Práctica 4: Entrenamiento de redes neuronales

Material proporcionado:

Fichero	Explicación
`ex4data1.mat`	Datos de entrenamiento con imágenes de números escritos a mano.
`ex4weights.mat`	Parámetros de la red neuronal.
`displayData.py`	Función que permite visualizar los datos.
`checkNNGradients.py`	Función que compara gradientes.

1. Función de coste

El objetivo de esta primera parte de la práctica es implementar el cálculo de la función de coste de una red neuronal para un conjunto de ejemplos de entrenamiento. Se utiliza el mismo conjunto de datos de la práctica 3. El fichero `ex4data1.mat` contiene 5000 ejemplos de entrenamiento en el formato nativo para matrices de Octave/Matlab[1]. Cada ejemplo de entrenamiento es una imagen de 20×20 píxeles donde cada píxel está representado por un número real que indica la intensidad en escala de grises de ese punto. Cada matriz de 20×20 se ha desplegado para formar un vector de 400 componentes que ocupa una fila de la matriz `X`. De esta forma, `X` es una matriz de 5000×400 donde cada fila representa la imagen de un número escrito a mano:

$$X = \begin{bmatrix} — & (x^{(1)})^T & — \\ — & (x^{(2)})^T & — \\ & \vdots & \\ — & (x^{(m)})^T & — \end{bmatrix}$$

El vector `y` es un vector de 5000 componentes que representan las etiquetas de los ejemplos de entrenamiento, donde la imagen del 0 se ha etiquetado como "10", manteniendo las etiquetas naturales del "1" al "9" para el resto de los números.

Eligiendo aleatoriamente 100 elementos del conjunto de datos y con ayuda de la función `displayData` proporcionada con la práctica se puede obtener una imagen como la que se muestra en la Figura 1.

[1] Una parte del conjunto de datos `http://yann.lecun.com/exdb/mnist/`

Figura 1: Muestra de los datos de entrenamiento

La red neuronal tiene la estructura que se muestra en la Figura 2, formada por tres capas, con 400 unidades en la primera capa (además de la primera fijada siempre a +1), 25 en la capa oculta y 10 en la capa de salida.

El fichero `ex4weights.mat` contiene las matrices $\Theta^{(1)}$ y $\Theta^{(2)}$ con el resultado de haber entrenado la red neuronal, que podemos cargar con la función `scipy.io.loadmat`:

```
weights = loadmat('ex4weights.mat')
Theta1, Theta2 = weights['Theta1'], weights['Theta2']
# Theta1 es de dimensión 25 x 401
# Theta2 es de dimensión 10 x 26
```

Con los valores proporcionados para las matrices $\Theta^{(1)}$ y $\Theta^{(2)}$ podrás comprobar si realizas correctamente el cálculo del coste. Recuerda que el coste de una red neuronal (sin regularización) viene dado por la expresión:

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

donde $h_\theta(x^{(i)})$ se calcula utilizando el algoritmo de propagación hacia delante que se resume en la figura 2. $K = 10$ es el número de etiquetas distintas y $(h_\theta(x^{(i)}))_k = a_k^{(3)}$ es el valor de salida de la k-ésima unidad de salida de la red neuronal. Recuerda que aunque los valores originales de las etiquetas son los números del 1 al 10, para poder entrenar la red neuronal es necesario codificar las etiquetas como vectores de 10 componentes con todos sus elementos a 0 salvo uno a 1:

$$y = \begin{bmatrix}1\\0\\0\\\vdots\\0\end{bmatrix}, \begin{bmatrix}0\\1\\0\\\vdots\\0\end{bmatrix}, \ldots \text{ o } \begin{bmatrix}0\\0\\0\\\vdots\\1\end{bmatrix}$$

donde, el vector con un 1 en la primera componente codifica al número 1, el vector con un 1 en la segunda componente al número 2 y así sucesivamente hasta el vector con un 1 en la última

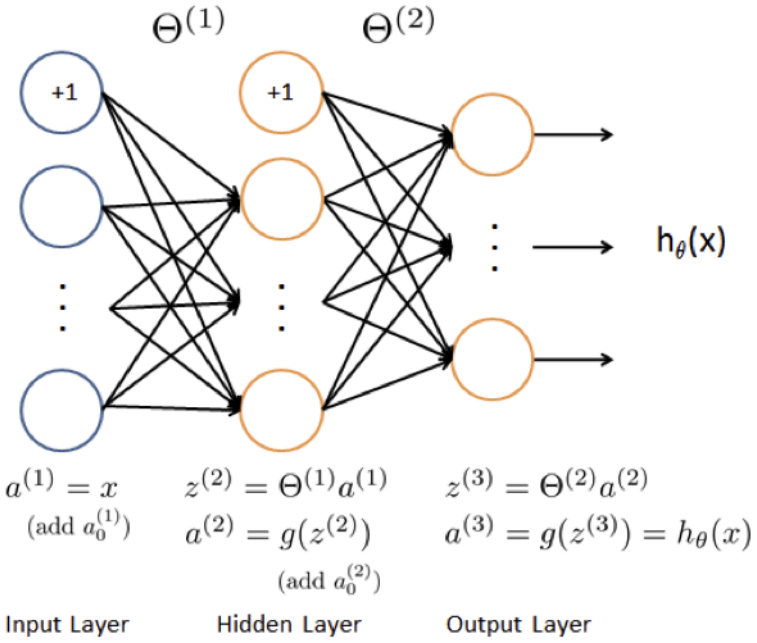

Figura 2: Propagación hacia delante en la red neuronal

componente que representa al 10 (que se corresponde con la imagen de un 0, por la manera como se han codificado los datos en el vector y leído del fichero `ex4data1.mat`, donde la imagen del 0 se codifica con la etiqueta "10").

Esta primera versión de la función de coste debería devolver un valor aproximado de 0.287629.

A continuación, debes completar el cálculo de la función de coste, añadiendo el término de regularización:

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

Aunque en la expresión, por claridad, se han incluido los límites explícitos de $\Theta^{(1)}$ y $\Theta^{(2)}$, tu código debe funcionar para cualquier red neuronal de tres capas. Recuerda que los pesos de los términos independientes $\Theta_{j,0}^{(1)}$ y $\Theta_{j,0}^{(2)}$, que corresponden, respectivamente, a la primera columna de la matriz `Theta1` y la primera columna de la matriz `Theta2`, no se incluyen en el cálculo del término de regularización. Con los valores proporcionados para `Theta1` y `Theta2` y un valor de $\lambda = 1$, el coste regularizado debería estar en torno a 0.383770.

2. Cálculo del gradiente

A continuación, has de implementar el cálculo del gradiente de una red neuronal de tres capas. El cálculo del gradiente lo debes implementar en una función que recibe todos los parámetros de la red neuronal desplegados en un array unidimensional y que los devuelve de la misma forma. Además, para poder utilizar la función de comprobación del gradiente, `checkNNGradients`,

que se proporciona con la práctica, es necesario que la función de cálculo del gradiente devuelva una tupla cuyo primer elemento es el coste y el segundo el gradiente, y que presente el siguiente perfil:

```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, reg)

# backprop devuelve una tupla (coste, gradiente) con el coste y el gradiente de
# una red neuronal de tres capas, con num_entradas, num_ocultas nodos en la capa
# oculta y num_etiquetas nodos en la capa de salida. Si m es el número de ejemplos
# de entrenamiento, la dimensión de 'X' es (m, num_entradas) y la de 'y' es
# (m, num_etiquetas)
```

Deberías implementar esta función de forma que funcione para cualquier número de ejemplos de entrenamiento y cualquier número de etiquetas (puedes suponer que el número de etiquetas es mayor o igual que 3).

Los parámetros de la red se reciben desplegados en `params_rn` como un array unidimensional y el gradiente se ha de devolver de la misma forma. El primer paso en la función de cálculo del gradiente será reconstruir las matrices de parámetros Θ_1 y Θ_2 a partir del array `params_rn`:

```
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)))
```

El algoritmo de retro-propagación permite calcular el gradiente de la función de coste de la red neuronal. Para ello, para cada ejemplo de entrenamiento $(x^{(t)}, y^{(t)})$ se ejecuta primero una pasada "hacia delante" para así calcular la salida de la red $h_\theta(x)$. A continuación, se ejecuta una pasada "hacia atrás" para computar en cada nodo j de cada capa l su contribución $\delta_j^{(l)}$ al error que se haya producido en la salida. En la figura 3 se resume el algoritmo.

La contribución de cada ejemplo de entrenamiento al gradiente se acumula en las matrices $\Delta^{(1)}$ y $\Delta^{(2)}$, inicializadas a 0, con las mismas dimensiones que, respectivamente, $\Theta^{(1)}$ y $\Theta^{(2)}$:

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)}(a^{(l)})^T$$

Una vez procesados los m ejemplos, el gradiente sin regularizar se obtiene dividiendo por m los valores acumulados en el bucle:

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

Una vez hayas comprobado que la implementación del gradiente sin regularizar es correcta, podrás añadir el término de regularización. Para ello, basta con actualizar los valores $\Delta_{ij}^{(l)}$ que se han calculado con el algoritmo de retro-propagación de la siguiente forma:

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

Recuerda que no se añade término de regularización a la primera columna de $\Theta^{(l)}$, y que en esta matriz $\Theta_{ij}^{(l)}$ el índice i empieza en 1 mientras que el índice j empieza en 0:

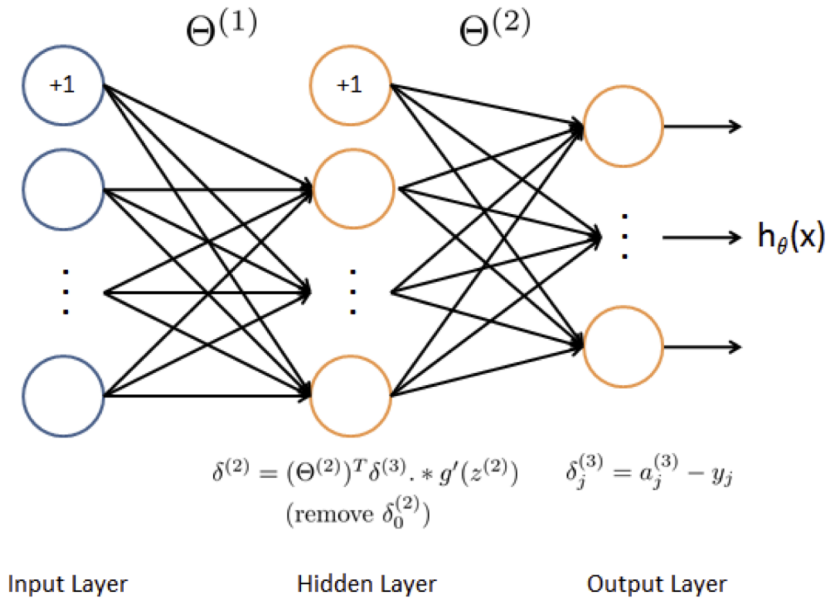

Figura 3: Retro-propagación

$$\Theta^{(l)} = \begin{bmatrix} \Theta_{1,0}^{(l)} & \Theta_{1,1}^{(l)} & \dots \\ \Theta_{2,0}^{(l)} & \Theta_{1,1}^{(l)} & \dots \\ \vdots & & \ddots \end{bmatrix}$$

Una vez modificado el gradiente para añadir el término de regularización, deberás ejecutar de nuevo el chequeo del gradiente para comprobar que es correcto.

2.1. Comprobación del gradiente

Para comprobar que el cálculo del gradiente de $J(\Theta)$ es correcto, se utiliza la función incluida en el archivo `checkNNGradients.py` proporcionado con la práctica.

La comprobación del gradiente se hace comparándolo con el resultado de un método aproximado que utiliza el cálculo del coste. Supongamos que tenemos una función $f_i(\theta)$ que calcula $\frac{\partial}{\partial \theta_i} J(\theta)$ y queremos comprobar si el cálculo es correcto. Dados los vectores $\theta^{(i+)}$, que es igual que θ excepto porque al i-ésimo elemento se le ha sumado ϵ, y $\theta^{(i-)}$, que es igual que θ excepto porque al i-ésimo elemento se le ha restado ϵ,

$$\theta^{(i+)} = \theta + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}, \text{y } \theta^{(i-)} = \theta - \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \epsilon \\ \vdots \\ 0 \end{bmatrix}$$

es posible comprobar numéricamente el valor de $f_i(\theta)$ comprobando que para todo i se cumple:

$$f_i(\theta) \approx \frac{J(\theta^{(i+)}) - J(\theta^{(i-)})}{2\epsilon} \approx \frac{\partial}{\partial \theta_i} J(\theta)$$

Cómo de próximos estén los valores dependerá de J, pero para un valor de $\epsilon = 10^{-4}$, los valores deberían coincidir al menos en las primeras 4 cifras significativas.

La función de comprobación del gradiente proporcionada tiene el siguiente perfil:

```
def checkNNGradients(costNN, reg_param):
```

que espera en *costNN* una función con el perfil de la función `backprop` descrita más arriba, que devuelve una tupla con el coste y el gradiente, y en *reg_param* espera el valor del parámetro de regularización λ.

Como el cálculo numérico del gradiente es una operación costosa, es mejor utilizarla para vectores pequeños, por ello, `checkNNGradients` construye una pequeña red neuronal inicializada con pesos aleatorios sobre la que se aplican los dos métodos de cálculo del gradiente para poder así comparar sus resultados. Si la función que le has pasado calcula correctamente el coste y el gradiente entonces la diferencia debería ser menor de 10^{-9}.

3. Aprendizaje de los parámetros

Una vez comprobada la corrección de cálculo del coste y el gradiente ya se puede utilizar la función `scipy.optimize.minimize` para entrenar a la red neuronal y obtener los valores óptimos para $\Theta^{(1)}$ y $\Theta^{(2)}$.

La función de optimización necesita que le pasemos el array de pesos inicializado. Los pesos de la red neuronal se deben inicializar con valores aleatorios pequeños en el rango $[-\epsilon_{ini}, \epsilon_{ini}]$. Puedes utilizar, por ejemplo, el valor $\epsilon_{ini} = 0{,}12$ para la inicialización[2]

Entrenando a la red con 70 iteraciones y un valor de $\lambda = 1$ deberías obtener una precisión en torno al 93 % (puede variar hasta un 1 % debido a la inicialización aleatoria de los parámetros). Es interesante que pruebes distintos valores de λ y del número de iteraciones para ver cómo afecta eso al resultado.

4. Entrega de la práctica

La práctica debe entregarse utilizando el mecanismo de entregas del campus virtual. Se entregará un único fichero en formato pdf que contenga la memoria de la práctica, incluyendo el código desarrollado y los comentarios y gráficas que se estimen más adecuados para explicar los resultados obtenidos.

[2] Una buena estrategia para elegir ϵ_{ini} es a partir del número de nodos de la red, por ejemplo con la expresión $\epsilon_{ini} = \frac{\sqrt{6}}{\sqrt{L_{in}+L_{out}}}$, donde $L_{in} = s_l$ y $L_{out} = s_{l+1}$ son los números de unidades en las capas adyacentes a $\Theta^{(l)}$.