

Práctica 5. Entrenamiento de redes neuronales

Descripción

La red neuronal dispone de tres capas, la primera capa donde se introducen los datos, la capa oculta y la capa de salida. Los datos de entrada constan de píxeles de imágenes de dígitos. Como las imágenes tienen un tamaño de 20X20 se dispondrá de un tamaño de 400 en la capa de entrada. Por diseño se ha decidido tener 25 unidades en la capa oculta.

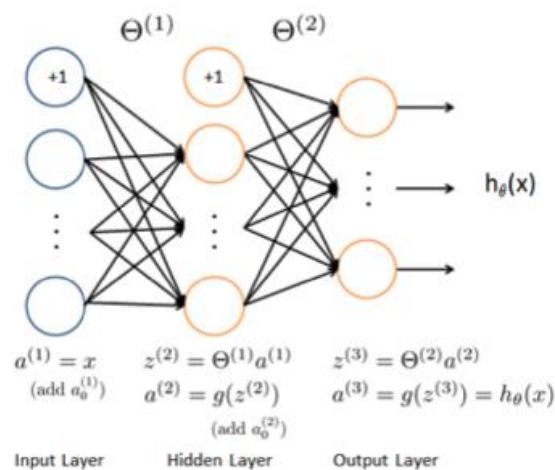


Figure 1.1: Neural network model

Carga de datos

Los datos que se van a utilizar en esta práctica vienen dados en el directorio *data* y se cargarán a través del método **load_data**

```
1 def load_data(file='./data/ex3data1.mat'):
2     data = loadmat(file, squeeze_me=True)
3     X = data['X']
4     y = data['y']
5     return X, y
```

Identificación de dígitos. Implementación de *one_hot*

La capa de salida tiene 10 unidades correspondientes a las 10 clases de dígitos. Mientras que las etiquetas originales (en la variable y) son 0, 1, ..., 9, con el propósito de entrenar una red neuronal, necesitamos codificar las etiquetas como vectores que contienen solo valores 0 o 1 (codificación "one-hot"), de manera que


$$y = \begin{bmatrix} 1 \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}, \dots \text{ or } \begin{bmatrix} 0 \\ 0 \\ 0 \\ \vdots \\ 1 \end{bmatrix}.$$

Por ejemplo, si $x(i)$ es una imagen con el dígito 5, entonces el correspondiente $y(i)$ que se tendrá con la red neuronal será un array de tamaño 10 con $y(5) = 1$ y los demás elementos de ese array serán 0.

Para poder implementar esto se desarrollará una función denominada ***one_hot***:

```
1  def calculate_exponent(value : int) -> int:
2      exponent : int = 0
3      while value > 1:
4          value >>= 1
5          exponent += 1
6      return exponent
7
8  def one_hot(y : list):
9      m = len(y)
10     y_onehot = np.zeros((m, 10))
11     for value in y:
12         value = calculate_exponent(value)
13
14     for i in range(m):
15         y_onehot[i][y[i]] = 1
16     return y_onehot
```

De esta manera calculamos la posición en la que se encuentra el número del que debemos de hacer el intercambio de 0 a 1.
Si se imprime el array construido en el método anterior junto con su valor verdadero se obtiene:



1	[0. 0. 0. 1. 0. 0. 0. 0. 0. 0.]	= 3
2	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]	= 4
3	[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]	= 6
4	[1. 0. 0. 0. 0. 0. 0. 0. 0. 0.]	= 0
5	[0. 0. 1. 0. 0. 0. 0. 0. 0. 0.]	= 2
6	[0. 0. 0. 0. 0. 0. 0. 0. 0. 1.]	= 9
7	[0. 0. 0. 0. 0. 0. 1. 0. 0. 0.]	= 6
8	[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]	= 8
9	[0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]	= 4
10	[0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]	= 1

Cálculo del coste

La ecuación para determinar el coste de una red neuronal es la siguiente:

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

Donde $h_{\Theta} x^{(i)}$ está computado como se ha mostrado en la figura 1.1 y $K = 10$ es el número total de posibles clases. Nótese que $h_{\Theta}(x^{(i)})_k$ es la activación (el valor de salida) para el valor k -ésimo del valor.

Se ha proporcionado un conjunto de redes de parámetros ($\Theta^{(1)}, \Theta^{(2)}$) ya entrenadas. Estos parámetros poseen dimensiones que concuerdan con la red neuronal de 25 unidades en la segunda capa y 10 unidades de salida (correspondiendo con los 10 dígitos de clases).

Para implementar la función de coste se desarrolla el siguiente método:

```
1 def cost(theta1, theta2, X, y, lambda_):
2     A1, A2, h = forward_propagation(X, theta1, theta2)
3
4     first_sum = y * np.log(h)
5     second_sum = (1 - y) * np.log(1 - h + 1e-6)
6
7     return (-1 / X.shape[0]) * np.sum(first_sum + second_sum)
```

Donde **forward_propagation** es un método auxiliar que calcula $h_{\theta}(x^{(i)})_k$

```
1 def sigmoid(z):
2     return 1 / (1 + np.exp(-z))
3
4 def forward_propagation(X, theta1, theta2):
5     n = X.shape[0]
6     X = np.hstack([np.ones([n, 1]), X],)
7
8     hidden = sigmoid(np.dot(X, theta1.T))
9     hidden = np.hstack([np.ones([n, 1]), hidden])
10
11    result = sigmoid(np.dot(hidden, theta2.T))
12    return X, hidden, result
```

A continuación, se explica un poco más en detalle la implementación de **forward_propagation**:

1. Se agrega una columna de unos a la matriz de características de entrada. Esta modificación se hace para añadir el término de sesgo y permitir que la red neuronal aprenda un término de sesgo para cada neurona en las capas ocultas y de salida.
2. Se calcula la salida de la capa oculta aplicando la función de activación sigmoide a la suma ponderada de las entradas y los pesos entre la capa de entrada y la capa oculta. Esto se realiza mediante la multiplicación de la matriz de entrada extendida por la matriz de pesos **theta1** transpuesta.

3. Se vuelve a agregar una columna de unos a la matriz de la capa oculta. Esta modificación tiene el mismo fin que en el primero punto, añadir el término de sesgo y permitir que la red neuronal aprenda un término de sesgo para cada neurona en las capas ocultas y de salida.
4. Se calcula la probabilidad de que un valor de entrada se corresponda con un valor de salida. Esto se consigue gracias a la función de activación o **sigmoide**.

Si comprobamos la implementación con los parámetros dados en el fichero *data/ex3weights.mat* obtenemos el siguiente resultado:



Cálculo del coste regularizado

Si se desea expandir la función coste de tal manera que devuelva el cálculo del coste regularizado de las redes neuronales, se añade el siguiente campo a la anterior función:

$$J(\Theta) = -\frac{1}{m} \left[\sum_{i=1}^m \sum_{k=1}^K y_k^{(i)} \log(h_{\Theta}(x^{(i)}))_k + (1 - y_k^{(i)}) \log(1 - (h_{\Theta}(x^{(i)}))_k) \right] + \frac{\lambda}{2m} \left[\sum_{l=1}^{L-1} \sum_{i=1}^{s_l} \sum_{j=1}^{s_{l+1}} (\Theta_{j,i}^{(l)})^2 \right]$$

En este caso concreto se puede especificar los campos de las capas y de las unidades:

$$\frac{\lambda}{2m} \left[\sum_{i=1}^{400} \sum_{j=1}^{25} (\Theta_{j,i}^{(1)})^2 + \sum_{i=1}^{25} \sum_{j=1}^{10} (\Theta_{j,i}^{(2)})^2 \right]$$

Por tanto, ampliamos el método anteriormente definido definiendo un nuevo método llamado **cost_reg**

```
1 def cost_reg(theta1, theta2, X, y, lambda_):  
2     costs = cost(theta1, theta2, X, y)  
3     m = X.shape[0]  
4     result = np.sum(np.sum(theta1[:, 1:] ** 2)) + np.sum(np.sum(theta2[:, 1:] ** 2))  
5  
6     return costs + (lambda_/(2*m)) * result
```

Si comprobamos la implementación con los parámetros dados en el fichero *data/ex3weights.mat* se obtiene el siguiente resultado:

```
1 0.38376070559687414
```

Backpropagation

Se implementará el algoritmo de **backpropagation** que calculará el gradiente del coste de la red neuronal. Se implementará inicialmente el algoritmo para que calcule los gradientes de los parámetros de una red neuronal no regularizada.

Teniendo en cuenta la explicación de las diapositivas

Training set $\{(x^{(1)}, y^{(1)}), \dots, (x^{(m)}, y^{(m)})\}$

Set $\Delta_{ij}^{(l)} = 0$ (for all l, i, j)

For $k = 1$ to m

Set $a^{(1)} = x^{(k)}$

Perform forward propagation to compute $a^{(l)}$ for $l = 2, 3, \dots, L$

Using $y^{(k)}$, compute $\delta^{(L)} = a^{(L)} - y^{(k)}$

Compute $\delta^{(L-1)}, \delta^{(L-2)}, \dots, \delta^{(2)}$

for all l, i, j

$$\Delta_{ij}^{(l)} := \Delta_{ij}^{(l)} + a_j^{(l)} \delta_i^{(l+1)}$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} \quad \text{if } j = 0$$

$$D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} \quad \text{if } j \geq 1$$

$$\frac{\partial}{\partial \Theta_{i,j}^{(l)}} J(\Theta) = \frac{1}{m} \sum_{k=1}^m (a_j^{(l)(k)} \delta_i^{(l+1)(k)})$$

vectorized, for all l

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

$$\frac{\partial}{\partial \Theta^{(l)}} J(\Theta) = D^{(l)}$$

Backpropagation no regularizado

Se realiza la implementación del algoritmo backpropagation no regularizado de la siguiente manera:

```
1 def backprop(theta1, theta2, X, y, lambda_):
2     # Obtención de los parámetros:
3     # X_: matriz con todos los valores de entrada incluyendo el término de sesgo como primera columna
4     # hidden: matriz con los valores de la capa de oculta incluyendo la columna donde se incluye el término de sesgo
5     # h: matriz con los valores de la capa de salida
6     X_, hidden, h = forward_propagation(X, theta1, theta2)
7
8     # Cálculo del coste no regularizado
9     J = cost(theta1, theta2, X, y)
10
11     # Gradientes
12     gradient_1, gradient_2 = gradients(theta1, theta2, X, y)
13
14     return J, gradient_1, gradient_2
```

Donde **gradients** es el método donde se calcula los diferentes gradientes no regularizados:

```
1 def gradients(theta1, theta2, X, y):
2
3     # Creación de los deltas respectivas con la forma de theta pero inicializados a cero
4     Delta_hidden = np.zeros(np.shape(theta1))
5     Delta_out = np.zeros(np.shape(theta2))
6
7     m = len(y)
8
9     # Se realiza la propagación hacia delante
10    # Obtención de los parámetros:
11    # X_: matriz con todos los valores de entrada incluyendo el término de sesgo como primera columna
12    # hidden: matriz con los valores de la capa de oculta incluyendo la columna donde se incluye el término de sesgo
13    # h: matriz con los valores de la capa de salida
14    X_, hidden, h = forward_propagation(X, theta1, theta2)
15
16    for k in range(m):
17        x_k = X_[k, :]
18        hidden_k = hidden[k, :]
19        out_k = h[k, :]
20        y_k = y[k, :]
21
22        delta_out = out_k - y_k
23        prime_g = (hidden_k * (1 - hidden_k))
24        delta_hidden = np.dot(theta2.T, delta_out) * prime_g
25
26        Delta_hidden = Delta_hidden + np.dot(delta_hidden[1:, np.newaxis], x_k[np.newaxis, :])
27        Delta_out = Delta_out + np.dot(delta_out[:, np.newaxis], hidden_k[np.newaxis, :])
28
29    return Delta_hidden / m, Delta_out / m
```


A continuación, se explica brevemente los pasos que sigue el método:

1. Se inicializan los deltas de las capas ocultas y de salida (Δ_{hidden} y Δ_{out}) con matrices de ceros del mismo tamaño que los parámetros θ_1 y θ_2 , respectivamente.
2. Se recorren los ejemplos de entrenamiento uno por uno.
3. Para cada ejemplo de entrenamiento, se realiza la propagación hacia adelante para calcular las activaciones en las capas ocultas y de salida (h). También se obtienen las activaciones de la capa oculta (A_2) y de la capa de salida (A_3).
4. Se calculan los errores de la capa de salida (δ_{out}) y de la capa oculta (δ_{hidden}) para el ejemplo de entrenamiento actual.
5. Se actualizan los deltas sumando el producto externo de δ_{hidden} y x^k para Δ_{hidden} , y el producto externo de δ_{out} y A_2 para Δ_{out} .
6. Se repiten los pasos **2-5** para todos los ejemplos de entrenamiento.
7. Se divide cada delta total por el número de ejemplos de entrenamiento (m) para obtener el gradiente promedio.

La función **main** ha sufrido ciertas modificaciones para conseguir implementar el algoritmo:

```
1 def main():
2     X, y = load_data()
3     y_onehot = one_hot(y)
4     theta1, theta2 = load_weights()
5     J, gradient_1, gradient_2 = backprop(theta1, theta2, X, y_onehot, lambda_=0.1)
6     print('Coste: {}'.format(J))
7     gradient = np.concatenate((np.ravel(gradient_1), np.ravel(gradient_2)))
8     print('Gradiente: {}'.format(gradient))
```

El resultado es el siguiente:

```
1 Coste: 0.2876200116672694
2 Gradiente: [ 6.18712766e-05 -2.11248326e-13  4.38829369e-14 ...  4.96169545e-04
3     1.10644527e-03  1.31443465e-03]
```

Backpropagation regularizado

Para el caso de implementar el algoritmo de **backpropagation** a una red neuronal regularizada modificamos el cálculo del gradiente y del coste en el método anterior

Para el cálculo del gradiente:

```
1 def gradients_reg(theta1, theta2, X, y, lambda_):
2     m = len(y)
3     delta_hidden, delta_out = gradients(theta1, theta2, X, y)
4
5     delta_hidden[:, 1:] = delta_hidden[:, 1:] + (lambda_ / m) * theta1[:, 1:]
6     delta_out[:, 1:] = delta_out[:, 1:] + (lambda_ / m) * theta2[:, 1:]
7
8     return delta_hidden, delta_out
```

Por lo que el algoritmo quedaría de la siguiente manera:

```
1 def backprop(theta1, theta2, X, y, lambda_):
2     # Obtención de los parámetros:
3     # X_: matriz con todos los valores de entrada incluyendo el término de sesgo como primera columna
4     # hidden: matriz con los valores de la capa de oculta incluyendo la columna donde se incluye el término de sesgo
5     # h: matriz con los valores de la capa de salida
6     X_, hidden, h = forward_propagation(X, theta1, theta2)
7
8     # Cálculo del coste regularizado
9     J = cost_reg(theta1, theta2, X, y, lambda_)
10
11     # Gradientes regularizados
12     gradient_1, gradient_2 = gradients_reg(theta1, theta2, X, y, lambda_)
13
14     return J, gradient_1, gradient_2
```

Y el resultado sería el siguiente:

```
1 Coste: 0.2972340810602299
2 Gradiente: [ 6.18712766e-05 -2.11248326e-13  4.38829369e-14 ...  4.96169545e-04
3             1.10644527e-03  1.31443465e-03]
```

Aprendizaje de parámetros

Después de implementar correctamente el cálculo del coste y del gradiente de la red neuronal el siguiente paso es usar el descendiente del gradiente para que aprenda un buen conjunto de parámetros. Por ello se inicializa las diferentes *thetas* de la siguiente manera:

$$\Theta^{(1)} = \Theta^{(1)} - \alpha \frac{\partial}{\partial \Theta^{(1)}} J(\Theta)$$
$$\Theta^{(2)} = \Theta^{(2)} - \alpha \frac{\partial}{\partial \Theta^{(2)}} J(\Theta)$$

Antes del entrenamiento es necesario inicializar aleatoriamente los parámetros. Una estrategia eficaz para la inicialización aleatoria consiste en seleccionar aleatoriamente valores para $\Theta^{(i)}$ de manera uniforme en el rango $[-\epsilon_{\text{init}}, \epsilon_{\text{init}}]$.

Se usará $\epsilon_{\text{init}} = 0,12$. Con este rango de valores se garantiza que los parámetros se mantengan pequeños y consigue un aprendizaje eficiente.

Se ha desarrollado la función ***learning_parameters()*** para la ejecución de esta parte de la práctica:

```
1 def train_neural_network(X, y, input_size, hidden_size, output_size, num_iterations, alpha, lambda_):
2
3     # Obtención de los pesos
4     theta1, theta2 = initialize_weights(input_size, hidden_size, output_size)
5
6     for i in range(num_iterations):
7
8         # Cálculo del coste y los diferentes gradientes
9         cost, grad1, grad2 = backprop(theta1, theta2, X, y, lambda_)
10
11        # Actualizar los parámetros con el descenso de gradiente
12        theta1 -= alpha * grad1
13        theta2 -= alpha * grad2
14
15        if i % 100 == 0:
16            print(f'Iteración {i}: Costo = {cost}')
17
18    return theta1, theta2
```

Donde ***train_neural_network()*** es el método descrito a continuación:

```
1 def train_neural_network(X, y, input_size, hidden_size, output_size, num_iterations, alpha, lambda_):
2
3     # Obtención de los pesos
4     theta1, theta2 = initialize_weights(input_size, hidden_size, output_size)
5
6     for i in range(num_iterations):
7
8         # Cálculo del coste y los diferentes gradientes
9         cost, grad1, grad2 = backprop(theta1, theta2, X, y, lambda_)
10
11        # Actualizar los parámetros con el descenso de gradiente
12        theta1 -= alpha * grad1
13        theta2 -= alpha * grad2
14
15        if i % 100 == 0:
16            print(f'Iteración {i}: Costo = {cost}')
17
18    return theta1, theta2
```

Se realiza la llamada a ***initialize_weights()*** para la creación de los Θ iniciales. Posteriormente estos Θ se verán afectados por los gradientes obtenidos en el bucle.

```
1 def initialize_weights(input_size, hidden_size, output_size):
2
3     INIT_EPSILON = 0.12
4     theta1 = np.random.random((hidden_size, (input_size + 1))) * (2 * INIT_EPSILON) - INIT_EPSILON
5     theta2 = np.random.random((output_size, (hidden_size + 1))) * (2 * INIT_EPSILON) - INIT_EPSILON
6
7     return theta1, theta2
```