|  |
| --- |
| Aprendizaje automático y minería de datos |
| Práctica 4 – Entrenamiento de redes neuronales |

|  |
| --- |
| Ramón Arjona Quiñones  Celia Castaños Bornaechea |

Contenido

[1. Función de coste 2](#_Toc31576551)

[Resumen 2](#_Toc31576552)

[1.1 Coste 2](#_Toc31576553)

[1.2 Coste regularizado 2](#_Toc31576554)

[2. Gradiente 3](#_Toc31576555)

[Resumen 3](#_Toc31576556)

[2.1 Generación de pesos aleatorios 3](#_Toc31576557)

[2.2 Retro-propagación 3](#_Toc31576558)

[2.3 Gradiente 4](#_Toc31576559)

[Código completo 4](#_Toc31576560)

[3. Aprendizaje de los parámetros 6](#_Toc31576561)

[Código del ejercicio 7](#_Toc31576562)

## Función de coste

### Resumen

Calcular el coste de una red neuronal respecto a un conjunto de ejemplos de entrenamiento.

### 1.1 Coste



def network\_cost(H, Y):

    '''

    Calcula el coste de manera vectorizada para la red neuronal,

    con una salida de la red H y la Y de los ejemplos de entrenamiento

    '''

    #Variables auxiliares

    m = Y.shape[0]

    # Usamos "multiply" en vez de "dot" para que haga multiplicación

     elemento a elemento, (no producto escalar) y así luego los sumamos

 todos en vez de hacer un doble bucle

    ## Coste cuando Y = 1

    costeUno = np.multiply(Y, np.log(H)).sum() # Suma todos los

    #elementos de la matriz (Y x H)

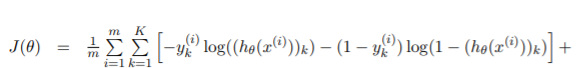
    ## Coste cuando Y = 0

    costeCero = np.multiply((1 - Y), np.log(1 - H)).sum()

    #Coste sin regularizar

    return -1 / m \* (costeUno + costeCero)

### 1.2 Coste regularizado



def reg\_network\_cost(H, Y, lamda, theta1, theta2):

    '''

    Calcula el coste (regularizado) para la red neuronal,

    con una salida de la red H y la Y de los ejemplos de entrenamiento

    '''

    #Variables auxiliares

    m = Y.shape[0]

    #Coste sin regularizar

    cost = network\_cost(H, Y)

    #Término de regularización (las columnas de 1's de thetas las quitamos)

    thetaSum = ((theta1[:, 1:]\*\*2).sum() + (theta2[:, 1:]\*\*2).sum())

    regTerm = lamda / (2 \* m) \* thetaSum

    #Coste regularizado

    return (cost + regTerm)

## Gradiente

### Resumen

En este punto se añade el cálculo del gradiente, para esto es necesario implementar la retro-propagación. Este se devolverá junto al coste.



### 2.1 Generación de pesos aleatorios

Se implementa una función que inicializa una matriz de pesos con valores aleatorios en el rango [ - 0’12, 0’12].

def pesosAleatorios(L\_in, L\_out):

    """

    Inicializa una matriz de pesos con valores aleatorios dentro de un rango epsilon

    """

    # Rango

    epsilon = 0.12

    # Inicializamos la matriz con 0s

    pesos = np.zeros((L\_out, 1 + L\_in))

    # Valores aleatorios en ese intervalo

    pesos = np.random.rand(L\_out, 1 + L\_in) \* (2 \* epsilon) - epsilon

    return pesos

### 2.2 Retro-propagación

Permite calcular el gradiente del coste de la red neuronal.

Para cada ejemplo de entrenamiento primero se ejecuta una pasada hacia adelante, que calcula la salida de la red. Y después una pasada hacia atrás, para computar la contribución de cada nodo de las capas al error producido en la salida.

[…]

# 4. RETRO - PROPAGACIÓN

    for t in range(m):

        a1t = a1[t, :] # (1, 401)

        a2t = a2[t, :] # (1, 26)

        ht = h[t, :] # (1, 10)

        yt = y[t] # (1, 10)

        #Error en la capa de salida

        d3t = ht - yt

        #Error en la capa oculta

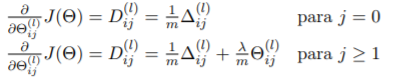
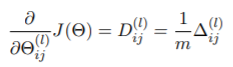
        d2t = np.dot(theta2.T, d3t) \* (a2t \* (1 - a2t)) # (1, 26)

        delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])

        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

[…]

### 2.3 Gradiente

Una vez se han procesado todos los ejemplos se calcula el gradiente. Se lleva a cabo sin término de regularización. Se dividen los valores acumulados durante la retro-propagación entre el número de ejemplos y después se regulariza.

Gradiente

Gradiente regularizado

[…]

    # 6. Calculamos el gradiente...

    delta1 = delta1 / m

    delta2 = delta2 / m

    # ... y lo regularizamos

    delta1[:,1:] = delta1[:,1:] + (lamda / m) \* theta1[:,1:]

    delta2[:,1:] = delta2[:,1:] + (lamda / m) \* theta2[:,1:]

[…]

### Código completo

Juntándolo todo, la implementación resultante para la función para el *back propagation* es:

def back\_prop (nn\_params, num\_entradas, num\_ocultas, num\_etiquetas, X,

lamda):

    """

    Implementa la propagación hacia atrás de la red neuronal con 2 capas

    Tenemos que convertir el vector "nn\_params" en dos matrices, ya que

viene desenrollado.

    Devuelve el coste y el vector de gradientes (desenrollado también)

    """

    # 1. Volvemos a construir las matrices de pesos

    theta1 = np.reshape(nn\_params[:num\_ocultas \* (num\_entradas + 1)],

                        (num\_ocultas, num\_entradas + 1)) # (25,401)

    theta2 = np.reshape(nn\_params[num\_ocultas \* (num\_entradas + 1):],

                        (num\_etiquetas, num\_ocultas + 1)) # (10,26)

    # Número de ejemplos de entrenamiento

    m = X.shape[0]

    X = np.hstack([np.ones([m, 1]), X]) #Para el término indep.

    # 2. Hacemos la propagación hacia delante para obtener las

activaciones

    a1, z2, a2, z3, h = forward\_prop(X, theta1, theta2)

    # 3. Inicializamos las matrices delta (con ceros)

    delta1 = np.zeros((num\_ocultas, num\_entradas + 1))

    delta2 = np.zeros((num\_etiquetas, num\_ocultas + 1))

    # 4. RETRO - PROPAGACIÓN

    for t in range(m):

        a1t = a1[t, :] # (1, 401)

        a2t = a2[t, :] # (1, 26)

        ht = h[t, :] # (1, 10)

        yt = y[t] # (1, 10)

        #Error en la capa de salida

        d3t = ht - yt

        #Error en la capa oculta

        d2t = np.dot(theta2.T, d3t) \* (a2t \* (1 - a2t)) # (1, 26)

        delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])

        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    # 5. Calculamos el coste regularizado

    regCost = reg\_network\_cost(h, y, lamda, theta1, theta2)

    # 6. Calculamos el gradiente...

    delta1 = delta1 / m

    delta2 = delta2 / m

    # ... y lo regularizamos

    delta1[:,1:] = delta1[:,1:] + (lamda / m) \* theta1[:,1:]

    delta2[:,1:] = delta2[:,1:] + (lamda / m) \* theta2[:,1:]

    # Desenrollamos el gradiente y lo devolvemos junto al coste

    grad = np.concatenate((delta1.ravel(), delta2.ravel()))

    return regCost, grad

## Aprendizaje de los parámetros

Ahora se entrena la red neuronal y se obtienen los valores para las Θ. Para ello se utiliza la función *scipy.optimize.minimize*

def trainNeutralNetwork(num\_entradas, num\_ocultas, num\_etiquetas, X, y,

lamda, num\_iter):

    '''

    Entrena una red neuronal de 2 capas y devuelve las matrices de pesos para cada capa

    La y debe estar en formato onehot

    '''

    # 1. Comenzamos con unos pesos aleatorios

    theta1 = pesosAleatorios(num\_entradas, num\_ocultas)

    theta2 = pesosAleatorios(num\_ocultas, num\_etiquetas)

    nn\_params = np.concatenate((theta1.ravel(), theta2.ravel())) #Los unimos en 1 solo vector

    # 2. Llamamos a la función minimize para obtener las matrices de pesos óptimos

    # (las que hacen que haya un mínimo en el coste devuelto, usando back\_prop)

    thetaOpt = minimize(fun=back\_prop,

                       x0=nn\_params,

                       args=(num\_entradas,

                             num\_ocultas,

                             num\_etiquetas,

                             X, y, lamda),

                       method='TNC',

                       jac=True,

                       options={'maxiter':num\_iter}).x

    # 3. Tenemos que reconstruir los pesos a partir del vector

    theta1 = np.reshape(thetaOpt[:num\_ocultas \* (num\_entradas + 1)],

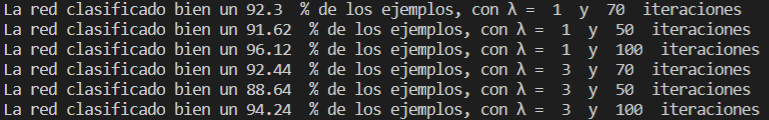
                        (num\_ocultas, num\_entradas + 1))

    theta2 = np.reshape(thetaOpt[num\_ocultas \* (num\_entradas + 1):],

                        (num\_etiquetas, num\_ocultas + 1))

    # Devolvemos los pesos óptimos

    return [theta1, theta2]

Ahora se prueba su efectividad con distintos valores para λ y cantidad de iteraciones.

### Código del ejercicio

def Ejercicio1(lamda, num\_iter):

    '''

    Redes neuronales

    '''

    # 1. Cargamos los datos

    data = loadmat('ex4data1.mat')

    X = data['X'] # (5000x400)

    y = data['y'].ravel() #(5000,)

    y = (y - 1) #Porque están de 1 - 10 y los queremos del 0 - 9

    m = X.shape[0]

    # 2. Atributos de la red neuronal

    num\_entradas = 400

    num\_ocultas = 25

    num\_etiquetas = 10

    # 3. Inicializamos y\_onehot

    y\_onehot = np.zeros((m, num\_etiquetas)) #5000 x 10

    for i in range(m):

        y\_onehot[i][y[i]] = 1

    # Visualizar 100 ejemplos

    #sample = np.random.choice(X.shape[0], 100)

    #fig, ax = displayData(X[sample])

    #plt.show()

    # 4. Entrenamos la red neuronal y sacamos los pesos óptimos

    theta1, theta2 = trainNeutralNetwork(num\_entradas, num\_ocultas, num\_etiquetas, X, y\_onehot, lamda, num\_iter)

    # 5. Con los pesos óptimos obtenidos, hacemos la propagación hacia delante y obtenemos la predicción de la red

    unosX = np.hstack([np.ones([m, 1]), X])

    a1, z2, a2, z3, h = forward\_prop(unosX, theta1, theta2)

    # Sacamos el porcentaje de aciertos

    porcentaje = calcula\_porcentaje(y, h, 3)

    print("La red clasificado bien un",  porcentaje, " % de los ejemplos, con λ = ", lamda, " y ",  num\_iter, " iteraciones")