Práctica 4:
Entrenamiento de redes neuronales

Rafael Herrera Troca
Rubén Ruperto Díaz

Aprendizaje Automático y Big Data
Doble Grado en Ingeniería Informática y Matemáticas

4 de septiembre de 2020

1. Introducción

En esta práctica hemos implementado las funciones de coste y gradiente de una red neuronal, así como la función de retro-propagación para su entrenamiento. Posteriormente, hemos estudiado como cambian los resultados de la red al variar ciertos parámetros del entrenamiento.

Hemos programado el código en el entorno *Spyder* con Python 3.7.

2. Procedimiento

Comenzamos por explicar las funciones más sencillas que hemos definido, mientras que las funciones más complejas para el cálculo del coste y su gradiente se desarrollarán en secciones posteriores.

En primer lugar, tenemos la función *sigmoide*, que sirve para transformar el resultado devuelto por una capa de la red en un valor en el intervalo $[0, 1]$, y que viene definida según la fórmula:

$$g(z) = \frac{1}{1 + e^{-z}}$$

La segunda función definida es la derivada de la sigmoide, *diffSigmoide*, cuya definición para una capa l de la red, siendo $a^{(l)}$ la entrada de dicha capa y $z^{(l)}$ la salida de la capa $l - 1$, viene dada por la fórmula:

$$g'(z^{(l)}) = a^{(l)}(1 - a^{(l)})$$

Si tenemos en cuenta que $a^{(l)} = [1 \sim g(z^{(l)})]$, esto es, calcular la sigmoide del resultado devuelto por la capa anterior y poner una columna de unos delante (normalmente referida como $a_0^{(l)}$), podemos expresar la derivada de la sigmoide según esta otra fórmula:

$$g'(z^{(l)}) = [1 \sim g(z^{(l)})](1 - [1 \sim g(z^{(l)})])$$

No obstante, por ser la implementación mucho más sencilla de este modo, en el código se ha utilizado la primera fórmula.

También hemos definido las funciones *applyLayer* y *applyNet* que, como su nombre indica, aplican una capa de la red dada la entrada y la matriz de pesos correspondientes, o la red entera dadas la entrada y un array con las matrices de pesos correspondientes a cada capa. Además, *applyNet* no sólo devuelve el resultado de aplicar la red, también devuelve la entrada utilizada en cada una de las capas, ya que necesitábamos este dato para calcular el gradiente de la función de coste.

Finalmente, también hemos definido las funciones *randomWeights*, que inicializa una matriz de pesos aleatorios para las dimensiones indicadas, y *acierto*, que calcula el porcentaje de acierto dadas la respuesta y las etiquetas correctas.

A continuación se explica con mayor grado de detalle el desarrollo de la función de coste y su gradiente.

2.1. Función de coste

Para la función de coste hemos aplicado la fórmula correspondiente manteniendo la generalidad todo lo posible. Dicha fórmula viene dada por:

$$J(\theta) = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \frac{\lambda}{2N}\sum_{l=1}^{L}\left[\sum_{i,j>0}(\Theta_{i,j}^{(l)})^2\right]$$

En esta fórmula, N indica el número de casos, K, el número de etiquetas y L, el número de matrices de pesos que tiene la red. Asimismo, y es la salida esperada de la red neuronal, x es la entrada, $h_\theta(x)$ es el resultado devuelto, λ es el parámetro de regularización y Θ es la matriz de pesos de una cierta capa. Además, el sumatorio que aparece al final del término de regularización expresa que se debe sumar el cuadrado de todos los términos de cada matriz de pesos teniendo en cuenta no sumar la primera columna (recordemos que el índice de las filas empieza en uno siempre).

En cualquier caso, para aprovechar la potencia del cálculo matricial que ofrece Python, hemos vectorizado la función de coste, evitando así tener que hacer el cálculo de la primera parte para cada uno de los casos de prueba. La fórmula que hemos utilizado es la siguiente:

$$J(\theta) = \frac{1}{N}\sum_{i,j}\left[y \odot \log(h_\theta(x)) - (1 \ominus y) \odot \log(1-h_\theta(x))\right]_{i,j} + \frac{\lambda}{2N}\sum_{l=1}^{L}\left[\sum_{i,j>0}(\Theta_{i,j}^{(l)})^2\right]$$

En esta fórmula, las letras vuelven a significar lo mismo que en la fórmula anterior, el término $(1 \ominus y)$ representa la matriz M dada por $M_{i,j} = 1 - y_{i,j}$, y el símbolo $\odot$ indica el producto elemento a elemento de dos matrices o producto de Hadamard.

Tras la construcción de la función, comprobamos con las matrices de pesos dadas como ejemplo que el resultado era el correcto.

2.2. Gradiente de la función de coste

Para el cálculo del gradiente implementamos la función *gradiente*, que lo calcula, nuevamente, de la forma más genérica posible para que se pueda utilizar para una red neuronal cualquiera sin tener que adaptarlo. Recibe como argumentos la salida

esperada para la red neuronal, la entrada de cada capa, el resultado de aplicar la red, la lista de matrices de pesos y el parámetro de regularización, y devuelve la lista de matrices de gradiente.

Para obtener este resultado vamos calculando la matriz del gradiente correspondiente a cada matriz de pesos empezando por la última y retrocediendo hasta llegar a la primera. Hacemos estos cálculos de forma vectorizada, procesando todos los casos a la vez, si bien cada matriz se calcula por separado en un bucle. La primera matriz que calculamos, que se corresponde a la última capa de la red, se obtiene aplicando las fórmulas siguientes:

$$\delta^{(L+1)} = a^{(L+1)} - y$$
$$D^{(L)} = \frac{(\delta^{(L+1)})^T a^{(L+1)} + \lambda \Theta^{(L)}_{\bullet, j>0}}{N}$$

El resto de matrices se calculan iterativamente con las fórmulas recursivas correspondientes, que para $0 < i < L$, donde L es el número de matrices de pesos que tiene la red, vienen dadas por:

$$\delta^{(i+1)} = (\delta^{(i+2)} \Theta(i+1)) \odot g'(z^{(i+1)})$$
$$D^{(i)} = \frac{(\delta^{(i+1)}_{\bullet, j>0})^T a^{(i)} + \lambda \Theta^{(i)}_{\bullet, j>0}}{N}$$

En estas fórmulas, N es el número de casos con el que se entrena la red, y es la respuesta esperada, $a^{(l)}$ es la entrada de la capa l de la red, z es la salida de la capa $l-1$, $\Theta^{(l)}$ es la matriz de pesos que pasa de la capa l a la $l+1$ y λ es el parámetro de regularización.

Una vez construida la función *gradiente*, utilizamos el código auxiliar proporcionado en el fichero *checkNNGradients.py* para comprobar que es correcta.

3. Resultados

Tras verificar que la función de coste y la de retro-propagación estaban implementadas correctamente, hemos procedido a entrenar la red neuronal con varios valores del parámetro de regularización λ y del número máximo de iteraciones para analizar el comportamiento de la red en cada caso. Para λ, hemos probado las potencias de 10 entre 10^{-5} y 10^2, y para el límite de iteraciones los múltiplos de 10 entre 10 y 100

$$\lambda \in \{10^i \mid i \in [-5, 2], i \in \mathbb{Z}\} \quad iter \in \{n \in [10, 100] \mid n \equiv 0 \,(\text{mód } 10)\}$$

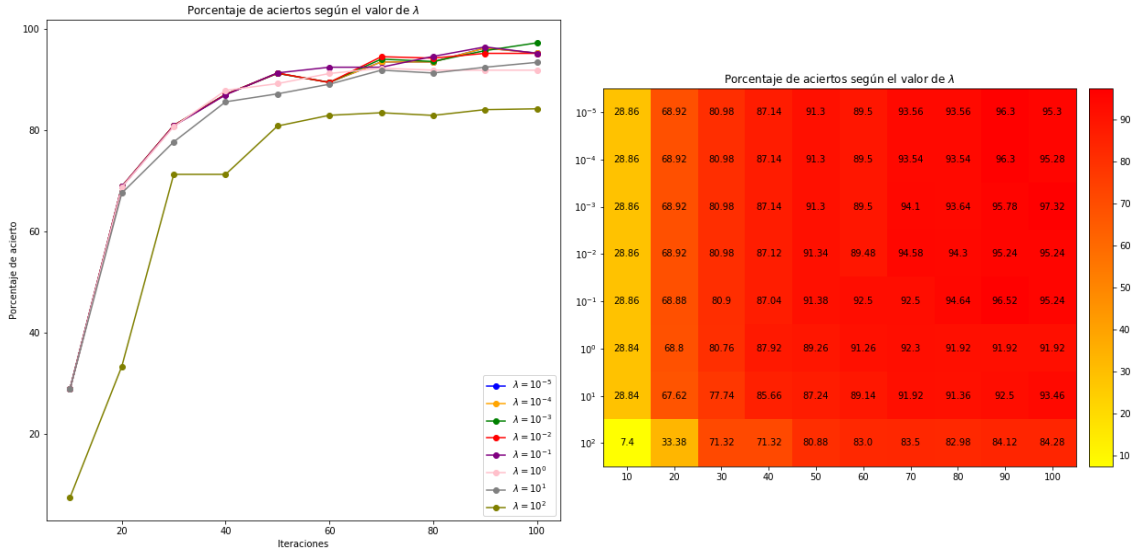

Figura 1: Estas gráficas muestran de dos formas distintas los mismos datos de tasas de acierto obtenidas tras entrenar nuestra red neuronal

Observamos que el porcentaje de aciertos de la red neuronal crece si permitimos un mayor número máximo de iteraciones. Esto se debe a que la función de minimización se va aproximando al mínimo progresivamente en cada iteración según le indica la dirección del gradiente. Si permitimos más iteraciones, nos aproximaremos más al mínimo de la función de coste y por tanto al mínimo error al predecir los datos de entrenamiento. Sin embargo, no es bueno minimizar excesivamente la función de coste puesto que, aunque permita acertar más con los datos de entrenamiento, puede perjudicar la predicción de datos nuevos si se produce sobreaprendizaje, haciendo que la red sólo prediga correctamente los datos de entrenamiento.

Asimismo, se obtienen más aciertos para valores más pequeños de λ. Para entender esto, recordamos que en el término de regularización que se añade a la función de coste, λ aparece multiplicando al sumatorio de los valores que aparecen en las matrices de pesos. Esto implica que, para minimizar la función de coste, mayores valores de λ fuerzan a elegir pesos más pequeños, mientras que λ más pequeños permiten mayor libertad al elegir los pesos. De este modo, con valores de λ pequeños la red tiene más flexibilidad para ajustarse a los datos de entrenamiento, y por tanto también existe la posibilidad del sobreaprendizaje si permitimos que la red se ajuste demasiado.

Por tanto, es posible que en los casos en los que nuestra red ha obtenido mejores tasas de acierto haya habido sobreaprendizaje, pero para confirmarlo necesitaríamos ver cómo se comporta la red con datos distintos de los utilizados para entrenarla.

4. Conclusiones

Gracias a esta práctica hemos comprobado que detrás del entrenamiento de una red neuronal hay muchos detalles técnicos y cálculos delicados que hay que realizar

con cuidado para que el resultado sea el deseado. Un fallo como sumar una columna de la matriz de pesos cuando no se debe puede pasar desapercibido al programador y provocar que los resultados de la red empeoren notablemente. En nuestro caso, al calcular el término de regularización de la función de coste incluimos las primeras columnas de las matrices de pesos, con lo que el resultado de la función de coste variaba ligeramente en el tercer decimal. Aunque al principio no dimos importancia a esta variación, provocó que el gradiente no se calculara bien y que el resultado no fuera igual que el obtenido por otro método implementado correctamente.

5. Anexo: Código

A continuación se ofrece el código escrito para resolver los ejercicios propuestos en la práctica.

```
'''
Práctica 4

Rubén Ruperto Díaz y Rafael Herrera Troca
'''

import os
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.axes_grid1 import make_axes_locatable
from scipy.io import loadmat
from scipy.optimize import minimize


os.chdir("./resources")


# Función sigmoide
def sigmoide(z):
    return 1 / (1 + np.exp(-z))

# Derivada de la función sigmoide
def diffSigmoide(a):
    return a * (1 - a)

# Devuelve una matriz de pesos aleatorios con la dimensión dada
def randomWeights(l_in, l_out):
    eps = np.sqrt(6)/np.sqrt(l_in + l_out)
    rnd = np.random.random((l_out, l_in+1)) * (2*eps) - eps
    return rnd

# Dada la entrada 'X' y los pesos 'theta' de una capa de una red
# neuronal, aplica los pesos y devuelve la salida de la capa
def applyLayer(X, theta):
    thetaX = np.dot(X, theta.T)
    return sigmoide(thetaX)
```

```
38
39 # Dada la entrada 'X' y el array de matrices de pesos 'theta',
40 # devuelve la entrada de cada capa y el resultado final devuelto
41 # por la red neuronal
42 def applyNet(X, theta):
43     lay = X.copy()
44     a = []
45     for i in range(len(theta)):
46         lay = np.hstack((np.array([np.ones(len(lay))]).T, lay))
47         a.append(lay.copy())
48         lay = applyLayer(lay, theta[i])
49
50     return lay,a
51
52 # Calcula la función de coste de una red neuronal para la
53 # salida esperada 'y', el resultado de la red 'h_theta', el array
54 # de matrices de pesos 'theta' y el término de regularización 'reg'
55 def coste(y, h_theta, theta, reg):
56     sumandos = -y * np.log(h_theta) - (1-y) * np.log(1-h_theta)
57     regul = 0
58     for i in range(len(theta)):
59         regul += np.sum(theta[i][:,1:]**2)
60     result = np.sum(sumandos) / len(y) + reg * regul / (2*len(y))
61     return result
62
63 # Calcula el gradiente de la función de coste haciendo
64 # retropropagación dada la salida esperada 'y', la entrada
65 # de cada capa 'a', la salida de la red 'h_theta', el array de
66 # matrices de pesos 'theta' y el término de regularización 'reg'
67 def gradiente(y, a, h_theta, theta, reg):
68     d = h_theta - y
69     delta = [np.dot(d.T, a[-1]) / len(y)]
70
71     for i in range(len(theta)-1,0,-1):
72         d = np.dot(d, theta[i]) * diffSigmoide(a[i])
73         d = d[:,1:]
74         delta.insert(0, np.dot(d.T, a[i-1]) / len(y))
75
76     for i in range(len(delta)):
77         delta[i][:,1:] += reg * theta[i][:,1:] / len(y)
78
79     return delta
80
81 # Calcula y devuelve el coste y el gradiente de una red neuronal
82 # dados todos los pesos en el array 'param_rn', las dimensiones
83 # de cada capa en 'capas', los datos de entrada 'X', la salida
84 # esperada 'y' y el término de regularización 'reg'
85 def backprop(params_rn, capas, X, y, reg):
86     # Convertimos el vector de todos los pesos en las distintas
87     # matrices
88     theta = [np.reshape(params_rn[:capas[1]*(capas[0]+1)],(capas
    [1],capas[0]+1))]
89     gastados = capas[1]*(capas[0]+1)
90     for i in range(len(capas)-2):
```

```
        theta.append(np.reshape(params_rn[gastados:gastados+capas[i
    +2]*(capas[i+1]+1)],(capas[i+2],capas[i+1]+1)))
        gastados += capas[i+2]*(capas[i+1]+1)

    # Calculamos el vector de salida esperada para la red neuronal
    Y = np.zeros((len(y), capas[-1]))
    for i in range(len(Y)):
        Y[i,y[i]-1] = 1

    # Aplicamos la red neuronal
    h_theta,a = applyNet(X, theta)

    cost = coste(Y, h_theta, theta, reg)
    grad = gradiente(Y, a, h_theta, theta, reg)

    g = np.array([])
    for i in range(len(grad)):
        g = np.concatenate((g, grad[i].ravel()))

    return cost, g

# Calcula el porcentaje de acierto obtenido con la respuesta dada
# en 'X' y las etiquetas correctas en 'Y'
def acierto(X, Y):
    resultados = X.argmax(axis=1) + 1
    return 100 * np.count_nonzero(resultados == Y.ravel()) / len(Y)


## Parte 1: Función de coste y gradiente

# Leemos los datos
data = loadmat('ex4data1.mat')
y = data['y'].ravel()
X = data['X']

# Leemos las matrices de pesos
weights = loadmat('ex4weights.mat')
theta1, theta2 = weights['Theta1'], weights['Theta2']

# Calculamos el coste y el gradiente
theta = np.concatenate((theta1.ravel(), theta2.ravel()))
res = backprop(theta, (400,25,10), X, y, 1)


## Parte 2: Entrenamos la red neuronal

# Creamos unas matrices de pesos iniciales de forma aleatoria
theta01 = randomWeights(400, 25)
theta02 = randomWeights(25, 10)
theta0 = np.concatenate((theta01.ravel(), theta02.ravel()))

# Entrenamos la red neuronal con distintos valores para el
# parámetro de regularización y representamos su porcentaje
# de acierto
reg = range(-5, 3)
```

```
145 itera = range(10, 110, 10)
146
147 pAc = np.zeros((len(reg), len(itera)))
148 for i in range(len(reg)):
149     for j in range(len(itera)):
150         theta = minimize(fun=backprop, x0=theta0, args=((400,25,10)
    , X, y, 10**reg[i]), method='TNC', jac=True, options={'maxiter':
    itera[j]})['x']
151
152         theta1 = np.reshape(theta[:25*401],(25,401))
153         theta2 = np.reshape(theta[25*401:],(10,26))
154         res =  applyNet(X, (theta1, theta2))[0]
155
156         pAc[i][j] = acierto(res, y)
157
158 colores = ['blue', 'orange', 'green', 'red', 'purple', 'pink', '
    gray', 'olive', 'cyan']
159 plt.figure(figsize=(10,10))
160 for i in range(len(pAc)):
161     plt.plot(range(10, 110, 10), pAc[i], color=colores[i], marker='
    o', label=(r"$\lambda = 10^{" + str(reg[i]) + "}$"))
162 plt.title(r"Porcentaje de aciertos según el valor de $\lambda$")
163 plt.xlabel("Iteraciones")
164 plt.ylabel("Porcentaje de acierto")
165 plt.legend(loc="lower right")
166 plt.savefig("LineasLambda.png")
167 plt.show()
168
169 xLabels = [str(it) for it in itera]
170 yLabels = [r'$10^{' + str(r) + '}$' for r in reg]
171 plt.figure(figsize=(10,10))
172 plt.title(r"Porcentaje de aciertos según el valor de $\lambda$")
173 fig = plt.subplot()
174 im = fig.imshow(pAc, cmap=cm.autumn_r)
175 cax = make_axes_locatable(fig).append_axes("right", size="5%", pad
    =0.2)
176 plt.colorbar(im, cax=cax)
177 for i in range(len(xLabels)):
178     for j in range(len(yLabels)):
179         text = fig.text(i, j, pAc[j][i], ha="center", va="center",
    color="k")
180 fig.set_xticks(np.arange(len(xLabels)))
181 fig.set_yticks(np.arange(len(yLabels)))
182 fig.set_xticklabels(xLabels)
183 fig.set_yticklabels(yLabels)
184 plt.savefig("CuadrosLambda.png")
185 plt.show()
```