PRÁCTICA 4

Entrenamiento de redes neuronales

María García Raldúa
Luisa Cardoso Macedo

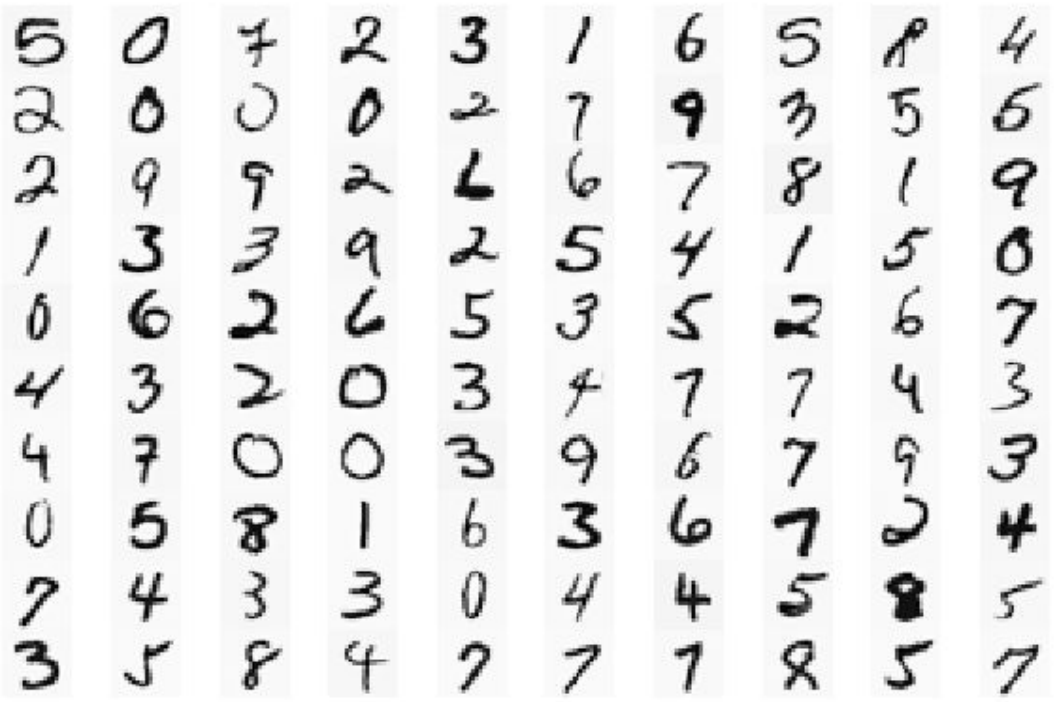
Librerías importadas:

```
from scipy.io import loadmat
import numpy as np
from matplotlib import pyplot as plt
from scipy import integrate as integrate
import random as rnd
import scipy.optimize as opt
import displayData as dp
import checkNNGradients as check

from pandas.io.parsers import read_csv
```

Lectura de datos: 100 números aleatorios

```
data = loadmat ("ex4data1.mat")

y = data ["y"]
X = data ["X"]



muestras = len(X)

Y = np.ravel(y)
weights = loadmat('ex4weights.mat')
thetha1,thetha2 = weights['Theta1'], weights['Theta2']
sample = np.random.choice(X.shape[0], 100)
im = X[sample,:]
dp.displayData(im) #esto muestra la tabla con 100 numeros random
dp.displayImage(X[4999]) #esto muestra solo una imagen
plt.show()
```

Resultado:



Funciones de activación de redes neuronales:

```
def sigmoid(z):
    return 1.0/(1.0 + np.exp(-z))

def sigmoidDeriv(z):
    return sigmoid(z) * (1.0 - sigmoid(z))
```

Función forward propagation:

Usa un valor de entrada para predecir una salida mediante matriz de pesos y una función de activación.

```
def forwardProp(thetas1, thetas2,X):
    z2 = thetas1.dot(X.T)
    a1 = sigmoid(z2)
    tuple = (np.ones(len(a1[0])), a1)
    a1 = np.vstack(tuple)
    z3 = thetas2.dot(a1)
    a2 = sigmoid(z3)
    return a1,z2,a2,z3
```

Predicción del elemento 1000 según X:

```
x_aux = np.hstack([np.ones((len(X),1),dtype = np.float),X])
print("valor del elemento 1000 de X según la hipotesis: ",
forwardProp(thetha1,thetha2,x_aux)[3].T[1000].argmax())
```

Salida: valor del elemento 1000 de X según la hipótesis: 1

El coste está implementado con regularización. Se prepara un vector de Y distinto al recibido, una matriz de n_elemtnos y n_etiquetas, donde cada fila corresponde a un caso.

Función one_hot_y:

```
def one_hot_Y(y,num_etiquetas):
    yaux = np.array(y) - 1
    y_onehot = np.zeros((len(y),num_etiquetas))
    for i in range(len(y_onehot)):
        y_onehot[i][yaux[i]] = 1

    return y_onehot
```

Función de coste según los apuntes:
J(θ) = 1 m Xm i=1 X K k=1 h −y (i) k log((hθ(x (i)))k) − (1 − y (i) k) log(1 − (hθ(x (i)))k) i
Función programada:

```
def fun_coste(X,Y, theta1,theta2,reg):
    X = np.array(X)
    Y = np.array(Y)
    theta1 = np.array(thetha1)
    theta2 = np.array(thetha2)
    h = forwardProp(theta1,theta2,X)[3]

    cost = np.sum((-Y.T) * (np.log(h)) - (1 - Y.T) * (np.log(1 -
h)))/len(Y)

    regcost = np.sum(np.power(theta1[:, 1:], 2)) +
np.sum(np.power(theta2[:,1:], 2))
    regcost = regcost * (reg/(2*len(Y)))

    return cost + regcost
```

Comprobamos con 10 etiquetas:

```
y_aux = one_hot_Y(Y,10)
```

```
print("El coste de thetas entrenados es: ",
fun_coste(x_aux,y_aux,thetha1,thetha2,1))
```

Resultado:

```
El coste de thetas entrenados es: 0.3837698590909236
```

Función backprop:

Para repartir el error entre las redes neuronales. Va desde la última capa hasta la penúltima, sin repartir error en la capa de entrada:

```
def backProp(params_rn,num_entradas,num_ocultas,num_etiquetas,X,Y,reg):
    th1 = np.reshape(params_rn[:num_ocultas * (num_entradas + 1)],
    (num_ocultas,(num_entradas + 1)))
    th2 = np.reshape(params_rn[num_ocultas * (num_entradas + 1):],
    (num_etiquetas,(num_ocultas + 1)))



    x_unos = np.hstack([np.ones((len(X),1)),X])
    y = np.zeros((len(X),num_etiquetas))
    y = one_hot_Y(Y,num_etiquetas)
    cost = fun_coste(x_unos, y, th1, th2, reg)
    z2,a1,z3,a2 = forwardProp(th1,th2,x_unos)



    gradW1 = np.zeros(th1.shape)
    gradW2 = np.zeros(th2.shape)

    delta3 = np.array(a2 - y.T)
    delta2 = th2.T[1:,:].dot(delta3)*sigmoidDeriv(z2)

    gradW1 = gradW1 + (delta2.dot(x_unos))
    gradW2 = gradW2 + (delta3.dot(a1.T))

    reg1 = (gradW1/float(len(X)))
    reg2 = (gradW2/float(len(X)))



    gradW1 = (gradW1/len(X))
    gradW2 = (gradW2/len(X))

    reg1[:,1:] = reg1[:,1:] + (float(reg)/float(len(X)) * th1[:, 1:])
    reg2[:, 1:] = reg2[:,1:] + (float(reg)/float(len(X)) * th2[:, 1:])
```

```
    gradient = np.concatenate((reg1,reg2),axis = None)

    return cost, gradient
```

Resultado:

```
diferencias entre gradientes:
 [-1.15261793e-11 -1.27858835e-12  3.71772058e-12  1.05840475e-11
 -6.54325899e-12 -9.02861119e-13  2.23431690e-12  1.21962440e-11
 -2.38443640e-11  2.71589695e-12 -6.86591062e-12 -1.00959241e-11
  5.90455462e-12 -2.61959898e-12  6.02981554e-12  2.47921128e-11
 -4.40363024e-13  2.67175171e-13  3.52642048e-12  1.03782816e-11
  6.20863361e-11  1.44565471e-11  6.42558229e-12  5.51581003e-12
  1.45199963e-11  2.23619179e-11  6.89308610e-11  1.42481582e-11
  7.72014397e-12  1.16234244e-11  1.58183466e-11  2.05401252e-11
  7.40916772e-11  1.15860654e-11  6.46410703e-12  1.74119608e-11
  1.72302173e-11  1.96654637e-11]
```

Función que inicializa los thetas aleatoriamente:

```
def pesosAleatorios(L_in,L_out):
    cini = 0.12
    aux = np.random.uniform(-cini, cini, size =(L_in, L_out))
    aux = np.insert(aux,0,1,axis = 0)
    return aux
```

Función que prueba la red neuronal:

Con matrices de pesos inicializados anteriormente usando optimize

```
def NNTest (num_entradas, num_ocultas, num_etiquetas, reg, X, Y, laps):
    t1 = pesosAleatorios(num_entradas, num_ocultas)
    t2 = pesosAleatorios(num_ocultas, num_etiquetas)

    params = np.hstack((np.ravel(t1),np.ravel(t2)))
    out = opt.minimize(fun=backProp,x0 = params,args =
(num_entradas,num_ocultas,num_etiquetas,X,Y,reg),method = 'TNC',jac =
True,options={'maxiter': laps})

    Thetas1 =
out.x[:(num_ocultas*(num_entradas+1))].reshape(num_ocultas,(num_entrada
s+1))
```

```
    Thetas2 =
out.x[(num_ocultas*(num_entradas+1)):].reshape(num_etiquetas,(num_ocult
as+1))

    input = np.hstack([np.ones((len(X), 1), dtype = np.float), X])
    hipo = forwardProp(Thetas1, Thetas2, input)[3]



    Ghipo = (hipo.argmax(axis = 0))+1
    print(Ghipo.shape)
    prec = (Ghipo == Y)*1

    precision = sum(prec) / len(X)

    print("Program precision: ", precision *100, "%")
```

Usando 400 entradas, 25 ocultas, 10 etiquetas, X e Y y 70 vueltas:
Precisión de **92.88%**
Esta precisión es variable cada vez que lo ejecutamos, pero ronda los 80-100%