Universidad Complutense de Madrid

IAAC - PRÁCTICA 4

Yaco Alejandro Santiago Pérez

Asignatura: INTELIGENCIA ARTIFICIAL APLICADA A INTERNET DE LAS COSAS

P4: Entrenamiento de redes neuronales

Master IOT

30 de marzo de 2020

Índice general

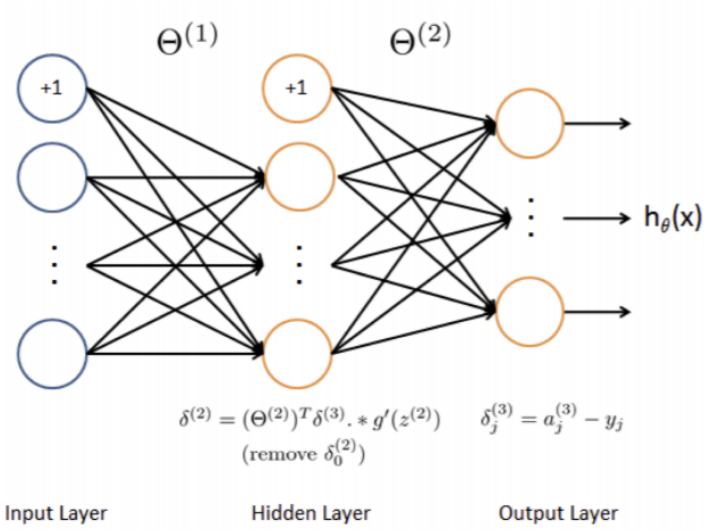
Capítulo 1

Introducción

Esta práctica consiste en el **entrenamiento de una red neuronal** encargada del reconocimiento de números manuscritos.

Figure 1.1: *Ejemplo de datos de entrenamiento*

Dicha red neuronal tendrá una **estructura** similar a la siguiente:



Capítulo 2

Objetivos

En esta práctica,los objetivos son los siguientes:

- Implementar la función de calculo del coste
- Modificar las funciones de manera que los resultados devueltos estén regularizados
- Comprobar los costes obtenidos
- Implementar la función de calculo del gradiente
- Implementar una función auxiliar que calcule la derivada de la función *sigmoide*
- Implementar el uso de las funciones para la *retro-propagación*
- Chequear que los valores devueltos son correctos, con la ayuda del archivo proporcionado por el profesor
- Entrenar a la red neuronal y obtener los valores para *theta1* y *theta2*

Capítulo 3

Parte 1: Implementación y Testeo

3.1. Funciones

Las funciones empleadas, en orden de llamada, son las siguientes:

- **backprop(params, num_entradas, num_ocultas, num_etiquetas, X, y, l):** Es la función encargada de hacer la retro-propagación, y devolver el coste y gradiente tras llamar a sus respectivas funciones para calcularlos.
- **cost(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, l):** Geneera *theta1* y *theta2* para calcular posteriormente la **h** que devuelve *forward_propagate(...)*. A continuación, realiza el calculo del coste con las siguientes formulas:

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

Figure 3.1: *Fórmula del coste **sin** regularización*

$$\begin{aligned} J(\theta) \quad = \quad & \frac{1}{m}\sum_{i=1}^{m}\sum_{k=1}^{K}\left[-y_k^{(i)}\log((h_\theta(x^{(i)}))_k) - (1-y_k^{(i)})\log(1-(h_\theta(x^{(i)}))_k)\right] + \\ & \frac{\lambda}{2m}\left[\sum_{j=1}^{25}\sum_{k=1}^{400}(\Theta_{j,k}^{(1)})^2 + \sum_{j=1}^{10}\sum_{k=1}^{25}(\Theta_{j,k}^{(2)})^2\right] \end{aligned}$$

Figure 3.2: *Fórmula del coste **con** regularización*

- **gradiente(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, l):** Función encargada de obtener el gradiente. Para ello, genera *theta1* y *theta2* y *delta1* y *delta2* con valores cero. A continuación llama a *forward_propagate(...)* para recuperar *a1, z2, a2, h* y realizar el cálculo.
- **forward_propagate(X, theta1, theta2):** Función que realiza la retro-propagación. Procesa los m ejemplos de entrenamiento,procesando cada ejemplo *(x(t), y(t))* siguiendo la siguiente formula:

$$\delta^{(2)} = (\Theta^{(2)})^T\delta^{(3)} .* g'(z^{(2)}) \qquad \delta_j^{(3)} = a_j^{(3)} - y_j$$
$$(\text{remove } \delta_0^{(2)})$$

Para luego calcular el gradiente dividiendo entre m los valores acumulados en el bucle. Siguiendo las siguientes formulas:

$$\frac{\partial}{\partial \Theta_{ij}^{(l)}} J(\Theta) = D_{ij}^{(l)} = \frac{1}{m} \Delta_{ij}^{(l)}$$

Figure 3.3: *Fórmula **sin** regularización*

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

Figure 3.4: *Fórmula **con** regularización*

- **sigmoid(x):** Función *sigmoide* tal y como la que hemos usado en las demás prácticas.
- **dSigmoid(z):** Derivada de la función *sigmoid(x)*:

$$g'(z) = \frac{d}{dz} g(z) = g(z)(1 - g(z))$$

- **pesosAleatorios(L_in, L_out):** Función que inicializa una matriz de pesos *theta* con valores aleatorios en el rango [-ini, ini]. Devuelve una matriz de dimensión (L_out, 1 + L_in).
- **min_coste(num_entradas, num_ocultas, num_etiquetas, X, y, reg):** Función encargada de calcular y devolver las *thetas* óptimas.
- **evaluar(h, y):** Evalúa cual es el porcentaje de acierto comparando los valores pronosticados contra los valores reales.
- **mainTest():** Función desde la cual se realizan las distintas llamadas y pruebas a la función de coste.
- **main():** Función desde la cual se realizan las llamadas para realizar el entrenamiento y ejecución de **backprop(...)**.

3.2. Ejecución de las pruebas

Se evalúan los siguientes valores obtenidos en las distintas ejecuciones.

3.2.1. Primera versión de la función de Coste

Esta función es la versión de la Función **sin** regularización.
Se debería devolver un valor aproximado de **0.287629**.

```
# valores iniciales
num_entradas = 400
num_ocultas = 25
num_etiquetas = 10
l = 1
X = np.hstack([np.ones((len(X), 1)), X])

encoder = OneHotEncoder(sparse=False, categories='auto')
y_cat = encoder.fit_transform(y)

theta_vec = np.concatenate((np.ravel(theta1), np.ravel(theta2)))
theta_vec = theta_vec.reshape((len(theta_vec), 1))
print("COSTE:")
print(cost(theta_vec,num_entradas, num_ocultas, num_etiquetas, X,↩
    y_cat, l))
print("---")
```

El resultado de la ejecución es **correcto**:

```
COSTE:
0.2876291651613187
---
```

3.2.2. Segunda versión de la función de Coste: Con regularización

El coste regularizado debería estar en torno a **0.383770**.
Para el calculo regularizado se anyade la regularización, traducida en la siguiente línea en la función de **Coste**:

```
coste += (float(l) / (2 * m)) * (np.sum(np.power(theta1[:,1:], 2)↩
   ) + np.sum(np.power(theta2[:,1:], 2)))
```

De esta manera con la misma ejecución se obtiene el resultado **correcto**:

```
COSTE:
0.3837698590909234
---
```

3.2.3. Testeo de backprop

Se ha proporcionado el fichero **checkNNGradients.py** que contiene una función que aproxima el valor de la derivada por este método.donde se construye una pequeña red neuronal y se aplican dos métodos de cálculo del gradiente: El numérico y la ejecución del que he implementado en la función **backprop**.
Si el gradiente está implementado correctamente, la diferencia debería ser menor de 10^{-9}.

Esta comprobación se realiza tanto sin regularización como con regularización.Mediante la siguiente llamada:

```
a = checkNNGradients(backprop, 0)
print(a)
```

En el caso de una implementación **errónea** de la función se mostraría un mensaje similar a este indicando un 47.4 % de diferencias:

```
Arrays are not almost equal to 7 decimals

Mismatch: 47.4%
Max absolute difference: 1.57334392
Max relative difference: 4.
 x: array([ 8.2208482e-02,  1.5659428e-03,  9.8839906e-03,  ↩
    9.1147431e-03,
       -8.0478877e-02,  1.9358252e-02, -3.3831887e-02, -5.5917146↩
          e-02,
       -1.3878659e-01,  2.7483267e-02, -6.7818432e-02, -1.0076818↩
          e-01,...
 y: array([ 0.0822085,  0.0015659,  0.009884 ,  0.0091147, ↩
    -0.0804789,
        0.0193583, -0.0338319, -0.0559171, -0.1387866,  ↩
           0.0274833,
       -0.0678184, -0.1007682, -0.0840919,  0.0065438, ↩
          -0.0259151,...
```

Tras mejoras, con la implementación correcta de la función **backprop** se obtiene el siguiente *array* al pintar lo que devuelve la llamada a **checkNNGradients(...)**:

```
[ 5.27761168e-11 -2.55029743e-12  3.45235299e-12  6.95584909e-12
 -6.52669585e-11  2.08456863e-12 -1.51965315e-11 -4.38069025e-11
 -9.07785513e-11  9.26888080e-12 -3.98116679e-11 -1.22385352e-10
 -2.17855040e-11  2.76547969e-12 -6.02735570e-12 -2.49761462e-11
  2.15736526e-11 -4.96176017e-13  1.19978506e-11  2.73879391e-11
  6.03760375e-11  1.55131741e-11  6.81166235e-12  5.26763355e-12
  1.90088223e-11  1.88441207e-11  7.15513759e-11  1.56080426e-11
  4.89146224e-12  1.37491546e-11  1.70987668e-11  1.79336823e-11
  7.32915950e-11  1.60134683e-11  8.61832827e-12  1.33683065e-11
  1.43913215e-11  2.26750840e-11]
```

Capítulo 4

Parte 2: Entrenamiento y ejecución

En esta parte de la práctica, concluida la implementación de la función backprop, usaré la función **scipy.optimize.minimize** para entrenar a la red neuronal y obtener los valores para *theta1* y *theta2*.

4.1. Entrenamiento

4.2. Ejecución

Entrenando a la red mediante **min_coste(...)** con 70 iteraciones y un valor de *lamda*=1 debería obtener una precisión en torno al **93 %**.
Para obtener el porcentaje de acierto voy a utilizar la función **evaluar(h, y)**:

```
    t1, t2= min_coste(num_entradas, num_ocultas, num_etiquetas, X↩
       , y_cat, l)

    X = np.hstack([np.ones((len(X), 1)), X])
    a1, z2, a2, z3, h = forward_propagate(X, t1, t2)
    yPred = np.array(np.argmax(h, axis=1) + 1)
    evaluar(yPred, y)
```

Usándose **minimize(...)** de la siguiente manera en **min_coste(...)**:

```
    result = minimize(fun=backprop, x0=params_rn, args=( ↩
       num_entradas, num_ocultas, num_etiquetas, X, y,reg), ↩
       method='TNC', jac=True, options={'maxiter':70})
```

Se espera obtener un 93 % (puede variar hasta un 1 %) y, efectivamente, se obtiene **correctamente**:

```
Acierta el 92.67999999999999 %
```

4.3. Ejecuciones adicionales

Probando diferentes configuraciones, obtengo distintos porcentajes de acierto:

Variando el valor de *Lamnda* entre 0,01 y 10, manteniendo las 70 iteraciones no he conseguido obtener cambios significativos, ya que el porcentaje de aciertos se mantenía dentro de los valores esperados, entre **92 %** y **94 %**.

Para un entrenamiento de **70 iteraciones** y *Lambda* = 100 obtengo:

```
Acierta el 82.89999999999999 %
```

Por lo que se interpreta que por aumentar *Lambda*, poco o mucho, no se consigue ninguna mejora.

Ejecutando para el entrenamiento **170 vueltas**, en vez de 70, se obtiene:

```
Acierta el 97.39999999999999 %
```

Lo que supone un incremento importante.

Si se realiza una ejecución de 170 vueltas y una *Lamnda*=0.01

```
Acierta el 99.44 %
```

Ese **99.44 %** supone un resultado prácticamente perfecto.

Capítulo 5

Código

```
import numpy as np
import copy
from scipy.io import loadmat
import matplotlib.pyplot as plt
from sklearn.preprocessing import OneHotEncoder
from scipy.optimize import minimize
import math

from checkNNGradients import checkNNGradients
from displayData import displayData


def backprop(params, num_entradas, num_ocultas, num_etiquetas, X, y, l):
    print("Entra en backprop")
    m = X.shape[0]
    X = np.hstack((np.ones((m, 1)), X))
    y = np.matrix(y)
    coste = 0
    grad= gradiente(params, num_entradas, num_ocultas, num_etiquetas, X, y, l)
    coste= cost(params, num_entradas, num_ocultas, num_etiquetas, X, y, l)
    return coste, grad

def cost(params_rn, num_entradas, num_ocultas, num_etiquetas, X, y, l):
    m = X.shape[0]
    theta1 = params_rn[0:(num_ocultas * (num_entradas + 1))].reshape(num_ocultas, (num_entradas + 1))
    theta2 = params_rn[(num_ocultas * (num_entradas + 1)):].reshape(num_etiquetas, (num_ocultas + 1))
```

```
27
28      a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2) #↩
           Para calcular la h
29      coste=0
30      for i in range(m):
31          coste1 = np.multiply(-y[i,:], np.log(h[i,:]))
32          coste2 = np.multiply((1 - y[i,:]), np.log(1 - h[i,:]))
33          coste += np.sum(coste1 - coste2)
34
35      coste = coste / m
36      #Termino de regularizacion
37      coste += (float(l) / (2 * m)) * (np.sum(np.power(theta1↩
           [:,1:], 2)) + np.sum(np.power(theta2[:,1:], 2)))
38      return coste
39
40
41  def gradiente(params_rn, num_entradas, num_ocultas, num_etiquetas↩
       , X, y, l):
42      theta1 = np.reshape(params_rn[:num_ocultas * (num_entradas + ↩
           1)], (num_ocultas, (num_entradas + 1)))
43      theta2 = np.reshape(params_rn[num_ocultas * (num_entradas + ↩
           1):], (num_etiquetas, (num_ocultas + 1)))
44      delta1 = np.zeros((num_ocultas, num_entradas + 1))
45      delta2 = np.zeros((num_etiquetas, num_ocultas + 1))
46      m = X.shape[0]
47      a1, z2, a2, z3, h = forward_propagate(X, theta1, theta2)
48      for t in range(m):
49          a1t = a1[t,:]
50          z2t = z2[t,:]
51          a2t = a2[t,:]
52          ht = h[t,:]
53          yt = y[t,:]
54
55          d3t = ht - yt
56
57          z2t = np.insert(z2t, 0, values=np.ones(1))
58          d2t = np.multiply((theta2.T * d3t.T).T, dSigmoid(z2t))
59
60          delta1 = delta1 + (d2t[:,1:]).T * a1t
61          delta2 = delta2 + d3t.T * a2t
62
63      delta1 = delta1 / m
64      delta2 = delta2 / m
65
66      # anyade el termino de regularizacion
```

```
67      delta1[:,1:] = delta1[:,1:] + (theta1[:,1:] * l) / m
68      delta2[:,1:] = delta2[:,1:] + (theta2[:,1:] * l) / m
69
70      grad = np.concatenate((np.ravel(delta1), np.ravel(delta2)))
71      return grad
72
73  def forward_propagate(X, theta1, theta2):
74      m = X.shape[0]
75      a1 = X
76      z2 = a1.dot(theta1.T)
77      a2 = np.insert(sigmoid(z2), 0, values = np.ones(m), axis = 1)
78      z3 = a2.dot(theta2.T)
79      h = sigmoid(z3) # = a3 = g(z3)
80
81      return a1, z2, a2, z3, h
82  def sigmoid(x):
83      s = 1 / (1 + np.exp(-x))
84      return s
85
86  def dSigmoid(z):
87      return np.multiply(sigmoid(z), (1 - sigmoid(z)))
88
89  def pesosAleatorios(L_in, L_out):
90    e = math.sqrt(6) / math.sqrt(L_in + L_out)
91    pesos = 2 * e * np.random.rand(L_out, L_in + 1) - e
92    return pesos
93
94  def min_coste(num_entradas, num_ocultas, num_etiquetas, X, y, reg↩
      ):
95      initialTheta1 = pesosAleatorios(num_entradas, num_ocultas)
96      initialTheta2 = pesosAleatorios(num_ocultas, num_etiquetas)
97      params_rn = np.concatenate((initialTheta1.ravel(), ↩
          initialTheta2.ravel()))
98      params = (np.random.random(size=num_ocultas * (num_entradas +↩
           1) + num_etiquetas * (num_ocultas + 1)) - 0.5) * 0.25
99
100     result = minimize(fun=backprop, x0=params_rn, args=( ↩
          num_entradas, num_ocultas, num_etiquetas, X, y,reg), ↩
          method='TNC', jac=True, options={'maxiter':70})
101     print(result)
102     theta1 = np.reshape(result.x[:num_ocultas * (num_entradas + ↩
          1)], (num_ocultas, (num_entradas + 1)))
103     theta2 = np.reshape(result.x[num_ocultas * (num_entradas + 1)↩
          :], (num_etiquetas, (num_ocultas + 1)))
104     return (theta1, theta2)
```

```
105
106 def evaluar(h, y):
107     correct = [1 if a == b else 0 for (a, b) in zip(h, y)]
108     accuracy = (sum(map(int, correct)) / float(len(correct)))
109     print('Acierta␣el␣'+str(accuracy * 100)+'%')
110
111
112 def main():
113     weights = loadmat('ex4weights.mat')
114     theta1, theta2 = weights['Theta1'], weights['Theta2']#Theta1 ↩
        dimensi n 25x401 ; #Theta2 dimensi n 10x26
115     data = loadmat ('ex4data1.mat')
116     y = data ['y']
117     X = data ['X']
118     # valores iniciales
119     num_entradas = 400
120     num_ocultas = 25
121     num_etiquetas = 10
122     l = 1
123
124     encoder = OneHotEncoder(sparse=False, categories='auto')
125     y_cat = encoder.fit_transform(y)
126     t1, t2= min_coste(num_entradas, num_ocultas, num_etiquetas, X↩
        , y_cat, l)
127
128     X = np.hstack([np.ones((len(X), 1)), X])
129     a1, z2, a2, z3, h = forward_propagate(X, t1, t2)
130     yPred = np.array(np.argmax(h, axis=1) + 1)
131     evaluar(yPred, y)
132
133
134 def mainTest():
135     weights = loadmat('ex4weights.mat')
136     theta1, theta2 = weights['Theta1'], weights['Theta2']#Theta1 ↩
        dimensi n 25x401 ; #Theta2 dimensi n 10x26
137     data = loadmat ('ex4data1.mat')
138     y = data ['y']
139     X = data ['X']
140     # valores iniciales
141     num_entradas = 400
142     num_ocultas = 25
143     num_etiquetas = 10
144     l = 1
145
146     X = np.hstack([np.ones((len(X), 1)), X])
```

```
147     encoder = OneHotEncoder(sparse=False, categories='auto')
148     y_cat = encoder.fit_transform(y)
149 
150     theta_vec = np.concatenate((np.ravel(theta1), np.ravel(theta2↩
            )))
151     theta_vec = theta_vec.reshape((len(theta_vec), 1))
152     print("COSTE:")
153     print(cost(theta_vec,num_entradas, num_ocultas, num_etiquetas↩
            , X, y_cat, 1))
154     print("---")
155 
156 
157 #mainTest()
158 #a = checkNNGradients(backprop, 0)
159 #print(a)
160 main()
```

Código: p4.py