Practica 4 (Entrenamiento de redes neuronales)

Aprendizaje automático y big data

Alberto García Doménech - Pablo Daurell Marina

In [1]:
```
import numpy as np
from scipy.io import loadmat
```

El objetivo de esta práctica será implementar los calculos y algoritmos necesarios para entrenar a una red neuronal. Usaremos una red neuronal para intentar clasificar 5000 imagenes de números del 0 al 9 escritos a mano. Cada imagen (de 20x20 pixeles) vendrá representada por un array de 400 componentes.

Para ello haremos uso de una red neuronal de 3 capas, con una capa de entrada de 5000 neuronas, una capa oculta de 25 neuronas y una capa de salida de 10 neuronas (una por cada digito).

Como se trata de un problema de multiclasificación codificaremos el array *y* en codificación one-hot (array de 9 elementos con un 1 en la clase correspondiente y 0s en el resto).

In [2]:
```
# Cargamos los datos: 5000 ejemplos de entrenamiento (imagenes de 20x20 pixeles)
data = loadmat('ex4data1.mat')

X = data['X'] # Shape: (5000, 400)
y = data['y'].ravel() # Shape: (5000,)

# Establecemos la estructura de la red neuronal
num_entradas = np.shape(X)[1]
num_etiquetas = 10 # Del 0 al 9
num_ocultas = 25

# Adaptamos 'y' para usarlo en la red neuronal
y = (y - 1)
y_onehot = np.zeros((len(y), num_etiquetas)) # Shape: (5000, 10)

for i in range(len(y)):
    y_onehot[i][y[i]] = 1
```

Vemos una pequeña muestra de las imagenes del dataset:

In [3]:
```
# seleccionamos 10 numeros al azar y los mostramos
from displayData import displayData
sample = np.random.choice(X.shape[0], 100)
img = displayData(X[sample,:])
```

Como función de activación para las neuronas usaremos la funcion sigmoide:

In [4]:
```
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

Definimos el algoritmo de propagación hacia delante que aplica los pesos correspondientes a cada capa de la red y nos permite clasificar los ejemplos.

In [5]:
```
def forward_propagation(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
```

Algoritmo de retropropagación (Back-propagation)

Coste regularizado:

Vamos a comenzar a implementar el algoritmo de retropropagación.

- Primero establecemos como calcular el error (coste) de la red una vez aplicado el algoritmo de propagación hacia delante.
- Ademas le añadimos el coeficiente de regularización para evitar el sobreaprendizaje.

In [6]:
```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    '''Funcion de back-propagation para red neuronal de 3 capas'''

    m = X.shape[0]

    # Desplegamos los paramas_rn en la matrices Theta
    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)))

    # Aplicamos forward-propagation para calcular la salidas de cada capa
    a1, z2, a2, z3, h = forward_propagation(X, theta1, theta2)

    # Calculo del coste
    cost = 0
    for i in range(m):
        a = np.dot(-y[i,:], np.log(h[i,:]))
        b = np.dot(1 - y[i,:], np.log(1 - h[i,:]))
        cost += np.sum(a - b)

    cost = cost/m
    cost += reg/(2*m) * (np.sum(theta1[:, 1:]**2) + np.sum(theta2[:, 1:]**2))

    return cost
```

- Cargamos los parametros de la red neuronal ya entrenada para comprobar si hemos implementado bien el cálculo del coste:

In [7]:
```
weights = loadmat('ex4weights.mat')

theta1 = weights['Theta1'] # Shape: (25, 401)
theta2 =weights['Theta2'] # Shape: (10, 26)
```

In [8]:
```
thetaVec = np.concatenate((np.ravel(theta1), np.ravel(theta2)))
print('Coste sin regularización: ', backprop(thetaVec, num_entradas, num_ocultas, num_etiquetas, X, y_onehot, 0))
print('Coste con regularización: ',backprop(thetaVec, num_entradas, num_ocultas, num_etiquetas, X, y_onehot, 1))
```

```
Coste sin regularización:  0.2876291651613187
Coste con regularización:  0.3837698590909234
```

Gradiente:

Seguimos con la implementación de la retropropagación.

- Ahora vamos a añadir el cálculo de los vectores de gradientes, para ello implementaremos el algoritmo de retropropagación propiamente dicho para calcular las derivadas parciales del error en cada capa y poder computar el gradiente adecuado.

> Antes de nada definimos la derivada de la función sigmoide (usada en el calculo del gradiente) y un método para inicializar las matrices de pesos con valores aleatorios, distintos pero cercanos a 0.

In [9]:
```
def sigmoid_derivative(x):
    return sigmoid(x)*(1 - sigmoid(x))
```

In [10]:
```
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
```

- Implementación del gradiente y la retropropagación:

In [11]:
```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg):
    '''Funcion de back-propagation para red neuronal de 3 capas'''

    m = X.shape[0]

    # Desplegamos los paramas_rn en la matrices Theta
    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)))

    # Aplicamos forward-propagation para calcular la salidas de cada capa
    a1, z2, a2, z3, h = forward_propagation(X, theta1, theta2)

    # Calculo del coste
    cost = 0
    for i in range(m):
        a = np.dot(-y[i,:], np.log(h[i,:]))
        b = np.dot(1 - y[i,:], np.log(1-h[i,:]))
        cost += np.sum(a - b)

    cost = cost/m
    cost += reg/(2*m) * (np.sum(theta1[:, 1:]**2) + np.sum(theta2[:, 1:]**2))

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

    for t in range(m):
        a1t = a1[t,:]
        a2t = a2[t,:]
        ht = h[t,:]
        yt = y[t]

        d3 = ht - yt
        d2 = np.dot(theta2.T, d3) * (a2t * (1 - a2t))

        delta1 += np.dot(d2[1:, np.newaxis], a1t[np.newaxis, :])
        delta2 += np.dot(d3[:, np.newaxis], a2t[np.newaxis, :])

    # Calculo del gradiente
    D1 = delta1 / m
    D2 = delta2 / m

    gradient = np.concatenate((np.ravel(D1), np.ravel(D2)))

    return cost, gradient
```

Haciendo uso de la función `checkNNGradients` comprobamos si hemos implementado bien el cálculo del gradiente.

In [12]:
```
from checkNNGradients import checkNNGradients
checkNNGradients(backprop, 0)
```

Out[12]:
```
array([ 5.27761168e-11,  1.89059467e-12,  7.89324509e-12,  6.95584909e-12,
       -6.30465125e-11,  2.08456863e-12, -1.07556394e-11, -5.04682407e-11,
       -9.07785513e-11,  7.04843475e-12, -3.98116679e-11, -1.22385352e-10,
       -2.17855040e-11,  2.76547969e-12, -6.02735570e-12, -2.49761462e-11,
        2.15736526e-11, -4.96176017e-13,  1.19978506e-11,  2.73879391e-11,
        6.25964836e-11,  1.55131741e-11,  9.03210839e-12,  5.26763355e-12,
        1.90088223e-11,  1.88441207e-11,  7.15513759e-11,  1.56080426e-11,
        7.11190828e-12,  1.37491546e-11,  1.70987668e-11,  1.79336823e-11,
        7.32915950e-11,  1.60134683e-11,  8.61832827e-12,  1.78091986e-11,
        1.43913215e-11,  2.26750840e-11])
```

Gradiente regularizado:

Finalmente, para terminar la implementación del algoritmo de retropropagación añadimos el coeficiente de regularización al calculo del gradiente:

In [13]:
```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg=0):
    '''Funcion de back-propagation para red neuronal de 3 capas'''

    m = X.shape[0]

    # Desplegamos los paramas_rn en la matrices Theta
    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)))

    # Aplicamos forward-propagation para calcular la salidas de cada capa
    a1, z2, a2, z3, h = forward_propagation(X, theta1, theta2)

    # Calculo del coste
    cost = 0
    for i in range(m):
        a = np.dot(-y[i,:], np.log(h[i,:]))
        b = np.dot((1-y[i,:]), np.log(1-h[i,:]))
        cost += np.sum(a - b)

    cost = cost/m

    # Regularizacion del coste
    cost += reg/(2*m) * (np.sum(theta1[:, 1:]**2) + np.sum(theta2[:, 1:]**2))

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

    for t in range(m):
        a1t = a1[t,:]
        a2t = a2[t,:]
        ht = h[t,:]
        yt = y[t]

        d3 = ht - yt
        d2 = np.dot(theta2.T, d3) * (a2t * (1 - a2t))

        delta1 += np.dot(d2[1:, np.newaxis], a1t[np.newaxis, :])
        delta2 += np.dot(d3[:, np.newaxis], a2t[np.newaxis, :])

    # Calculo del gradiente
    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

    gradient = np.concatenate((np.ravel(D1), np.ravel(D2)))

    return cost, gradient
```

Volvemos a comprobar si la implementación es correcta:

In [14]:
```
from checkNNGradients import checkNNGradients
checkNNGradients(backprop, 1)
```

Out[14]:
```
array([ 5.27761168e-11,  7.32719441e-13,  8.82988127e-12,  7.53047624e-12,
       -6.30465125e-11,  2.10970130e-12, -1.16537613e-11, -4.92537400e-11,
       -9.07785513e-11,  5.59484403e-12, -3.90588950e-11, -1.22203025e-10,
       -2.17855040e-11,  4.35645964e-12, -7.00919878e-12, -2.43030734e-11,
        2.15736526e-11,  2.27623476e-13,  1.19978506e-11,  2.84505197e-11,
        6.25964836e-11,  1.38673517e-11,  8.50600146e-12,  5.29278010e-12,
        2.03311395e-11,  1.78381754e-11,  7.15513759e-11,  1.63749014e-11,
        7.86468113e-12,  1.39315087e-11,  1.64833286e-11,  1.95246597e-11,
        7.32915950e-11,  1.66865410e-11,  8.55090998e-12,  1.63125624e-11,
        1.34624811e-11,  2.22044327e-11])
```

Entrenamiento

Una vez implementado el algoritmo de retropropagación al completo, entrenamos a la red neuronal usando la función `minimize` de `scipy.optimize` y estudiamos los resultados.

In [15]:
```
import scipy.optimize as opt
```

In [16]:
```
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_propagation(X, theta1, theta2)

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

In [20]:
```
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: 307
Numero de aciertos: 4693

Porcentaje de aciertos:  93.86
```

> Con un parametro de regularización a 1 y un máximo de 70 iteraciones obtenemos una pracisión bastante buena

Probamos a variar la regularizción y/o el número de iteraciones:

In [18]:
```
predictions = train(X, y_onehot, reg=1, iters=10)

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: 4500
Numero de aciertos: 500

Porcentaje de aciertos:  10.0
```

In [21]:
```
predictions = train(X, y_onehot, reg=0, iters=10)

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: 1808
Numero de aciertos: 3192

Porcentaje de aciertos:  63.839999999999996
```

> Vemos que con pocas iteraciones no conseguimos una buena precisión. Esto se debe a que al aplicar pocas veces la retropropagación minimizamos muy poco el error.

In [23]:
```
predictions = train(X, y_onehot, reg=20, 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: 532
Numero de aciertos: 4468

Porcentaje de aciertos:  89.36
```

> Si aumentamos la regularización obtenemos peor precisión, probablemente porque la red esté sobreaprendiendo