Practica 4

Antonio Rodriguez Hurtado y Miguel Ferreras Chumillas

Librerias y datos usados en la practica

```
#   Librerias
from scipy.io import loadmat
import numpy as np
import checkNNGradients as ch
import scipy.optimize as opt



#   Extraemos los ejemplos de entrenamiento del archivo que los contiene
data = loadmat("ex4data1.mat")

#   Separamos los atributos de los ejemplos y sus etiquetas (El 0 viene etiquetado como 10)
y = data['y'].ravel()
X = data['X']
m = len(y)
input_size = X.shape[1]

#   Obtenemos las distintas etiquetas
etiquetas = np.unique(y)
num_labels = etiquetas.size

#   Definimos nuestra termino de regularizacion
reg = 1

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

#   Obtenemos las matrices de pesos del archivo
weights = loadmat('ex4weights.mat')
theta1, theta2 = weights['Theta1'], weights['Theta2']
```

Calculo del coste

Primero se define la funcion del coste sin regularizar

```
def cost(H, Y, Theta1, Theta2, reg):
    m = len(Y)
    th1 = np.delete(Theta1, 0, axis=1)
    th2 = np.delete(Theta2, 0, axis=1)
    suma = 0
    for i in range(m):
        suma += np.sum((np.matmul(-Y[i,:], np.log(H[i,:])) - np.matmul((1 - Y[i,:]), np.log(1 - H[i,:]))))
    return ((1 / m) * suma) + ((reg / (2 * m)) * (np.sum(np.power(th1, 2)) + np.sum(np.power(th2, 2))))

A1, Z2, A2, Z3, H = forward_propagate(X, theta1, theta2)
coste = cost(H, y_onehot, theta1, theta2, reg)
print(coste)
```

```
0.28762916516131876
```

A continuacion se añade el termino de regularizacion al coste y se vuelve a calcular

```
def cost(H, Y, Theta1, Theta2, reg):
    th1 = np.delete(Theta1, 0, axis=1)
    th2 = np.delete(Theta2, 0, axis=1)
    suma = 0
    for i in range(m):
        suma += np.sum((np.matmul(-Y[i,:], np.log(H[i,:])) - np.matmul((1 - Y[i,:]), np.log(1 - H[i,:]))))
    return ((1 / m) * suma) + ((reg / (2 * m)) * (np.sum(np.power(th1, 2)) + np.sum(np.power(th2, 2))))

A1, Z2, A2, Z3, H = forward_propagate(X, theta1, theta2)
coste = cost(H, y_onehot, theta1, theta2, reg)
print(coste)
```

```
0.3837698590909235
```

Calculo del gradiente

Se crea la funcion backprop que coge el coste y calcula el gradiente para devolver una tupla con ambos.

```
#   Definimos la funcion de propagacion hacia
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    m = X.shape[0]
    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)
    #Llamada a la funcion para calcular el coste
    coste = cost(H, y, Theta1, Theta2, reg)

    #Back-propagation
    delta1 = np.zeros(theta1.shape)
    delta2 = np.zeros(theta2.shape)

    Delta1, Delta2 = np.zeros(Theta1.shape), np.zeros(Theta2.shape)
    sigma3 = (H - y)
    Delta2 += np.dot(sigma3.T, A2)
    Delta1 += np.dot(np.delete(np.dot(sigma3, Theta2) * (A2 * (1 - A2)), 0, axis=1).T, A1)
    D1 = Delta1 / m
    D2 = Delta2 / m
    #Regularizacion del gradiente
    D1[:, 1:] = D1[:, 1:] + (reg * Theta1[:, 1:]) / m
    D2[:, 1:] = D2[:, 1:] + (reg * Theta2[:, 1:]) / m

    return coste, np.concatenate((D1, D2), axis=None)
```

A continuacion se comprueba con 'checkNNGradients' tanto con el termino de regularizacion añadido como sin el.

```
param = np.concatenate((theta1, theta2), axis=None)
ch.checkNNGradients(backprop, 0)
```

```
grad shape:  (38,)
num grad shape:  (38,)

array([ 5.27761168e-11, -3.29852031e-13,  7.89324162e-12,  9.17629167e-12,
       -6.30465125e-11,  2.08456863e-12, -1.07556464e-11, -5.04682407e-11,
       -9.29989974e-11,  7.04843475e-12, -4.20321139e-11, -1.22385352e-10,
       -1.95650579e-11,  2.76548055e-12, -6.02735223e-12, -2.27557001e-11,
        2.15736456e-11, -4.96176017e-13,  1.19978472e-11,  2.73879391e-11,
        6.25964836e-11,  1.55131741e-11,  1.12525544e-11,  7.48809348e-12,
        1.90088223e-11,  2.10645668e-11,  7.15513759e-11,  1.56080426e-11,
        4.89147611e-12,  1.37491685e-11,  1.93192129e-11,  1.79336823e-11,
        7.32915950e-11,  1.60134683e-11,  1.08387743e-11,  1.78091986e-11,
        1.43913215e-11,  2.04546380e-11])
```

```
param = np.concatenate((theta1, theta2), axis=None)
ch.checkNNGradients(backprop, reg)
```

```
grad shape:  (38,)
num grad shape:  (38,)

array([ 5.27761168e-11, -1.48772661e-12,  8.82988127e-12,  9.75091535e-12,
       -6.30465125e-11,  2.10970130e-12, -1.16537752e-11, -4.92537400e-11,
       -9.29989974e-11,  5.59484403e-12, -4.34997871e-11, -1.22203025e-10,
       -1.95650579e-11,  2.13601359e-12, -7.00919878e-12, -2.43030734e-11,
        2.15736456e-11,  2.27623476e-13,  1.19978472e-11,  2.62300737e-11,
        6.25964836e-11,  1.38673517e-11,  1.07264475e-11,  7.51324003e-12,
        2.03311395e-11,  2.00586214e-11,  7.15513759e-11,  1.63749014e-11,
        3.42380291e-12,  1.39315226e-11,  1.87037746e-11,  1.95246597e-11,
        7.32915950e-11,  1.66865410e-11,  1.07713560e-11,  1.63125624e-11,
        1.34624811e-11,  2.22044327e-11])
```

Aprendizaje de los parametros

En primer lugar se definen las funciones necesarias para el entrenamiento

```
def pesosAleatorios(L_in, L_out):
    ini_epsilon = 0.12
    theta = np.random.rand(L_out, 1 + L_in) * (2*ini_epsilon) - ini_epsilon
    return theta

def train(X, y, reg, iters):
    num_entradas = X.shape[1]
    num_ocultas = 25
    num_etiquetas = 10

    theta1 = pesosAleatorios(num_entradas, num_ocultas)
    theta2 = pesosAleatorios(num_ocultas, num_etiquetas)
    params = np.concatenate((np.ravel(theta1), np.ravel(theta2)))

    fmin = opt.minimize(fun=backprop, x0=params,
                args=(num_entradas, num_ocultas, num_etiquetas, X, y, reg),
                method='TNC', jac=True, options={'maxiter' : iters})

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

    a1, z2, a2, z2, h = forward_propagate(X, theta1, theta2)
```

```
    predictions = np.argmax(h, axis=1)
    return predictions
```

A continuacion se prueba con los parametros dados para comprobar que funciona la red

```
predictions = train(X, y_onehot, reg=1, iters=70)

fallos =  np.where([predictions != y])[1]
print('Numero de fallos:', len(fallos))

aciertos = np.where([predictions == y])[1]
print('Numero de aciertos:', len(aciertos))

accuracy = 100 * np.mean(predictions == y)
print("\nPorcentaje de aciertos: ", accuracy)
```

```
Numero de fallos: 310
Numero de aciertos: 4690

Porcentaje de aciertos:  93.8
```

Una vez comprobado que la red funciona bien con el coeficiente regulador y las iteraciones estandar, comenzamos a variarlos para ver como cambia el resultado.

Probamos con numeros bajos y altos de iteraciones:

```
predictions = train(X, y_onehot, reg=1, iters=10)
accuracy = 100 * np.mean(predictions == y)
print("\nPorcentaje de aciertos: ", accuracy)
```

```
Porcentaje de aciertos:  10.100000000000001
```

```
Porcentaje de aciertos:  40.62
```

```
Porcentaje de aciertos:  29.459999999999997
```

```
Porcentaje de aciertos:  53.76
```

Con tan solo 10 iteraciones la red no solo consigue resultados muy bajos de acierto, si no que ademas son totalmente volatiles, cambiando hasta en un 40% de un intento a otro. Esto implica que con tan pocas iteraciones la red no es capaz de sacar resultados concluyentes de los datos y los clasifica en gran medida a boleo.

```
predictions = train(X, y_onehot, reg=1, iters=200)
accuracy = 100 * np.mean(predictions == y)
print("\nPorcentaje de aciertos: ", accuracy)
```

```
Porcentaje de aciertos:  99.3
```

Por el contrario con 200 iteraciones el resultado no baja del 98% lo que implica que la red ha sobreaprendido los datos y probablemente no sea capaz de procesar correctamente nuevos datos

Lo siguiente que hicimos fue cambiar el coeficiente de regularizacion

```
predictions = train(X, y_onehot, reg=0, iters=70)
accuracy = 100 * np.mean(predictions == y)
print("\nPorcentaje de aciertos: ", accuracy)
```

```
Porcentaje de aciertos:  92.54
```

```
Porcentaje de aciertos:  90.08
```

```
Porcentaje de aciertos:  76.72
```

Sorprendentemente con un coeficiente de regulacion de 0 el resultado suele ser bastante bueno, y aunque el porcentaje baja alguna vez al 70-80% suele mantenerse en torno al 92 en la mayoria de ejecuciones

```
predictions = train(X, y_onehot, reg=100, iters=70)
accuracy = 100 * np.mean(predictions == y)
print("\nPorcentaje de aciertos: ", accuracy)
```

```
Porcentaje de aciertos:  81.46
```

Con un coeficiente de 100 el porcentaje que se obtiene esta entre el 75-85% lo cual sigue siendo aceptable para la red. Asi que probamos a subirlo más.

```
predictions = train(X, y_onehot, reg=500, iters=70)
accuracy = 100 * np.mean(predictions == y)
print("\nPorcentaje de aciertos: ", accuracy)
```

```
Porcentaje de aciertos:  25.779999999999998
```

Esta vez el porcentaje si baja mucho al no regularse correctamente.

Funciones adicionales de la práctica

```
#   Definimos la funcion sigmoide
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

```
#   Defininmos la funcion de hipotesis
def hip(a, b):
    return sigmoid(np.matmul(a, b))

#   Definimos la funcion de propagacion hacia adelante
def forward_propagate(X, Theta1, Theta2):
    m = X.shape[0]
    A1 = np.hstack([np.ones([m, 1]), X])
    Z2 = np.dot(A1, Theta1.T)
    A2 = np.hstack([np.ones([m, 1]), sigmoid(Z2)])
    Z3 = np.dot(A2, Theta2.T)
    H = sigmoid(Z3)
    return A1, Z2, A2, Z3, H
```