Javier Cordero Calvo y Juan Ruiz Jiménez

Aprendizaje Automático y Minería de Datos – Práctica 4

1. Función de coste:

En primer lugar, cargamos los elementos de la matriz de datos “ex4data1.mat” en la variable data, y de ella sacamos los conjuntos de datos X e y (este último “estirado” en un vector 1xN siendo N la dimensión de cada fila de la matriz)

Tras haber hecho esto, extraemos el número de entradas que tiene cada ejemplo de entrenamiento, así como asignamos a variables el número de elementos con el que cuenta la red neuronal en la capa oculta y el número de etiquetas que tenemos, así como el valor de lambda.

Preparamos la variable y para la práctica dentro de la nueva variable “y_onehot”

```
data = loadmat("ex4data1.mat")

y = data["y"].ravel()
X = data["X"]

num_entradas = X.shape[1]
capa_oculta = 25
num_labels = 10
landa = 1

lenY = len(y)
y = (y - 1)
y_onehot = np.zeros((lenY, num_labels))
for i in range(lenY):
    y_onehot[i][y[i]] = 1
```

Tras tenerlo todo preparado, para la primera prueba que es la de la función del coste cogeremos una matriz de pesos ya definida, ex4weights.mat, y guardaremos en Theta1 y Theta2 su resultado.

Aunque esto lo hagamos para la primera prueba, más adelante las matrices de thetas las tendremos que calcular mediante una función que nos dará pesos aleatorios para ambas matrices.

Una vez tenemos los pesos, independientemente de la manera en la que los hayamos conseguido, tenemos que juntarlos en un único vector de dimensión 1xM, siendo M la longitudo de ambas matrices de Thetas ya “estiradas”

```
#Calculo de pesos por una red ya entrenada
weights = loadmat ("ex4weights.mat")
Theta1, Theta2 = weights["Theta1"], weights["Theta2"]
```

```
# Crea una lista de Thetas
Thetas = [Theta1, Theta2]

# Concatenación de las matrices de pesos en un solo vector
unrolled_Thetas = [Thetas[i].ravel() for i,_ in enumerate(Thetas)]
params = np.concatenate(unrolled_Thetas)
```

Una vez realizado todo esto, procederemos a crear la función propuesta “backprop”, en la que, para comenzar, desmenuzaremos params_rn en Theta1 y Theta2 para después proceder a su uso y guardaremos en m la longitud de una de las filas de X.

El siguiente paso es utilizar la función “PropagaciónHaciaDelante” para calcular la h (salida de la red neuronal) que necesitaremos para el coste regularizado.

Tras esto, finalmente contamos con la llamada a la función de coste regularizado que nos dará el coste óptimo de la red con los parámetros previamente establecidos.

```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y,
reg):
    m = X.shape[0]

    # Sacamos ambas thetas
    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)))

    a1, z2, a2, z3, h = PropagacionHaciaDelante(X, theta1, theta2)

    coste = costeRegularizado(m, h, y, reg, theta1, theta2)
```

```
#Devuelve la salida de la red neuronal
def PropagacionHaciaDelante(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]), sigmoide(z2)])
    z3 = np.dot(a2, theta2.T)
    h = sigmoide(z3)
    return a1, z2, a2, z3, h
```

Como función auxiliar, vemos que cuenta con la función sigmoide:

```
# Función sigmoide
def sigmoide(x):
    return 1 / (1 + np.exp(-x))
```

Para el calculo del coste, tenemos que llamar al coste regularizado, que es la suma del coste normal más la regularización.

```
# Cálculo del coste regularizado
def costeRegularizado(m, h, Y, reg, theta1, theta2):
    return coste(m, h, Y) + ((reg / (2 * m)) * ((np.sum(np.square(t
heta1[:, 1:]))) + (np.sum(np.square(theta2[:,1:])))))
```

```
# Cáculo del coste no regularizado
def coste(m, h, y):
    J = 0
    for i in range(m):
        J += np.sum(-y[i] * np.log(h[i]) \
             - (1 - y[i]) * np.log(1 - h[i]))
    return (J / m)
```

Cabe destacar dentro de estas dos últimas funciones, que la función costeRegularizado está vectorizada mientras que la función coste no lo está.

Tras realizar todos estos pasos, encontramos que la salida del coste es de 0.383770…, que es el valor óptimo que el enunciado nos pedía para el coste y los datos dados, con un valor de lambda = 1

Destacamos también que los pesos de la red para esta parte de la práctica son los proporcionados por la matriz de pesos, y no está calculado en base a los pesos aleatorios.

2. Cálculo del gradiente:

Para esta parte, el primer paso es inicializar la matriz con pesos aleatorios en lugar de leyendo la matriz de pesos.

```
# Inicializa una matriz de pesos aleatorios
def pesosAleatorios(L_in, L_out):
    ini = 0.12
    out = np.random.uniform(low=-ini, high=ini, size=(L_out, L_in))
    out = np.hstack((np.ones((out.shape[0], 1)), out))
    return out
```

Una vez obtenemos dicha matriz, dentro de nuestra función backprop, después de calcular el coste, añadiremos la función de cálculo de gradiente proporcionada en la práctica.

```
# Inicialización de dos matrices "delta" a 0 con el tamaño de los thethas
    delta1 = np.zeros_like(theta1)
    delta2 = np.zeros_like(theta2)

    # Por cada ejemplo
    for t in range(m):
        a1t = a1[t, :] # (1, 401)
        a2t = a2[t, :] # (1, 26)
        ht = h[t, :] # (1, 10)
        yt = y[t]

        d3t = ht - yt
        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, :])

    delta1 = delta1 / m
    delta2 = delta2 / m

    # Gradiente perteneciente a cada delta
    delta1[:, 1:] = delta1[:, 1:] + (reg * theta1[:, 1:]) / m
    delta2[:, 1:] = delta2[:, 1:] + (reg * theta2[:, 1:]) / m
```

Obtenidos delta1 y delta2, ambas matrices, tenemos que devolverlas igual que fueron proporcionadas a la práctica, es decir, en un único vector cuya dimensión sea la 1xN, siendo N la suma de todos los elementos de ambas matrices concatenados uno detrás de otro.

Finalmente, devolvemos el coste calculado en el apartado anterior así como el gradiente ya preparado:

```
#Unimos los gradientes
  gradiente = np.concatenate((np.ravel(delta1), np.ravel(delta2)))

  return coste, gradiente
```

Además de la función sigmoide, en este apartado también hemos implementado la función derivada del sigmoide:

```
# Cálculo de la derivada de la función sigmoide
def derivada_sigmoide(x):
    return (sigmoide(x) * (1.0 - sigmoide(x)))
```

Finalmente, mediante la función minimize calcularemos el valor mínimo de theta para que sea óptimo para nuestro sistema. Tras tenerlo, Calcularemos que el error cometido en el cálculo del gradiante sea mínimo, por lo que utilizaremos la función proporcionada checkNNGradientes. Esta nos devolverá el error cometido y tendremos que asegurarnos que sea menor que 10^-9

```
# Obtención de los pesos óptimos entrenando una red con los pesos
aleatorios
optTheta = opt.minimize(fun=backprop, x0=params,
        args=(num_entradas, capa_oculta, num_labels,
        X, y_onehot, 1), method='TNC', jac=True,
        options={'maxiter': 70})

#Calculo de la precision del gradiante gracias a checkNNGradients
print("Diferencia de precision de gradiantes: ", str(np.sum(checkNN
Gradients(backprop, 1))), ", maximo aceptado = 10e-9")
```

Salida del sistema en este punto: "***Diferencia de precision de gradiantes: 1.1755403248048246e-10 , maximo aceptado = 10e-9***"

3. Aprendizaje de los parámetros:

Para terminar esta práctica, calcularemos el porcentaje de aciertos que obtiene nuestra red.

Para ello, desglosaremos los valores de optTheta (la theta óptima calculada anteriormente) y utilizaremos la propagación hacia delante para calcular la salida del sistema el cual ya ha sido entrenado anteriormente.

```
# Desglose de los pesos óptimos en dos matrices
Theta1Final = np.reshape(optTheta.x[:capa_oculta * (num_entradas + 1)],
    (capa_oculta, (num_entradas + 1)))

Theta2Final = np.reshape(optTheta.x[capa_oculta * (num_entradas + 1): ],
    (num_labels, (capa_oculta + 1)))

# H, resultado de la red al usar los pesos óptimos
a1, z2, a2, z3, h = PropagacionHaciaDelante(X, Theta1Final, Theta2Final)
```

Finalmente, mediante la función calcAciertos, comprobaremos el número de errores que comete nuestra red comparándola con la salida y aportada por el enunciado.

```
# Cálculo de la precisión de la red neuronal
print("{0:.2f}% de precision".format(calcAciertos(y,h)))
```

Esto nos deja una salida final de nuestro sistema con el mensaje: "***90.90% de precisión***", que es el porcentaje final de aciertos de nuestro sistema.