Practical No.2

Source Code :-

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, h))

# 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 blue)

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 # theta

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

def load\_extra\_datasets():

N = 200

noisy\_circles = sklearn.datasets.make\_circles(n\_samples=N, factor=.5, noise=.3)

noisy\_moons = sklearn.datasets.make\_moons(n\_samples=N, noise=.2)

blobs = sklearn.datasets.make\_blobs(n\_samples=N, random\_state=5, n\_features=2, centers=6)

gaussian\_quantiles = sklearn.datasets.make\_gaussian\_quantiles(mean=None, cov=0.5, n\_samples=N, n\_features=2, n\_classes=2, shuffle=True, random\_state=None)

no\_structure = np.random.rand(N, 2), np.random.rand(N, 2)

return noisy\_circles, noisy\_moons, blobs, gaussian\_quantiles, no\_structure

X, Y = load\_planar\_dataset()

plt.scatter(X[0, :],X[1, :], c=Y, s=35,cmap=plt.cm.Spectral)

shape\_X = X.shape

shape\_Y = Y.shape

m = shape\_X[1] # training set size

print ('The shape of X is: ' + str(shape\_X))

print ('The shape of Y is: ' + str(shape\_Y))

print ('We have m = %d training examples!' % (m))

clf = sklearn.linear\_model.LogisticRegressionCV()

clf.fit(X.T, Y.T)

plot\_decision\_boundary(lambda j: clf.predict(j), X, Y)

plt.title("Logistic Regression")

# Print accuracy

LR\_predictions = clf.predict(X.T)

print ('Accuracy of logistic regression: %d ' % float((np.dot(Y,LR\_predictions) + np.dot(1-Y,1-LR\_predictions))/float(Y.size)\*100) + '% ' + "(percentage of correctly labelled datapoints)")

def layer\_sizes(X, Y):

"""

Arguments:

X -- input dataset of shape (input size, number of examples)

Y -- labels of shape (output size, number of examples)

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

"""

n\_x = X.shape[0] # size of input layer

n\_h = 1

n\_y = Y.shape[0] # size of output layer

return (n\_x, n\_h, n\_y)

X\_assess, Y\_assess = layer\_sizes\_test\_case()

(n\_x, n\_h, n\_y) = layer\_sizes(X\_assess, Y\_assess)

(n\_x, n\_h, n\_y) = layer\_sizes(X\_assess, Y\_assess)

print("The size of the input layer is: n\_x = " + str(n\_x))

print("The size of the hidden layer is: n\_h = " + str(n\_h))

print("The size of the output layer is: n\_y = " + str(n\_y))

def initialize\_parameters(n\_x, n\_h, n\_y):

"""

Returns:

params: python dict 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(2) # We set up a seed so that our output remains identical.

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

n\_x, n\_h, n\_y = initialize\_parameters\_test\_case()

parameters = initialize\_parameters(n\_x, n\_h, n\_y)

print("W1 = \n" + str(parameters["W1"]))

print("b1 = \n" + str(parameters["b1"]))

print("W2 = \n" + str(parameters["W2"]))

print("b2 = \n" + str(parameters["b2"]))

def forward\_propagation(X, parameters):

"""

Argument:

X -- input data of size (n\_x, m)

parameters -- python dictionary containing our parameters (output of initialization function)

Returns:

A2 -- The sigmoid output of the second activation

cache -- a dictionary containing "Z1", "A1", "Z2" and "A2"

"""

# 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)

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

X\_assess, parameters = forward\_propagation\_test\_case()

A2, cache = forward\_propagation(X\_assess, parameters)

# Note: we use the mean here just to make sure that our output matches test cases.

print(np.mean(cache['Z1']) ,np.mean(cache['A1']),np.mean(cache['Z2']),np.mean(cache['A2']))

def compute\_cost(A2, Y, parameters):

"""

Computes the cross-entropy cost given in equation (13)

Arguments:

A2 -- The sigmoid output of the second activation, of shape (1, number of examples)

Y -- "true" labels vector of shape (1, number of examples)

parameters -- python dictionary containing your parameters W1, b1, W2 and b2

Returns:

cost -- cross-entropy cost given equation (13)

"""

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

# Compute the cross entropy cost

logprobs = np.multiply(np.log(A2), Y) + np.multiply(np.log(1 - A2), 1-Y)

cost = -np.sum(logprobs) / m

cost = np.squeeze(cost) # makes sure the cost is the dimensions we expect.

assert(isinstance(cost, float))

return cost

A2, Y\_assess, parameters = compute\_cost\_test\_case()

print("cost = " + str(compute\_cost(A2, Y\_assess, parameters)))

def backward\_propagation(parameters, cache, X, Y):

"""

Implement the backward propagation using the instructions above.

Arguments:

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)

Returns:

grads -- python dictionary containing your gradients with respect to different parameters

"""

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"]

# Backward propagation: calculate dW1, db1, dW2, db2

dZ2 = A2 - Y

dW2 = np.dot(dZ2, A1.T) / m

db2 = np.sum(dZ2, axis = 1, keepdims = True) / m

dZ1 = np.dot(W2.T, dZ2) \* (1 - np.power(A1, 2))

dW1 = np.dot(dZ1, X.T) / m

db1 = np.sum(dZ1, axis = 1, keepdims = True) / m

grads = {

"dW1": dW1,

"db1": db1,

"dW2": dW2,

"db2": db2

}

return grads

parameters, cache, X\_assess, Y\_assess = backward\_propagation\_test\_case()

grads = backward\_propagation(parameters, cache, X\_assess, Y\_assess)

print ("dW1 = \n"+ str(grads["dW1"]))

print ("db1 = \n"+ str(grads["db1"]))

print ("dW2 = \n"+ str(grads["dW2"]))

print ("db2 = \n"+ str(grads["db2"]))

def update\_parameters(parameters, grads, learning\_rate = 1.2):

"""

Updates parameters using the gradient descent update rule given above

Arguments:

parameters -- python dictionary containing your parameters

grads -- python dictionary containing your gradients

Returns:

parameters -- python dictionary containing your updated parameters

"""

# 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

parameters, grads = update\_parameters\_test\_case()

parameters = update\_parameters(parameters, grads)

print("W1 = \n" + str(parameters["W1"]))

print("b1 = \n" + str(parameters["b1"]))

print("W2 = \n" + str(parameters["W2"]))

print("b2 = \n" + str(parameters["b2"]))

def nn\_model(X, Y, n\_h, num\_iterations = 10000, print\_cost=False):

"""

Arguments:

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

Returns:

parameters -- parameters learnt by the model. They can then be used to predict.

"""

np.random.seed(3)

n\_x = layer\_sizes(X, Y)[0]

n\_y = layer\_sizes(X, Y)[2]

# Initialize parameters, then retrieve W1, b1, W2, b2. Inputs: "n\_x, n\_h, n\_y". Outputs = "W1, b1, W2, b2, parameters".

parameters = initialize\_parameters(n\_x, n\_h, n\_y)

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

# 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: "grads".

grads = backward\_propagation(parameters, cache, X, Y)

# Gradient descent parameter update. Inputs: "parameters, grads". Outputs: "parameters".

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

X\_assess, Y\_assess = nn\_model\_test\_case()

parameters = nn\_model(X\_assess, Y\_assess, 3, num\_iterations=6000, print\_cost=True)

print("W1 = \n" + str(parameters["W1"]))

print("b1 = \n" + str(parameters["b1"]))

print("W2 = \n" + str(parameters["W2"]))

print("b2 = \n" + str(parameters["b2"]))

def predict(parameters, X):

"""

Using the learned parameters, predicts a class for each example in X

Arguments:

parameters -- python dictionary containing your parameters

X -- input data of size (n\_x, m)

Returns

predictions -- vector of predictions of our model (red: 0 / blue: 1)

"""

# Computes probabilities using forward propagation, and classifies to 0/1 using 0.5 as the threshold.

A2, cache = forward\_propagation(X, parameters)

predictions = A2 > 0.5

return predictions

parameters, X\_assess = predict\_test\_case()

predictions = predict(parameters, X\_assess)

print("predictions mean = " + str(np.mean(predictions)))

# Build a model with a n\_h-dimensional hidden layer

parameters = nn\_model(X, Y, n\_h = 3, num\_iterations = 6000, print\_cost=True)

# Plot the decision boundary

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X, Y)

plt.title("Decision Boundary for hidden layer size " + str(4))

# Print accuracy

predictions = predict(parameters, X)

print ('Accuracy: %d' % float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float(Y.size)\*100) + '%')

plt.figure(figsize=(16, 32))

hidden\_layer\_sizes = [1, 2, 3, 4, 5, 20, 50]

for i, n\_h in enumerate(hidden\_layer\_sizes):

plt.subplot(5, 2, i+1)

plt.title('Hidden Layer of size %d' % n\_h)

parameters = nn\_model(X, Y, n\_h, num\_iterations = 5000)

plot\_decision\_boundary(lambda x: predict(parameters, x.T), X, Y)

predictions = predict(parameters, X)

accuracy = float((np.dot(Y,predictions.T) + np.dot(1-Y,1-predictions.T))/float(Y.size)\*100)

print ("Accuracy for {} hidden units: {} %".format(n\_h, accuracy))

Output :-

Screenshot 2024-03-01 094101



