Práctica 4. Memoria Técnica

Sergio Gavilán Fernández [sgavil01@ucm.es](mailto:sgavil01@ucm.es)

Alejandro Villar Rubio [alvill04@ucm.es](mailto:alvill04@ucm.es)

# Proceso

En primer lugar, hemos implementado el cálculo de la función de coste de la red neuronal, para ello hemos creado la función backprop que utilizando las matrices de pesos dados y la propagación hacia adelante nos permite calcular el coste regularizado y no regularizado. Para poder calcular esta función hemos tenido que crear una nueva matriz Y donde cada vector está formado por ceros exceptuando el valor marcado en la Y del conjunto de datos, que se pone a uno.

En la segunda parte hemos añadido el calculo del gradiente a la función que habíamos definido en la parte anterior, para ello en primer lugar inicializamos la matriz de pesos aleatoriamente entre un rango definido. Después, para cada ejemplo de entrenamiento (x (t) , y(t) ) se ejecuta primero una pasada “hacia adelante” para así calcular la salida de la red hθ(x). A continuación, se ejecuta una pasada “hacia atrás” para computar en cada nodo j de cada capa l su contribución δ (l) j al error que se haya producido en la salida. En esa parte hemos utilizado finalmente el fichero *checkNNGradients.py* que contiene una función que aproxima el valor de la derivada por este método para comprobar nuestro cálculo del gradiente.

Por último, hemos añadido a la función backprop el término de regularización y comprobado la diferencia entre nuestro cálculo y el resultado usando el archivo *checkNNGradients.py.* Finalmente hemos utilizado ***scipy.optimize.minimize*** para entrenar a la red neuronal y obtener los valores para Θ(1) y Θ(2).

# Implementación

## Parte Genérica

import numpy as np

from pandas.io.parsers import read\_csv

import matplotlib.pyplot as plt

import scipy.optimize as opt

from scipy.io import loadmat

from checkNNGradients import checkNNGradients

from displayData import displayData

# Cáculo del coste no regularizado

def coste\_no\_reg(m, K, h, Y):

    return ((1 / m) \* np.sum((0 - Y \* np.log(h)) -

    ((1 - Y) \* np.log(1 - h)), initial=1))

# Cálculo del coste regularizado

def coste\_reg(m, K, h, Y, reg, theta1, theta2):

    return (coste\_no\_reg(m, K, h, Y) +

            ((reg / (2 \* m)) \* (np.sum(theta1, initial=1) \*\* 2 +

            np.sum(theta2, initial=1) \*\* 2)))

# Función sigmoide

def sigmoid(z):

    return 1 / (1 + np.exp(-z))

# Cálculo de la derivada de la función sigmoide

def der\_sigmoid(z):

    return (z \* (1.0 - z))

# Inicializa una matriz de pesos aleatorios

def pesosAleatorios(L\_in, L\_out):

    ini = 0.12

    theta = np.random.uniform(low=-ini, high=ini, size=(L\_out, L\_in))

    theta = np.hstack((np.ones((theta.shape[0], 1)), theta))

    return theta

# Devuelve "Y" a partir de una X y no unos pesos determinados

def forward\_propagate(X, theta1, theta2):

    m = X.shape[0]

    a1 = np.hstack([np.ones([m, 1]), X])    # (5000, 401)

    z2 = np.dot(a1, theta1.T)   # (5000, 25)

    a2 = np.hstack([np.ones([m, 1]), sigmoid(z2)])  # (5000, 26)

    z3 = np.dot(a2, theta2.T)   # (5000, 10)

    h = sigmoid(z3) # (5000, 10)

    return a1, z2, a2, z3, h

# Devuelve el coste y el gradiente de una red neuronal de dos capas

def backprop(params\_rn, num\_entradas, num\_ocultas, num\_etiquetas, X, y, reg):

    m = X.shape[0]

    # Despliegue de params\_rn para sacar las Thetas

    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)))

    a1, z2, a2, z3, h = forward\_propagate(X, theta1, theta2)

    coste = coste\_no\_reg(m, num\_etiquetas, h, y) # Coste sin regularizar

    print(coste)

    costeReg = coste\_reg(m, num\_etiquetas, h, y, reg, theta1, theta2) # Coste regularizado

    print(costeReg)

    # Inicialización de dos matrices "delta" a 0 con el tamaño de los thethas respectivos

    delta1 = np.zeros\_like(theta1)

    delta2 = np.zeros\_like(theta2)

    # Por cada ejemplo

    for t in range(m):

        a1t = a1[t, :] # (1, 401)

        a2t = a2[t, :] # (1, 26)

        ht = h[t, :] # (1, 10)

        yt = y[t]

        d3t = ht - yt

        d2t = np.dot(theta2.T, d3t) \* (a2t \* (1 - a2t)) # (1, 26)

        delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])

        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    delta1 = delta1 / m

    delta2 = delta2 / m

    # Gradiente perteneciente a cada delta

    delta1[:, 1:] = delta1[:, 1:] + (reg \* theta1[:, 1:]) / m

    delta2[:, 1:] = delta2[:, 1:] + (reg \* theta2[:, 1:]) / m

    # Concatenación de los gradientes

    grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))

    return costeReg, grad

## Parte 1

def main():

    data = loadmat("ex4data1.mat")

    y = data["y"].ravel()

    X = data["X"]

    num\_entradas = X.shape[1]

    num\_ocultas = 25

    num\_etiquetas = 10

    # Transforma Y en una matriz de vectores, donde cada vector está formado por todo

    # 0s excepto el valor marcado en Y, que se pone a 1

    # 3 ---> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

    lenY = len(y)

    y = (y - 1)

    y\_onehot = np.zeros((lenY, num\_etiquetas))

    for i in range(lenY):

        y\_onehot[i][y[i]] = 1

    # Crea una X nueva con 100 valores aleatorios de X

    X\_show = np.zeros((100, X.shape[1]))

    for i in range(100):

        random = np.random.randint(low=0, high=X.shape[0])

        X\_show[i] = X[random]

    # Muestra por pantalla algunos ejemplos formados por la nueva X

    displayData(X\_show)

    plt.show()

    # Lectura de los pesos del archivo

    weights = loadmat("ex4weights.mat")

    theta1 = weights["Theta1"] # (25, 401)

    theta2 = weights["Theta2"] # (10, 26)

    # Concatenación de las matrices de pesos en un solo vector

    thetaVec = np.concatenate((np.ravel(theta1), np.ravel(theta2)))

    # Cálculo del coste

    backprop(thetaVec, X.shape[1], num\_ocultas, num\_etiquetas, X, y\_onehot, 0.1)

main()

## Parte2

def main():

    data = loadmat("ex4data1.mat")

    y = data["y"].ravel()

    X = data["X"]

    num\_entradas = X.shape[1]

    num\_ocultas = 25

    num\_etiquetas = 10

    # Transforma Y en una matriz de vectores, donde cada vector está formado por todo

    # 0s excepto el valor marcado en Y, que se pone a 1

    # 3 ---> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

    lenY = len(y)

    y = (y - 1)

    y\_onehot = np.zeros((lenY, num\_etiquetas))

    for i in range(lenY):

        y\_onehot[i][y[i]] = 1

    # Inicialización de dos matrices de pesos de manera aleatoria

    theta1 = pesosAleatorios(400, 25) # (25, 401)

    theta2 = pesosAleatorios(25, 10) # (10, 26)

    # Concatenación de las matrices de pesos en un solo vector

    thetaVec = np.concatenate((np.ravel(theta1), np.ravel(theta2)))

    # Chequeo del gradiente

    checkNNGradients(backprop, 0)

main()

## Parte3

# Cálculo de la precisión

def testClassificator(h, Y):

    aciertos = 0

    for i in range (h.shape[0]):

        max = np.argmax(h[i])

        print(max, "---------------->", Y[i])

        if max == Y[i]:

            aciertos += 1

    precision = (aciertos / h.shape[0]) \* 100

    print("La precisión es del", precision)

def main():

    data = loadmat("ex4data1.mat")

    y = data["y"].ravel()

    X = data["X"]

    num\_entradas = X.shape[1]

    num\_ocultas = 25

    num\_etiquetas = 10

    # Transforma Y en una matriz de vectores, donde cada vector está formado por todo

    # 0s excepto el valor marcado en Y, que se pone a 1

    # 3 ---> [0, 0, 0, 1, 0, 0, 0, 0, 0, 0]

    lenY = len(y)

    y = (y - 1)

    y\_onehot = np.zeros((lenY, num\_etiquetas))

    for i in range(lenY):

        y\_onehot[i][y[i]] = 1

    # Inicialización de dos matrices de pesos de manera aleatoria

    theta1 = pesosAleatorios(400, 25) # (25, 401)

    theta2 = pesosAleatorios(25, 10) # (10, 26)

    # Concatenación de las matrices de pesos en un solo vector

    thetaVec = np.concatenate((np.ravel(theta1), np.ravel(theta2)))

    # Obtención de los pesos óptimos entrenando una red con los pesos aleatorios

    optTheta = opt.minimize(fun=backprop, x0=thetaVec,

            args=(num\_entradas, num\_ocultas, num\_etiquetas,

            X, y\_onehot, 1), method='Newton-CG', jac=True,

            options={'maxiter': 70})

    # Desglose de los pesos óptimos en dos matrices

    newTheta1 = np.reshape(optTheta.x[:num\_ocultas \* (num\_entradas + 1)],

        (num\_ocultas, (num\_entradas + 1)))

    newTheta2 = np.reshape(optTheta.x[num\_ocultas \* (num\_entradas + 1): ],

        (num\_etiquetas, (num\_ocultas + 1)))

    # H, resultado de la red al usar los pesos óptimos

    a1, z2, a2, z3, h = forward\_propagate(X, newTheta1, newTheta2)

    # Cálculo de la precisión

    testClassificator(h, y)

main()