# Aprendizaje Automático y minería de datos: Práctica 4

***Autores:*** *Tatiana Duarte Balvís, Miguel Mur Cortés*

## 1. Objetivo:

El objetivo de esta práctica se divide en dos partes. En primer lugar, se deberá implementar el cálculo de la función de coste de una red neuronal para un conjunto de ejemplos de entrenamiento con el mismo conjunto de datos de la práctica anterior. En segundo lugar, se deberá implementar el cálculo del gradiente de una red neuronal.

## 2. Función de coste:

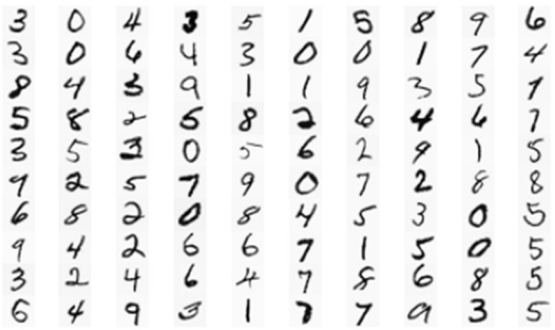
Mediante la función displayData(), usando los datos proporcionados se puede obtener una imagen con la que podemos visualizar una muestra de los datos de entrenamiento.

```
datos = loadmat("ex4data1.mat")
X = datos["X"]
y = datos["y"]
y = np.ravel(y)

m = len(y)
n_input = np.shape(X)[1]
n_hidden = 25
n_labels = 10

aux_y = (y-1)
y_onehot = np.zeros((m, n_labels))
for i in range(m):
    y_onehot[i][aux_y[i]] = 1

#pinta 100 ejemplos
sample = np.random.choice(X.shape[0], 100)
plt.figure()
dD.displayData(X[sample])
plt.savefig("fig1.png")
```



Con los valores proporcionados, comprobamos si realizamos correctamente el cálculo del coste de manera regularizada y sin regularizar.
Para el cálculo del coste sin regularizar se requiere implementar el algoritmo de propagación hacia delante.

```
def forward_propagate(X, Theta1, Theta2):
    m = np.shape(X)[0]

    #Input layer
    A1 = np.hstack([np.ones([m, 1]), X])
    #Hidden layer
    Z2 = np.dot(A1, Theta1.T)
    A2 = np.hstack([np.ones([m, 1]), sigmoide(Z2)])
    #Output layer
    Z3 = np.dot(A2, Theta2.T)
    H = sigmoide(Z3)

    return A1, A2, H

def sigmoide(z): #g(z)
    return 1 / (1 + np.exp(-z))
```

Con la propagación hacia delante, podemos calcular el coste de la siguiente manera:

```
def coste(X, y, Theta1, Theta2):
    a1, a2, h = forward_propagate(X, Theta1, Theta2)
    op1 = -(y * np.log(h))
    op2 = -((1-y) * np.log(1-h))

    return np.sum(op1 + op2) / np.shape(X)[0]
```

Esta primera versión de la función de coste debería devolver un valor aproximado de 0.287629

```
print("Coste sin regularizar: " + str(coste(X, y_onehot, Theta1,
Theta2))[:5])
```

Output: `Coste sin regularizar: 0.287`

Para el cálculo de la función de coste regularizada, implementamos la siguiente función:

```
def coste_reg(X, y, Theta1, Theta2, lamda):
    op1 = coste(X, y, Theta1, Theta2)
    op2 = lamda * (np.sum(Theta1[:, 1:]**2) + np.sum(Theta2[:,
1:]**2)) / (2*np.shape(X)[0])

    return op1 + op2
```

Que debería devolver un valor aproximado de 0.38770 para λ = 1

```
print("Coste regularizado con lambda=1: " + str(coste_reg(X, y_onehot,
Theta1, Theta2, 1))[:5])

Output: Coste regularizado con lambda=1: 0.383
```

## 3. Cálculo del gradiente:

En este apartado se calcula el gradiente mediante una función back_propagate que además devuelva el resultado de la función de coste.

```
def back_propagate (params_rn, n_input, n_hidden, n_labels, X, y,
lamda):
    Theta1 = np.reshape(params_rn[:n_hidden * (n_input+1)],
(n_hidden, (n_input+1)))
    Theta2 = np.reshape(params_rn[n_hidden * (n_input+1):],
(n_labels, (n_hidden+1)))

    m = np.shape(X)[0]
    A1, A2, H = forward_propagate(X, Theta1, Theta2)

    Delta1 = np.zeros_like(Theta1)
    Delta2 = np.zeros_like(Theta2)

    for t in range(m):
        a1t = A1[t, :]
        a2t = A2[t, :]
        ht = H[t, :]
        yt = y[t]

        d3t = ht - yt
        d2t = np.dot(Theta2.T, d3t) * (a2t * (1 - a2t))

        Delta1 = Delta1 + np.dot(d2t[1:, np.newaxis],
a1t[np.newaxis, :])
```

```
            Delta2 = Delta2 + np.dot(d3t[:, np.newaxis],
a2t[np.newaxis, :])

        #Calculos de gradiente
        gradiente1 = Delta1 / m
        gradiente2 = Delta2 / m
        lamda1 = lamda * Theta1 / m
        lamda2 = lamda * Theta2 / m
        lamda1[:, 0] = lamda2[:, 0] = 0
        gradiente1 += lamda1
        gradiente2 += lamda2

        coste = coste_reg(X, y, Theta1, Theta2, lamda)
        gradiente = np.concatenate((np.ravel(gradiente1),
np.ravel(gradiente2)))

        return coste, gradiente
```

Para calcular la veracidad de los resultados, utilizamos la función *checkNNGradients()*.

```
lamda = 1
print("--Comprobación del gradiente--")
diff = checkNNG.checkNNGradients(back_propagate, lamda)
print("Menor diferencia: " + str(min(diff)))
print("Mayor diferencia: " + str(max(diff)))
```

```
Output:

--Comprobación del gradiente--

Menor diferencia: -1.2664391757510884e-10

Mayor diferencia: 7.551204106448495e-11
```

## 4. Aprendizaje de los parámetros:

En este último apartado se debe entrenar la red neuronal mediante la función scipy.optimize.minime con 70 iteraciones. El objetivo es obtener una precisión alrededor del 93%.

Los pesos de la red neuronal se deben inicializar con valores aleatorios en el rango [-0.12, 0.12]

```
def pesosAleatorios(L_ini, L_out):
    E_ini = 0.12
    return np.random.random((L_out, L_ini + 1)) * (2*E_ini) - E_ini
```

Una vez inicializados, entrenamos a la red neuronal con la función minimize:

```
#n_iters = 70
resOpt = opt.minimize(fun=back_propagate, x0=params_rn,
args=(n_input, n_hidden, n_labels, X, y_onehot, lamda),
method='TNC', jac=True, options={'maxiter': 70})
```

Para validar los resultados, implementamos una función de validación.

```
def comprobar(resOpt, n_input, n_hidden, n_labels, X, y):
    Theta1 = np.reshape(resOpt.x[:n_hidden * (n_input + 1)] ,
(n_hidden, (n_input+1)))
    Theta2 = np.reshape(resOpt.x[n_hidden * (n_input + 1):] ,
(n_labels, (n_hidden+1)))

    A1, A2, H = forward_propagate(X, Theta1, Theta2)

    aux = np.argmax(H, axis=1)
    aux += 1

    return np.sum(aux == y) / np.shape(H)[0]

evaluacion = comprobar(resOpt, n_input, n_hidden, n_labels, X, y)
print("Evaluación del entrenamiento de la red:
{}%".format(str(evaluacion*100)[:5]))

Output: Evaluación del entrenamiento de la red: 93.58%
```

## 5. Código:

```
from scipy.io import loadmat
import scipy.optimize as opt
import numpy as np
import matplotlib.pyplot as plt

import displayData as dD
import checkNNGradients as checkNNG


def sigmoide(z): #g(z)
    return 1 / (1 + np.exp(-z))


def coste(X, y, Theta1, Theta2):
    a1, a2, h = forward_propagate(X, Theta1, Theta2)
    op1 = -(y * np.log(h))
    op2 = -((1-y) * np.log(1-h))

    return np.sum(op1 + op2) / np.shape(X)[0]


def coste_reg(X, y, Theta1, Theta2, lamda):
    op1 = coste(X, y, Theta1, Theta2)
    op2 = lamda * (np.sum(Theta1[:, 1:]**2) + np.sum(Theta2[:, 1:]**2))
/ (2*np.shape(X)[0])

    return op1 + op2


def forward_propagate(X, Theta1, Theta2):
    m = np.shape(X)[0]

    #Input layer
    A1 = np.hstack([np.ones([m, 1]), X])
    #Hidden layer
    Z2 = np.dot(A1, Theta1.T)
    A2 = np.hstack([np.ones([m, 1]), sigmoide(Z2)])
    #Output layer
    Z3 = np.dot(A2, Theta2.T)
    H = sigmoide(Z3)

    return A1, A2, H


def back_propagate (params_rn, n_input, n_hidden, n_labels, X, y,
lamda):
```

```
    Theta1 = np.reshape(params_rn[:n_hidden * (n_input+1)], (n_hidden,
(n_input+1)))
    Theta2 = np.reshape(params_rn[n_hidden * (n_input+1):], (n_labels,
(n_hidden+1)))

    m = np.shape(X)[0]
    A1, A2, H = forward_propagate(X, Theta1, Theta2)

    Delta1 = np.zeros_like(Theta1)
    Delta2 = np.zeros_like(Theta2)

    for t in range(m):
        a1t = A1[t, :]
        a2t = A2[t, :]
        ht = H[t, :]
        yt = y[t]

        d3t = ht - yt
        d2t = np.dot(Theta2.T, d3t) * (a2t * (1 - a2t))

        Delta1 = Delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis,
:])
        Delta2 = Delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    #Calculos de gradiente
    gradiente1 = Delta1 / m
    gradiente2 = Delta2 / m
    lamda1 = lamda * Theta1 / m
    lamda2 = lamda * Theta2 / m
    lamda1[:, 0] = lamda2[:, 0] = 0
    gradiente1 += lamda1
    gradiente2 += lamda2

    coste = coste_reg(X, y, Theta1, Theta2, lamda)
    gradiente = np.concatenate((np.ravel(gradiente1),
np.ravel(gradiente2)))

    return coste, gradiente


def pesosAleatorios(L_ini, L_out):
    E_ini = 0.12
    return np.random.random((L_out, L_ini + 1)) * (2*E_ini) - E_ini


def comprobar(resOpt, n_input, n_hidden, n_labels, X, y):
    Theta1 = np.reshape(resOpt.x[:n_hidden * (n_input + 1)] , (n_hidden,
(n_input+1)))
```

```
    Theta2 = np.reshape(resOpt.x[n_hidden * (n_input + 1):] , (n_labels,
(n_hidden+1)))

    A1, A2, H = forward_propagate(X, Theta1, Theta2)

    aux = np.argmax(H, axis=1)
    aux += 1

    return np.sum(aux == y) / np.shape(H)[0]


def main():
    datos = loadmat("ex4data1.mat")
    X = datos["X"]
    y = datos["y"]
    y = np.ravel(y)

    m = len(y)
    n_input = np.shape(X)[1]
    n_hidden = 25
    n_labels = 10

    aux_y = (y-1)
    y_onehot = np.zeros((m, n_labels))
    for i in range(m):
        y_onehot[i][aux_y[i]] = 1

    #pinta 100 ejemplos
    sample = np.random.choice(X.shape[0], 100)
    plt.figure()
    dD.displayData(X[sample])
    plt.savefig("fig1.png")

    weights = loadmat("ex4weights.mat")
    Theta1 = weights["Theta1"]
    Theta2 = weights["Theta2"]

    print("Coste sin regularizar: " + str(coste(X, y_onehot, Theta1,
Theta2))[:5])
    print("Coste regularizado con lambda=1: " + str(coste_reg(X,
y_onehot, Theta1, Theta2, 1))[:5])
    print()

    lamda = 1
    print("--Comprobación del gradiente--")
    diff = checkNNG.checkNNGradients(back_propagate, lamda)
    print("Menor diferencia: " + str(min(diff)))
    print("Mayor diferencia: " + str(max(diff)))
    print()
```

```
    #Aprendizaje de los parámetros
    Theta1 = pesosAleatorios(n_input, n_hidden)
    Theta2 = pesosAleatorios(n_hidden, n_labels)
    params_rn = np.concatenate((np.ravel(Theta1), np.ravel(Theta2)))

    #n_iters = 70
    resOpt = opt.minimize(fun=back_propagate, x0=params_rn,
args=(n_input, n_hidden, n_labels, X, y_onehot, lamda), method='TNC',
jac=True, options={'maxiter': 70})
    evaluacion = comprobar(resOpt, n_input, n_hidden, n_labels, X, y)
    print("Evaluación del entrenamiento de la red:
{}%".format(str(evaluacion*100)[:5]))


main()
```