Aprendizaje de ANN

En el anterior ejercicio implementamos el forward propagation (feedforward) de la red neuronal para predecir dígitos escritos a mano. En este ejercicio vamos a implementar el backpropagation (retro-propagación) para aprender los valores de los pesos de la red.

La red de nueronas será la misma del ejercicio anterior (véase Figura 1)

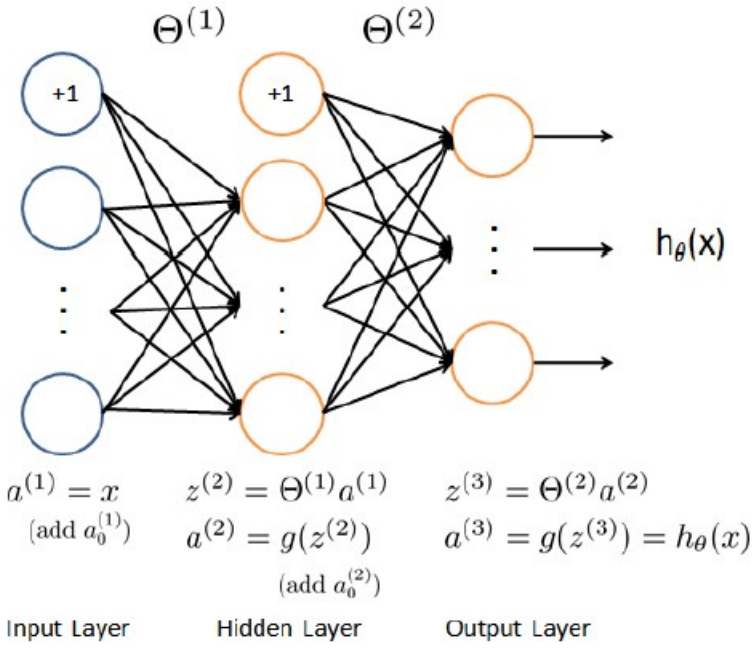

Figura 1: Modelo de NN a crear.

Tiene 3 capas: una capa de entrada, una capa oculta y una capa de salida. Recordemos que nuestras entradas tienen un tamaño de 400 unidades (excluyendo la unidad de sesgo adicional que siempre da como resultado +1) debido a que las imágenes tienen un tamaño de 20x20 píxeles.

La capa oculta tiene 25 unidades y la capa de salida 10 unidades (correspondientes a las 10 clases de dígitos).

Hay que tener en cuenta que para poder entrenar la red, la variable Y que contiene las clases conocidas, debe ser codificadas como un vector de 10 elementos, donde sólo uno de ellos puede valer 1. Esta codificación se denomina **One-Hot-Encoding** y debe realizarse sobre los valores reales de la clase (etiquetas, labels...), para poder compararlo directamente con la salida de la red.

Podéis utilizar **SKLearn** para codificar la variable como One-hot-encoding o hacerlo manualmente.

```
data = loadmat('data/ex3data1.mat', squeeze_me=True)
y = data['y']
X = data['X']
Y = One-Hot-Encoding(y)
print(y.shape) #(400,10)
```

Ejercicio 1 Compute cost:

La función de coste de la red neuronas es la siguiente:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_{k,i}log(y'_{k,i}) + (1 - y_{k,i})log(1 - y'_{k,i})]$$

Donde $y'_{k,i} = (h_0(x^{(i)})_k)$ en la figura 1 y representa la salida de la neurona K de la red e $y_{k,i}$ es el valor real para la neurona k (recordad, hemos codificado la salida con One-hot-encoding) para el ejemplo i.

Para comprobar que funciona, podemos utilizar los datos de las matrices del ejercicio anterior $\Theta(1)$ y $\Theta(2)$ ya entrenadas.

Completa la función cost para implementar el cálculo del coste. Debes implementar el cálculo feedforward que calcula h(x(i)) para cada ejemplo i y sumar el coste de todos los ejemplos. Tu código también debe funcionar para un conjunto de datos de cualquier tamaño, con cualquier número de etiquetas

```
def cost(theta1, theta2, X, y, lambda_):
    """
    Compute cost for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
```

```
    1-hot encoding of labels for the input, having shape
    (number of examples x number of labels).

lambda_ : float
    The regularization parameter.

Returns
-------
J : float
    The computed value for the cost function.

"""
return J
```

Puedes comprobar si tu implementación es correcta calculando el coste con los parámetros proporcionados en el archivo data/ex3weights.mat con la red pre-entrenada del ejercicio anterior. El resultado esperado de la función de coste con esos parámetros es aproximadamente 0,287629.

Coste con regularización L2:

Extiende el computo del coste con la regularización L2. Recordemos que la regularización L2 consiste en sumar al coste la suma de todos los parámetros de la red al cuadrado, multiplicado por $\frac{\lambda}{2m}$ donde λ es el parámetro de regularización y m el número de ejemplos:

$$JL2(\Theta) = J(\Theta) + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{j=1}^{Dim(l)} \sum_{i=1}^{Dim(l+1)} (\Theta_{i,j}^{l})^2$$

El código debe funcionar para cualquier número de capas ocultas, para cualquier numero de neuronas por capa y para cualquier número de salidas, pero en este caso concreto, para resolver el ejercicios la implementación es la siguiente:

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

Nota importante: El sesgo, umbral o bias de la red no debe normalizarse, por este motivo los valores sean 400 y 25 y no 401 y 26. No debemos procesar la primera columna de las matrices Theta 1 y 2. (i = 0 en la ecuación anterior).

Puedes comprobar si tu implementación es correcta calculando el coste con los parámetros proporcionados en el archivo data/ex3weights.mat. El resultado esperado de la función de coste regularizada con esos parámetros y = 1 es aproximadamente 0,383770.

Ejercicio 2 Backpropagation:

Implementar el algoritmo de retropropagación (backpropagation) para calcular el gradiente de la función de coste de la red neuronal. Primero implementa el algoritmo de retropropagación para calcular los gradientes de los parámetros de la red neuronal no regularizada. Después de haber verificado que el cálculo del gradiente para el caso no regularizado es correcto, implementa el gradiente para la red neuronal regularizada.

Completa la función backprop para implementar el cálculo del gradiente. Dado que la retropropagación requiere un primer paso de propagación hacia adelante, la función también devolverá el valor del coste.

```
def backprop(theta1, theta2, X, y, lambda_):
    """
    Compute cost and gradient for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

    lambda_ : float
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.

    grad1 : array_like
        Gradient of the cost function with respect to weights
        for the first layer in the neural network, theta1.
        It has shape (2nd hidden layer size x input size + 1)
```

```
    grad2 : array_like
        Gradient of the cost function with respect to weights
        for the second layer in the neural network, theta2.
        It has shape (output layer size x 2nd hidden layer size + 1)

    """
    return (J, grad1, grad2)
```

Puedes comprobar si tu implementación es correcta llamando a utils.checkNNGradients con tu función de retropropagación como primer parámetro. utils.checkNNGradients crea una pequeña red neuronal para comprobar los gradientes de retropropagación comparando el gradiente devuelto por tu función de retropropagación con una **aproximación numérica** (Puedes consultar en los apuntes **Numérical estimation of gradients** en el tema 3.2 si quieres recordar la base teórica de este método) que utiliza el coste también devuelto por su función de retropropagación (Hacer esto cuando hayamos comprobado que el coste es correcto). Estos dos cálculos de gradiente deberían dar como resultado valores muy similares.

Una vez que la versión no regularizada sea correcta, amplíe backprop para que devuelva el coste y el gradiente regularizados y compruebe de nuevo que es correcto utilizando utils.checkNNGradients.

Como comentamos en en clase, hay diferentes implementaciones del algoritmo backpropagantion. Todas optimizan los pesos, solo que unas van a más velocidad que otras. Para esta versión vamos a asumir la implementación de delta Last más simple, donde no calculamos la derivada de la función de activación.

$$\delta_j^L = (y'_j - y_j)$$

Si calculáis la derivada

$$\delta_j^L = (y'_j - y_j)g'(y'_j) = (a_j^L - y_j)g'(a_j^L)$$

Los valores en el check final serna mas altos.

A modo de recordatorio, la regla delta generalizada para todas las capas menos la ultima se mantiene igual que en la teoría:

$$\delta_i^{L-1} = \Theta_{i,j}^{L-1}\delta_j^L g'(a_i^{L-1})$$

$$g'(a_i^l) = a_i^l(1 - a_i^l)$$

Los gradientes se calculan de la siguiente forma:

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

Ejercicio 3 Aprendizaje de los parámetros:

Una vez comprobado que los gradientes funcionan, hay que aprender los parámetros de aprendizaje.

$$\Theta_{i,j}^{l} = \Theta_{i,j}^{l} - \alpha \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^{l}}$$

Repetir hasta convergencia:

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

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

Inicializacion:

Antes del entrenamiento, hay que inicializar aleatoriamente los parámetros de Theta 1 y 2. Una estrategia eficaz para inicialización es seleccionar aleatoriamente valores para theta uniformemente en el rango [-ϵ; ϵ]. Utiliza $\epsilon = 0{:}12$. Este rango de valores asegura que los parámetros se mantienen pequeños y hace que el aprendizaje sea más eficiente.

Puedes comprobar tu implementación calculando la precisión de entrenamiento de tu clasificador, es decir, el porcentaje de ejemplos de entrenamiento correctos. Si su implementación es correcta, debería obtener una precisión (accuracy) de entrenamiento de alrededor del 95% (puede variar en torno a un 1% debido a la inicialización aleatoria) después de ejecutar 1000 iteraciones de descenso de gradiente con $\lambda = 1$ y $\alpha = 1$.

Parte opcional: Aprendizaje de los parámetros usando métodos de optimización scipy.optimize.minimize

Utiliza ahora scipy.optimize.minimize para aprender los parámetros de la red neuronal. scipy.optimize.minimize minimiza una función escalar de una o más variables. En este caso la utilizaremos para encontrar los parámetros que minimizan el coste de la red neuronal. Usando el método 'TNC', $\lambda = 1$ y sólo 100 iteraciones deberías obtener de nuevo una precisión de entrenamiento de alrededor del 95%.

Las redes neuronales son modelos muy potentes que pueden formar límites de decisión muy complejos. Sin regularización, es posible que una red neuronal

"sobreajuste" un conjunto de entrenamiento de forma que obtenga una precisión cercana al 100% en el conjunto de entrenamiento pero no lo haga tan bien en nuevos ejemplos que no haya visto antes. Puedes ajustar la regularización λ a un valor menor y el parámetro **maxfun** a un mayor número de iteraciones para comprobarlo.

scipy.optimize.minimize.html

optionsdict, optional

A dictionary of solver options. All methods except TNC accept the following generic options:

- maxiter:int Maximum number of iterations to perform. Depending on the method each iteration may use several function evaluations. For TNC use maxfun instead of maxiter.
- disp: bool Set to True to print convergence messages

English version

In the previous exercise we implemented feedforward propagation of the neural network to predict handwritten digits. In this exercise we will implement backpropagation to learn the values of the network weights.

The network of nueronas will be the same as in the previous exercise (see Figure 1).

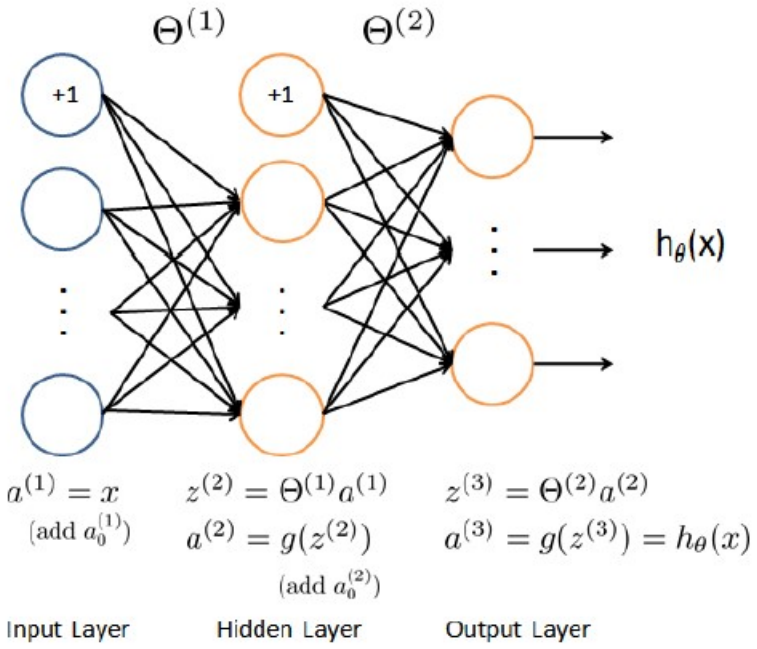

Figure 1: NN model to create

It has 3 layers: an input layer, a hidden layer and an output layer. Recall that our inputs have a size of 400 units (excluding the additional bias unit which always results in +1) because the images have a size of 20x20 pixels.

The hidden layer has 25 units and the output layer has 10 units (corresponding to the 10 digit classes).

It must be taken into account that in order to train the network, the Y variable containing the known classes or labels, must be encoded as a vector of 10 elements, where only one of them can be worth 1. This encoding is called **One-Hot-Encoding** and must be done on the real values of the labels, to be able to compare it directly with the network output.

You can use **SKLearn** to encode the variable as One-hot-encoding or do it manually.

```
data = loadmat('data/ex3data1.mat', squeeze_me=True)
y = data['y']
X = data['X']
Y = One-Hot-Encoding(y)
print(y.shape) #(400,10)
```

Exercise 1 Compute cost:

The equation for the cost function for neural networks is:

$$J(\Theta) = -\frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}[y_{k,i}log(y'_{k,i}) + (1 - y_{k,i})log(1 - y'_{k,i})]$$

Where $y'_{k,i} = (h_0(x^{(i)})_k)$ i8n figure 1 represent the output o neuron K and $y_{k,i}$ is the real value for output neuron k (recall: we have encoded the output with One-hot-encoding) for the example i.

Complete the cost function to implement the computation of the cost. You should implement the feedforward computation that computes h0(x(i)) for every example i and sum the cost over all examples. Your code should also work for a dataset of any size, with any number of labels.

```
def cost(theta1, theta2, X, y, lambda_):
    """
    Compute cost for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

    lambda_ : float
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.
```

```
    """
    return J
```

You can check if your implementation is correct by computing the cost with the parameters provided in the file data/ex3weights.mat. The expected output of the cost function with those parameters is about 0:287629.

Next you have to extend the implementation of cost to return the computation of regularized cost for neural networks, adding the regularization term to the previous function. Recall that the L2 regularisation consists of adding to the cost the sum of all network parameters squared, multiplied by $\frac{\lambda}{2m}$ where λ is the regularisation parameter and m is the number of examples:

$$JL2(\Theta) = J(\Theta) + \frac{\lambda}{2m} \sum_{l=1}^{L-1} \sum_{j=1}^{Dim(l)} \sum_{i=1}^{Dim(l+1)} (\Theta_{i,j}^{l})^2$$

Although your code should work for any number of input units, hidden units and outputs units, assuming that the neural network will only have 3 layers, in this particular case the regularization term is:

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

Important Note that you should not be regularizing the terms that correspond to the bias. For the matrices theta1 and theta2, this corresponds to the first column of each matrix (i = 0 in the equation above).

You can check if your implementation is correct by computing the cost with the parameters provided in the file data/ex3weights.mat. The expected output of the regularized cost function with those parameters and $\lambda = 1$ is about 0:383770.

Exercise 2 Backpropagation:

You will implement the backpropagation algorithm to compute the gradient for the neural network cost function. You will first implement the backpropagation algorithm to compute the gradients for the parameters for the unregularized neural network. After you have verified that your gradient computation for the unregularized case is correct, you will implement the gradient for the regularized neural network.

Complete the nn.backprop function to implement the computation of the gradient. Since the backpropagation requieres a first step of forward propagation, the function will also return the cost value with no additional computational cost:

```
def backprop(theta1, theta2, X, y, lambda_):
    """
```

```
    Compute cost and gradient for 2-layer neural network.

    Parameters
    ----------
    theta1 : array_like
        Weights for the first layer in the neural network.
        It has shape (2nd hidden layer size x input size + 1)

    theta2: array_like
        Weights for the second layer in the neural network.
        It has shape (output layer size x 2nd hidden layer size + 1)

    X : array_like
        The inputs having shape (number of examples x number of dimensions).

    y : array_like
        1-hot encoding of labels for the input, having shape
        (number of examples x number of labels).

    lambda_ : float
        The regularization parameter.

    Returns
    -------
    J : float
        The computed value for the cost function.

    grad1 : array_like
        Gradient of the cost function with respect to weights
        for the first layer in the neural network, theta1.
        It has shape (2nd hidden layer size x input size + 1)

    grad2 : array_like
        Gradient of the cost function with respect to weights
        for the second layer in the neural network, theta2.
        It has shape (output layer size x 2nd hidden layer size + 1)

    """
    return (J, grad1, grad2)
```

You can check if your implementation is correct by calling utils.checkNNGradients with your backpropagation function as first parameter. utils.checkNNGradients creates a small neural network to check the backpropagation gradients by comparing the gradient returned by your backpropagation function with a **numerical approximation** (You can refer to class material **Numérical estimation of gradients** in topic 3.2 if you want to recall the theoretical basis of this method).

that uses the cost also returned by your backpropagation function. These two gradient computations should result in very similar values.

Once the unregularized version is correct, extend nn.backprop to return regularized cost and gradient and check again that it is correct using utils checkNNGradients.

As we discussed in class, there are different implementations of the backpropagantion algorithm. All of them optimise the weights, only some go faster than others. For this version we will assume the simplest delta Last implementation, where we do not calculate the derivative of the activation function.

$$\delta_j^L = (y'_j - y_j)$$

if you calculate the derivative:

$$\delta_j^L = (y'_j - y_j)g'(y'_j) = (a_j^L - y_j)g'(a_j^L)$$

The values in the final check will be higher.

As a reminder, the delta rule generalized for all but the last layer remains the same as in theory:

$$\delta_i^{L-1} = \Theta_{i,j}^{L-1}\delta_j^L g'(a_i^{L-1})$$

$$g'(a_i^l) = a_i^l(1 - a_i^l)$$

The gradients are calculated as follows:

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

Exercise 3 Learning parameters:

Once it has been proven that gradients work, the learning parameters must be learned.

$$\Theta_{i,j}^l = \Theta_{i,j}^l - \alpha \frac{\partial J(\Theta)}{\partial \Theta_{i,j}^l}$$

Repeat until convergence:

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

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

Before training you will need to randomly initilize the parameters. One effective strategy for random initialization is to randomly select values for Theta uniformly in the range [-ϵ; ϵ]. You should use $\epsilon = 0{:}12$. This range of values ensures that the parameters are kept small and makes the learning more efficient.

You can check your implementation by computing the training accuracy of your classifier, i.e, the percentage of training examples it got correct. If your implementation is correct, you should see a reported training accuracy of about 95% (this may vary by about 1% due to the random initialization) after running 1000 iterations of gradient descent with $\lambda = 1$ and $\alpha = 1$.

Optional part: Learning parameters using optimisation methods scipy.optimise.minimise

Use now the scipy.optimize.minimize to learn the parameters for the neural network. scipy.optimize.minimize minimize a scalar function of one or more variables. In this case we will use it to find the parameters that minimize the cost of the neural network. Using the 'TNC' method, $\lambda = 1$ and just 100 iterations you should get again a training accuracy of about 95%.

Neural networks are very powerful models that can form highly complex decision boundaries. Without regularization, it is possible for a neural network to "overfit" a training set so that it obtains close to 100% accuracy on the training set but does not as well on new examples that it has not seen before. You can set the regularization λ to a smaller value and the **maxfun** parameter to a higher number of iterations to see this for yourself.

scipy.optimize.minimize.html

optionsdict, optional

A dictionary of solver options. All methods except TNC accept the following generic options:

- maxiter:int Maximum number of iterations to perform. Depending on the method each iteration may use several function evaluations. For TNC use maxfun instead of maxiter.
- disp: bool Set to True to print convergence messages.