***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 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.**

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, theta1, theta2, y, m, tasa\_aprendizaje)

    else :

        delta1, delta2 = backProp\_Deltas(a1, z2, a2, z3, h, theta1, theta2, 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 and

    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 gradients.

    Outputs the analytical gradients produced by the back prop code and the

    numerical gradients computed using the computeNumericalGradient function.

    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 correspondinetes 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\_ocultas, 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, theta2\_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.