Aprendizaje de ANN

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

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

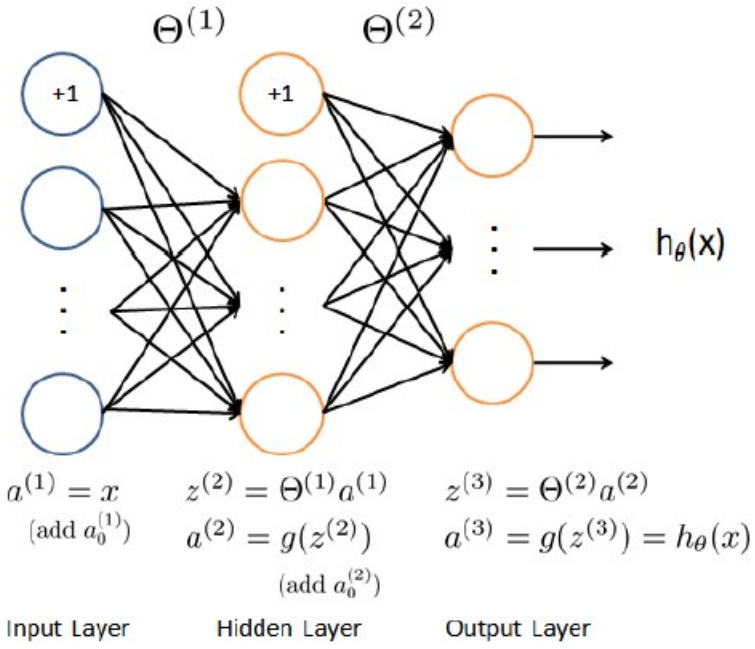

Figura 1: Modelo de NN a crear.

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.

Ejercicio 1 compute_cost:

Modifica el método compute_cost de la práctica anterior para añadirle la regularización L2

```
    """
    Computes only the cost of a previously generated output (private)

    Args:
        yPrime (array_like): output generated by neural network.
        y (array_like): output from the dataset
        lambda_ (scalar): regularization parameter

    Return
    ------
    J (scalar): the cost.
    """
    def compute_cost(self, yPrime,y, lambda_):
        ##TO-DO
        J = 0
        return J
```

Inicializacion:

Antes del entrenamiento, hay que inicializar aleatoriamente los parámetros de Theta 1 y 2 (en el constructor). 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.

Ejercicio 2 compute_gradients:

Implementar el cómputo de los gradientes de la retropropagación de la clase MLP. 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 compute_gradients 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.

```
    """
    Compute the gradients of both theta matrix parámeters and cost J

    Args:
```

```
        x (array_like): input of the neural network.
        y (array_like): output of the neural network.
        lambda_ (scalar): regularization.

    Return
    ------
    J: cost
    grad1, grad2: the gradient matrix
    (same shape than theta1 and theta2)
    """
    def compute_gradients(self, x, y, lambda_):
        ##TO-DO
        J,grad1,grad2 = 0
        return (J, grad1, grad2)
```

Puedes comprobar si tu implementación es correcta llamando a gradientTest que 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 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. El test ya comprueba que los resultados son correctos por lo que sólo hay que ejecutarlo para comprobarlo.

Una vez que la versión no regularizada sea correcta, amplía compute_gradients para que devuelva el coste y el gradiente regularizados y comprueba de nuevo que es correcto utilizando gradientTest. (Puedes observar que hay tres test, dos de ellos con lambda_ diferente de 0)

Ejercicio 3 Aprendizaje de los parámetros:

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

Para comprobar que funciona, utiliza el Test 2. Si estableces los mismos valores que indica el test deberías obtener en tu implmentació unos resultados muy similares a los que se marcan como esperados.

Ejercicio 4 Utiliza MLPClassifier de SKLearn

Para comprobar que tu implementación es correcta, utiliza el MLPClassifier de sklearn. Establece los parámetros de alpha y lambda igual que en tus pruebas y asegúrate que usas la función sigmoidal, así como el mismo número de iteraciones. Los resultados no serán iguales, ya que MLPClassifier utiliza ADAM como optimizador y además la inicialización es diferente pero el orden de magnitud debe ser similar al tu resultado.

Ejercicio 5 Generaliza para cualquier número de capas y neuronas por capa (opcional)

Una vez sepas que el perceptrón para 1 capa oculta funciona correctamente, créate una nueva clase denominada MLP_Complete y realiza los cambios necesarios para que la implementación anterior funcione para cualquier número de capas ocultas.

El constructor tendrá que cambiar de la siguiente forma:

```
"""
Constructor: Computes MLP_Complete.

Args:
    inputLayer (int): size of input
    hiddenLayers (array-like): number
    of layers and size of each layers.
    outputLayer (int): size of output layer
    seed (scalar): seed of the random numeric.
    epislom (scalar) : random initialization range.
    e.j: 1 = [-1..1], 2 = [-2,2]...
"""

def __init__(self,inputLayer,hiddenLayers, outputLayer,
 seed=0, epislom = 0.12):
```

Una llamada a este constructor nuevo podría ser la siguiente:

```
mlp_Complete = MLP_Complete(400,[100,50,25],10)
```

Esto indicaría que la red que estamos creando tiene 3 capas ocultas de 100,50 y 25 neuronas cada una.

Comprueba con SKLearn que la implementación es correcta.

Recordatorio de funciones:

Reglas deltas generalizadas, para la ultima capa:

$$\delta_j^L(k) = y_j\prime(k) - y_j(k)$$

Para el resto de capas ocultas:

$$\delta_i^l(k) = \sum_{j=1}^{size(a^{l+1})} (\Theta_{j,i}^l \delta_j^{l+1}(k) g'(a_i^l(k)))$$

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

Los gradientes se calculan de la siguiente forma:

Gradientes: Para cada peso i y j de la matriz Θ^l

$$\frac{\partial J(\Theta)}{\partial \Theta^l_{j,i}} = \frac{1}{m}\sum_{k=1}^{m} \delta_j^{l+1}(k) \cdot a_i^l(k)$$

Donde K es el ejemplo correspondiente, $a_i^l(k)$ es la activación de la neurona i de la capa l para el ejemplo k

Actualización de los pesos:

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

Regularización L2

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

El término **gradiente en la regularización** (derivada del anterior):

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

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 neural network will be the same as in the previous exercise (see Figure 1).

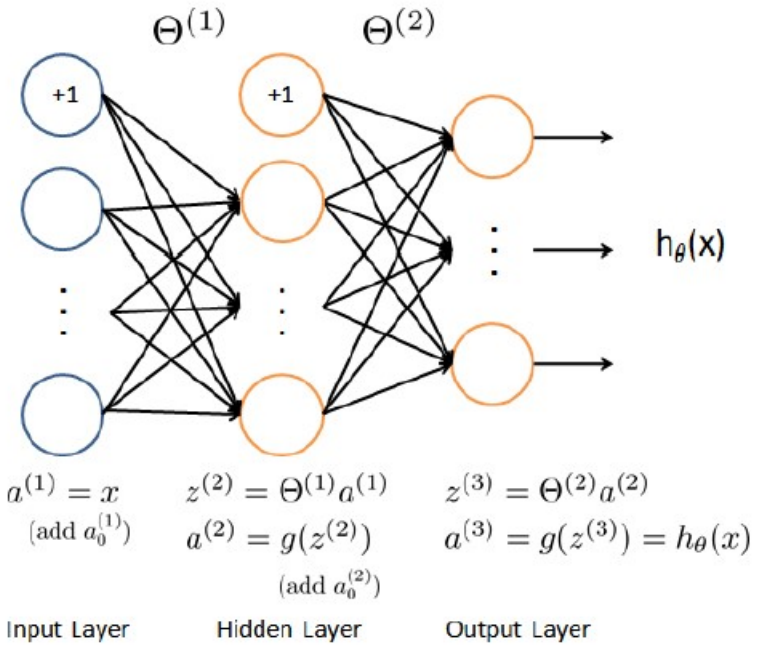

Figura 1: NN model to create.

It has 3 layers: an input layer, a hidden layer and an output layer. Remember that our inputs have a size of 400 units (excluding the additional bias unit which always results in +1) because the images are 20x20 pixels in size. The hidden layer has 25 units and the output layer has 10 units (corresponding to the 10 digit classes).

Note that in order to train the network, the Y variable containing the known classes, 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 performed on the actual values of the class (labels, 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.

Exercise 1 compute_cost:

Modify the compute_cost method from the previous practice to add the L2 regularization to it.

```
"""
Computes only the cost of a previously generated output (private)
```

```
    Args:
        yPrime (array_like): output generated by neural network.
        y (array_like): output from the dataset
        lambda_ (scalar): regularization parameter

    Return
    ------
    J (scalar): the cost.
    """
    def compute_cost(self, yPrime,y, lambda_):
        ##TO-DO
        J = 0
        return J
```

Initialization:

Prior to training, one must randomly initialize theta parameters 1 and 2 (in the constructor). An effective strategy for initialization is to randomly select values for theta uniformly in the range [-ϵ; ϵ]. Use $\epsilon = 0.12$. This range of values ensures that the parameters are kept small and makes learning more efficient.

Exercise 2 compute_gradients:

Implement the compute gradients backpropagation of the MLP class. First implement the backpropagation algorithm to compute the gradients of the unregularized neural network parameters. After verifying that the gradient computation for the unregularized case is correct, implement the gradient for the regularized neural network.

Complete the compute_gradients function to implement the gradient calculation. Since backpropagation requires a first step of forward propagation, the function will also return the cost value.

```
    """
    Compute the gradients of both theta matrix parámeters and cost J

    Args:
        x (array_like): input of the neural network.
        y (array_like): output of the neural network.
        lambda_ (scalar): regularization.

    Return
    ------
    J: cost
    grad1, grad2: the gradient matrix
    (same shape than theta1 and theta2)
    """
    def compute_gradients(self, x, y, lambda_):
```

```
        ##TO-DO
        J,grad1,grad2 = 0
        return (J, grad1, grad2)
```

You can check if your implementation is correct by calling gradientTest which creates a small neural network to check the backpropagation gradients by comparing the gradient returned by your backpropagation function with a **numerical estimation** (You can refer to the notes **Numerical estimation of gradients** in Chapter 3 if you want to remember the theoretical base of this method) which uses the cost also returned by your backpropagation function (Do this when we have checked that the cost is correct). These two gradient calculations should result in very similar values. The test already checks that the results are correct so just run it to verify.

Once the unregularized version is correct, extend compute_gradients to return the regularized cost and gradient and check again that it is correct using gradientTest. (You can see that there are three tests, two of them with lambda_ different from 0)

Exercise 3 Parameter learning:

Once you have verified that the gradients work, you have to learn the learning parameters of the theta matrices. To check that it works, use Test 2. If you set the same values as indicated in the test, you should obtain in your implementation results very similar to the expected ones.

Exercise 4 Use MLPClassifier from SKLearn

To check that your implementation is correct, use MLPClassifier of sklearn. Set the alpha and lambda parameters the same as in your tests and make sure you use the sigmoidal function, as well as the same number of iterations.

The results will not be the same, since MLPClassifier uses ADAM as an optimizer and also the initialization is different but the order of magnitude should be similar to your result.

Exercise 5 Generalize for any number of layers and neurons per layer (optional).

Once you know that the perceptron for 1 hidden layer works correctly, create a new class named MLP_Complete and make the necessary changes to make the above implementation work for any number of hidden layers. The constructor will have to be changed as follows:

```
    """
    Constructor: Computes MLP_Complete.

    Args:
```

```
        inputLayer (int): size of input
        hiddenLayers (array-like): number
        of layers and size of each layers.
        outputLayer (int): size of output layer
        seed (scalar): seed of the random numeric.
        epislom (scalar) : random initialization range.
        e.j: 1 = [-1..1], 2 = [-2,2]...
    """

    def __init__(self,inputLayer,hiddenLayers, outputLayer,
     seed=0, epislom = 0.12):
```

The invocation of the new constructor could look like this:

```
mlp_Complete = MLP_Complete(400,[100,50,25],10)
```

This would indicate that the network we are creating has 3 hidden layers of 100,50 and 25 neurons each.

Check with SKLearn that the implementation is correct.

Reminder of functions:

Generalized delta rules, for the last layer:

$$\delta_j^L(k) = y_j\prime(k) - y_j(k)$$

For the rest of the layers:

$$\delta_i^l(k) = \sum_{j=1}^{size(a^{l+1})} (\Theta_{j,i}^l \delta_j^{l+1}(k) g'(a_i^l(k)))$$

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

The gradients are calculated as follow:

GRadients: For each weight i and j of the matrix Θ^l

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

Where K is the corresponding example $a_i^l(k)$ is the activation of the neuron i in the layer l for the example k

Weights update:

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

Regularization L2

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

The term **gradient in the regularization** (derived from the previous one):

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