Práctica 4 (Clasificador)

¿Cómo he planteado la práctica?

Primero necesitamos los datos, en este caso leemos el .MAT 'ex4data1', y separamos los datos en X e Y, como está en un diccionario, accedemos como un array en el que, en lugar de posiciones, tenemos el nombre de las columnas: data['X'] y data['Y'].

Mostramos 100 ejemplos como se pide en la práctica.

Además, debemos implementar One Hot Encoding para la Y, para así, al operar sobre todos los valores, que 'aporten' a su salida los implicados.

También he implementado la función sigmoide y las funciones de coste y gradiente regularizadas, que tienen como parámetros:

- Theta
- X
- Y
- lam

En el siguiente paso, como pide el enunciado, he calculado la función de backpropagation y creado funciones de apoyo (como 'recolocar')

Posteriormente, he comprobado la gradiente y creado una función que permite inicializar las θ con valores aleatorios.

Finalmente he predicho los resultados, multiplicando cada ejemplo X_i por cada etiqueta para θ (10 distintas), y he calculado su precisión con $\frac{aciertos}{total}$.

La clasificación ha arrojado unos resultados cercanos al 93%.

Código comentado y gráficas

1. Importar las librerías

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

from scipy.io import loadmat
from displayData import displayData, displayImage
from checkNNGradients import checkNNGradients
```

2. Obtener los datos del CSV:

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

Y = data['y'].ravel()
X = data['X']

w = loadmat('ex4weights.mat')
```

Es interesante destacar que en la Y necesitamos hacer ravel(), para que la dimensión sea de (5000,) y no de (5000, 1), ya que, si no lo hacemos, la función FMIN_TNC suele dar problemas.

```
sample = np.random.choice(X.shape[0], 100)
fig, ax = displayData(X[sample, :])
```

3. Ahora creamos la Y_onehot para que al operar entre vectores sea más cómodo.

```
m = len(Y)
input_size = X.shape[1]
num_labels = 10

Y = (Y - 1)
Y_onehot = np.zeros((m, num_labels))

for i in range(m):
    Y_onehot[i][Y[i]] = 1
```

4. Ahora creamos las funciones:
   - Sigmoide

```
def sigmoid(z):
    return 1 / (1 + np.exp(-z))
```

   - Forward Propagation: en la devuelvo a1, z2, a2, z3 y h (=a3)

```
def forward_prop(X, w):
    a1 = X
    a1 = np.hstack([np.ones([X.shape[0], 1]), a1])

    z2 = np.dot(w['Theta1'], a1.T)
    a2 = sigmoid(z2).T
    a2 = np.hstack([np.ones([a2.shape[0], 1]), a2])

    z3 = np.dot(w['Theta2'], a2.T)
    h = sigmoid(z3).T

    return a1, z2, a2, z3, h
```

- De coste: En la que solo me interesa quedarme con h_θ de la Forward Propagation. Como hay que ir fila por fila, lo pongo en un bucle (#7), hago las operaciones y aplico la regularización. Como el diccionario de θs tiene keys que no nos interesan ("__"), solo nos quedamos con las que nos interesan → $Theta1, Theta2$ (#18)
  Finalmente aplicamos la regularización a la J_θ

```
1  def coste(X, Y, w, K, lam):
2      m = X.shape[0]
3      J_theta = 0
4      aux = 0
5      _, _, _, _, h_theta = forward_prop(X, w)
6
7      for i in range(m):
8          aux +=np.sum(-Y[i] * np.log(h_theta[i])
9                       - (1 - Y[i])* np.log(1 - h_theta[i]))
10
11
12     J_theta = (1 / m) * aux
13
14     # Regularizacion
15     reg = 0
16
17     for theta in w.keys():
18         if "__" not in theta:
19             reg += np.sum(np.square(w[theta][:, 1:]))
20
21     reg *= (lam /(2 * m))
22
23     # Coste Regularizado
24     J_theta += reg
25
26     return J_theta
```

- De gradiente: Uso diccionarios para las Δ, ahora sí que nos interesa quedarnos con todos los términos que se devuelven en la Forward Propagation (#11).
  Se aplican los cálculos dentro del bucle, y se acumulan la suma en los Δ. Al terminar, se divide toda la acumulación y se aplica la regularización
  → Sin la primera columna θ (#28 y #29)
  → Sin contar j=0 (#30 y #31)

  Hay que destacar que es obligatorio el uso de np.newaxis (#21 y #22), que aumenta un eje, para así poder hacer las operaciones.

```
1  def gradiente(X, Y, w, lam):
2      """
3      Calcula el coste de la gradiente
4      """
5      m = X.shape[0]
6
7      d = dict()
8      d["delta1"] = np.zeros(w["Theta1"].shape)
9
10     d["delta2"] = np.zeros(w["Theta2"].shape)
11     a1, z2, a2, z3, h_theta = forward_prop(X, w)
12
13     for i in range(m):
14         # Calcular d2 y d3
15         d3 = h_theta[i] - Y[i]                  # (10, )
16         g_z2 = a2[i] * (1 - a2[i])              # (26, )
17         d2 = np.dot(d3, w["Theta2"]) * g_z2     # (26, )
18         #d2 = d2[1:]                             # (25, )
19
20         # Actualizar deltas
21         d["delta1"] += np.dot(d2[1:, np.newaxis], a1[i][np.newaxis, :])
22         d["delta2"] += np.dot(d3[:, np.newaxis], a2[i][np.newaxis, :])
23
24     d["delta1"] /= m
25     d["delta2"] /= m
26
27     #Regularizar deltas
28     reg1 = ((lam / m) * w["Theta1"][:, 1:]) # No theta primera columna
29     reg2 = ((lam / m) * w["Theta2"][:, 1:]) # No theta primera columna
30     d["delta1"][:, 1:] += reg1              # j = 0 no tiene regularización
31     d["delta2"][:, 1:] += reg2              # j = 0 no tiene regularización
32
33
34     return np.concatenate(
35         (np.ravel(d["delta1"]),
36          np.ravel(d["delta2"]))
37     )
38
```

5. Tras ello implementamos la función de Back Propagation, como se pide en el enunciado, en el que devuelve la tupla del coste y la gradiente, ambas regularizadas.

```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, Y, reg)
    """
    Esta función devuelve una tupla (coste, gradiente) con el coste y
    el gradiente de una red neuronal de tres capas, con num_entradas,
    num_ocultas nodos en la capa oculta y num_etiquetas nodos en la
    capa de salida. Si m es el número de ejemplos de entrenamiento,
    la dimensiónde de 'X' es (m, num_entradas) y la de 'y'
    es (m, num_etiquetas)
    """
    w = dict()
    w["Theta1"], w["Theta2"] = recolocar(params_rn, num_entradas, num_oculta

    return coste(X, Y, w, Y.shape[0], reg), gradiente(X, Y, w, reg)
```

Además, usamos una función de apoyo ('recolocar') que nos permitirá partir un vector con los tamaños deseados.

```
def recolocar(params_rn, num_entradas, num_ocultas, num_etiquetas):
    """
    A partir de un único vector y el número de entradas, de nodos ocultos
    y etiquetas se devuelven los vectores Theta1 y Theta2, con los tamaños
    establecidos
    """
    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)))
    return Theta1, Theta2
```

Para poder usar la función 'backprop', necesitamos unir todas las θs en un vector, para ello concatenamos y usamos la función 'ravel'

```
params_rn = np.concatenate(
    (
        np.ravel(w["Theta1"]),
        np.ravel(w["Theta2"])
    )
)
```

6. Ahora comprobamos la gradiente con la función checkGradients:

```
print(checkNNGradients(backprop, 0))
print(checkNNGradients(backprop, 1))
```

```
grad shape:  (38,)
num grad shape:  (38,)
[ 5.27761168e-11 -2.55029895e-12  5.67280077e-12  9.17629861e-12
 -6.52669724e-11  2.08457210e-12 -1.07556533e-11 -4.38069164e-11
 -9.29989974e-11  7.04843822e-12 -4.64730199e-11 -1.24605812e-10
 -1.95650579e-11  5.45034504e-13 -8.24779828e-12 -2.49761462e-11
  2.15736456e-11 -4.96176017e-13  7.55695853e-12  2.51674931e-11
  6.03760375e-11  1.32927280e-11  9.03210839e-12  7.48807960e-12
  1.67883762e-11  2.10645668e-11  7.15513759e-11  1.56080704e-11
  7.11190828e-12  1.37491407e-11  1.70987668e-11  1.79336823e-11
  7.55120411e-11  1.60134683e-11  1.08387743e-11  1.78091986e-11
  1.43913215e-11  2.04546380e-11]
grad shape:  (38,)
num grad shape:  (38,)
[ 5.27761168e-11 -3.70814490e-12  6.60943522e-12  9.75092229e-12
 -6.52669724e-11  2.10972906e-12 -1.16537890e-11 -4.48128618e-11
 -9.29989974e-11  5.59485791e-12 -4.79407070e-11 -1.24423472e-10
 -1.95650579e-11 -8.44324610e-14 -9.22964483e-12 -2.43030734e-11
  2.15736456e-11  2.27595720e-13  7.55695853e-12  2.62300737e-11
  6.03760375e-11  1.38673517e-11  8.50597370e-12  7.51322615e-12
  1.81106935e-11  2.00586214e-11  7.15513759e-11  1.63749569e-11
  7.86468113e-12  1.39314948e-11  1.64833147e-11  1.73042136e-11
  7.55120411e-11  1.66865410e-11  1.07713560e-11  1.63125347e-11
  1.34624811e-11  2.22044327e-11]
```

7. Para poder aplicar correctamente una red neuronal, necesitamos inicializar los valores de θ aleatoriamente, tanto en positivo como en negativo.

```
1  def random_thetas(shape, E):
2      """
3      Recibe como parámetros las dimensiones y el Epsilon (rango -> [-E, E])
4
5      Primero creamos una matriz de positivos y negativos
6      Posteriormente creamos una matriz con números aleatorios positivos < Epsilon
7      Multiplicamos las dos matrices, tenemos aleatorios positivos y negativos.
8
9      Devuelve las dimensiones con valores aleatorios
10
11     """
12     posNeg = np.random.random((shape))
13     pos = np.where(posNeg < .5)
14     neg = np.where(posNeg >= .5)
15     posNeg[pos] = 1
16     posNeg[neg] = -1
17
18     return (np.random.random((shape)) %E ) * posNeg
```

8. Inicializamos los valores.
   En las líneas #6 y #7 creamos un par con las dimensiones que deberían tener las θs, y posteriormente las inicializamos aleatoriamente (#12 y #13)

```
1  eIni = 0.12
2  it = 70
3  lambd = 0
4  hidden_layer = 25
5
6  Theta1_sh = (hidden_layer, X.shape[1] + 1)
7  Theta2_sh = (Y_onehot.shape[1], hidden_layer + 1)
8
9
10 thetas_random = dict()
11
12 thetas_random["Theta1"] = random_thetas(Theta1_sh, eIni)
13 thetas_random["Theta2"] = random_thetas(Theta2_sh, eIni)
14
15 th_random = np.concatenate(
16     (
17         np.ravel(thetas_random["Theta1"]),
18         np.ravel(thetas_random["Theta2"])
19     )
20 )
```

9. Obtenemos las $\theta_{OPTIMAS}$ con MINIMIZE y recolocamos el vector que nos devuelve ($\theta_{OPTIMAS}$)

```python
def entrenar(backprop, thetas, X, Y, reg = 1, iterations = 70):

    # Thetas optimas
    res = opt.minimize(
        fun = backprop,
        x0 = thetas,
        args=(X.shape[1], 25, Y.shape[1], X, Y, reg),
        options={'maxiter': iterations},
        method='TNC',
        jac=True
    )

    # Recolocamos
    w = dict()
    w["Theta1"], w["Theta2"] = recolocar(res.x,
                                         X.shape[1],
                                         hidden_layer,
                                         Y_onehot.shape[1])
    return w

w_opt = entrenar(backprop, th_random, X, Y_onehot, 1, 70)
```

10. Ahora predecimos el resultado

```python
def predecir_nn(X, Y, w):

    Y_hat = []
    _, _, _, _, pred = forward_prop(X, w)

    for i in range(pred.shape[0]):
        ejemplo = pred[i]
        num = np.argmax(ejemplo) + 1    # Va de 0 a 9, no de 1 a 10 -> +1
        Y_hat.append(num)

    Y_hat = np.array(Y_hat)

    return Y_hat
```

11. Y comprobamos la precisión que tiene nuestra Red Neuronal

```
def precision_nn(X, Y, w):
    Y_hat = predecir_nn(X, Y, w)

    return np.round(
        np.sum(Y_hat == Y) / m * 100,
        decimals = 2
    )
```

```
print("La precisión de la Red Neuronal es de aproximadamente un: {}%"
      .format(precision_nn(X, Y, w_opt)))
```

```
La precisión de la Red Neuronal es de aproximadamente un: 93.32%
```

12. Ahora comparamos las iteraciones y las λ

```
iterations = [ 10, 50, 100, 200, 300 ]
lambdas = [ 0.01, 0.03, 0.1, 0.3, 1, 3, 10 ]

for lambd in lambdas:
    for i in iterations:
        thetas = entrenar(backprop, th_random, X, Y_onehot, lambd, i)
        p = precision_nn(X, Y, thetas)
        print("Lambda: {}".format(lambd),
              "\tIteraciones: {}".format(i),
              "\tPrecisión:{}%\n".format(p)
             )
```

```
Lambda: 0.01    Iteraciones: 10         Precisión:60.32%
Lambda: 0.01    Iteraciones: 50         Precisión:89.94%
Lambda: 0.01    Iteraciones: 100        Precisión:95.34%
Lambda: 0.01    Iteraciones: 200        Precisión:99.58%
Lambda: 0.01    Iteraciones: 300        Precisión:99.98%
Lambda: 0.03    Iteraciones: 10         Precisión:60.32%
Lambda: 0.03    Iteraciones: 50         Precisión:89.94%
Lambda: 0.03    Iteraciones: 100        Precisión:95.44%
Lambda: 0.03    Iteraciones: 200        Precisión:99.94%
Lambda: 0.03    Iteraciones: 300        Precisión:100.0%
Lambda: 0.1     Iteraciones: 10         Precisión:60.32%
Lambda: 0.1     Iteraciones: 50         Precisión:89.76%
Lambda: 0.1     Iteraciones: 100        Precisión:95.3%
Lambda: 0.1     Iteraciones: 200        Precisión:99.72%
Lambda: 0.1     Iteraciones: 300        Precisión:99.98%
Lambda: 0.3     Iteraciones: 10         Precisión:60.4%
Lambda: 0.3     Iteraciones: 50         Precisión:89.54%
Lambda: 0.3     Iteraciones: 100        Precisión:96.66%
Lambda: 0.3     Iteraciones: 200        Precisión:99.84%
Lambda: 0.3     Iteraciones: 300        Precisión:99.9%
Lambda: 1       Iteraciones: 10         Precisión:60.34%
Lambda: 1       Iteraciones: 50         Precisión:88.12%
Lambda: 1       Iteraciones: 100        Precisión:94.78%
Lambda: 1       Iteraciones: 200        Precisión:98.78%
Lambda: 1       Iteraciones: 300        Precisión:99.38%
Lambda: 3       Iteraciones: 10         Precisión:60.3%
Lambda: 3       Iteraciones: 50         Precisión:91.6%
Lambda: 3       Iteraciones: 100        Precisión:96.06%
Lambda: 3       Iteraciones: 200        Precisión:97.44%
Lambda: 3       Iteraciones: 300        Precisión:97.5%
Lambda: 10      Iteraciones: 10         Precisión:60.18%
Lambda: 10      Iteraciones: 50         Precisión:86.28%
Lambda: 10      Iteraciones: 100        Precisión:93.9%
Lambda: 10      Iteraciones: 200        Precisión:94.1%
Lambda: 10      Iteraciones: 300        Precisión:94.34%
```

Como conclusiones sacamos:

- Lo que más aporta son las iteraciones, ya que las distintas λ tienen pocas diferencias para las mismas iteraciones, por ejemplo, para 100 iteraciones $\lambda = 0.01$, la precisión es de 95'34%, mientras que para $\lambda = 0.3$ la precisión es de 96'66%.

- Por otro lado, para la misma λ, en este caso, $\lambda = 0.3$, la diferencia entre 50 y 100 iteraciones es de aproximadamente un 7%.

- Otro dato a destacar es que, por norma general, **a más iteraciones mejor precisión**, en cambio, y como es lógico, si **la regularización es muy alta, tiende a mostrar resultados peores**, por ejemplo, para $\lambda = 0.01$ y 300 iteraciones, la precisión es del 99'98%, mientras que para $\lambda = 10$ y 300 iteraciones la precisión es del 94'34%

- El mejor resultado lo ofrece $\lambda = 0.03$ y 300 iteraciones, con un 100% de precisión.