***Practica 4***

*Entrenamiento de redes neuronales*

Aaron Reboredo Vázquez, Pablo Martín García

Previo a la implementación de las funciones necesarias para llevar a cabo la práctica es interesante tener métodos que nos permitan cargar los datos de archivos externos y otros que nos permitan visualizar en pantalla una primera representación de esos valores.

Utilizaremos los métodos **load\_data** para volcar en variables los datos obtenidos de los diferentes archivos para las partes uno y dos.

def load\_data(file\_name):

#Carga los valores de entrada y salida de una red neuronal

    data = loadmat(file\_name)

    y = data['y']

    y\_2 = np.ravel(y)

    X = data['X']

    return y\_2, X

def load\_wwights\_neuronal\_red(file\_name):

#Carga los parámetros de una red neuronal

    weights = loadmat(file\_name)

    theta1 , theta2 = weights ['Theta1'] , weights ['Theta2']

    return theta1, theta2

Es interesante también en primera instancia tener métodos que nos permitan mostrar en pantalla de manera visual los valores con los que vamos a trabajar, en este caso imágenes de números.

Las siguientes funciones son las que nos permiten generar e imprimir en pantalla una imagen a través de los

def displayData(X):

#Convierte los valores de entrada en un elemento representable pro una imagen

    num\_plots = int(np.size(X, 0)\*\*.5)

    fig, ax = plt.subplots(num\_plots, num\_plots, sharex=True, sharey=True)

    plt.subplots\_adjust(left=0, wspace=0, hspace=0)

    img\_num = 0

    for i in range(num\_plots):

        for j in range(num\_plots):

            # Convert column vector into 20x20 pixel matrix

            # transpose

            img = X[img\_num, :].reshape(20, 20).T

            ax[i][j].imshow(img, cmap='Greys')

            ax[i][j].set\_axis\_off()

            img\_num += 1

    return (fig, ax)

def displayImage(im):#Nos premite imprimir una imagen

    fig2, ax2 = plt.subplots()

    image = im.reshape(20, 20).T

    ax2.imshow(image, cmap='gray')

    return (fig2, ax2)

Para la parte final de la práctica y cuando tratemos el entrenamiento de la red neuronal sustituiremos los valores de pesos por defecto dados por un par de arrays de pesos generados de manera aleatoria. Para ello haremos uso de la función **generate\_Random\_Weights,** que nos permite generar un array con elementos dentro de un rango y con las dimensiones correspondientes para la capa de la red neuronal con entrada L\_in y salida L\_out.

def generate\_Random\_Weights(L\_in, L\_out):

#Genera un array de pesos para una capa de una red neuronal con entrada L\_in y salida L\_out

    e\_ini = math.sqrt(6)/math.sqrt(L\_in + L\_out)

    e\_ini= 0.12

    weights = np.zeros((L\_out, 1 + L\_in))

    for i in range(L\_out):

        for j in range(1 + L\_in):

            rnd = random.uniform(-e\_ini, e\_ini)

            weights[i,j] = rnd

    return weights

De la misma manera utilizaremos un par de métodos auxiliares que nos facilitarán trabajar con los métodos de implementación de la red neuronal.

El primero nos permite desplegar en un vector otros dos:

def unrollVect(a, b): #nos permite desplegar en un vector otros dos

    thetaVec\_ = np.concatenate((np.ravel(a), np.ravel(b)))

    return thetaVec\_

El segundo nos convierte la salida de nuestra base de datos en una matriz gestionable por nuestros métodos vectorizados que implementan la red neuronal:

def y\_onehot(y, X, num\_etiquetas):

    #Devuelve la salida en forma de matriz lista para ser utilizada por nuestros métodos de la red neuronal

    m = X.shape[0]

    y = (y - 1)

    y\_onehot = np.zeros((m, num\_etiquetas))  # 5000 x 10

    for i in range(m):

        y\_onehot[i][y[i]] = 1

    return y\_onehot

**Red neuronal:**

**Función de coste:**

Para nuestra práctica implementaremos dos versiones de la función de coste que nos permitirán generar nuestra red neuronal. Estamos hablando de sus versiones básicas y su versión regularizada que tienen el siguiente aspecto:

def cost(params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje):

    m = X.shape[0]

    X = np.matrix(X)

    y = np.matrix(y)

    theta1 = np.matrix(np.reshape(params[:num\_ocultas

\* (num\_entradas + 1)], (num\_ocultas, (num\_entradas + 1))))

    theta2 = np.matrix(np.reshape(params[num\_ocultas

\* (num\_entradas + 1):], (num\_etiquetas, (num\_ocultas + 1))))

    h = forward(X, theta1, theta2)[4]

    J = (np.multiply(y, np.log(h)) - np.multiply((1 - y),

np.log(1 - h))).sum() / m

    return J, theta1, theta2

def cost\_Regularized(params, num\_entradas, num\_ocultas, num\_etiquetas, X, y, tasa\_aprendizaje):

    m = X.shape[0]

    J\_, theta1, theta2 =

cost(params, num\_entradas, num\_ocultas, num\_etiquetas,

X, y, tasa\_aprendizaje)

    J\_regularized =  J\_ + (float(tasa\_aprendizaje) /

            (2 \* m)) \* (np.sum(np.power(theta1[:, 1:], 2))

+ np.sum(np.power(theta2[:, 1:], 2)))

    return J\_regularized

Podemos comprobar que en el cálculo del coste y para el cálculo de h se hace uso de una función **forward** que representa la pasada hacia adelante que constituye el primer paso para la implementación de nuestra red neuronal.

La función tiene este aspecto :

def forward(X, theta1, theta2):

#Método pasada hacia adelante para la implementación de la red neuronal

#Nos devuelve los parámetros de activación de la red neuronal y el valor h

    m = X.shape[0]

    a1 = np.insert(X, 0, values=np.ones(m), axis=1)

    z2 = a1 \* theta1.T

    a2 = np.insert(sigmoid(z2), 0, values=np.ones(m), axis=1)

    z3 = a2 \* theta2.T

    h = sigmoid(z3)

    return a1, z2, a2, z3, h

Tras realizar las comprobaciones propuestas por el guion de la práctica podemos concluir que los métodos están correctamente implementados.

**Cálculo de gradiente:**

El siguiente paso es el cálculo del gradiente, también en sus dos versiones.

El cálculo del gradiente pasa por la implementación de la segunda función fundamental en la implementación de una red neuronal, la retro propagación o nuestra función **backprop.**