Forward propagation ANN

En esta práctica vamos a implementar una red de neuronas simple. Para ello vamos a leer un dataset de 5000 imágenes de entrenamiento de dígitos escritos a mano. Están en formato Octave/Matlab y para leerlo necesitamos usar scipy.io.loadmat [1].

Para hacerlo, podemos utilizar la función **load_data** de **utils**

Cada ejemplo de entrenamiento es una imagen en escala de grises de 20x20 píxeles . Cada píxel está representado por un número de coma flotante que indica la intensidad de la escala de grises en ese lugar. La cuadrícula de 20 por 20 píxeles está aplanada en forma de vector de 400 dimensiones. Esto nos da una matriz X de 5000 por 400 en la que cada fila es una imagen manuscrita.

La segunda parte del conjunto de entrenamiento es un vector de tamaño 5000 y que contiene etiquetas para el conjunto de entrenamiento etiquetadas de 0 a 9.

Figura 1: Ejemplo del dataset de entrenamiento.

Con estos datos de entrenamiento vamos a implementar una red de neuronas, utilizando los parámetros de una red ya entrenada.

El objetivo de esta parte es programar el algoritmo de feedforward (fordward

[1]This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/).

propagation), utilizando los pesos en la predicción. La red se muestra en la Figura 2.

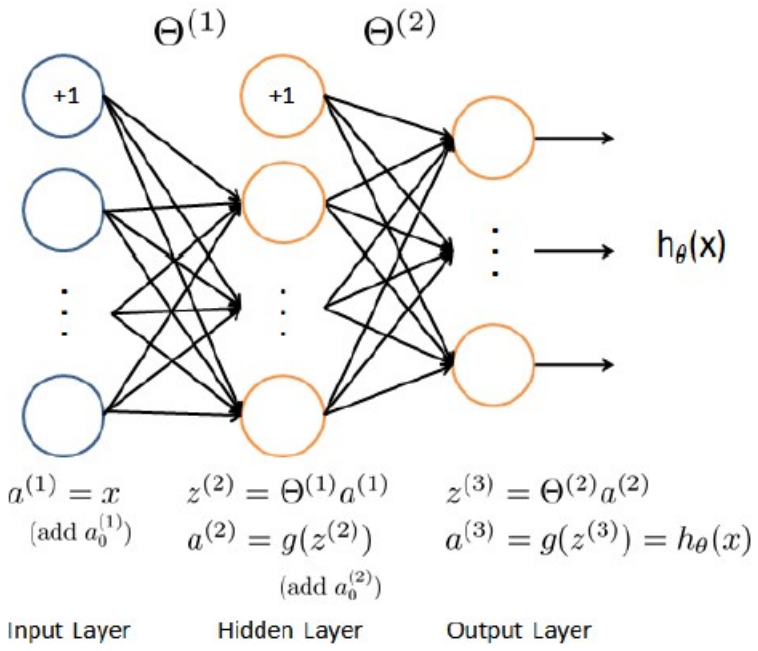

Figura 2: Modelo de NN a crear.

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).

def load_weights(file): Los parámetros pre-entrenados tienen 25 unidades en la segunda capa y 10 unidades de salida (correspondientes a las 10 clases de dígitos). Estos parámetros se pueden leer usando la función **load_weights** del fichero **utils**.

Ejercicio 1:

Completa el código MLP para obtener la predicción de la red neuronal. Debes implementar el cálculo feedforward que calcula $h(x(i))$ para cada ejemplo i y devuelve las predicciones asociadas. La predicción de la red neuronal será la etiqueta que tenga la mayor salida $(h(x))_k$.

Sugerencia: Para obtener una solución vectorizada del problema, es conveniente añadir una columna de 1s a la matriz X antes de calcular la propagación de la red neuronal (simula los valores de los umbrales que recordemos pueden modelarse

como una entrada siempre asignada a 1) usando la función **np.hstack** que permite apilar matrices por columnas.

https://numpy.org/doc/stable/reference/generated/numpy.hstack.html

```
X1s = np.hstack([np.ones((m, 1)), X])
```

```
class MLP:
    """
    Constructor: Computes MLP.

    Args:
        theta1 (array_like): Weights for the first layer in the neural network.
        theta2 (array_like): Weights for the second layer in the neural network.
    """
    def __init__(self,theta1,theta2):
        self.theta1 = theta1
        self.theta2 = theta2

    """
    Num elements in the training data. (private)

    Args:
        x (array_like): input data.
    """
    def _size(self,x):
        return x.shape[0]

    """
    Computes de sigmoid function of z (private)

    Args:
        z (array_like): activation signal received by the layer.
    """
    def _sigmoid(self,z):
        return 0

    """
    Run the feedwordwar neural network step

    Args:
        z (array_like): activation signal received by the layer.

    Return
    ------
    a1,a2,a3 (array_like): activation functions of each layers
    z2,z3 (array_like): signal fuction of two last layers
```

```
    """
    def feedforward(self,x):
        a1,a2,a3,z2,z3 = 0
        return a1,a2,a3,z2,z3 # devolvemos a parte de las activaciones, los valores sin eje

    """
    Get the class with highest activation value

    Args:
        a3 (array_like): output generated by neural network.

    Return
    ------
    p (scalar): the class index with the highest activation value.
    """
    def predict(self,a3):
        p = -1
        return p
```

Ejecuta el test **predict_test** en el fichero **public_test** para comprobar que el resultado es correcto. Para ello necesitas implmentar la función **accuracy** del fichero **utils** que debe dar un resultado cercano a 97.52%.

Ejercicio 2:

Calcula el coste implementando la función compute_cost.

```
    """
    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

    Return
    ------
    J (scalar): the cost.
    """
    def compute_cost(self, yPrime,y): # calcula solo el coste, para no ejecutar nuevamente
        J = 0
        return J
```

Ejecuta el test **compute_cost_test** para comprobar que es correcto. Para ello necesitas implementar la función **one_hot_encoding** del fichero **utils**.

Ejercicios 3:

Calcula la matriz de confusión para la predicción del 0. Es decir, asumimos que la clase positiva es 0 y el resto de clases son negativas. Calcula tambien precission, recall y F1-Score.

NOTA: En utils disponéis de una función que permite mostrar las imágenes que podéis utilizar.

English version

In this practice we are going to implement a simple neural network. For this we are going to read a dataset of 5000 training images of handwritten digits. They are in Octave/Matlab format and we need to use scipy.io.loadmat[2] to read it.

To do this, we can use the **load_data** function of **utils**.

Each training example is a grayscale image of 20x20 pixels . Each pixel is represented by a floating point number indicating the grayscale intensity at that location. The 20 by 20 pixel grid is flattened into a 400 dimensional vector shape. This gives us a 5000 by 400 matrix X in which each row is a handwritten image. The second part of the training set is a vector of size 5000 and containing labels for the training set labeled 0 through 9.

[2]This is a subset of the MNIST handwritten digit dataset (http://yann.lecun.com/exdb/mnist/).

Figure 1: Example of the training dataset.

With this training data we are going to implement a neural network, using the parameters of an already trained network.

The aim of this part is to implement the feedforward propagation algorithm, using the weights in the prediction. The network is shown in Figure 2.

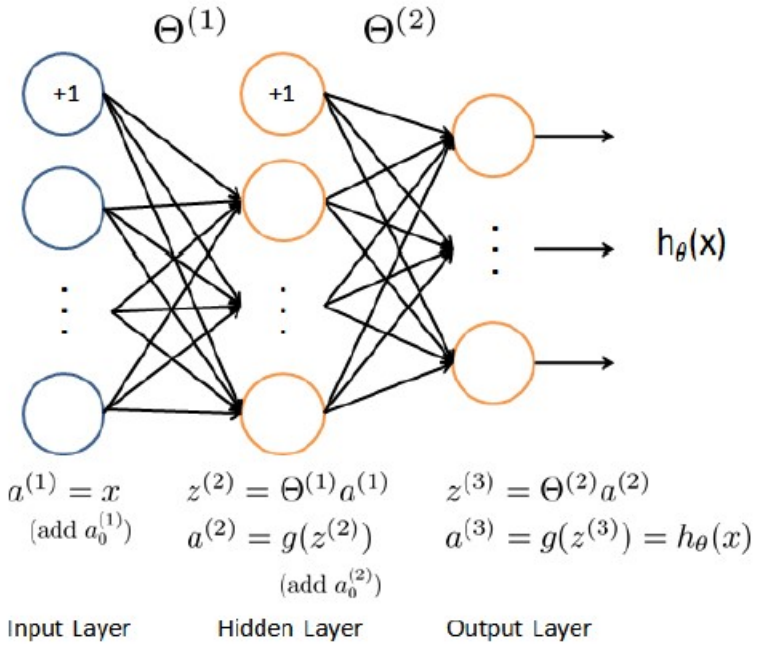

Figure 2: NN model to be created.

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). def load_weights(file): The pre-entered parameters have 25 units in the second layer and 10 output units (corresponding to the 10 digit classes). These parameters can be read using the **load_weights** function of the **utils** file.

Exercise 1:

Complete the MLP code to obtain the neural network prediction. You must implement feedforward computation that computes $h(x(i))$ for each example i and returns the associated predictions. The neural network prediction will be the label with the highest output $(h(x))_k$. Hint: To obtain a vectorized solution to the problem, it is convenient to add a column of 1s to the matrix X before

calculating the neural network propagation (simulates the threshold values which we recall can be modeled as an input always assigned to 1) using the **np.hstack** function which allows to stack matrices by columns.

https://numpy.org/doc/stable/reference/generated/numpy.hstack.html

```
X1s = np.hstack([np.ones((m, 1)), X])
```

```
class MLP:
    """
    Constructor: Computes MLP.

    Args:
        theta1 (array_like): Weights for the first layer in the neural network.
        theta2 (array_like): Weights for the second layer in the neural network.
    """
    def __init__(self,theta1,theta2):
        self.theta1 = theta1
        self.theta2 = theta2

    """
    Num elements in the training data. (private)

    Args:
        x (array_like): input data.
    """
    def _size(self,x):
        return x.shape[0]

    """
    Computes de sigmoid function of z (private)

    Args:
        z (array_like): activation signal received by the layer.
    """
    def _sigmoid(self,z):
        return 0

    """
    Run the feedwordwar neural network step

    Args:
        z (array_like): activation signal received by the layer.

    Return
    ------
```

```
    a1,a2,a3 (array_like): activation functions of each layers
    z2,z3 (array_like): signal fuction of two last layers
    """
    def feedforward(self,x):
        a1,a2,a3,z2,z3 = 0
        return a1,a2,a3,z2,z3 # devolvemos a parte de las activaciones, los valores sin eje

    """
    Get the class with highest activation value

    Args:
        a3 (array_like): output generated by neural network.

    Return
    ------
    p (scalar): the class index with the highest activation value.
    """
    def predict(self,a3):
        p = -1
        return p
```

Run the **predict_test** test on the **public_test** file to check that the result is correct. For this you need to implement the **accuracy** function of the **utils** file which should give a result close to 97.52%.

Exercise 2:

Calculate the cost by implementing the compute_cost function.

```
    """
    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

    Return
    ------
    J (scalar): the cost.
    """
    def compute_cost(self, yPrime,y): # calcula solo el coste, para no ejecutar nuevamente
        J = 0
        return J
```

Run the **compute_cost_test** test to check that it is correct. For this you need to implement the **one_hot_encoding** function from the **utils** file.

Exercises 3:

Calculate the confusion matrix for the prediction of 0. That is, we assume that the positive class is 0 and all other classes are negative. Calculate also precission, recall and F1-Score. NOTE: In utils you have a function that allows you to display the images you can use.