Práctica 4

Alejandro Aizel Boto
Miguel Robledo Blanco

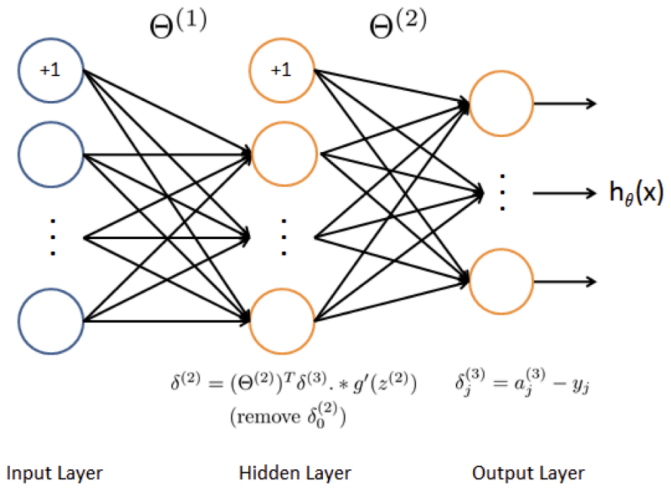
Índice

1. Descripción General

Esta práctica consta de 3 partes. En la primera vamos a calcular la función coste, en la segunda vamos a calcular el gradiente y en la última vamos a realizar el entrenamiento.

Seguimos con el mismo caso que en la práctica anterior, tenemos una serie de dígitos que tenemos que identificar y vamos a crear una red neuronal para resolver el problema.

La red neuronal que vamos a utilizar tiene la siguiente forma:



Como vemos es una red neuronal de 3 capas. En este caso la primera capa va a disponer de 400 entradas, la segunda capa 25 y en la última 10 que sería la de salida.

2. Explicación de la Solución

En este caso vamos a comentar el desarrollo de la práctica sin la división de apartados ya que consideramos que resulta más cómodo entender la práctica por el total.

Comenzamos con la llamada al método `main()` que realiza la lectura de los datos y los trata.

```
def main():
    data = loadmat('Práctica 4/Recursos/ex4data1.mat')
    weights = loadmat('Práctica 4/Recursos/ex4weights.mat')

    theta1_sol, theta2_sol = weights['Theta1'], weights['Theta2']

    X = data['X']
    y = data['y'].ravel()

    m = len(y)
    input_size = X.shape[1]
    num_hidden = 25
    num_labels = 10
    reg = 1

    y = (y - 1)
    y_onehot = np.zeros((m, num_labels))

    for i in range(m):
        y_onehot[i][y[i]] = 1

    theta1 = random_weights(len(theta1_sol), len(theta1_sol[0]))
    theta2 = random_weights(len(theta2_sol), len(theta2_sol[0]))

    params_rn = np.concatenate((theta1.flatten(), theta2.flatten()))

    fmin = minimize(fun=backprop, x0=params_rn, args=(input_size,
         num_hidden, num_labels, X, y_onehot, reg), method='TNC', jac=True,
         options={'maxiter': 70})

    print('El coste es de: {}'.format(fmin['fun']))

    correct_classified(X, y, fmin.x, input_size, num_hidden, num_labels)
```

Tenemos en este caso dos arrays, el primero es el de los datos (el que contiene lo que llamamos las X y la y) y el segundo es un array de pesos de una red ya entrenada. Después de separar las X y la y creamos una serie de variables con la información de nuestro caso, que es una red neuronal de 3 capas siendo 400 unidades en la primera, 25 en la segunda y 10 en la capa de salida.

La primera operación que hacemos es restar a la y 1 para facilitarnos el tratamiento de los datos. Después creamos la matriz `y_onhot` que va a contener por cada ejemplo un array de ceros, con 1 solo en la posición que corresponda con su valor.

A continuación inicializamos las thetas con un valor aleatorio. Para ello llamamos a la función `random_weights()`:

```
def random_weights(L_in, L_out, E_ini=0.12):
    return np.random.uniform(-E_ini, E_ini, (L_in, L_out))
```

Esta función lo que hace es devolver una matriz con las dimensiones, pasadas por parámetro, con valores aleatorios entre -0,12 y 0,12 (valor por defecto en caso de no introducir ninguno). Una vez ya tenemos los datos podemos comenzar con el desarrollo de la función `backprop()` que es la que se va a encargar de devolver el coste y el gradiente y de hacer la propagación hacia delante y hacia atrás.

```
def backprop(params_rn, num_entries, num_hidden, num_labels, X, y, reg):
    theta1 = np.reshape(params_rn[:num_hidden * (num_entries + 1)],
         (num_hidden, (num_entries + 1)))
    theta2 = np.reshape(params_rn[num_hidden * (num_entries + 1):],
         (num_labels, (num_hidden + 1)))
    M = len(y)

    X = np.hstack([np.ones([len(X), 1]), X])

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

    cost = cost_reg(np.transpose(h), y, theta1, theta2, M, reg)

    m = X.shape[0]
    delta1, delta2 = np.zeros(np.shape(theta1)), np.zeros(np.shape(theta2))

    for t in range(m):
        a1t = a1[t, :]
        a2t = a2[t, :]
        ht = h[t,:]
        yt = y[t]
        d3t = ht-yt
        d2t = np.dot(theta2.T, d3t) * (a2t * (1 - a2t))
        delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])
        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    gradient1 = delta1 / m
    gradient2 = delta2 / m

    reg1 = (reg / m) * theta1
    reg2 = (reg / m) * theta2

    reg1[:, 0] = 0
    reg2[:, 0] = 0

    gradient1 += reg1
    gradient2 += reg2

    gradient = np.concatenate((np.ravel(gradient1), np.ravel(gradient2)))

    return cost, gradient
```

Lo primero que hacemos es dividir las thetas que habíamos concatenado previamente con la función `np.concatenate()`. Guardamos el valor de M en una variable para hacer más cómodo hacer la función de coste.

Lo primero que hacemos es añadir la columna de unos y llamar a la función `forward_propagate()` que es la que nos va a devolver los valores de a1, z2, a2, z3 y h. Esta función es de la forma:

```
def forward_propagate(theta1, theta2, X):
    hidden_layer_z2 = np.matmul(X, np.transpose(theta1))
    hidden_layer_a2 = sigmoid(hidden_layer_z2)
    hidden_layer_a2 = np.hstack([np.ones([np.shape(hidden_layer_a2)[0],
1]), hidden_layer_a2])

    output_layer_z3 = np.matmul(hidden_layer_a2, np.transpose(theta2))
    output_layer_a3 = sigmoid(output_layer_z3)

    return X, hidden_layer_z2, hidden_layer_a2, output_layer_z3,
output_layer_a3
```

La explicación y las funciones que se utilizan se explicaron en la anterior práctica así que las vamos a omitir. En caso de duda consultar "Práctica 3: Regresión Logística Multi-clase y Redes Neuronales".

Una vez tenemos los datos vamos a calcular el coste. Para ello llamamos a la función `cost_reg()` que nos calcula el coste regularizado:

```
def cost_reg(output_layer, y, theta1, theta2, M, reg):
    H = cost(output_layer, y, M)

    return H + reg / (2 * M) * (np.sum(theta1[:, 1:] * theta1[:, 1:]) +
np.sum(theta2[:, 1:] * theta2[:, 1:]))
```

Este valor es el que vamos a utilizar para devolver en la función `backprop()` como primer valor.

Lo que hacemos a continuación lo podemos describir en 4 pasos:

1. Asignamos el valor de x(t) a la capa de entrada de la red, a(1), y realizamos una pasada hacia adelante calculando las salidas de las capas 2 y 3 (a(2) y a(3)). Añadimos un +1 al principio de a(1) y a(2).

2. Para cada unidad k de la capa 3 (la capa de salida), calculamos:

$$\delta_k^{(3)} = (a_k^{(3)} - y_k),$$

donde yk ∈ {0, 1} representa si el ejemplo de entrenamiento pertenece a la clase k (yk = 1) o a una clase diferente (yk = 0).

3. Para la capa oculta l = 2, calculamos:

$$\delta^{(2)} = (\Theta^{(2)})^T \delta^{(3)} .* g'(z^{(2)})$$

4. Acumulamos el gradiente de este ejemplo utilizando la fórmula:

$$\Delta^{(l)} = \Delta^{(l)} + \delta^{(l+1)} (a^{(l)})^T$$

Por último, una vez procesados los m ejemplos, calculamos el gradiente dividiendo por m los valores acumulados en el bucle:

$$\begin{aligned} \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} &= \frac{1}{m} \Delta_{ij}^{(l)} && \text{para } j = 0 \\ \frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} &= \frac{1}{m} \Delta_{ij}^{(l)} + \frac{\lambda}{m} \Theta_{ij}^{(l)} && \text{para } j \geq 1 \end{aligned}$$

Por último volvemos a concatenar las deltas y lo devolvemos como segundo valor junto al coste.

Para realizar el entrenamiento de la red vamos a utilizar la función `minimize()`. Esta función nos va a devolver el coste y los pesos (concatenados). Para comprobar que todo ha funcionado de forma correcta hemos creado la función `correct_classified()`:

```
def correct_classified(X, y, params_rn, num_entries, num_hidden,
     num_labels):
    X = np.hstack([np.ones([len(X), 1]), X])
    theta1 = np.reshape(params_rn[:num_hidden * (num_entries + 1)],
         (num_hidden, (num_entries + 1)))
    theta2 = np.reshape(params_rn[num_hidden * (num_entries + 1):]
         (num_labels, (num_hidden + 1)))

    hidden_layer = sigmoid(np.matmul(theta1, np.transpose(X)))
    hidden_layer = np.vstack([np.ones([len(X)]), hidden_layer])

    output_layer = sigmoid(np.matmul(theta2, hidden_layer))

    max_values = np.argmax(output_layer, axis=0)

    correct_values = y == max_values

    print("Se han clasificado correctamente: {}
         %".format(np.sum(correct_values) / len(correct_values) * 100))
```

Lo que hace esta función es simplemente hacer una propagación hacia adelante por las distintas capas hasta obtener el resultado final. Una vez obtenido el resultado de la `output_layer` nos vamos a quedar solo con el índice mayor de cada una y vamos a compararlo con el y (que es el valor real). Lo devolvemos en forma de porcentaje.

3. Resultados

En este apartado vamos a ir comentando los distintos resultados que hemos obtenido.

Los primeros resultados que obtuvimos fueron los respectos al coste. Comprobamos que la función coste nos daba correcto comparándola con el resultado esperado que aparecía en la práctica (en este punto solamente teníamos implementado la devolución del coste en la función `backprop()`):

```
> print(backprop(params_rn, num_entries, num_hidden, num_labels, X, y,
        reg))
```

```
  0.383770
```

También, otro resultado que obtuvimos fue el chequeo del gradiente. Para ello usamos la función `checkNNGradients()` que se nos daba junto a la documentación de la práctica y nos quedamos con el mayor valor de estos para poder sacar conclusiones. El mayor valor que obtuvimos fue el siguiente:

```
print('La mayor diferencia es de:{}'.format(max(checkNNGradients(backprop,
     reg))))
```

```
  La mayor diferencia es de: 7.329159501523463e-11
```

Como el mayor valor es más pequeño que 10-9 que es lo que nos indicaba el enunciado podemos afirmar que se realiza de forma correcta.

Por último nos queda comprobar cuantos casos se clasifican de forma correcta:

```
correct_classified(X, y, fmin.x, input_size, num_hidden, num_labels)
```

```
  Se han clasificado correctamente: 92.64 %
```

4. Conclusión

Esta práctica nos ha servido para entender el funcionamiento de las redes neuronales. Hemos comprendido cómo funcionan y como implementar de forma correcta la propagación hacia atrás.

En el siguiente apartado se muestra el código completo por si se quiere ver seguido o hacer alguna prueba compilada.

5. Código

```
import numpy as np
import matplotlib.pyplot as plt
import scipy
from scipy.io import loadmat
from scipy.optimize import minimize


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)

def sigmoid(X):
    return 1 / (1 + np.exp(-X))

def sigmoid_der(X):
    return sigmoid(X) * (1 - sigmoid(X))

def cost(output_layer, y, M):
    return 1 / M * np.trace((np.dot(-y, np.log(output_layer)) - np.dot((1 -
y), np.log(1 - output_layer))))

def cost_reg(output_layer, y, theta1, theta2, M, reg):
    H = cost(output_layer, y, M)

    return H + reg / (2 * M) * (np.sum(theta1[:, 1:] * theta1[:, 1:]) +
np.sum(theta2[:, 1:] * theta2[:, 1:]))

def random_weights(L_in, L_out, E_ini=0.12):
    return np.random.uniform(-E_ini, E_ini, (L_in, L_out))

def forward_propagate(theta1, theta2, X):
    hidden_layer_z2 = np.matmul(X, np.transpose(theta1))
    hidden_layer_a2 = sigmoid(hidden_layer_z2)
```

```
    hidden_layer_a2 = np.hstack([np.ones([np.shape(hidden_layer_a2)[0], 1]),
hidden_layer_a2])

    output_layer_z3 = np.matmul(hidden_layer_a2, np.transpose(theta2))
    output_layer_a3 = sigmoid(output_layer_z3)

    return X, hidden_layer_z2, hidden_layer_a2, output_layer_z3,
output_layer_a3

def backprop(params_rn, num_entries, num_hidden, num_labels, X, y, reg):
    theta1 = np.reshape(params_rn[:num_hidden * (num_entries + 1)],
(num_hidden, (num_entries + 1)))
    theta2 = np.reshape(params_rn[num_hidden * (num_entries + 1):],
(num_labels, (num_hidden + 1)))
    M = len(y)

    X = np.hstack([np.ones([len(X), 1]), X])

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

    cost = cost_reg(np.transpose(h), y, theta1, theta2, M, reg)

    m = X.shape[0]
    delta1, delta2 = np.zeros(np.shape(theta1)), np.zeros(np.shape(theta2))

    for t in range(m):
        a1t = a1[t, :]
        a2t = a2[t, :]
        ht = h[t,:]
        yt = y[t]
        d3t = ht-yt
        d2t = np.dot(theta2.T, d3t) * (a2t * (1 - a2t))
        delta1 = delta1 + np.dot(d2t[1:, np.newaxis], a1t[np.newaxis, :])
        delta2 = delta2 + np.dot(d3t[:, np.newaxis], a2t[np.newaxis, :])

    gradient1 = delta1 / m
    gradient2 = delta2 / m

    reg1 = (reg / m) * theta1
    reg2 = (reg / m) * theta2

    reg1[:, 0] = 0
    reg2[:, 0] = 0

    gradient1 += reg1
    gradient2 += reg2

    gradient = np.concatenate((np.ravel(gradient1), np.ravel(gradient2)))

    return cost, gradient

def correct_classified(X, y, params_rn, num_entries, num_hidden, num_labels):
    X = np.hstack([np.ones([len(X), 1]), X])
    theta1 = np.reshape(params_rn[:num_hidden * (num_entries + 1)],
(num_hidden, (num_entries + 1)))
    theta2 = np.reshape(params_rn[num_hidden * (num_entries + 1):],
(num_labels, (num_hidden + 1)))

    hidden_layer = sigmoid(np.matmul(theta1, np.transpose(X)))
    hidden_layer = np.vstack([np.ones([len(X)]), hidden_layer])
```

```
    output_layer = sigmoid(np.matmul(theta2, hidden_layer))

    max_values = np.argmax(output_layer, axis=0)

    correct_values = y == max_values

    print("Se han clasificado correctamente: {}
%".format(np.sum(correct_values) / len(correct_values) * 100))

def main():
    data = loadmat('Práctica 4/Recursos/ex4data1.mat')
    weights = loadmat('Práctica 4/Recursos/ex4weights.mat')

    theta1_sol, theta2_sol = weights['Theta1'], weights['Theta2']

    X = data['X']
    y = data['y'].ravel()

    m = len(y)
    input_size = X.shape[1]
    num_hidden = 25
    num_labels = 10
    reg = 1

    y = (y - 1)
    y_onehot = np.zeros((m, num_labels))

    for i in range(m):
        y_onehot[i][y[i]] = 1

    theta1 = random_weights(len(theta1_sol), len(theta1_sol[0]))
    theta2 = random_weights(len(theta2_sol), len(theta2_sol[0]))

    params_rn = np.concatenate((theta1.flatten(), theta2.flatten()))

    difference = checkNNGradients(backprop, reg)

    print('La mayor diferencia es de:
{}'.format(max(checkNNGradients(backprop, reg))))

    fmin = minimize(fun=backprop, x0=params_rn, args=(input_size, num_hidden,
num_labels, X, y_onehot, reg), method='TNC', jac=True, options={'maxiter':
70})

    print('El coste es de: {}'.format(fmin['fun']))

    correct_classified(X, y, fmin.x, input_size, num_hidden, num_labels)


main()
```