Neural Networks training

In this practice, we continue working with the dataset with handwritten digits, but the objective is to construct forward and back propagations from scratch for a Neural Network model.

<u>Our code:</u>

```
import numpy as np
import matplotlib.pyplot as plt
from scipy import optimize as opt
from scipy.io import loadmat
from sklearn.preprocessing import PolynomialFeatures
from displayData import displayData
from checkNNGradients import checkNNGradients
import scipy.optimize as opt

data = loadmat('ex3data1.mat')
#se pueden consultar las claves con data.keys( )
y = data ['y']
X = data ['X']
#almacena los datos leídos en X, y

m = len(y)
input_size = X.shape[1]
num_labels = 10

weights = loadmat('ex4weights.mat')
theta1, theta2 = weights['Theta1'], weights['Theta2']
# Theta1 es de dimensión 25 x 401
# Theta2 es de dimensión 10 x 26

y = (y - 1)
y_onehot = np.zeros((m, num_labels))  # 5000 x 10
for i in range(m):
   y_onehot[i][y[i]] = 1

'''
def one_hot(y):
   y = (y - 1)
   y_onehot = np.zeros((m, num_labels))  # 5000 x 10
   for i in range(m):
       y_onehot[i][y[i]] = 1
```

```python
        return y_onehot
'''
def sigmoid(z):
    return 1 / (1 + np.exp(-z))

def derivade_sigmoid(dA):
    return dA * (1 - dA)

def pesosAleatorios(L_in, L_out, epsilon = 0.12):
    #devolverá una matriz de dimensión (L_out, 1 + L_in)
    return np.random.rand(L_out, 1 + L_in) * (epsilon + epsilon) - epsilon

def linear_activation_forward(A_prev, theta):
    Z = np.dot(A_prev, theta.T)
    A = sigmoid(Z)
    return A

def L_model_forward(X, parameters):
    A = X
    cache = {}
    L = len(parameters)
    for l in range(L):
        A_prev = A
        A_prev = np.hstack([np.ones([A_prev.shape[0], 1]), A_prev])
        cache["A" + str(l + 1)] = A_prev
        A = linear_activation_forward(A_prev, parameters['theta' + str(l +
1)])
    cache["A" + str(L + 1)] = A
    return (A, cache)

def cost(parameters, A, Y, lambd):
    reg = (lambd / (2 * m)) * (np.sum(parameters["theta1"][:, 1:] ** 2) +
np.sum(parameters["theta2"][:, 1:] ** 2))
    coste = (Y * np.log(A)) + ((1 - Y) * np.log( 1 - A) )
    return (- 1 / m) * coste.sum() + reg

def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    parameters = {}
    grads = {}
    grads["dT1"] = 0
    grads["dT2"] = 0
    params_rn = params_rn.reshape(len(params_rn), 1)
```

```
    theta1 = np.reshape(params_rn[: num_ocultas * (num_entradas + 1)],
(num_ocultas, (num_entradas + 1)))
    theta2 = np.reshape(params_rn[num_ocultas * (num_entradas + 1) :],
(num_etiquetas, (num_ocultas + 1)))

    parameters['theta1'] = theta1
    parameters['theta2'] = theta2
    AL, cache = L_model_forward(X, parameters)
    coste = cost(parameters, AL, y, reg)
    grads["dA3"] = AL - y
    grads["dA2"] = np.dot(grads["dA3"], parameters['theta2']) *
derivade_sigmoid(cache["A2"])

    grads["dT1"] += (np.dot(grads["dA2"][:, 1:].T, cache["A1"]) / m)
    grads["dT1"][:, 1:] += theta1[:, 1:] * reg / m

    grads["dT2"] += (np.dot(grads["dA3"].T, cache["A2"]) / m)
    grads["dT2"][:, 1:] += theta2[:, 1:] * reg / m

    theta_grads = np.concatenate((grads["dT1"].ravel(), grads["dT2"].ravel()))

    return (coste, theta_grads)



#params_rn = np.concatenate((theta1.ravel(), theta2.ravel()))
#coste, grads = backprop(params_rn, input_size, theta1.shape[0], num_labels,
X, y_onehot, 1)
#print(checkNNGradients(backprop, 1))

def modelo(input_size, num_labels, X, Y, y_onehot, reg, iterations):
    parameters = {}
    inner_layer = 25
    params_rn = np.concatenate((pesosAleatorios(input_size,
inner_layer).ravel(),pesosAleatorios(inner_layer, num_labels).ravel()))

    min = opt.minimize(backprop, params_rn, args=(input_size, inner_layer,
num_labels, X, y_onehot, reg), method='TNC', options={'maxiter': iterations},
jac=True)
    params_rn = min.x

    params_rn = params_rn.reshape(len(params_rn), 1)
```

```
    theta1 = np.reshape(params_rn[: inner_layer * (input_size + 1)],
(inner_layer, (input_size + 1)))
    theta2 = np.reshape(params_rn[inner_layer * (input_size + 1) :],
(num_labels, (inner_layer + 1)))

    parameters['theta1'] = theta1
    parameters['theta2'] = theta2
    AL, _ = L_model_forward(X, parameters)
    indexes = np.argmax(AL, axis=1)
    return str(np.sum(indexes == Y.ravel())/len(X)*100)+"%"


lambd = [1, 1.5, 2]
iterations = [50, 70, 100, 200, 300]

for i in iterations:
    for rate in lambd:
        print(f'Accuracy {modelo(input_size, num_labels, X, y, y_onehot, rate,
i)},when λ= {rate} and № of iterations={i}')
    print("------------------------------------------")
```

Results testing with different learning rate and number of iterations:

Accuracy 85.48%,when **λ**= 1 and № of iterations=50
Accuracy 87.46000000000001%,when **λ**= 1.5 and № of iterations=50
Accuracy 87.0%,when **λ**= 2 and № of iterations=50

Accuracy 92.17999999999999%,when **λ**= 1 and № of iterations=70
Accuracy 93.42%,when **λ**= 1.5 and № of iterations=70
Accuracy 94.06%,when **λ**= 2 and № of iterations=70

Accuracy 95.94%,when **λ**= 1 and № of iterations=100
Accuracy 94.84%,when **λ**= 1.5 and № of iterations=100
Accuracy 95.7%,when **λ**= 2 and № of iterations=100

Accuracy 99.03999999999999%,when **λ**= 1 and № of iterations=200
Accuracy 98.32%,when **λ**= 1.5 and № of iterations=200
Accuracy 97.88%,when **λ**= 2 and № of iterations=200

Accuracy 99.32%,when **λ**= 1 and № of iterations=300
Accuracy 98.92%,when **λ**= 1.5 and № of iterations=300
Accuracy 98.32%,when **λ**= 2 and № of iterations=300

Conclusion:

In this practice, we have implemented both forward and back propagations for a Neural Network model capable of predicting handwritten digits. Additionally, we have tested with different number of iterations and learning rate. From the above-selected **λ**s, our results indicate that “1” and “1.5” give a higher percentage of corrected classified examples. As expected letting the model train longer, higher number of iterations, increments its accuracy.

Gasan Nazer and Veronika Yankova