Google Colab and upload the necessary file/s

from google.colab import files

files.upload()

#Import Necessary Libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from scipy.io import loadmat

# Use loadmat to load matlab files

mat=loadmat("p5.mat")

X=mat["X"]

y=mat["y"]

#Original data visualization

import matplotlib.image as mpimg

fig, axis = plt.subplots(10,10,figsize=(8,8))

for i in range(10):

    for j in range(10):

        axis[i,j].imshow(X[np.random.randint(0,5001),:].reshape(20,20,order="F"), cmap="hot") #reshape back to 20 pixel by 20 pixel

        axis[i,j].axis("off")

A picture containing text

Description automatically generated

#Sigmoid Function

def sigmoid(z):

    return 1/ (1 + np.exp(-z))

#Cost Function

def lrCostFunction(theta, X, y, Lambda):

    m=len(y)

    predictions = sigmoid(X @ theta)

    error = (-y \* np.log(predictions)) - ((1-y)\*np.log(1-predictions))

    cost = 1/m \* sum(error)

    regCost= cost + Lambda/(2\*m) \* sum(theta[1:]\*\*2)

    # compute gradient

    j\_0= 1/m \* (X.transpose() @ (predictions - y))[0]

    j\_1 = 1/m \* (X.transpose() @ (predictions - y))[1:] + (Lambda/m)\* theta[1:]

    grad= np.vstack((j\_0[:,np.newaxis],j\_1))

    return regCost[0], grad

#Parameters modification Function Calling

theta\_t = np.array([-2,-1,1,2]).reshape(4,1)

X\_t =np.array([np.linspace(0.1,1.5,15)]).reshape(3,5).T

X\_t = np.hstack((np.ones((5,1)), X\_t))

y\_t = np.array([1,0,1,0,1]).reshape(5,1)

J, grad = lrCostFunction(theta\_t, X\_t, y\_t, 3)

print("Cost:",J,"Expected cost: 2.534819")

print("Gradients:\n",grad,"\nExpected gradients:\n 0.146561\n -0.548558\n 0.724722\n 1.398003")

#Gradient Descent Function

def gradientDescent(X,y,theta,alpha,num\_iters,Lambda):

    m=len(y)

    J\_history =[]

    for i in range(num\_iters):

        cost, grad = lrCostFunction(theta,X,y,Lambda)

        theta = theta - (alpha \* grad)

        J\_history.append(cost)

    return theta , J\_history

#One Vs All Function

def oneVsAll(X, y, num\_labels, Lambda):

    m, n = X.shape[0], X.shape[1]

    initial\_theta = np.zeros((n+1,1))

    all\_theta = []

    all\_J=[]

    # add intercept terms

    X = np.hstack((np.ones((m,1)),X))

    for i in range(1,num\_labels+1):

        theta , J\_history = gradientDescent(X,np.where(y==i,1,0),initial\_theta,1,300,Lambda)

        all\_theta.extend(theta)

        all\_J.extend(J\_history)

    return np.array(all\_theta).reshape(num\_labels,n+1), all\_J

all\_theta, all\_J= oneVsAll(X, y, 10, 1)

#Loss curve visualization

plt.plot(all\_J[0:300])

plt.xlabel("Iteration")

plt.ylabel("$J(\Theta)$")

plt.title("Cost function using Gradient Descent")

Chart

Description automatically generated

#Predicting the Accuracy

def predictOneVsAll(all\_theta, X):

    m= X.shape[0]

    X = np.hstack((np.ones((m,1)),X))

    predictions = X @ all\_theta.T

    return np.argmax(predictions,axis=1)+1

pred = predictOneVsAll(all\_theta, X)

print("Training Set Accuracy:",sum(pred[:,np.newaxis]==y)[0]/5000\*100,"%"

**Problem no. 6** In the previous part of this exercise, you implemented multi-class logistic regression to recognizehandwritten digits. However, logistic regression cannot form more complex hypotheses as it is only a linearclassifier (You could add more features - such as polynomial features - to logistic regression, but that canbe very expensive to train). In this part of the exercise, you will implement a neural network to recognize handwritten digits using the same training set as before. The neural network will be able to represent complex models that form nonlinear hypotheses. For this experiment, you will be using parameters from a neural network that are already trained. Your goal is to implement the feedforward propagation algorithm to use the weights for prediction. In the next exercise, you will write the backpropagation algorithm for learning the neural network parameters.

When solving the problem, you must comply with the following conditions.

* Implement the feedforward propagation algorithm to use the weights (Θ(1), Θ(2)) for prediction.
* Use python (and Jupyter notebook) to solve the problem.

#Connect the Google Colab and upload File/s

from google.colab import files

files.upload()

#Import Necessary libraries

import numpy as np

import pandas as pd

import matplotlib.pyplot as plt

from scipy.io import loadmat

# Use loadmat to load matlab files

mat=loadmat("p5.mat")

X=mat["X"]

y=mat["y"]

mat2=loadmat("p6.mat")

Theta1=mat2["Theta1"] # Theta1 has size 25 x 401

Theta2=mat2["Theta2"] # Theta2 has size 10 x 26

#Sigmoid function

def sigmoid(z):

    return 1/ (1 + np.exp(-z))

#Using feedforward propagation for prediction

def predict(Theta1, Theta2, X):

    m= X.shape[0]

    X = np.hstack((np.ones((m,1)),X))

    a1 = sigmoid(X @ Theta1.T)

    a1 = np.hstack((np.ones((m,1)), a1)) # hidden layer

    a2 = sigmoid(a1 @ Theta2.T) # output layer

    return np.argmax(a2,axis=1)+1

pred2 = predict(Theta1, Theta2, X)

print("Training Set Accuracy:",sum(pred2[:,np.newaxis]==y)[0]/5000\*100,"%")

#Sigmoid Gradient

def sigmoidGradient(z):

    sigmoid = 1/(1 + np.exp(-z))

    return sigmoid \*(1-sigmoid)

#Cost Function and Gradient

def nnCostFunction(nn\_params,input\_layer\_size, hidden\_layer\_size, num\_labels,X, y,Lambda):

    # Reshape nn\_params back into the parameters Theta1 and Theta2

    Theta1 = nn\_params[:((input\_layer\_size+1) \* hidden\_layer\_size)].reshape(hidden\_layer\_size,input\_layer\_size+1)

    Theta2 = nn\_params[((input\_layer\_size +1)\* hidden\_layer\_size ):].reshape(num\_labels,hidden\_layer\_size+1)

    m = X.shape[0]

    J=0

    X = np.hstack((np.ones((m,1)),X))

    y10 = np.zeros((m,num\_labels))

    a1 = sigmoid(X @ Theta1.T)

    a1 = np.hstack((np.ones((m,1)), a1)) # hidden layer

    a2 = sigmoid(a1 @ Theta2.T) # output layer

    for i in range(1,num\_labels+1):

        y10[:,i-1][:,np.newaxis] = np.where(y==i,1,0)

    for j in range(num\_labels):

        J = J + sum(-y10[:,j] \* np.log(a2[:,j]) - (1-y10[:,j])\*np.log(1-a2[:,j]))

    cost = 1/m\* J

    reg\_J = cost + Lambda/(2\*m) \* (np.sum(Theta1[:,1:]\*\*2) + np.sum(Theta2[:,1:]\*\*2))

    # Implement the backpropagation algorithm to compute the gradients

    grad1 = np.zeros((Theta1.shape))

    grad2 = np.zeros((Theta2.shape))

    for i in range(m):

        xi= X[i,:] # 1 X 401

        a1i = a1[i,:] # 1 X 26

        a2i =a2[i,:] # 1 X 10

        d2 = a2i - y10[i,:]

        d1 = Theta2.T @ d2.T \* sigmoidGradient(np.hstack((1,xi @ Theta1.T)))

        grad1= grad1 + d1[1:][:,np.newaxis] @ xi[:,np.newaxis].T

        grad2 = grad2 + d2.T[:,np.newaxis] @ a1i[:,np.newaxis].T

    grad1 = 1/m \* grad1

    grad2 = 1/m\*grad2

    grad1\_reg = grad1 + (Lambda/m) \* np.hstack((np.zeros((Theta1.shape[0],1)),Theta1[:,1:]))

    grad2\_reg = grad2 + (Lambda/m) \* np.hstack((np.zeros((Theta2.shape[0],1)),Theta2[:,1:]))

    return cost, grad1, grad2,reg\_J, grad1\_reg,grad2\_reg

#Parameter Initialization

input\_layer\_size  = 400

hidden\_layer\_size = 25

num\_labels = 10

nn\_params = np.append(Theta1.flatten(),Theta2.flatten())

J,reg\_J = nnCostFunction(nn\_params, input\_layer\_size, hidden\_layer\_size, num\_labels, X, y, 1)[0:4:3]

print("Cost at parameters (non-regularized):",J,"\nCost at parameters (Regularized):",reg\_J)

#Weight Initialization

def randInitializeWeights(L\_in, L\_out):

    epi = (6\*\*1/2) / (L\_in + L\_out)\*\*1/2

    W = np.random.rand(L\_out,L\_in +1) \*(2\*epi) -epi

    return W

initial\_Theta1 = randInitializeWeights(input\_layer\_size, hidden\_layer\_size)

initial\_Theta2 = randInitializeWeights(hidden\_layer\_size, num\_labels)

initial\_nn\_params = np.append(initial\_Theta1.flatten(),initial\_Theta2.flatten())

#Gradient Descent Function

def gradientDescentnn(X,y,initial\_nn\_params,alpha,num\_iters,Lambda,input\_layer\_size, hidden\_layer\_size, num\_labels):

    Theta1 = initial\_nn\_params[:((input\_layer\_size+1) \* hidden\_layer\_size)].reshape(hidden\_layer\_size,input\_layer\_size+1)

    Theta2 = initial\_nn\_params[((input\_layer\_size +1)\* hidden\_layer\_size ):].reshape(num\_labels,hidden\_layer\_size+1)

    m=len(y)

    J\_history =[]

    for i in range(num\_iters):

        nn\_params = np.append(Theta1.flatten(),Theta2.flatten())

        cost, grad1, grad2 = nnCostFunction(nn\_params,input\_layer\_size, hidden\_layer\_size, num\_labels,X, y,Lambda)[3:]

        Theta1 = Theta1 - (alpha \* grad1)

        Theta2 = Theta2 - (alpha \* grad2)

        J\_history.append(cost)

    nn\_paramsFinal = np.append(Theta1.flatten(),Theta2.flatten())

    return nn\_paramsFinal , J\_history

nnTheta, nnJ\_history = gradientDescentnn(X,y,initial\_nn\_params,0.8,800,1,input\_layer\_size, hidden\_layer\_size, num\_labels)

Theta1 = nnTheta[:((input\_layer\_size+1) \* hidden\_layer\_size)].reshape(hidden\_layer\_size,input\_layer\_size+1)

Theta2 = nnTheta[((input\_layer\_size +1)\* hidden\_layer\_size ):].reshape(num\_labels,hidden\_layer\_size+1)

#Predict the Accuracy

pred3 = predict(Theta1, Theta2, X)

print("Training Set Accuracy:",sum(pred3[:,np.newaxis]==y)[0]/5000\*100,"%")