Práctica 10

Redes Neuronales

En esta práctica trabajaremos con un sistema de aprendizaje basado en ejemplos que ya hemos visto con anterioridad (k-vecinos) y una implementación de las redes neuronales.

1 Redes neuronales

Aunque son muchos los toolboxes de redes neuronales para MATLAB que se pueden encontrar por Internet, para esta práctica utilizaremos el toolbox de neuronales que viene incorporado en la instalación de MATLAB (Neural Network Toolbox).

1.1 Aprendiendo la función XOR

Para comprender mejor las redes neuronales vamos a ver un ejemplo que resuelve la función XOR:

A	B	A xor B
1	1	0
1	0	1
0	1	1
0	0	0

Necesitamos una red neuronal con dos neuronas de entrada y una de salida. No es un problema separable linealmente, por tanto, necesitamos una capa oculta en la que colocaremos dos neuronas.

[NOTA] Os vendría bien hacer un script para no tener que volver a repetir todos lo pasos cuando tengáis que modificar algo.

Para crear la red utilizaremos la función `newff`:

```
net = newff([0 1; 0 1],[2 1],{'logsig','logsig'})
```

Como parámetros necesita el rango de valores de las neuronas de entrada (`[0 1; 0 1]`), el número de celdas en la capa oculta y en la de salida (`[2 1]`) y la función de activación de cada capa (`{'logsig','logsig'}` en este caso, ver figura 1).

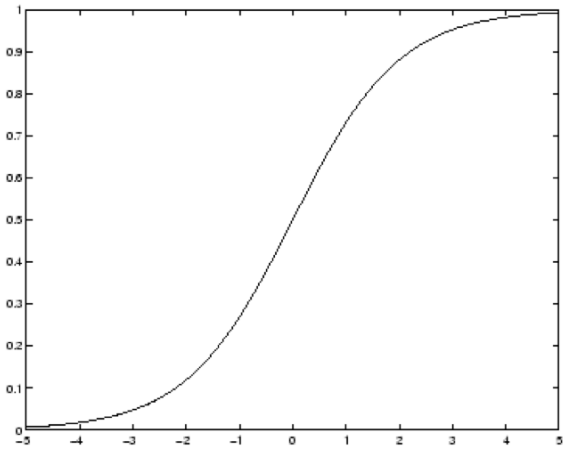

Figura 1.- La función de activación `logsig`

Podríamos haber utilizado otras funciones de activación (ver figura 2):

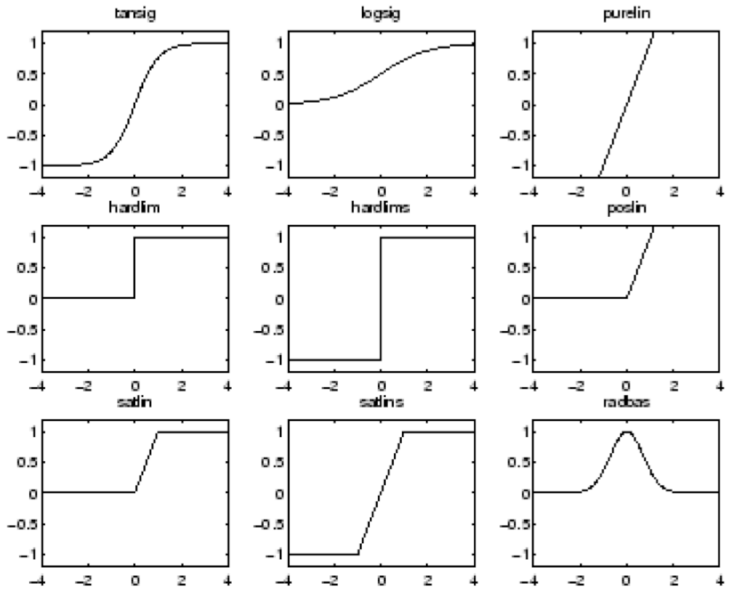

Figura 2.- Otras funciones de activación

Las funciones de activación tienen que estar acordes con el tipo de número que van a recibir en la capa de entrada. Si reciben algo en [0,1] ponemos logsig, si es algo entre [-1,1] ponemos tansig. Lo mismo sucede con la capa de salida: si queremos devolver algo en [-1,1] ponemos tansig y si es entre [0,1] ponemos logsig.

Vamos a ver cómo es de "buena" la red sin entrenar. Necesitamos una matriz con las entradas. Las entradas de la red son las columnas de la matriz. Si queremos una matriz con las entradas: "1 1", "1 0", "0 1" y "0 0" debemos escribir:

```
>> input = [1 1 0 0; 1 0 1 0]
input =
     1     1     0     0
     1     0     1     0
```

Veamos qué salidas obtenemos si le damos esta entrada a la red:

```
>> output=sim(net,input)
output =
    0.3394    0.0659    0.0769    0.1025
```

La función `sim` se utiliza para simular la red y así calcular las salidas. La salida no es muy buena. Lo deseable era (0 1 1 0) y nos hemos encontrado con (0.3394 0.0659 0.0769 0.1025). Esto es así porque los pesos se han inicializado aleatoriamente y la red no ha sido entrenada (seguramente que cuando lo ejecutes tú saldrán otros valores).
El objetivo de esta red es ser capaz de producir:

```
>> target = [0 1 1 0]
target =
     0     1     1     0
```

Con el comando `plot` podemos ver el objetivo y lo que hemos conseguido hasta el momento:

```
>> plot(target, 'o')
>> hold on
>> plot(output, '+r')
```

Veamos el resultado en la figura 3:

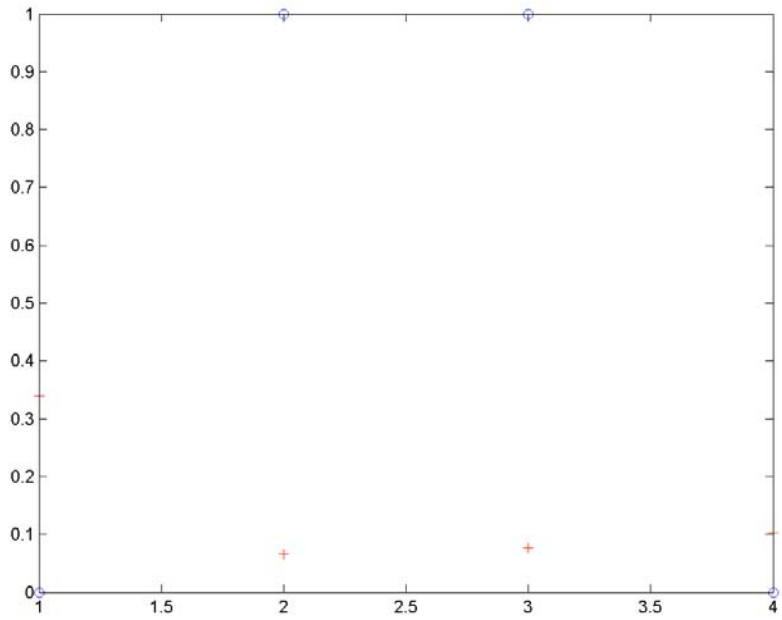

Figura 3.- Objetivo (círculos en azul) y solución obtenida sin entrenar la red (+ en rojo)

Parece que con los pesos que tiene la red sin entrenar no se obtiene una buena solución. Veamos los pesos que hay entre la capa de entrada y la capa oculta:

```
>> net.IW{1,1}
ans =
    7.7055    1.8290
   -7.9089   -0.4123
```

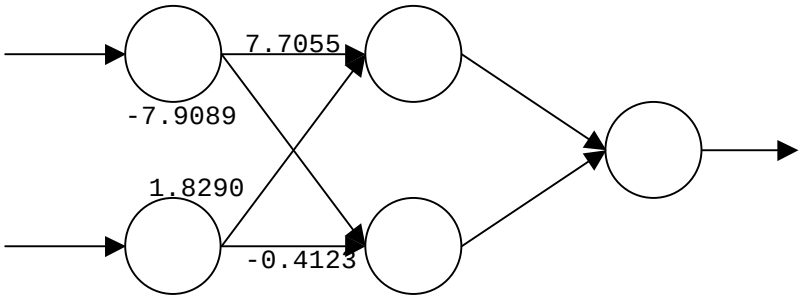

Figura 4.- La red con los pesos iniciales

Podríamos cambiar cualquier peso de la red:

```
>> net.IW{1,1}(1,2)=5;
>> net.IW{1,1}
ans =
    7.7055    5.0000
   -7.9089   -0.4123
```

Los pesos entre la capa oculta y la capa de salida se almacenan en `LW`:

```
>> net.LW{2,1}
ans =
    4.6527    3.1164
```

Así la red con todos los pesos sería (incluyendo el cambio realizado):

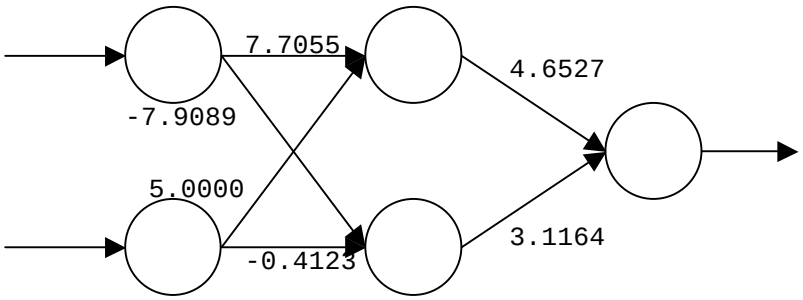

Figura 5.- La red con todos los pesos iniciales

Hemos cambiado un peso, así que podemos volver a evaluar la red:

```
>> output=sim(net,input)
output =
    0.6645    0.0659    0.0846    0.1025
>> plot(output,'g*')
```

En la figura vemos que las salidas han cambiado ligeramente:

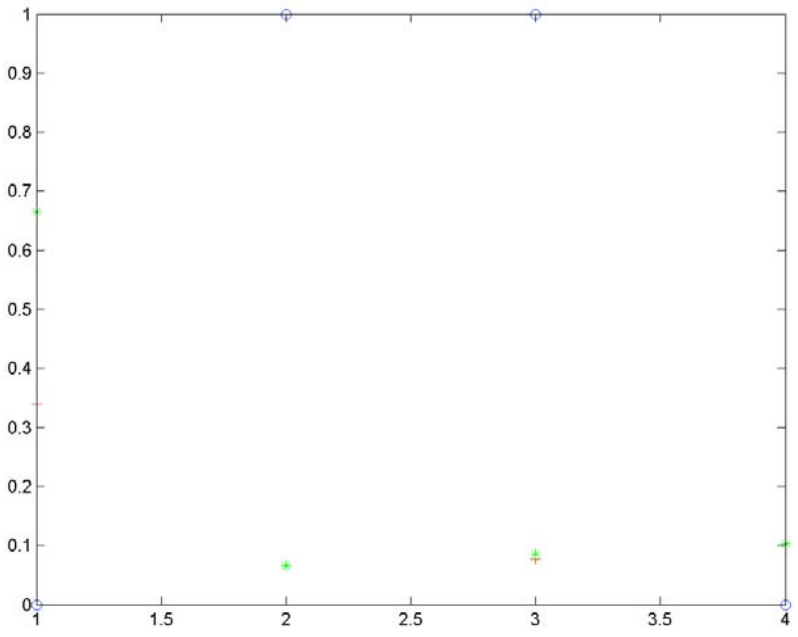

Figura 6.- Objetivo (círculos en azul) y solución obtenida sin entrenar la red (+ en rojo). En (* verde) las salidas habiendo modificado un peso a mano.

Podríamos pasarnos horas modificando los pesos tratando de acercarlos a la solución (prueba a hacer un par de cambios). Pero parece más sencillo dejar que los pesos se calculen automáticamente. Para ello tenemos que entrenar la red.

Para entrenar la red tenemos que utilizar la función `train`:

```
>> net  = train(net,input,target);
```

Aparece un gráfico en el que vemos la evolución del entrenamiento. El objetivo a alcanzar es bajar de un gradiente de 1E-10 (por defecto). Vemos también que el número de iteraciones máximo, por defecto, es 100.
Como hay pocos ejemplos el entrenamiento es muy rápido. Veamos si hemos aprendido bien:

```
>> output = sim(net,input)
output =
    0.0000    1.0000    1.0000    0.0000
```

Y ahora vamos a ver los pesos que se han calculado en el entrenamiento:

```
>> net.IW{1,1}
ans =
    9.2325  -11.5070
  -12.0289   13.7543
>> net.LW{2,1}
ans =
   27.6393   30.3009
```

1.2 Trabajo

Haz tú una red que resuelva la función OR.

1.3 Un problema real

Vayamos al enlace:

http://archive.ics.uci.edu/ml/datasets/Breast+Cancer+Wisconsin+%28Diagnostic%29

y descarguemos el fichero `wdbc.data`. En este fichero tenemos los datos de 569 mujeres con cáncer de mama. Cada mujer está descrita por 32 atributos. El primero es un identificador, el segundo el tipo de cáncer (Maligno o Benigno) y el resto son el resultado de otros análisis clínicos. Pretendemos aprender el tipo de cáncer. En este conjunto la distribución de clases es: 357 benignos y 212 malignos.

Una vez descargado el conjunto hay que sustituir las M's por 1 y las B's por 0. Después cargamos el conjunto en MATLAB descartando la primera columna (el identificador) ya que no aporta nada al problema:

```
>> bcdata=csvread('wdbc.data',0,1);
>> size(bcdata)
ans =
   569    31
```

[NOTA] Acordaros de hacer un script.

Como la red necesita que se le suministre cada ejemplo en una columna, necesitamos trasponer la matriz de datos:

```
>> bcdata=bcdata';
```

```
>> size(bcdata)
ans =
    31   569
```

Ahora tenemos que definir cuál es el objetivo y cuáles son los datos de entrada:

```
>> target=bcdata(1,:);
>> indata=bcdata(2:31,:);
```

Para obtener los rangos de las variables de entrada podemos utilizar:

```
>> input_ranges=minmax(indata);
```

Ahora creamos la red. No sabemos ni cuantas capas ocultas necesita ni cuantas neuronas necesita en esas capas. Lo que si sabemos que necesita 30 neuronas de entrada y una de salida. Vamos a crear la red con una capa oculta de 10 neuronas (porque algo hay que probar). Como algoritmo de entrenamiento utilizaremos `trainlm`.

```
>> net=newff(input_ranges,[10 1],{'tansig','logsig'},'trainlm');
```

Como tenemos bastantes ejemplos podemos utilizar una parte para el entrenamiento y otra parte para validar lo aprendido mientras se está entrenando la red. Lo ideal sería parar el entrenamiento cuando el error sobre el conjunto de validación se incremente (estaríamos empezando a sobre ajustarnos). Esto se puede hacer separando una parte del conjunto de entrenamiento para validar la solución. Se haría de la siguiente manera:

```
>> training_in = indata(:,1:2:length(indata));
>> training_target = target(1:2:length(target));
>> testset.P = indata(:,2:2:length(indata));
>> testset.T = target(2:2:length(target));
```

Ahora podemos entrenar la red:

```
>> net.trainParam.show=1;
>> net=train(net,training_in,training_target,[],[],testset);
```

Ponemos el parámetro `show` a 1 para que nos muestre los datos de cada iteración. En la gráfica vemos en azul el error de entrenamiento y en verde el de validación.
Para calcular la precisión de la solución aportada por la red tenemos que implementar una función:

```
function return_value = accuracy(net,input,target)
output=sim(net,input);
correct=0;
for i=1:length(output)
      if ((target(i)==1) & (output(i)>=0.5))
      correct=correct+1;
  elseif ((target(i)==0) & (output(i)<0.5))
      correct=correct+1;
      end;
end;
return_value = correct./length(output);
```

En este caso tenemos 2 clases etiquetadas como 0 y 1, por eso ponemos el 0.5 como punto de división entre las dos clases. Si las clases fuesen +1 y -1, la separación entre las dos clases estaría en el 0. Esta función podremos utilizarla de la siguiente manera:

```
>> porc_err_todo = 100*(1-accuracy(net,indata,target))
```

y así obtenemos el error en el conjunto original.

Éste es el script que he elaborado para hacer las pruebas (se supone que `wdbc.data` ya tiene sustituidas las M's por 1 y las B's por 0):

```
bcdata=csvread('wdbc.data',0,1);
size(bcdata)
bcdata=bcdata';
size(bcdata)
%temp_target=bcdata(1,:);
target=bcdata(1,:);
indata=bcdata(2:31,:);

input_ranges=minmax(indata);
net=newff(input_ranges,[10 1],{'tansig','logsig'},'trainlm');

training_in = indata(:,1:2:length(indata));
testset.P = indata(:,2:2:length(indata));

training_target = target(1:2:length(target));
testset.T = target(2:2:length(target));

net.trainParam.show=1;
net=train(net,training_in,training_target,[],[],testset);

porc_err_ent = 100*(1-accuracy(net,training_in,training_target))
porc_err_valida = 100*(1-accuracy(net,testset.P,testset.T))
porc_err_todo = 100*(1-accuracy(net,indata,target))
```

1.4 Trabajo

Modifica parámetros de la red para tratar de minimizar el error en el conjunto de validación. ¿Cuál es el mejor error que has conseguido?

1.5 Spider

Unos alumnos de esta asignatura desarrollaron un objeto `Spider` para el manejo de redes neuronales. El objeto se llama `nno` y podéis descargarlo en:

http://www.aic.uniovi.es/SSII/P10/nno.zip

Al descomprimirlo aparece un directorio `@nno`. En este directorio se encuentran los métodos del objeto `nno`, el cual utiliza el toolbox de `Matlab` para redes neuronales que hemos visto en los epígrafes anteriores (Neural Network Toolbox).

Debéis incluirlo en el `path`, pero tened en cuenta que los directorios que empiezan por @ no se pueden incluir directamente, sino que debemos incluir el directorio superior. Si creáis la carpeta Z:\spider\nno para contener el directorio @nno deberías añadirlo al path:

```
addpath('Z:\spider\nno')
```

Ahora ya podemos utilizarlo como un objeto más del `Spider`:

```
>> X=rand(200,8)-0.5;
>> Y=sign(X(:,1)+X(:,2));
>> d=data(X,Y);
>> s=nno;
>> s.n_occ=8; % nº de celdas en la capa oculta
>> [res sist] = train(s,d,'class_loss');
>> [res] = test(sist,d,'class_loss')
```

1.6 Trabajo

En las prácticas anteriores hemos trabajado con `knn` y `j48`. Ejecuta la red neuronal en los problemas que descargaste de:

http://www.ics.uci.edu/~mlearn/MLRepository.html

¿Qué sistema parece mejor?