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.m`	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, el "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` se puede obtener una imagen como la que se muestra en la Figura 1.

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.

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

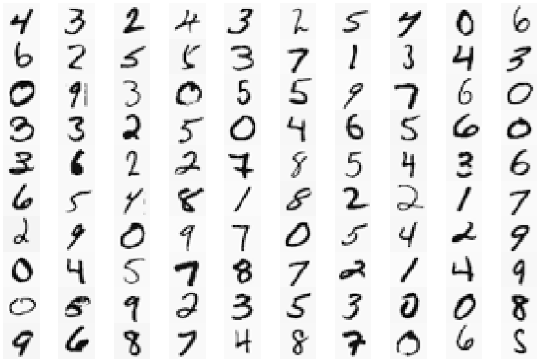

Figura 1: Muestra de los datos de entrenamiento

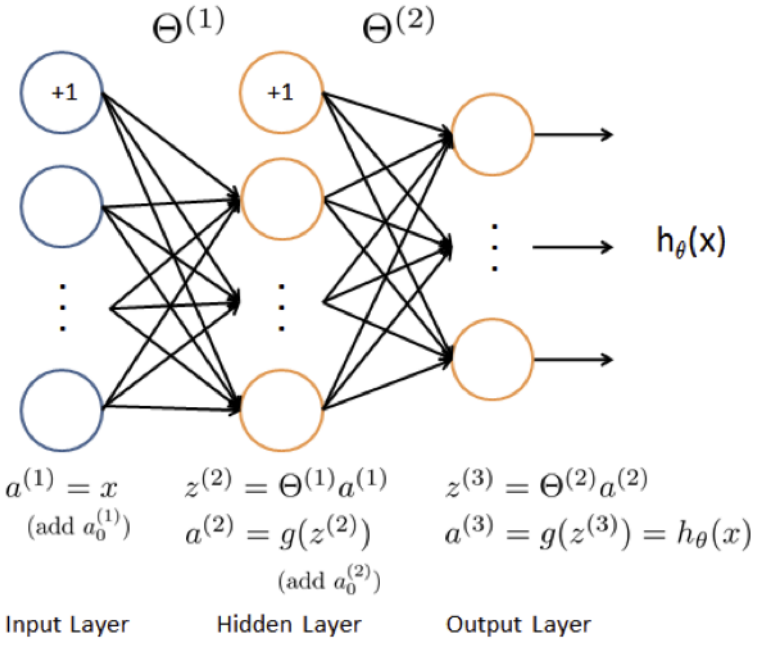

Figura 2: Red neuronal

El fichero `ex4weights.mat` contiene las matrices $\Theta^{(1)}$ y $\Theta^{(2)}$ con el resultado de haber entrenado la red neuronal y 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. Para ello has de implementar la función `backprop` que calcula el coste y el gradiante de una red neuronal de dos capas, con el siguiente perfil:

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

# backprop devuelve el coste y el gradiente de una red neuronal de dos capas.
```

implementada 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 vector columna y el gradiente se ha de devolver de la misma forma. El primer paso en `backprop` será reconstruir las matrices de parámetros a partir del vector `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)))
```

A continuación, has de implementar 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 como se indica 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. 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, por ejemplo, si $x^{(i)}$ es una imagen del dígito 5, entonces la correspondiente $y^{(i)}$ que has de usar para evaluar la función de coste es un vector de 10 componentes que tiene $y_5 = 1$ y el resto de componentes a 0.

Recuerda que en la implementación de `backprop` debes añadir una columna de 1s a la matriz `X` que se recibe como parámetro. Cada fila de las matrices `theta1` y `theta2` se corresponde con los parámetros de una unidad de la red. Por ejemplo, la primera fila de `theta1` contienen los parámetros de la primera unidad de la capa oculta.

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:

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

Aunque en la expresión, por claridad, se han incluido los límites explícitos de $\Theta^{(1)}$ y $\Theta^{(2)}$, recuerda que la implementación de `backprop` debe funcionar para cualquier red neuronal de tres capas. 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

En esta parte de la práctica has de implementar el algoritmo de retro-propagación para añadir el cálculo del gradiente a la función `backprop` que ha de devolverlo junto con el coste (como una tupla cuyo primer elemento sea el coste y el segundo el vector gradiente).

Para empezar, implementa una función auxiliar que calcule la derivada de la función sigmoide:

$$g'(z) = \frac{d}{dz}g(z) = g(z)(1-g(z))$$

donde

$$g(z) = \frac{1}{1+\mathrm{e}^{-z}}$$

La función ha de ser capaz de calcular la derivada del sigmoide tanto para un valor concreto, como para un vector o una matriz en cuyo caso lo aplicará a cada uno de sus componentes.

A continuación, implementa una función que inicialice una matriz de pesos $\Theta^{(l)}$ con valores aleatorios en el rango $[-\epsilon_{ini}, \epsilon_{ini}]$. Puedes utilizar el valor $\epsilon_{ini} = 0{,}12$ para la inicialización en una función con esta cabecera[2]:

```
def pesosAleatorios(L_in, L_out)
```

que devolverá una matriz de dimensión `(L_out, 1 + L_in)`.

2.1. Retro-propagación

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 $(x^{(t)}, y^{(t)})$ ejecuta primero una

[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 $\Theta^{(l)}$.

pasada "hacia adelante" 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.

Como se muestra en la figura 3, en los nodos de salida ($\delta_j^{(3)}$ en este caso en que tenemos 3 capas) el error se calcula simplemente como la diferencia entre la salida de la red y el valor $y^{(t)}$ especificado por el ejemplo de entrenamiento. Para los nodos de una capa oculta l, el error $\delta_j^{(l)}$ se calcula como una media ponderada de los términos de error de la capa $(l+1)$.

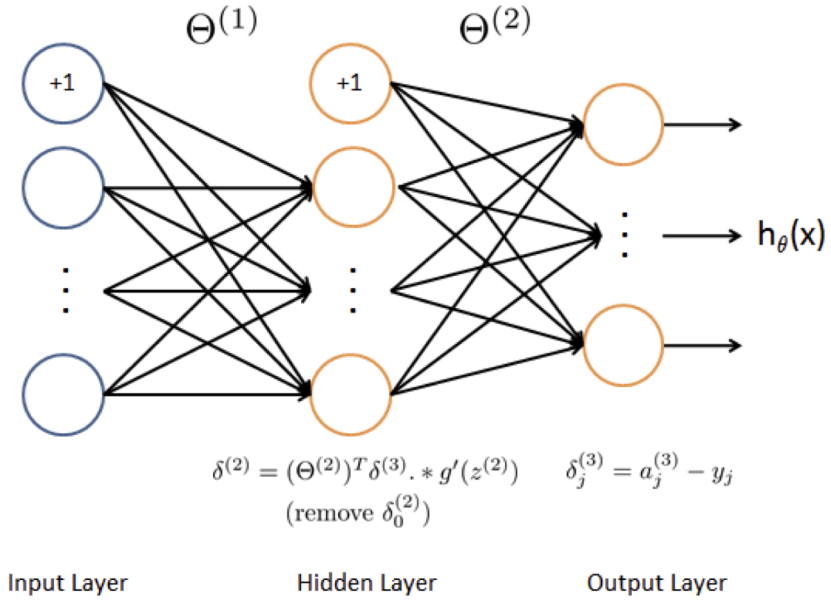

Figura 3: Retro-propagación

En concreto, debes implementar un bucle que procese los m ejemplos de entrenamiento, procesando cada ejemplo $(x^{(t)}, y^{(t)})$ de la siguiente forma:

1. Asigna el valor de $x^{(t)}$ a la capa de entrada de la red, $a^{(1)}$, y realiza una pasada hacia adelante calculando las salidas de las capas 2 y 3 ($a^{(2)}$ y $a^{(3)}$). Recuerda que has de añadir un `+1` al principio de $a^{(1)}$ y $a^{(2)}$.

2. Para cada unidad k de la capa 3 (la capa de salida), calcula

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

   donde $y_k \in \{0, 1\}$ representa si el ejemplo de entrenamiento pertenece a la clase k ($y_k = 1$) o a una clase diferente ($y_k = 0$).

3. Para la capa oculta $l = 2$, calcula

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} . * g'(z^{(2)})$$

4. Acumula el gradiente de este ejemplo utilizando la fórmula:

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

   y recuerda que debes eliminar o ignorar $\delta_0^{(2)}$.

Por último, una vez procesados los m ejemplos, calcula el gradiente (sin regularizar) dividiendo por m los valores acumulados en el bucle:

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

2.2. Chequeo del gradiente

Para comprobar que el cálculo del gradiente de $J(\Theta)$ es correcto, desplegaremos los parámetros $\Theta^{(1)}$ y $\Theta^{(2)}$ para convertirlos en un vector θ y poder así utilizar el método de chequeo del gradiente que se describe a continuación

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:

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

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.

Con la práctica se ha proporcionado el fichero `checkNNGradients.py` que continene una funcion que aproxima el valor de la derivada por este método. Como el cálculo numérico del gradiente es una operación costosa, es mejor utilizarla para vectores pequeños, por ello, se incluye también otra función donde se construye una pequeña red neuronal y se aplican los dos métodos de cálculo del gradiente (el numérico y el que tú hayas implementado en la función `backprop`) para poder comparar su resultado. Si el gradiente está implementado correctamente, la diferencia debería ser menor de 10^{-9}.

2.3. Redes neuronales regularizadas

Para completar la implementación de `backprop` has de añadir el término de regularización al cálculo del gradiente. 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:

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

Una vez modificado el gradiente para añadir el término de regularización, deberás ejecutar de nuevo el chequeo del gradiente para comprobar de nuevo que la diferencia entre los dos métodos es inferior a 10^{-9}

3. Aprendizaje de los parámetros

Una vez concluida la implementación de la función `backprop` ya se puede utilizar la función `scipy.optimize.minimize` para entrenar a la red neuronal y obtener los valores para $\Theta^{(1)}$ y $\Theta^{(2)}$. 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.