Práctica 4: Entrenamiento de redes neuronales

Álvar Domingo Fernández y Pablo Jurado López

Preparación inicial

A continuación se importan todas las librerías externas que serán utilizadas en esta práctica.

[ ]:
```
import numpy as np
from numpy.lib.function_base import gradient
import checkNNGradients as cnn
from displayData import displayData
import operator
import matplotlib.pyplot as plt
from numpy.lib.npyio import load
import scipy.optimize as opt
from scipy.io import loadmat
```

Muestra de los datos

A partir de las matrices de datos proporcionadas, se han elegido 100 elementos distintos y se han guardado en una imagen haciando uso de la función displayData.

[ ]:
```
weights = loadmat('ex4weights.mat')
data = loadmat('ex4data1.mat')
theta1, theta2 = weights['Theta1'], weights['Theta2']
X = data['X']
y = data['y'].ravel()

sample = np.random.choice(X.shape[0], 100)
imgs = displayData(X[sample, :])
plt.savefig('numbers')
```

1 - Función de coste

Se ha implementado una función de coste que funciona con un término de regularización, de acuerdo con la siguiente fórmula:

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[-y_k^{(i)}log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})log(1-(h_\theta(x^{(i)}))_k)] + \frac{\lambda}{2m}[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2]$$

[ ]:
```
def coste_neuronal(X, y, theta1, theta2, reg):
    a1, a2, h = propagar(X, theta1, theta2)
    m = X.shape[0]

    J = 0
    for i in range(m):
        J += np.sum(-y[i]*np.log(h[i]) - (1 - y[i])*np.log(1-h[i]))
    J = J/m

    sum_theta1 = np.sum(np.square(theta1[:, 1:]))
    sum_theta2 = np.sum(np.square(theta2[:, 1:]))

    reg_term = (sum_theta1 + sum_theta2) * reg / (2*m)

    return J + reg_term
```

La función es similar a la que teníamos antes, solo que ahora se le ha sumado el término de regularización (los dos sumatorios + lambda partido de 2m)
Con lambda = 1, el coste de como resultado aproximadamente 0.38

2 - Función del gradiente

Se ha implementado una función que calcula el gradiente de una red neuronal de tres capas, añadiendo el término de regularización tal y como se hizo con el coste (excepto a la primera columna de $\theta^{(l)}$), de acuerdo a la siguiente fórmula:
$\frac{\delta}{\delta\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)}$ para j = 0
$\frac{\delta}{\delta\Theta_{ij}^{(l)}}J(\Theta) = D_{ij}^{(l)} = \frac{1}{m}\Delta_{ij}^{(l)} + \frac{\lambda}{m}\Theta_{ij}^{(l)}$ para j > 0

[ ]:
```
def propagar(X, theta1, theta2):
    m = np.shape(X)[0]

    a1 = np.hstack([np.ones([m,1]), X])
    z2 = np.dot(a1, theta1.T)
    a2 = np.hstack([np.ones([m, 1]), sigmoide(z2)])
    z3 = np.dot(a2, theta2.T)
    a3 = sigmoide(z3)
    return a1, a2, a3

def gradiente(X, y, Theta1, Theta2, reg):
    m = X.shape[0]

    delta1 = np.zeros(Theta1.shape)
    delta2 = np.zeros(Theta2.shape)

    a1, a2, h = propagar(X, Theta1, 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, :])
        delta1[:, 1:] += Theta1[:, 1:] * reg/m
        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])
        delta2[:, 1:] += Theta2[:, 1:] * reg/m

    return np.concatenate((np.ravel(delta1/m), np.ravel(delta2/m)))
```

Con las funciones de coste y gradiente, ya podemos implementar la función de retropropagación, que viene dada por el siguiente código:

[ ]:
```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    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 coste_neuronal(X, y, Theta1, Theta2, reg), gradiente(X, y, Theta1,␣
↪Theta2, reg)
```

2.1 - Comprobación del gradiente

Con las funciones que se han descrito antes, además de con otra función auxiliar para sacar matrices de pesos aleatorios, ya podemos prepararlo todo para utilizar la función checkNNGradients, que comprobará si las funciones proporcionadas funcionan correctamente:

[ ]:
```
def random_weights(L_in, L_out):
    epsilon = np.sqrt(6)/np.sqrt(L_in + L_out)
    return np.random.random((L_in, L_out)) * epsilon - epsilon/2

weights = loadmat('ex4weights.mat')
data = loadmat('ex4data1.mat')
theta1, theta2 = weights['Theta1'], weights['Theta2']
X = data['X']
y = data['y'].ravel()

sample = np.random.choice(X.shape[0], 100)
imgs = displayData(X[sample, :])
plt.savefig('numbers')

m = len(y)
input_size = X.shape[1]
num_labels = 10
num_ocultas = 25

y = y -1
y_onehot = np.zeros((m, num_labels))

for i in range(m):
    y_onehot[i][y[i]] = 1
a1, a2, h = propagar(X, theta1, theta2)

#params_rn = np.concatenate((np.ravel(theta1), np.ravel(theta2)))
theta1 = random_weights(theta1.shape[0], theta1.shape[1])
theta2 = random_weights(theta2.shape[0], theta2.shape[1])
params_rn = np.concatenate((np.ravel(theta1), np.ravel(theta2)))
reg_param = 1
```

```
cost, grad = backprop(params_rn, input_size, num_ocultas, num_labels, X,␣
 ↪y_onehot, reg_param)
a = cnn.checkNNGradients(backprop, 1)
```

Al ejecutar checkNNGradients, el resultado devuelto está conformado por números muy pequeños (menores que -10^9), lo cual nos permite concluir que nuestras funciones son correctas al ser casi despreciable la diferencia entre nuestro resultado y la aproximación que establece la función de comprobación.

3 - Aprendizaje de los parámetros

Una vez realizadas todas las comprobaciones, ya podemos comenzar a entrenar la red neuronal con la función scipy.optimize.minimize y nuestra función que obtiene pesos aleatorios (con un valor de ϵ calculado por nosotros mismos en *random_weights*).

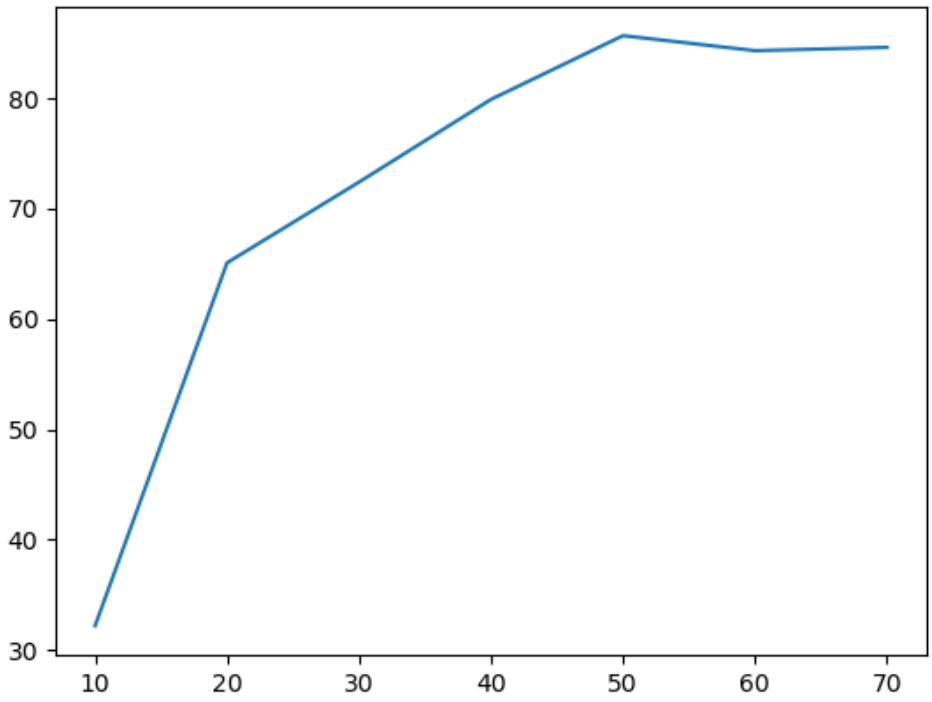
[ ]:
```
fmin = opt.minimize(fun=backprop, x0=params_rn, args=(input_size, num_ocultas,␣
 ↪num_labels, X, y_onehot, reg_param), method='TNC', jac=True,␣
 ↪options={'maxiter': 70})

theta1_opt = np.reshape(fmin.x[:num_ocultas * (input_size + 1)], (num_ocultas,␣
 ↪(input_size + 1)))
theta2_opt = np.reshape(fmin.x[num_ocultas * (input_size + 1):], (num_labels,␣
 ↪(num_ocultas + 1)))

a1, a2, h = propagar(X, theta1_opt, theta2_opt)

print("El porcentaje de acierto del modelo es: {}%".format(np.sum((y ==␣
 ↪predict_nn(X, h)))/X.shape[0] * 100))
```

Con unas 70 iteraciones, el porcentaje de acierto del modelo sale en torno al 93,74%.

A continuación se han calculado dos gráficas distintas. La primera plasma el porcentaje de acierto del modelo con distintos números de iteraciones. Se puede observar que, a mayor número de iteraciones, más eficaz de vuelve el modelo:

[ ]:
```
lambdas = np.linspace(10, 70, 7)

accuracy = []

for lamb in lambdas:
    reg_param = lamb
    fmin = opt.minimize(fun=backprop, x0=params_rn, args=(input_size,␣
 ↪num_ocultas, num_labels, X, y_onehot, reg_param), method='TNC', jac=True,␣
 ↪options={'maxiter': int(lamb)})

    theta1_opt = np.reshape(fmin.x[:num_ocultas * (input_size + 1)],␣
 ↪(num_ocultas, (input_size + 1)))
```

```
    theta2_opt = np.reshape(fmin.x[num_ocultas * (input_size + 1):],␣
 ↪(num_labels, (num_ocultas + 1)))

    a1, a2, h = propagar(X, theta1_opt, theta2_opt)
    accuracy.append((np.sum((y == predict_nn(X, h)))/X.shape[0] * 100))

plt.plot(lambdas, accuracy)
```



La segunda evalúa el porcentaje de acierto del modelo según el valor de lambda (que va desde 0 hasta 1), siempre con el mismo número de iteraciones (70)

[ ]:
```
lambdas = np.linspace(0, 1, 10)

accuracy = []

for lamb in lambdas:
    reg_param = lamb
    fmin = opt.minimize(fun=backprop, x0=params_rn, args=(input_size,␣
 ↪num_ocultas, num_labels, X, y_onehot, reg_param), method='TNC', jac=True,␣
 ↪options={'maxiter': 70})
```

```
    theta1_opt = np.reshape(fmin.x[:num_ocultas * (input_size + 1)],␣
↪(num_ocultas, (input_size + 1)))
    theta2_opt = np.reshape(fmin.x[num_ocultas * (input_size + 1):],␣
↪(num_labels, (num_ocultas + 1)))

    a1, a2, h = propagar(X, theta1_opt, theta2_opt)
    accuracy.append((np.sum((y == predict_nn(X, h)))/X.shape[0] * 100))

plt.plot(lambdas, accuracy)
plt.savefig("accuracychart")
```