Aprendizaje Automático

Redes Neuronales Multicapa

Curso 2019-2020

Roberto Paredes, Francisco Casacuberta, Enrique Vidal, Jorge Civera
Departamento de Sistemas Informáticos y Computación
Universitat Politècnica de València

1. Introducción

El objetivo de esta práctica es experimentar con redes neuronales multicapa dentro de `octave`, en concreto con redes *Multilayer Perceptron* (MLP). Para ello utilizaremos la librería `nnet` para `octave` que se puede descargar en:

`http://octave.sourceforge.net/nnet`

La instalación es muy sencilla. Simplemente se necesita disponer de los ficheros que definen las funciones requeridas por la librería en el directorio de trabajo o en el `PATH`. Se ha dejado disponible el fichero `nnet_apr.tgz` que contiene dos directorios:

1. El directorio `data` contiene los conjuntos de datos que usaremos para realizar los distintos experimentos.
2. El directorio `nnet` contiene todos los ficheros de la librería de redes neuronales.

Por tanto, para utilizar dicha librería se aconseja disponer de ambas carpetas en el directorio de trabajo y simplemente añadir la ruta de la librería `nnet` al `PATH` desde `Octave` mediante el comando:

```
octave:1> addpath("nnet");
```

Nótese que la ruta anterior es relativa al directorio de trabajo.

2. Ejemplo de entrenamiento y clasificación con `MLP`

Para ilustrar el funcionamiento de la librería `MLP` utilizaremos el corpus `hart` que ya hemos visto en la práctica anterior. Seguidamente, definiremos los conjuntos de entrenamiento, validación y test, y su preproceso para el entrenamiento, clasificación y evaluación.

2.1. Carga de datos

Como recordarás, el corpus `hart` es un problema de clasificación binaria de datos bidimensionales disponible en el directorio `data/hart`.

Los comandos utilizados para realizar la carga de los datos son los mismos que en la práctica anterior:

```
octave:2> load data/hart/tr.dat;
octave:3> load data/hart/trlabels.dat;
octave:4> load data/hart/ts.dat;
octave:5> load data/hart/tslabels.dat;
```

Es importante hacer notar que la librería `nnet` trabaja con los datos por columnas en lugar de por filas, por lo que es necesario trasponer los datos:

```
octave:6> mInput = tr';
octave:7> mOutput = trlabels';
octave:8> mTestInput = ts';
octave:9> mTestOutput = tslabels';
```

2.2. Conjuntos de entrenamiento y validación

El entrenamiento de `MLP` tiene dos posibles criterios de parada: entrenar hasta un número máximo de *epochs* o entrenar hasta que el error de clasficación de un conjunto de validación no mejore. Normalmente se suele utilizar el segundo criterio, por lo que debemos dedicar una parte del conjunto de entrenamiento a conjunto de validación.

Con las siguientes órdenes obtenemos el número de muestras (`nSamples`) y nos quedaremos con el 80 % de las mismas para entrenamiento y el 20 % restante para validación:

```
octave:10> [nFeat, nSamples] = size(mInput);
octave:11> nTr=floor(nSamples*0.8);
octave:12> nVal=nSamples-nTr;
```

Queremos que la elección de qué muestras forman parte de cada conjunto sea aleatoria. Para ello utilizamos la función `randperm(n)` que devuelve una permutación aleatoria de elementos desde 1 hasta n:

```
octave:13> rand('seed',23);
octave:14> indices=randperm(nSamples);
```

Este vector `indices` aplicado sobre las muestras (columnas del conjunto `mInput`) y sobre el vector de etiquetas de clase produce un barajado aleatorio. Seguidamente, se toman las `nTr` primeras muestras para entrenamiento, y las restantes para validación:

```
octave:15> mTrainInput=mInput(:,indices(1:nTr));
octave:16> mTrainOutput=mOutput(indices(1:nTr));
octave:17> mValiInput=mInput(:,indices((nTr+1):nSamples));
octave:18> mValiOutput=mOutput(indices((nTr+1):nSamples));
```

2.3. Preproceso y formato de los datos

Dado un problema de clasificación en C clases, la capa de salida de la `MLP` suele tener C neuronas de forma que la neurona i-ésima se activa para la clase i. Por ejemplo, en un problema de clasificación en 3 clases, éstas se codificarían como $[0, 0, 1]$, $[0, 1, 0]$ y $[1, 0, 0]$, lo que es conocido como *one-hot encoding*. Por ello, debemos transformar las etiquetas de clase al formato correspondiente en forma de vector.

Ejercicio: Implementa en `octave` la codificación de las etiquetas de clase en el formato requerido por la librería `MLP`.

Una vez disponemos de los datos de entrenamiento y validación, solo quedan realizar un par de pasos más. En primer lugar, es conveniente normalizar los datos de modo que tengan media cero y desviación típica uno. Esto se consigue con el comando:

```
octave:26> [mTrainInputN,cMeanInput,cStdInput] = prestd(mTrainInput);
```

tal que `mTrainInputN` son los datos normalizados, y `cMeanInput` y `cStdInput` son la media y desviación típica de los datos originales, respectivamente.

Por último, el conjunto de validación debe representarse en una estructura especial con dos campos: el campo `P` para los datos de entrada y el campo `T` para la salida, siendo `validoutDisp` las etiquetas de clase del conjunto de validación en el formato previamente descrito:

```
octave:27> VV.P = mValiInput;
octave:28> VV.T = validoutDisp;
```

También es necesario normalizar los datos del conjunto de validación utilizando la media y desviación típica del conjunto de entrenamiento:

```
octave:29> VV.P = trastd(VV.P,cMeanInput,cStdInput);
```

2.4. Entrenamiento

Antes entrenar una red neuronal debemos especificarla mediante la siguiente función:

```
net = newff (Pr,ss,trf,btf,blf,pf)
```

siendo

- `Pr`: una matriz $D \times 2$ con los valores máximo y mínimo de los datos de entrenamiento en cada dimensión. Se puede calcular mediante la función `min_max`.
- `ss`: un vector fila con el número de neuronas en cada capa oculta y en la capa de salida.
- `trf`: la lista de funciones de activación de cada capa.
- `btf`: el algoritmo de entrenamiento de la red neuronal. En la versión actual, el único algoritmo implementado es `trainlm` que es el algoritmo *backpropagation*.
- `blf`: un parámetro que no se utiliza en la versión actual.

- `pf`: la función objetivo a minimizar. En la versión actual, la única función objetivo es `mse` que es el error cuadrático medio.

Una posible configuración para una topología de una capa oculta de `nHidden` neuronas y con `nOutput` neuronas en la capa de salida, y funciones de activación `tansig` $\in [-1, 1]$ y `logsig` $\in [0, 1]$ para la capa oculta y de salida, respetivamente, la crearíamos mediante:

```
octave:30> MLPnet = newff(min_max(mTrainInputN),[nHidden nOutput],\
                          {"tansig","logsig"},"trainlm","","mse");
```

Con la red creada en `MLPnet` podemos establecer algunos parámetros de entrenamiento:

```
octave:31> MLPnet.trainParam.show = 10;
octave:32> MLPnet.trainParam.epochs = 300;
```

donde el parámetro `show` indica cada cuántas *epochs* de entrenamiento queremos que se imprima información, y el parámetro `epochs` es el número máximo de epochs que queremos que realice, adicionalmente al criterio de parada del conjunto de validación.

Finalmente, podemos entrenar la red siendo `trainoutDisp` las etiquetas de clase del conjunto de entrenamiento en *one-hot encoding*:

```
octave:32> net = train(MLPnet,mTrainInputN,trainoutDisp,[],[],VV);
TRAINLM, Epoch 0/300, MSE 0.378153/0, Gradient 149.445/1e-10
TRAINLM, Epoch 10/300, MSE 0.0798137/0, Gradient 7.6783/1e-10
TRAINLM, Epoch 20/300, MSE 0.0672102/0, Gradient 12.3743/1e-10
TRAINLM, Epoch 30/300, MSE 0.0316977/0, Gradient 4.10449/1e-10
TRAINLM, Validation stop.
```

De esta forma obtenemos en `net` la red entrenada. Además, se mostrará una gráfica con la evolución del error. Al principio del entrenamiento aparecerán avisos (*warnings*) que pueden ser ignorados.

2.5. Clasificación y estimación de la tasa de acierto

Antes de clasificar el conjunto de test con la red neuronal entrenada en la etapa anterior (variable `net`), debemos aplicar la normalización a los datos del conjunto de test como hemos hecho con los conjuntos de entrenamiento y validación:

```
octave:33> mTestInputN = trastd(mTestInput,cMeanInput,cStdInput);
```

y ya podemos clasificar los datos mediante la orden:

```
octave:34> simOut = sim(net,mTestInputN);
```

A diferencia de otras librerías, no obtenemos el porcentaje de error o precisión, sino que obtenemos los valores de salida de la red para cada clase (por filas) y para cada muestra (por columnas). A continuación mostramos el valor de la variable `simOut` para las 5 primeras muestras de test de la tarea de clasificación binaria del corpus `hart` :

```
simOut =

   2.3448e-12   1.7738e-06   6.0816e-08   9.9966e-01   8.4825e-05
   1.0000e+00   1.0000e+00   1.0000e+00   1.7935e-04   9.9992e-01
```

Por tanto, para realizar la estimación de la clase por cada muestra de test, debemos calcular la clase con mayor puntuación en cada muestra, comparar esta etiqueta estimada con la etiqueta real y estimar el error de clasificación correspondiente.

Ejercicio: Implementa una función que reciba la salida de la `MLP` para el conjunto de test (`simOut`) y sus etiquetas (`tslabels`) y devuelva el porcentaje de error de clasificación.

Es muy recomendable integrar el proceso descrito de definición de conjuntos de entrenamiento y validación, preproceso y formato de datos, entrenamiento, clasificación y estimación del error, en una función que tome como entrada los datos de entrenamiento y test junto con los parámetros de entrenamiento de la red neuronal y devuelva la tasa de error conseguida:

```
function err = nnexp(tr,trlabels,ts,tslabels,nhidden,...)
```

Esta función puede ser invocada desde un script que tome los parámetros de entrada que proporcione el usuario, cargue los datos y realice cualquier preproceso previo (reducción de dimensionalidad PCA).

Pista: Las tasas de error en la tarea hart (80 % entrenamiento, 20 % validación) utilizando 1, 2 y 5 neuronas en la capa oculta con función `tansig`*, y en la capa de salida con función* `logsig` *son 14.90 %, 11.30 % y 2.40 %, respectivamente.*

3. Redes neuronales en MNIST (a entregar)

Como se puede observar en la web de MNIST[1] la utilización de redes neuronales puede llegar a proporcionar los mejores resultados en esta tarea. Sin embargo, por cuestiones de eficiencia, la librería `nnet` no permite entrenar redes neuronales con un elevado número de muestras cuyos vectores de características de entrada tengan un número elevado de dimensiones, o una configuración de red con un elevado número de neuronas en la capa oculta.

Se propone la realización de una serie de experimentos de clasificación para determinar los parámetros que minimicen el error de clasificación sobre el conjunto de test. Se deberá presentar una tabla y/o gráfica donde se muestre no solo el mejor resultado obtenido, sino también otros resultados relevantes que permitan poner de manifiesto la tendencia a mejorar o empeorar del modelo según varían los parámetros considerados.

Algunos de los parámetros a considerar en la configuración de la red neuronal es el número de capas ocultas y el número de neuronas en cada capa. Asimismo, dadas las limitaciones computacionales de la librería `nnet` mencionadas anteriormente, será necesario la proyección previa de los datos con PCA a una dimensionalidad reducida que permita realizar el entrenamiento de la red.

Otra estrategia posible a explorar en la reducción del coste computacional que nos permitirá entrenar redes neuronales más complejas es la reducción del número de muestras en el conjunto de entrenamiento. Es decir, es posible seleccionar un subconjunto del conjunto de entrenamiento de acuerdo a algún criterio (i.e. aleatorio) para entrenar una

[1] http://yann.lecun.com/exdb/mnist

red más compleja. Esta estrategia posibilita la combinación de clasificadores entrenados en subconjuntos del conjunto de entrenamiento que es propia de las técnicas de *Bagging*.

Recordad que toda estimación de la probabilidad de error de un clasificador, debe ir acompañada de sus correspondientes intervalos de confianza al 95 %.