Aprendizaje automático y minería de datos

Práctica 4 – Entrenamiento de redes neuronales

Ramón Arjona Quiñones
Celia Castaños Bornaechea

Contenido

1. Función de coste

Resumen

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

1.1 Coste

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right]$$

```
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

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

```
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 quitamo
s)
    thetaSum = ((theta1[:, 1:]**2).sum() + (theta2[:, 1:]**2).sum())
    regTerm = lamda / (2 * m) * thetaSum

    #Coste regularizado
    return (cost + regTerm)
```

2. 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.

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1 - g(z)) \qquad g(z) = \frac{1}{1 + e^{-z}}$$

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 ra
ngo 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.

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

Gradiente

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \qquad \text{para } j = 0$$
$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \qquad \text{para } j \geq 1$$

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
```

3. 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 uni
mos en 1 solo vector

    # 2. Llamamos a la función minimize para obtener las matrices de peso
s 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.

```
La red clasificado bien un 92.3  % de los ejemplos, con λ =  1  y  70  iteraciones
La red clasificado bien un 91.62  % de los ejemplos, con λ =  1  y  50  iteraciones
La red clasificado bien un 96.12  % de los ejemplos, con λ =  1  y  100  iteraciones
La red clasificado bien un 92.44  % de los ejemplos, con λ =  3  y  70  iteraciones
La red clasificado bien un 88.64  % de los ejemplos, con λ =  3  y  50  iteraciones
La red clasificado bien un 94.24  % de los ejemplos, con λ =  3  y  100  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_e
tiquetas, X, y_onehot, lamda, num_iter)

    # 5. Con los pesos óptimos obtenidos, hacemos la propagación hacia de
lante 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")
```