Practica 4

Aurora García Jover
Anass Carreau Allagui

Primero calculamos la función de coste:
Con los valores de las matrices Theta1 y theta2 comprobamos que nuestra función de coste funciona correctamente.
Para el cálculo del coste tendremos que calcular la hipótesis hθ(x (i)) que calcularemos con la propagación hacia adelante.
Por último añadimos a la función de coste el término de regularización.
Con los valores de theta dados conseguimos un coste de 0.3837698590909236.

```
def costFun(X, y, theta1, theta2, reg):
    #Cambios para poder operar
    X = np.array(X)
    y = np.array(y)
    muestras = len(y)

    theta1 = np.array(theta1)
    theta2 = np.array(theta2)

    #predecimos la salida de los valores para la matriz de pesos de
theta y nos quedamos con el valor predicho en la
    #variable hipothesis y calculamos el coste con el valor de dicha
variable
    hipothesis  = forward_propagate(theta1, theta2, X)[3]
    cost = np.sum((-y.T)*(np.log(hipothesis)) - (1-y.T)*(np.log(1-
hipothesis)))/muestras

    #calculo del coste con regularización
    regcost = np.sum(np.power(theta1[:, 1:], 2)) +
np.sum(np.power(theta2[:,1:], 2))
    regcost = regcost * (reg/(2*muestras))

    return cost + regcost
```

función que utiliza un valor de entrada para predecir el de salida utilizando una matriz de pesos

```
def fwd_propagate(Theta1, Theta2, X):
    z1 = Theta1.dot(X.T)
    a1 = sigmoid(z1)
    tuple = (np.ones(len(a1[0])), a1)
    a1 = np.vstack(tuple)
```

```
    z2 = Theta2.dot(a1)
    a2 = sigmoid(z2)
    return z1, a1, z2, a2
```

```
X_aux = np.hstack([np.ones((len(X), 1)), X])
print("Valor predicho para el elemento 0 de X segun la hipotesis: ",
(forward_propagate(thetas1, thetas2, X_aux)[3]).T[0].argmax())
```

```
def getYMatrix(Y, nEtiquetas):
    nY = np.zeros((len(Y), nEtiquetas))
    yaux = np.array(Y) -1

    for i in range(len(nY)):
        z = yaux[i]
        if(isinstance(z, np.uint8)):
            if(z == 10): z = 0
            nY[i][z] = 1
        else:
            z = yaux[i].all()
            if(z == 10): z = 0
            nY[i][z] = 1

    return nY
```

```

Y_aux = getYMatrix(Y, 10)

print("El coste con thetas entrenados es: ", costFun(X_aux, Y_aux,
thetas1, thetas2, 1))
```

Después calculamos el gradiente:

primero reconstruimos las matrices Theta1 y Theta2 a partir de los parámetros de la red

```
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)))
    # theta2 es un array de (num_etiquetas, num_ocultas)
    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])
    nMuestras = len(X)
    y = np.zeros((nMuestras, num_etiquetas))

    y = y + getYMatrix(Y, num_etiquetas)

    coste = costFun(X_unos, y, th1, th2, reg)

    #Backpropagation

    # Forward propagation para obtener una hipótesis y los valores
intermedios
    # de la red neuronal
    z2, a2, z3, a3 = forward_propagate(th1, th2, X_unos)

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

    # Coste por capas
    delta3 = np.array(a3 - y.T)
    delta2 = th2.T[1:, :].dot(delta3)*sigmoideDerivada(z2)

    # Acumulacion de gradiente
    gradW1 = gradW1 + (delta2.dot(X_unos))
    gradW2 = gradW2 + (delta3.dot(a2.T))

    G1 = gradW1/float(nMuestras)
    G2 = gradW2/float(nMuestras)

    # suma definitiva
    G1[:, 1: ] = G1[:, 1:] + (float(reg)/float(nMuestras))*th1[:, 1:]
```

```
    G2[:, 1: ] = G2[:, 1:] + (float(reg)/float(nMuestras))*th2[:, 1:]

    gradients = np.concatenate((G1, G2), axis = None)

    return coste, gradients
```

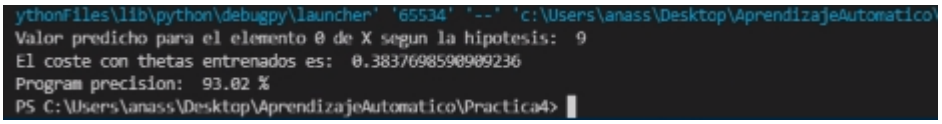
```
params = np.concatenate((theta1, theta2), axis = None)
print("Diferencias al comprobar gradientes:\n",
check.checkNNGradients(backprop, 1))
```

Probamos la red neuronal

```
def NNTest (num_entradas, num_ocultas, num_etiquetas, reg, X, Y, laps):
    t1 = InitRandomWeight(num_entradas, num_ocultas)
    t2 = InitRandomWeight(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)), X])
    hipo = forward_propagate(Thetas1, Thetas2, input)[3]


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

    precision = sum(prec) / len(X)

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

```
NNTest(400, 25, 10, 1, X, Y, 70)
```

nos devuelve este resultado:



La función sigmoide es la misma que la de la practica anterior