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 representando la población de datos.

```
def displayData(X):
#Convierte los valores de entrada en un elemento representable pro una im
agen
    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 n
uestros 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
```

<u>Red neuronal:</u>

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_apr
endizaje):
    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.**

```
def backprop(params, num_entradas, num_ocultas, num_etiquetas, X, y, tasa
_aprendizaje, regularize = True):
#Pasada hacia adelante y hacia atrás en nuestra red neuronal, nos calcula
 el gradiente y el coste correspondientes a nuestra red neuronal
    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))))

    a1, z2, a2, z3, h = forward(X, theta1, theta2)

    if regularize:
        delta1, delta2 = backProp_Deltas_regularized(a1, z2, a2, z3, h, t
heta1, theta2, y, m, tasa_aprendizaje)
    else :
        delta1, delta2 = backProp_Deltas(a1, z2, a2, z3, h, theta1, theta
2, y, m)

    J = cost_Regularized(params, num_entradas, num_ocultas, num_etiquetas
, X, y, tasa_aprendizaje)

    grad = unrollVect(delta1, delta2)

    return J, grad
```

En este caso hemos implementado en un mismo método las versiones normalizadas y básicas de el cálculo del gradiente o de la función de retro propagación mediante el uso de un booleano que nos permite calcular los deltas o gradientes de la propagación hacia atrás en su versión básica o regularizada.

Las funciones de las que hablamos son **backProp_Deltas** y **backProp_Deltas_regularized**, que nos permiten calcular los valores de gradiente correspondientes para cada capa:

```
def backProp_Deltas(a1, z2, a2, z3, h, theta1, theta2, y, m):

    delta1 = np.zeros(theta1.shape)
    delta2 = np.zeros(theta2.shape)

    d3 = h - y

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

    d2 = np.multiply((theta2.T * d3.T).T, sigmoid_Gradient(z2))

    delta1 += (d2[:, 1:]).T * a1
    delta2 += d3.T * a2

    delta1 = delta1 / m
    delta2 = delta2 / m

    return delta1, delta2
```

```
def backProp_Deltas_regularized(a1, z2, a2, z3, h, theta1, theta2, y, m,
tasa_aprendizaje):

    delta1, delta2 =
backProp_Deltas(a1, z2, a2, z3, h, theta1, theta2, y, m)

    delta1[:, 1:] =
delta1[:, 1:] + (theta1[:, 1:] * tasa_aprendizaje) / m

    delta2[:, 1:] =
delta2[:, 1:] + (theta2[:, 1:] * tasa_aprendizaje) / m

    return delta1, delta2
```

El método en su versión regularizada es una ampliación de la versión básica con el añadido del término de regularización correspondiente.

Para ambos casos se nos pide que hagamos una comprobación de la correcta implementación de los métodos correspondientes. Para ello usamos unos métodos facilitados por el profesor y que nos permiten comprobar mediante la creación de una pequeña red neuronal si los cálculos elaborados usando nuestros métodos se aproximan a los esperados. Las funciones son las siguientes:

```
def debugInitializeWeights(fan_in, fan_out):
    """
    Initializes the weights of a layer with fan_in incoming connections a
nd
    fan_out outgoing connections using a fixed set of values.
    """

    # Set W to zero matrix
    W = np.zeros((fan_out, fan_in + 1))

    # Initialize W using "sin". This ensures that W is always of the same
    # values and will be useful in debugging.
    W = np.array([np.sin(w) for w in
                  range(np.size(W))]).reshape((np.size(W, 0), np.size(W,
1)))

    return W


def computeNumericalGradient(J, theta):
    """
    Computes the gradient of J around theta using finite differences and
    yields a numerical estimate of the gradient.
    """

    numgrad = np.zeros_like(theta)
    perturb = np.zeros_like(theta)
    tol = 1e-4

    for p in range(len(theta)):
        # Set perturbation vector
        perturb[p] = tol
        loss1 = J(theta - perturb)
        loss2 = J(theta + perturb)

        # Compute numerical gradient
        numgrad[p] = (loss2 - loss1) / (2 * tol)
        perturb[p] = 0

    return numgrad
```

```
def checkNNGradients(costNN, reg_param):
    """
    Creates a small neural network to check the back propogation gradient
s.
    Outputs the analytical gradients produced by the back prop code and t
he
    numerical gradients computed using the computeNumericalGradient funct
ion.
    These should result in very similar values.
    """
    # Set up small NN
    input_layer_size = 3
    hidden_layer_size = 5
    num_labels = 3
    m = 5

    # Generate some random test data
    Theta1 = debugInitializeWeights(hidden_layer_size, input_layer_size)
    Theta2 = debugInitializeWeights(num_labels, hidden_layer_size)

    # Reusing debugInitializeWeights to get random X
    X = debugInitializeWeights(input_layer_size - 1, m)

    # Set each element of y to be in [0,num_labels]
    y = [(i % num_labels) for i in range(m)]

    ys = np.zeros((m, num_labels))
    for i in range(m):
        ys[i, y[i]] = 1

    # Unroll parameters
    nn_params = np.append(Theta1, Theta2).reshape(-1)

    # Compute Cost
    cost, grad = costNN(nn_params,
                        input_layer_size,
                        hidden_layer_size,
                        num_labels,
                        X, ys, reg_param)
```

```
    def reduced_cost_func(p):
        """ Cheaply decorated nnCostFunction """
        return costNN(p, input_layer_size, hidden_layer_size, num_labels,
                      X, ys, reg_param)[0]

    numgrad = computeNumericalGradient(reduced_cost_func, nn_params)

    # Check two gradients
    np.testing.assert_almost_equal(grad, numgrad)
    return (grad - numgrad)
```

Tras lanzar los métodos podemos comprobar como las predicciones que se nos facilitan en el guion de la práctica coinciden con nuestros resultados, dándonos a entender que el método programado se acerca o es el correcto.

Aprendizaje de los parámetros:

Para entrenar la red neuronal utilizaremos la función minimize utilizando como parámetro de entrada la función backprop, arrays de pesos aleatorios generados por nuestra función **generate_Random_Weights** y los valores correspondientes a en número de entradas y salidas y elementos en la capa oculta, así como la tasa de aprendizaje.

```
def minimice(backprop, params, num_entradas, num_ocultas, num_etiquetas,
X, y, tasa_aprendizaje):
    #Calcula los parámetros optimos de pesos para nuestra red neuronal
    fmin = opt.minimize(fun=backprop, x0=params, args=(num_entradas, num_
ocultas, num_etiquetas, X, y, tasa_aprendizaje),
    method='TNC', jac=True, options={'maxiter': 70})

    result = fmin.x
    return result
```

Esta función nos devuelve un array que contiene los pesos óptimos o pesos mínimos de nuestra red neuronal en función del número de iteraciones “maxiter” y del valor de la tasa de aprendizaje. A partir de aquí podemos volcarnos en el cálculo de la tasa de acierto de nuestra red neuronal para los pesos óptimos obtenidos.

Para ello necesitaremos una serie de métodos y métodos auxiliares que nos permitan hacer el cálculo de ese porcentaje, que en nuestro caso será el grado de similitud entre los valores de salida dados y los valores de salida predichos por la red neuronal implementada.

Para llevar a cabo este proceso primero tenemos que convertir nuestro array de pesos en los arrays de pesos correspondientes a cada capa de la red neuronal. Para ello plegamos el vector de pesos en otros dos mediante la función ***rollVector***

```
def rollVector(params, num_entradas, num_ocultas, num_etiquetas):
    #pliega el vector params en dos vectores de parámetros correspondinet
es a los vectores de pesos de cada una de las capas de nuestra red
    vector1 = np.matrix(np.reshape(params[:num_ocultas * (num_entradas +
1)], (num_ocultas, (num_entradas + 1))))
    vector2 = np.matrix(np.reshape(params[num_ocultas * (num_entradas + 1
):], (num_etiquetas, (num_ocultas + 1))))

    return vector1, vector2
```

El método principal será el que hemos llamado ***neuronal_succes_percentage:***

```
def neuronal_succes_percentage(X, y, weights1, weights2) :
    #Compara los valores predichos por la red neuronal para unos
    sigmoids_matrix = forward(X, weights1, weights2)[4]
    y_ = neuronal_prediction_vector(sigmoids_matrix)
    percentage = vectors_coincidence_percentage(y_, y)

    return percentage
```

Para el cálculo de la matriz de sigmoides utilizamos el método forward anteriormente implementado y lo procesamos mediante el método ***neuronal_prediction_vector*** para calcular el array de predicciones de la red neuronal para la entrada dada.

```
def neuronal_prediction_vector(sigmoids_matrix) :
    #calcula los valores predichos por la red neuronal dada una matriz de
 sigmoides
    # Será generada por la funcón forward.

    samples = sigmoids_matrix.shape[0]
    y = np.zeros(samples)

    for i in range(samples):
        y[i] = np.argmax(sigmoids_matrix[i, :]) +1

    return y
```

Finalmente tenemos un método auxiliar que simplemente nos devuelve el porcentaje de coincidencia entre dos vectores dados, y que, en este caso, introduciendo los vectores de salidas reales y el predicho por la red neuronal nos dará como resultado el porcentaje de acierto de nuestra red neuronal.

```
def vectors_coincidence_percentage(a, b):
    #Calcula el porcentaje de coincidencia dados dos vectores a, b
    coincidences_array = a == b

    coincidences = sum(map(lambda coincidences_array : coincidences_array
 == True, coincidences_array  ))
    percentage =100 * coincidences/coincidences_array.shape

    return percentage
```

De esta manera nuestro main() tiene el siguiente aspecto :

```
def main():

    X, y = load_data("ex4data1.mat")

    tasa_aprendizaje = 1
    num_etiquetas = 10 #num_etiquetas = num_salidas
    num_entradas = 400
    num_ocultas = 25

    theta1 = generate_Random_Weights(num_entradas, num_ocultas)
    theta2 = generate_Random_Weights(num_ocultas, num_etiquetas)

    params_rn = unrollVect(theta1, theta2)
    y_ = y_onehot(y, X, num_etiquetas)

    params_optimiced = minimice(backprop, params_rn, num_entradas, num_oc
ultas, num_etiquetas, X, y_, tasa_aprendizaje)

    theta1_optimiced, theta2_optimiced = rollVector(params_optimiced, num
_entradas, num_ocultas, num_etiquetas)

    percentage = neuronal_succes_percentage(X, y, theta1_optimiced, theta
2_optimiced)

    print("Percentage neuronal red : ", percentage)
```

El resultado obtenido tras lanzar el programa varias veces (nótese que generamos los pesos de manera aleatoria para el cálculo de la variable *params* y que nos ayuda a entrenar nuestra red neuronal) los resultados que obtenemos rondan el 93% de acierto, dejando patente que nuestra red neuronal es bastante fiable a la hora de predecir valores para la entrada dada.