Practica 4 Aprendizaje Automático y Minería de Datos

Mario Jimenez y Manuel Hernández

Esta práctica consiste en reconocer digitos manuscritos mediante una red neuronal.

Inclusión de librerías

In [35]:
```
import displayData
import numpy as np
import displayData as dp
from scipy.io import loadmat
import matplotlib.pyplot as plt
import scipy.optimize as opt
import checkNNGradients as check
```

Lectura de datos

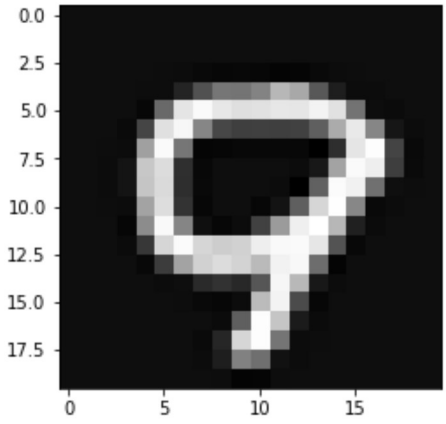
In [36]:
```
data = loadmat('ex4data1.mat')
Y = data['y']  # Representa el valor real de cada ejemplo de entrenamiento de X (y para cada X)
X = data['X']  # Cada fila de X representa una escala de grises de 20x20 desplegada linearmente (400 pixeles)
nMuestras = len(X)
Y = np.ravel(Y)
```

Las entradas son mapas de bits de 20x20, que se desdoblan en columnas de 400 elementos a los que se les asocia un valor.

In [37]:
```
print(Y[4999])
plt.figure()
dp.displayImage(X[4999])
plt.savefig("Input_sample")
plt.show()
```

```
9
```

```
<Figure size 432x288 with 0 Axes>
```



Carga de matrices de pesos preentrenadas

In [38]:
```
weights = loadmat('ex4weights.mat')
theta1, theta2 = weights['Theta1'], weights ['Theta2']
```

Función de activación y derivada

En esta red neuronal utilizaremos la función sigmoide como función de activación para las neuronas.

In [39]:
```
def sigmoid(z):
    return 1/(1 + np.exp(-z))
def sigmoidDerivative(z):
    z = sigmoid(z)
    return z*(1-z)

print("Sigmoid (0.25) = ",sigmoid(0.25))
```

```
Sigmoid (0.25) =  0.5621765008857981
```

Forward propagation y función de coste

La función de hipótesis o de forward propagation utiliza un valor de entrada(401 entradas, con termino de sesgo ya añadido) para predecir una salida mediante una matriz de pesos y una función de activación. Además, durante el proceso, añadirá el termino de sesgo o *bias* para el computo final. Devolvemos todas las matrices intermedias, ya que nos podrán ser de utilidad.

In [40]:
```
def forwardProp(thetas1, thetas2, X):
    z2 = thetas1.dot(X.T)
    a2 = sigmoid(z2)
    tuple = (np.ones(len(a2[0])), a2)
    a2 = np.vstack(tuple)
    z3 = thetas2.dot(a2)
    a3 = sigmoid(z3)
    return z2, a2, z3, a3

X_aux = np.hstack([np.ones((len(X), 1), dtype = np.float), X])
print("Valor predicho para el elemento 0 de X según la hipótesis: ",(forwardProp(th
eta1, theta2, X_aux)[3]).T[0].argmax())
```

```
Valor predicho para el elemento 0 de X según la hipótesis:  9
```

En cuanto a la función de coste, implementaremos la función de coste con regularización. Como entrada a dicha función, hemos de preparar un vector de Y distinto al recibido. Será una matriz de *(numElementos, numEtiquetas)* donde cada fila corresponde a un caso. Cada fila tendrá todos los valores a cero menos el valor real que representa ese caso, que estará a 1.

In [41]:
```
def costFun(X, y, theta1, theta2,  reg):
    #Here we assert that we can operate with the parameters
    X = np.array(X)
    y = np.array(y)
    muestras = len(y)

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

    hipo  = forwardProp(theta1, theta2, X)[3]
    cost = np.sum((-y.T)*(np.log(hipo)) - (1-y.T)*(np.log(1- hipo)))/muestras

    regcost = np.sum(np.power(theta1[:, 1:], 2)) + np.sum(np.power(theta2[:,1:], 2)
)
    regcost = regcost * (reg/(2*muestras))

    return cost + regcost

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(z == 10): z = 0
        nY[i][z] = 1
    return nY
```

In [42]:
```
Y_aux = getYMatrix(Y,10)
```

In [43]:
```
print("El coste con thetas entrenados es: ", costFun(X_aux, Y_aux, theta1, theta2,1
))
```

```
El coste con thetas entrenados es:  0.3837698590909236
```

Backpropagation

Función de backpropagation para repartir el error entre las neuronas de la red neuronal. Comienza desde la ultima capa y desde esa desciende hasta la penúltima, ya que no se puede repartir error para la capa de entrada.

In [44]:
```
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,(n
um_ocultas+1)))

    X_unos = np.hstack([np.ones((len(X), 1), dtype = np.float), X])
    nMuestras = len(X)

    y = np.zeros((nMuestras, num_etiquetas))
    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 = forwardProp(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)*sigmoidDerivative(z2)


    # Acumulación 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
```

In [45]:
```
params =  np.concatenate((theta1, theta2), axis = None)
print("Diferencias al comprobar gradientes:\n", check.checkNNGradients(backprop, 1)
)
```

```
Diferencias al comprobar gradientes:
 [ 4.33315606e-11 -5.85087534e-13  5.24080779e-13  6.94293928e-12
-3.86019966e-11  8.55844562e-12 -7.97453770e-12 -3.25843796e-11
-5.90238414e-11  3.02491365e-11 -2.21222485e-11 -9.52720680e-11
-4.15551621e-11  9.12638021e-13 -2.03395634e-12 -1.80884266e-11
 1.25427238e-11 -4.09060286e-12  6.03773281e-12  2.41384690e-11
 5.28279642e-11  1.03140274e-11  6.16659501e-12  8.66137717e-12
 9.34252675e-12  1.71084258e-11  6.32419117e-11  1.16325005e-11
 1.02217609e-11  1.49442403e-11  1.12843068e-11  1.66834324e-11
 7.14504289e-11  1.10622622e-11  8.72196759e-12  2.17894591e-11
 8.18731194e-12  1.53632801e-11]
```

Inicialización aleatoria de thetas

In [46]:
```
def weightInitialize(L_in, L_out):
    cini = 0.12
    aux = np.random.uniform(-cini, cini, size =(L_in, L_out))
    aux = np.insert(aux,0,1,axis = 0)
    return aux
```

Prueba para la red Neuronal

Con esta función probaremos la red con matrices de pesos inicializadas aleatoriamente y comprobaremos su precisión después de ser optimizada con la función optimize.

In [63]:
```
def NNTest (num_entradas, num_ocultas, num_etiquetas, reg, X, Y, laps):
    t1 = weightInitialize(num_entradas, num_ocultas)
    t2 = weightInitialize(num_ocultas, num_etiquetas)

    params = np.hstack((np.ravel(t1), np.ravel(t2)))
    out = opt.minimize(fun = backprop, x0 = params, args = (num_entradas, num_ocult
as, num_etiquetas, X, Y, reg), method='TNC', jac = True, options = {'maxiter': laps
})

    Thetas1 = out.x[:(num_ocultas*(num_entradas+1))].reshape(num_ocultas,(num_entra
das+1))
    Thetas2 = out.x[(num_ocultas*(num_entradas+1)):].reshape(num_etiquetas,(num_ocu
ltas+1))

    input = np.hstack([np.ones((len(X), 1), dtype = np.float), X])
    hipo = forwardProp(Thetas1, Thetas2, input)[3]


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

    precision = sum(prec) / len(X)

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

In [67]:
```
NNTest(400, 25, 10, 1, X, Y, 70)
```

```
Program precision:  91.8 %
```