Universidad Complutense de Madrid
Facultad de Informática
Aprendizaje Automático y Big Data

Memoria Práctica 4:
Entrenamiento de redes neuronales.

Profesor:
- Alberto Díaz Esteban.

Alumnos:
- Marina de la Cruz López.
- Diego Alejandro Rodríguez Pereira.

Introducción

El objetivo de esta práctica es desarrollar un código para la implementación de una red neuronal, cuyo diseño es similar a la del realizado en la práctica anterior (práctica 3), con la diferencia en que esta práctica se implementará todo el proceso de entrenamiento de la red neuronal, lo que incluye el cálculo de los pesos (thetas) aplicando la propagación hacia atrás (*backpropagation)*, de tal forma que se optimizarán estos pesos, los cuales posteriormente se usarán en cada capa de la red neuronal, la cual a su vez, devolverá una predicción usando estos pesos calculados. Esta predicción se cuantificará en función del porcentaje de aciertos realizados por el algoritmo.

Parte 1: Cálculo del coste

El objetivo de este primer apartado es el de implementar la función de coste de nuestra red neuronal para un conjunto de ejemplos de entrenamiento. Este conjunto de entrenamiento es el mismo que el utilizado en la práctica anterior. Para trabajar con él, inicialmente cargamos los datos del fichero 'ex4data1.mat' que está en formato nativo para matrices de Octave/Matlab, por lo que se ha de utilizar la función *'loadmat'* para cargar estos datos. Este fichero contiene los ejemplos de entrenamiento para nuestra red neuronal, cada ejemplo es una imagen de 20x20 píxeles, donde cada píxel está representado por un número real que indica la intensidad en escala de grises de ese punto. A su vez cada matriz de 20x20 está desplegada para formar un vector de componentes, que en nuestro caso será la matriz 'X'.

La variable 'X' que es nuestra variable de entrada, es una matriz donde cada fila corresponde a una imagen, la cual es una imagen de un número entero de entre 0 y 9 (de un dígito) escrito a mano. En esta matriz 'X' tenemos 5000 imágenes para el entrenamiento de la red neuronal.

Para visualizar cada una de las imágenes de los ejemplos del dataset se hace uso de la función *'displayImage'* de *'displayData.py'* la cual nos permite mostrar por pantalla la imagen que escribe estos números. Lo que se imprime por pantalla es lo siguiente:

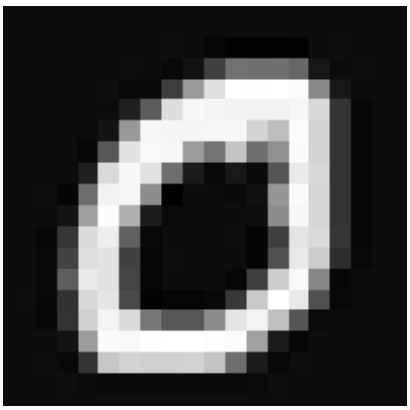

Figura 1

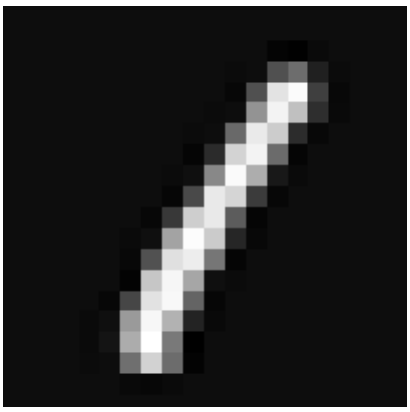

Figura 2

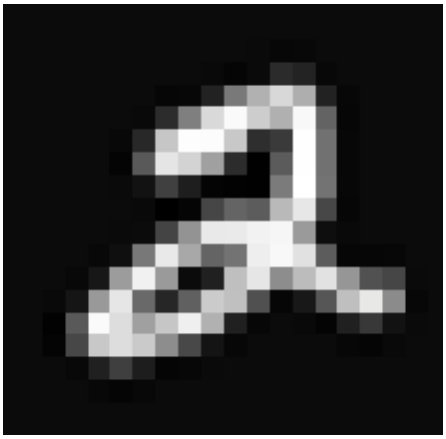

Figura 3

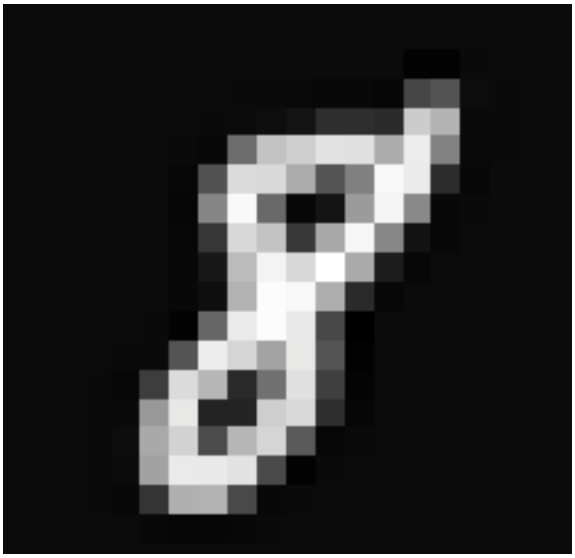

Figura 4

La variable *'y'* contiene la salida, es decir los números del 1 al 9 que corresponde a su respectivo número natural, y siendo el 0 el número representado por la etiqueta 10.

Nuestra red neuronal tiene 400 neuronas en la capa entrada, 25 neuronas en la capa oculta y 10 neuronas en la capa de salida, que se corresponden con las 10 clases en las que vamos a clasificar cada una de las imágenes de los ejemplos del conjunto de datos de entrenamiento.

Lo primero que vamos a implementar es la función que calcula el coste de la red neuronal para unas matrices de Thetas. Para ello programamos la función de coste descrita, sin regularización, como se muestra en la figura5, la cual usa la salida de la propagación hacia adelante (como se muestra en la figura6) y posteriormente realizamos la suma del coste para cada ejemplo de entrenamiento, para cada clase de las 10 que tenemos en la y_onehot partido por la cantidad de ejemplos de entrenamiento.

$$J(\theta) = \frac{1}{m} \sum_{i=1}^{m} \sum_{k=1}^{K} \left[-y_k^{(i)} \log((h_\theta(x^{(i)}))_k) - (1 - y_k^{(i)}) \log(1 - (h_\theta(x^{(i)}))_k) \right]$$

Figura 5

$$a^{(1)} = x^{(1)}$$

$$z^{(2)} = \Theta^{(1)} \cdot a^{(1)}$$

$$a^{(2)} = g(z^{(2)})$$

$$\text{add } a_0^{(2)} \rightarrow a^{(2)}$$

$$z^{(3)} = \Theta^{(2)} a^{(2)}$$

$$a^{(3)} = g(z^{(3)})$$

Figura 6

Probamos esta función de coste sin regularizar y comprobamos que nos da como resultado 0.287629, que es el valor que esperamos.

Una vez implementada la función de coste sin regularización pasamos a programar el término de regularización, que se va a sumar a la función ya implementada, para así obtener la función de coste con regularización. Para calcular la función de regularización pasamos como parámetro un vector con la matrices Thetas, para cada una de ellas hacemos la suma de sus valores elevados al cuadrado excepto por la primera columna que la quitamos debido a que Thetaj,0 no se regulariza en ningún caso.

Probamos la función de coste regularizado con los pesos que vienen incluidos en la práctica y lambda = 1. Obtenemos como resultado 0.383769, que es el valor que esperábamos.

Parte 2: Cálculo del gradiente

Programamos el gradiente de la siguiente forma: Primero sacamos las Thetas con las que vamos a calcular el *forward propagation* de la red neuronal, como las obtenemos en forma de lista las tenemos que reconstruir en matrices, y devolvemos *a1, z2, a2,* y *z3* del *forward propagation*. Con el valor de *z3* calculamos el coste que luego devolveremos. Aplicamos la fórmula que nos calcula *δ3* y *δ2*, posteriormente calculamos el valor de los gradientes con *δ3*y *a2*, *δ2*, menos su primera fila y *a1*, ambos partidos por la cantidad de ejemplos de entrenamiento *m*, esto nos devuelve el valor del gradiente sin regularizar. Usando la función '*checkNNGradients'* comprobamos que el valor es correcto y añadimos el término de regularización.

El término de regularización es añadir a los dos gradientes lambda partido la cantidad de ejemplos por la matriz de la Thetas. Esto se hace para cada columna de gradientes menos por la primera que no se regulariza, para poder realizar los cálculos vectorizados hemos cambiado la primera columna de las Thetas con ceros tal que a la primera columna de gradientes se le suma 0 siempre. Las matrices de gradientes se concatenan en una lista y se devuelve junto con el coste.

Nuevamente comprobamos que funciona el cálculo usando *'checkNNGradients'*

Parte 3: Aprendizaje de los parámetros

En este apartado, una vez ya culminados las funciones bases para nuestra red neuronal, como lo son la función de coste regularizada y el cálculo del gradiente, pasamos a entrenar la red neuronal y con ellos obtener los valores óptimos de Theta1 y Theta2.

El aprendizaje de red neuronal se hace desde la función *'optimize_backprop_and_check'*, la cual inicializa las Thetas iniciales a valores en el rango -*ini* y *ini* y las concatena en una lista. En nuestro caso el valor de nuestra *épsilon* inicial para la inicialización ha sido de valor 0,12.

Posteriormente llamamos a *opt.minimize* de la librería *spicy*, pasando como primer parámetro la función *'backprop'* que hemos implementado anteriormente y definiendo una cantidad límite de iteraciones, que en nuestro caso se ha establecido en unas 70 iteraciones. A su vez se han pasado como parámetro el número de entradas, el número de etiquetas, el número de capas ocultas, la matriz X, y_onehot y el valor de regularización que se ha establecido en lambda = 1. Los resultados los reconstruimos en dos matrices de Thetas y ejecutamos el forward propagation que nos devuelve la clasificación final obtenida. Usando argmax y el vector original de y - 1 calculamos el porcentaje final de aciertos que mostramos por pantalla.

Resultados obtenidos

Apartado 1. Cálculo de la función de coste.

La función de coste sin regularizar nos ha devuelto como resultado: 0.287629.
La función de coste con regularización nos ha dado como resultado: 0.383769

Ambos valores se corresponden con los valores esperados.

Apartado 2. Cálculo de la función de gradiente

Se ha comprobado que la función de gradiente es correcta para reg= 0 (no regularización)

grad shape: (38,)
num grad shape: (38,)
[5.49965629e-11 1.89059402e-12 7.89324162e-12 6.95584562e-12
-6.08260664e-11 2.08456863e-12 -1.29760924e-11 -4.60273486e-11
-9.29989974e-11 9.26888080e-12 -4.20321139e-11 -1.26826244e-10
-2.40059500e-11 2.76548055e-12 -8.24779828e-12 -2.49761462e-11
2.15736456e-11 -4.96176017e-13 7.55695506e-12 2.73879391e-11
6.25964836e-11 1.32927280e-11 4.59121630e-12 3.04720138e-12
1.45679302e-11 1.44032286e-11 6.93309299e-11 1.56080426e-11
7.11192216e-12 1.15287224e-11 1.70987668e-11 1.79336823e-11
7.32915950e-11 1.60134683e-11 6.39788222e-12 1.78091986e-11
1.66117675e-11 1.82341919e-11]

También para la función regularizada, por ejemplo, con reg = 1
grad shape: (38,)
num grad shape: (38,)
[5.49965629e-11 7.32747196e-13 8.82988127e-12 7.53046930e-12
-6.08260664e-11 2.10970130e-12 -1.38742212e-11 -4.70332939e-11
-9.29989974e-11 7.81530396e-12 -4.12793411e-11 -1.26643918e-10
-2.40059500e-11 4.35645964e-12 -7.00919878e-12 -2.43030734e-11
2.15736456e-11 2.27595720e-13 7.55695506e-12 2.84505197e-11
6.25964836e-11 1.38673517e-11 4.06508160e-12 3.07234793e-12
1.58902475e-11 1.33972833e-11 6.93309299e-11 1.41544554e-11
7.86469501e-12 1.17110766e-11 1.64833147e-11 1.95246597e-11
7.32915950e-11 1.66865410e-11 6.33046393e-12 1.85329807e-11
1.79033732e-11 1.99839867e-11]

Apartado 3. Aprendizaje de los parámetros

Luego de entrenar la Red Neuronal y obtenidos los valores óptimos para Theta1 y Theta2, el algoritmo ha obtenido una precisión de en torno a 92% en promedio. Como se muestra en la siguiente figura:

```
90.38000000000001
94.48
93.38
92.58
94.12
91.53999999999999
93.4
92.5
92.72
92.80000000000001
93.22
93.76
91.94
93.78
94.56
93.12
92.04
```

Lambda

A su vez, como se ha pedido para la práctica, hemos utilizado varios valores para Lambda, pero, como se puede apreciar en los resultados obtenidos, el valor que se establezca para lambda no afecta significativamente al porcentaje de aciertos obtenido porque este depende también de los pesos fijados aleatoriamente.

Ya que el valor de los pesos es aleatorio hemos ejecutado el clasificador para cada lambda 5 veces y estos son los resultados:

lambda: 0

90.56 %
94.76 %
93.38 %
89.98 %
93.82000000000001 %

lambda: 0.05

92.67999999999999 %
89.8 %
92.47999999999999 %
89.96 %
93.97999999999999 %

lambda: 0.1

92.22 %
83.62 %
90.36 %
92.80000000000001 %
90.60000000000001 %

lambda: 0.25

91.64 %

94.46 %
86.94 %
94.92 %
92.25999999999999 %

lambda: 0.5

94.17999999999999 %
93.38 %
92.42 %
93.7 %
92.92 %

lambda: 1

94.12 %
94.56 %
92.9 %
93.74 %
92.88 %

lambda: 2

93.46 %
94.02000000000001 %
92.78 %
92.47999999999999 %
91.60000000000001 %

lambda: 3

94.34 %
93.22 %
93.34 %
92.96 %
92.30000000000001 %

lambda: 4

92.14 %
91.7 %
93.26 %
92.64 %
92.14 %

lambda: 5

92.16 %
91.96 %
92.30000000000001 %
89.3 %
92.9 %

lambda: 10

91.18 %
91.25999999999999 %
90.24 %
92.17999999999999 %
91.44 %

lambda: 20

89.72 %
89.1 %
91.3 %

88.96 %
91.46 %

lambda 100

82.58 %
81.74 %
84.08 %
79.92 %
81.38 %

Para valores de lambda muy pequeños o muy grandes el porcentaje de aciertos tiende a ser menor, aunque depende de la ejecución, por lo general nos ha dado mejores valores con lambda entre 0.5 y 3.

Iteraciones

En el caso de las iteraciones se puede observar que para mayor cantidad de iteraciones mejor es capaz de optimizar los parámetros, aunque también más tarda en ejecutarse. Por esto a partir de cierto punto no merece la pena aumentarlas porque el coste en ejecución es muy alto y no ganamos apenas precisión.

iteraciones: 30

72.64 %
82.86 %
78.2 %
64.62 %
71.06 %

iteraciones: 50

88.4 %
86.83999999999999 %
89.18 %
87.6 %
88.58 %

iteraciones: 70

91.46 %
94.38 %
93.28 %
93.04 %
93.16 %

iteraciones: 100

95.72 %
93.78 %
94.89999999999999 %
95.38 %
96.02000000000001 %

iteraciones: 200

99.16 %
99.03999999999999 %
99.14 %
99.08 %
97.16 %

iteraciones: 300

99.4 %
99.4 %
99.53999999999999 %
99.46000000000001 %
99.5 %

Código

```
#Imports
import numpy as np
from pandas.io.parsers import read_csv
import matplotlib.pyplot as plt
from matplotlib.axes import Axes
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import math
import scipy.optimize as opt
from sklearn.preprocessing import PolynomialFeatures
from scipy.io import loadmat
from scipy.optimize import minimize

import displayData as disp
import checkNNGradients as grad
RANDOM_STATE = 100
```

Función de coste (con y sin regularización) y sigmoide

```
def func_coste_reg(Thetas, X, Y, lmb):
    m = np.shape(X)[0]
    return cost_funct(Y, X, m) + regularizacion(Thetas, lmb, m)

def sigmoid_funct(z):
    return 1.0 / (1.0 + np.exp(-z))

def cost_funct(Y, g, m):
    J = np.sum(-1.0*Y* np.log(g) -1.0*(1 - Y)* np.log(1-g))
    return J/m

def regularizacion(Thetas, lmb, m):
    suma = 0
    for i in Thetas:
```

```
        i = i[:,1:]
        suma += (np.sum(i**2))

    return (lmb/(2*m))*suma
```

Propagación hacia delante

```
def forward_propagation(X, theta1, theta2, m, Y):

    #Capa entrada asignamos la X con los unos incluidos
    a1 = np.hstack([np.ones([m, 1]), X])

    #capa intermedia (hidden) calculamos las ecuaciones de la anterior, aplicamos
la sigmoide e incluimos los unos de la neurona 0
    z2 = np.dot(a1, theta1.T)
    a2 = np.hstack([np.ones([m, 1]), sigmoid_funct(z2)])

    #Capa salida calcuamos las ecuaciones con theta2 y aplicamos la sigmoide, nos
devuelve la matriz de salida 5000x10
    z3 = np.dot(a2, theta2.T)

    a3 = sigmoid_funct(z3)

    return a1, z2, a2, z3, a3
```

Propagación hacia atrás

```
def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas, X, Y, reg):

    #Recostruimos las Thetas
    Theta1 = np.reshape(params_rn[:num_ocultas *(num_entradas + 1)],(num_ocultas,
(num_entradas+1)))
    Theta2 = np.reshape(params_rn[num_ocultas*(num_entradas + 1): ],
(num_etiquetas,(num_ocultas+1)))

    m = X.shape[0]
    y_onehot = Y

    a1, z2, a2, z3, a3 = forward_propagation(X, Theta1, Theta2, m, y_onehot)

    #Calculamos el coste
    coste = func_coste_reg([Theta1, Theta2], a3, y_onehot, reg)


    d3 = a3 - y_onehot
    d2 = np.matmul(Theta2.T,d3.T)*(a2*(1-a2)).T

    #Calculamos los gradientes no regularizados
    G1 = np.matmul(d2[1:,:], a1)/m
    G2 = np.matmul(d3.T,a2)/m
```

```
    #Calculamos los gradientes regularizados
    G1 = G1 + ((reg*1.0)/m)*np.insert(Theta1[:, 1:], 0, 0, axis = 1)
    G2 = G2 + ((reg*1.0)/m)*np.insert(Theta2[:, 1:], 0, 0, axis = 1)

    #Ponemos los gradientes en forma de lista
    gradientes = np.concatenate((G1, G2), axis = None)

    return coste, gradientes
```

Función que optimiza los pesos de la red neuronal

```
#Genera los pesos aleatorios
def RandomWeights(entradas, salidas, ini):
    Theta = np.random.uniform(-ini, ini, size = (salidas,entradas+1))
    return Theta
```

```
def optimize_backprop_and_check (num_entradas, num_ocultas, num_etiquetas, reg, X,
y_onehot, laps, Y, ini):

    #Inicializamos los pesos y los ponemos en forma de lista
    Theta1 = RandomWeights(num_entradas, num_ocultas, ini)
    Theta2 = RandomWeights(num_ocultas, num_etiquetas, ini)
    pesos = np.concatenate((Theta1, Theta2), axis=None)

    #Optimizamos
    out = opt.minimize(fun = backprop, x0 = pesos, args = (num_entradas,
num_ocultas, num_etiquetas, X, y_onehot, reg), method='TNC', jac = True, options =
{'maxiter': laps})

    #Reconstruimos las Thetas
    Theta1 =
out.x[:(num_ocultas*(num_entradas+1))].reshape(num_ocultas,(num_entradas+1))
    Theta2 =
out.x[(num_ocultas*(num_entradas+1)):].reshape(num_etiquetas,(num_ocultas+1))

    m = X.shape[0]
    #Hacemos el forward propagation
    a1, z2, a2, z3, a3 = forward_propagation(X, Theta1, Theta2, m, y_onehot)

    #Sacamos los aciertos
    indexes = np.argmax(a3, axis=1)
    acc = (np.sum(indexes == (Y))/m)*100

    print("Porcentaje aciertos: ", acc)
    return acc
```

Punto de entrada (main)

```
def main():

    data = loadmat ("ex4data1.mat")

    #almacenamos los datos leídos en X e y
```

```
    X = data['X']
    m = X.shape[0]
    #X = np.hstack([np.ones([m, 1]), X])
    y = data['y']
    y = np.ravel(y)
    num_labels = 10
    num_entries = np.shape(X)[1]
    num_hiden_layers = 25

    disp.displayImage(X[4])

    #Ponemos la y en forma onehot
    y = y-1
    y_onehot = np.zeros((m, num_labels))  # 5000 x 10

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

    #Cargamos los pesos de prueba
    weights = loadmat("ex4weights.mat")
    theta1, theta2 = weights["Theta1"], weights["Theta2"]

    reg = 1
    a1, z2, a2, z3, a3 = forward_propagation(X, theta1, theta2, m, y_onehot)

    print("Coste no regularizado: ",cost_funct(y_onehot, a3, m))
    print("Coste regularizado: ",func_coste_reg([theta1, theta2], a3,y_onehot,
reg))

    #Check del gradiente con reg
    print(grad.checkNNGradients(backprop, reg))


    acc = optimize_backprop_and_check(num_entries, num_hiden_layers, num_labels,
reg, X, y_onehot, 70, y, 0.12)

    #Pruebas
    '''l = [0,0.05,0.1, 0.25, 0.5, 1,2 ,3 ,4, 5,10,20,100]
    g = [30, 50, 70, 100, 200, 300]
    for i in g:
        print("iteraciones: " ,i)
        for j in range (0, 5):
            acc = optimize_backprop_and_check(num_entries, num_hiden_layers,
num_labels, reg, X, y_onehot, i, y, 0.12)

    for i in l:
        print("Lambda: " ,i)
        for j in range (0, 5):
            acc = optimize_backprop_and_check(num_entries, num_hiden_layers,
num_labels, i, X, y_onehot, 70, y, 0.12)
    '''
```

```
    return acc

main()
```