#Planar data classification with one hidden layer

# Package imports

import numpy as np

import matplotlib.pyplot as plt

from testCases\_v2 import \*

import sklearn

import sklearn.datasets

import sklearn.linear\_model

from planar\_utils import plot\_decision\_boundary, sigmoid, load\_planar\_dataset, load\_extra\_datasets

%matplotlib inline

np.random.seed(1) # set a seed so that the results are consistent

# 2.Dataset

X, Y = load\_planar\_dataset()

# Visualize the data:

plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);

### START CODE HERE ### (≈ 3 lines of code)

shape\_X = X.shape

shape\_Y = Y.shape

m = np.size(X[0]) # training set size

### END CODE HERE ###

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

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

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

# thuc hien hoi quy logstic

# Train the logistic regression classifier

clf = sklearn.linear\_model.LogisticRegressionCV();

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

# draw data

plot\_decision\_boundary(lambda x: clf.predict(x), 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)")

# thuc hien mang no\_ron

# GRADED FUNCTION: layer\_sizes

# xay dung so lop an, lop dauvao va lop ra

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

"""

### START CODE HERE ### (≈ 3 lines of code)

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

n\_h = 4

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

### END CODE HERE ###

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)

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))

# khoi tao cac tham so ngau nghien

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

(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))

# lan truyen xuoi

def forward\_propagation(X, parameters):

"""

Argument:

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

parameters -- python dictionary containing your 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"

### START CODE HERE ### (≈ 4 lines of code)

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

### END CODE HERE ###

# Implement Forward Propagation to calculate A2 (probabilities)

### START CODE HERE ### (≈ 4 lines of code)

Z1 = np.dot(W1,X)+b1

A1 = np.tanh((Z1))

Z2 = np.dot(W2,A1)+b2

A2 = 1/(1+np.exp(-Z2))

### END CODE HERE ###

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 your output matches ours.

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

# thuc hien ham mat mat

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

[Note that the parameters argument is not used in this function,

but the auto-grader currently expects this parameter.

Future version of this notebook will fix both the notebook

and the auto-grader so that `parameters` is not needed.

For now, please include `parameters` in the function signature,

and also when invoking this function.]

Returns:

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

"""

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

# Compute the cross-entropy cost

### START CODE HERE ### (≈ 2 lines of code)

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

cost = - (1/m)\*np.sum(logprobs)

### END CODE HERE ###

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

# E.g., turns [[17]] into 17

assert(isinstance(cost, float))

return cost

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

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

# lan truyen nguoc

# GRADED FUNCTION: backward\_propagation

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".

### START CODE HERE ### (≈ 2 lines of code)

W1 =parameters["W1"]

W2 =parameters["W2"]

### END CODE HERE ###

# Retrieve also A1 and A2 from dictionary "cache".

### START CODE HERE ### (≈ 2 lines of code)

A1 = cache["A1"]

A2 = cache["A2"]

### END CODE HERE ###

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

### START CODE HERE ### (≈ 6 lines of code, corresponding to 6 equations on slide above)

dZ2 = A2-Y

dW2 = (1/m)\*np.dot(dZ2,A1.T)

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

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

dW1 = (1/m)\*np.dot(dZ1,X.T)

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

### END CODE HERE ###

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 = "+ str(grads["dW1"]))

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

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

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

# cap nhat cactham so

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"

### START CODE HERE ### (≈ 4 lines of code)

W1 = parameters["W1"]

b1 = parameters["b1"]

W2 = parameters["W2"]

b2 = parameters["b2"]

### END CODE HERE ###

# Retrieve each gradient from the dictionary "grads"

### START CODE HERE ### (≈ 4 lines of code)

dW1 = grads["dW1"]

db1 = grads["db1"]

dW2 = grads["dW2"]

db2 = grads["db2"]

## END CODE HERE ###

# Update rule for each parameter

### START CODE HERE ### (≈ 4 lines of code)

W1 = W1-learning\_rate\*dW1

b1 = b1 - learning\_rate\*db1

W2 = W2 - learning\_rate \*dW2

b2 = b2 - learning\_rate \* db2

### END CODE HERE ###

parameters = {"W1": W1,

"b1": b1,

"W2": W2,

"b2": b2}

return parameters

# thuc hien gop tat cacac chuong trinh vs nhau

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

### START CODE HERE ### (≈ 1 line of code)

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

### END CODE HERE ###

# Loop (gradient descent)

for i in range(0, num\_iterations):

### START CODE HERE ### (≈ 4 lines of code)

# 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)

### END CODE HERE ###

# 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, 4, num\_iterations=10000, print\_cost=True)

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

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

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

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

# du doan lay nguong la 0.5

# GRADED FUNCTION: predict

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.

### START CODE HERE ### (≈ 2 lines of code)

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

predictions = (A2 > 0.5)

### END CODE HERE ###

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 = 4, num\_iterations = 10000, 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))

predictions = predict(parameters, X)

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

# draw predict

# This may take about 2 minutes to run

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))

# Datasets

noisy\_circles, noisy\_moons, blobs, gaussian\_quantiles, no\_structure = load\_extra\_datasets()

datasets = {"noisy\_circles": noisy\_circles,

"noisy\_moons": noisy\_moons,

"blobs": blobs,

"gaussian\_quantiles": gaussian\_quantiles}

### START CODE HERE ### (choose your dataset)

dataset = "noisy\_moons"

### END CODE HERE ###

X, Y = datasets[dataset]

X, Y = X.T, Y.reshape(1, Y.shape[0])

# make blobs binary

if dataset == "blobs":

Y = Y%2

# Visualize the data

plt.scatter(X[0, :], X[1, :], c=Y, s=40, cmap=plt.cm.Spectral);