Aprendizaje Automático y Big Data, Práctica 4

Romain Contrain y Guillermo Martín Sánchez

Abril 2020

1 Objetivo

El objetivo de esta práctica es implementar el aprendizaje de una red neuronal para clasificación multiclase. Nuestra arquitectura consiste de tres capas, como se muestra en la Figura 1. Por lo tanto, buscamos las matrices de pesos $\Theta^{(1)}$ y $\Theta^{(2)}$ tales que se minimice la función coste regularizada:

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

El cálculo de $h_\theta(x)$ se hace mediante el método de *forward propagation* representado en la Figura 1.

En este caso, vamos a intentar discernir unos dígitos escritos a mano y clasificarlos en los posibles valores: $0, 1, ..., 9$. Estos dígitos están escritos en una imagen de 20×20 píxeles, por lo que $x^{(i)}$ es un vector con 400 elementos (la linearización de la imagen) que representan la intensidad en cada píxel. Por tanto tenemos un número de 400 neuronas en la capa de entrada y 10 neuronas (una por cada posible clasificación) en la de salida. En la capa oculta disponemos de 25 neuronas.

2 Método

Hemos usado la función *minimize* con método *TNC* de la librería *scipy.optimize* para calcular el mínimo de la función de coste regularizada (Eq 1).

Para ello hemos programado una función *backprop* que nos calcula el coste regularizado (Eq 1) y el gradiente de $\Theta^{(1)}$ y $\Theta^{(2)}$ mediante el método de *back-propagation* regularizado.

Inicializando los valores de $\Theta^{(1)}$ y $\Theta^{(2)}$ de forma aleatoria entre $[-\epsilon_{ini}, \epsilon_{ini}]$ con $\epsilon_{ini} = 0.12$ para asegurar una rápida convergencia.

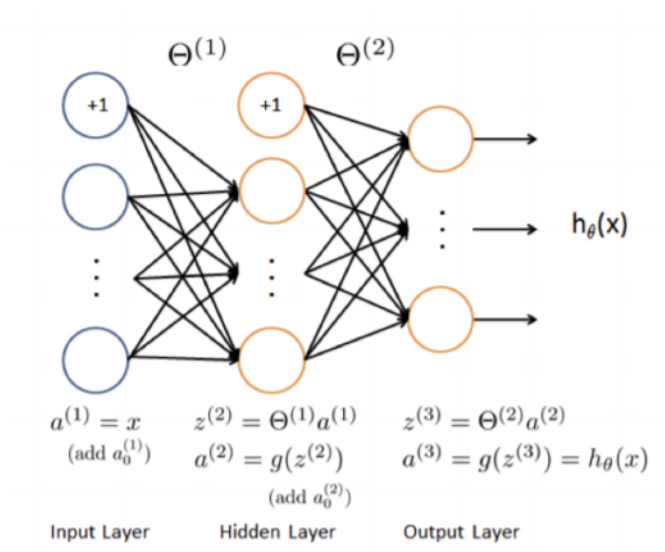

Figure 1: Red neuronal usada. Fórumla de *forward propagation*

El $\Theta^{(1)}$ y $\Theta^{(2)}$ óptimos obtenidos se han usado en una función del cálculo de precisión. Para determinar a qué clase pertenece un ejemplo concreto $x^{(j)}$, calculamos $h_\theta(x^{(j)})$ que nos da la activación de las 10 neuronas de la capa de salida. Así, clasificamos el ejemplo como el número de la neurona más activada en esta capa. De esta manera, usando los ejemplos de entrenamiento, calculamos cuántos ejemplos han sido clasificados correctamente mediante nuestra red neuronal y dividimos por el número de ejemplos para calcular la precisión.

Vemos luego cómo cambia esta precisión si aumentamos el parámetro de regularización λ y si aumentamos el número de iteraciones del método *minimize*.

3 Resultados

Con parámetro de regularización $\lambda = 1$ y 70 iteraciones obtenemos una precisión del 0.921.

Si aumentamos λ vemos que la red neuronal sufre de $underfitting$ y por tanto perdemos precisión. Para $\lambda = 100$ obtenemos una precisión de 0.809.

Por otro lado, si aumentamos el número de iteraciones, obtenemos que la red neuronal se ajusta más y más a los ejemplos de entrenamiento y la precisión aumenta. Para 700 iteraciones obtenemos una precisión de 0.996. Hay que tener en cuenta que esta no es la precisión "real" puesto que se está calculando con respecto a los datos de entrenamiento y por tanto un sobreajuste a ellos (que se obtiene con un gran número de iteraciones) probablemente resulte en

overfitting y una menor precisión para datos de test.

4 Conclusiones

El entrenamiento de redes neuronales se realiza de manera estándar: mediante la minimización de una función de coste. El cálculo del gradiente de los pesos se realiza mediante *backpropagation* pero una vez programado esto la minimización de la función coste se hace igual que en los otros métodos de aprendizaje automático. Hemos visto también como afecta el parámetro de regularización y el número de iteraciones a la precisión y hemos concluido que el uso de los datos de entrenamiento para el cálculo de la precisión no notifica del *overfitting* o una peor precisión para datos nuevos.

5 Código

```
"""
Coursework 4: Neural network training
"""

#%%
"""
Imports and definitions
"""


import numpy as np
from scipy.io import loadmat
import matplotlib.pyplot as plt
import scipy.optimize as opt
import time

def load_dat(file_name):
    """
    load the file dat (Matlab) specified
    and return it as an numpy array
    """
    data = loadmat(file_name)
    y = data['y']
    X = data['X']
    return X,y

def sigmoid(z):
    """
    sigmoid function
```

```
    can be applied on numbers and on numpy arrays of any dimensions
    """
    return 1 / (1 + np.exp(-z))

def sigmoid_prime(z):
    """
    derivative of a sigmoid function
    can be applied on numbers and on numpy arrays of any dimensions
    """
    gz = sigmoid(z)
    return gz*(1-gz)

def propagation(theta1,theta2,X):
    """
    forward propagation function
    performs the forward propagation in a neural network
    with the stucture decribed in the subject,
    with weights given by theta1 and theta2
    on the dataset X
    returns :
        -H : the activation of the output layer
        -a2 : the activation of the hidden layer
    """
    m = np.shape(X)[0]
    z2 = (np.dot(X,theta1.T))
    a2 = sigmoid(z2)
    a2 = np.hstack([np.ones([m,1]),a2])
    H = sigmoid(np.dot(a2,theta2.T))
    return H, a2

def coste_reg(theta1, theta2, X, Y, lamb):
    """
    cost function with regularization
    computes J for a given neural network,
    described by the weights matrices theta1 and theta2
    on a given dataset, described by X and Y
    with a regularization factor of lamb
    """
    m = np.shape(X)[0]
    H, _ = propagation(theta1,theta2, X)
    J = -1/m * np.sum( np.log(H)*Y
                + np.log(1-H)*(1-Y))
    reg = lamb/(2*m)*(np.sum(theta1[:,1:]**2) + np.sum(theta2[:,1:]**2))
    return J + reg

def gradiente_reg(theta1, theta2, X, Y, lamb):
```

```
    """
    computes the gradient of the cost function
    using backpropagation
    for a neural network described by theta1 and theta2
    on a given dataset described by X and Y
    with a regularization factor of lamb
    returns :
        -d1 : the gradients for the weights of theta1
        -d2 : the gradient for the weights of theta2
    """
    m = np.shape(X)[0]

    a3, a2 = propagation(theta1, theta2, X)

    delta3 = a3 - Y
    delta2 = delta3.dot(theta2)*(a2*(1-a2))
    DELTA1 = delta2[:,1:].T.dot(X)
    DELTA2 = delta3.T.dot(a2)
    reg1 = np.zeros_like(theta1)
    reg1[:,1:] = lamb / m * theta1[:,1:]
    reg2 = np.zeros_like(theta2)
    reg2[:,1:] = lamb / m * theta2[:,1:]
    return DELTA1/m + reg1, DELTA2/m + reg2

def backprop(params_rn, num_entradas, num_ocultas, num_etiquetas , X, y,  reg) :
    """
    computes the cost function and its gradient
    for a given neural network having 3 layers
    and for a given dataset, while performing
    regularization
    parameters :
        -params_rn : vector of the weights of the network
        -num_entradas : size of the input layer
        -num_ocultas : size of the hidden layer
        -num_etiquetas : size of the output layer
        -X : matrix containing the training examples as vectors in its lines
        -y : matrix containing the labels as vectors in its lines
        -reg : regularization factor
    returns :
        -coste : the value of the cost function
        -grad : the gradient as a vector
    """
    # Add the column of 1s
    m = np.shape(X)[0]
    X = np.hstack([np.ones([m,1]),X])
```

```
    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)))

    coste = coste_reg(theta1, theta2, X, y, reg)
    d1, d2 = gradiente_reg(theta1, theta2, X, y, reg)
    grad = np.concatenate([d1.ravel() , d2.ravel()])
    return coste, grad

def pesosAleatorios(L_in, L_out) :
    """
    returns a matrix of random floats
    of shape (L_out, L_in + 1)
    the values belong in interval [-0.12,0.12]
    """
    eps = 0.12
    theta = np.random.rand(L_out, L_in + 1)
    theta = (theta - 0.5)*2*eps
    return theta

def precision_neur(theta1,theta2,X,Y):
    """
    accuracy function
    computes the accuracy of the neural network described by theta1 and theta2
    on dataset X with true target variable Y
    """
    m = np.shape(X)[0]
    X_new = np.hstack([np.ones([m,1]),X])
    H,_ = propagation(theta1,theta2,X_new)
    labels = np.argmax(H,axis = 1) + 1
    Y = Y.ravel()
    return np.sum(labels == Y)/m


#%%
"""
Testing the backprop function on the provided network
"""


# Load the data
X,y = load_dat('ex4data1.mat')
m = np.shape(X)[0]
weights = loadmat('ex4weights.mat')
```

```
theta1 ,  theta2 = weights ['Theta1'], weights['Theta2']

# Reshape the weights
num_entradas = np.shape(theta1)[1] -1
num_ocultas = np.shape(theta2)[1] -1
num_etiquetas = np.shape(theta2)[0]
params_rn = np.concatenate([theta1.ravel() , theta2.ravel()])

# Transform y into a usable matrix
labels = np.zeros((m,num_etiquetas))
for i in range(10) :
    labels[(y == i+1).ravel(), i] = 1;

# Calculate the cost and gradient for testing purposes
c, grad = backprop(params_rn , num_entradas, num_ocultas, num_etiquetas, X, labels, 1)


#%%
"""
Training a neural network
"""

# Initializing the data and parameters
X,y = load_dat('ex4data1.mat')
m = np.shape(X)[0]
theta1 = pesosAleatorios(400,25)
theta2 = pesosAleatorios(25,10)
reg = 1
maxIter = 70

# Reshape the weights
num_entradas = np.shape(theta1)[1] -1
num_ocultas = np.shape(theta2)[1] -1
num_etiquetas = np.shape(theta2)[0]
params_rn = np.concatenate([theta1.ravel() , theta2.ravel()])

# Transform y into a usable matrix
labels = np.zeros((m,num_etiquetas))
for i in range(10) :
    labels[(y == i+1).ravel(), i] = 1;

# Train the network
print("Training a network with lambda = " + str(reg)
        + " for " + str(maxIter) + " iterations")
before = time.process_time()
trainingresult = opt.minimize(fun=backprop, x0=params_rn,
```

```
                        args=(num_entradas, num_ocultas, num_etiquetas, X, labels, reg),
                        method='TNC', jac=True,
                        options={'maxiter': maxIter})
after = time.process_time()
print("training time : " + str(after-before))
params_rn_result = trainingresult.x

# Compute the precision
theta1_res = np.reshape(params_rn_result[:num_ocultas*(num_entradas + 1)] ,
        (num_ocultas, (num_entradas + 1)))
theta2_res = np.reshape(params_rn_result[num_ocultas*(num_entradas + 1):] ,
        (num_etiquetas, (num_ocultas + 1)))
precision = precision_neur(theta1_res,theta2_res,X,y)
print("Precision : " + str(precision))
```