Introducción a las RR.NN.AA. en Matlab

Sistemas Inteligentes

Matlab posee un Toolbox denominado Deep Learning Toolbox (Neural Network Toolbox en versiones más antiguas de Matlab), que tiene implementadas las funciones necesarias para desarrollar y trabajar con RR.NN.AA. en ese entorno. Por otra parte, el Toolbox de estadística proporciona una serie de bases de datos conocidas para poder trabajar en distintos problemas. Para cargar cada una de estas bases de datos, se puede utilizar simplemente la función load. Por ejemplo:

```
load iris_dataset
```

Esta línea cargará en memoria la conocida base de datos de clasificación de flores iris. Esta base de datos tiene 150 instancias y 4 atributos. Como consecuencia de la ejecución de esta línea, se generarán en memoria dos variables:

- irisInputs: de dimensión 4x150, con los valores de los 4 atributos para cada instancia.
- irisTargets: de dimensión 3x150, con la clasificación de cada instancia. Esta variable contiene, para cada instancia, 3 valores, siendo 0 para aquellas clases a las que no pertenece, y 1 para la que pertenece.

Es necesario tener en cuenta que la función load permite cargar aquellas bases de datos que vienen con Matlab, y no será el caso de las usadas en estas prácticas. Para cargarlas, es necesario llamar a las funciones que se especifican al final de la práctica anterior.

Como se puede ver por la dimensión de las variables, Matlab entiende que en las bases de datos las distintas instancias se disponen en columnas, y los atributos y salidas deseadas en filas, al contrario de lo que intuitivamente se suele pensar.

1.- Creación y entrenamiento de RR.NN.AA.

A la hora de desarrollar RR.NN.AA., Matlab distingue tres tipos principales de problemas, y para cada uno proporciona una función para crear una RNA: regresión, clasificación y ajuste de curvas.

1.1.- Problemas de regresión

Para resolver este tipo de problemas está disponible la función feedforwardnet, que recibe, como mínimo, un único parámetro, que es un vector con la arquitectura que tendrá la red. El número de elementos del vector será el número de capas ocultas, y el valor de cada elemento del vector será el número de neuronas en cada capa oculta. Por ejemplo, para crear una red con una capa oculta y 10 neuronas en esa capa se podría escribir lo siguiente:

```
rna = feedforwardnet([10]);
```

Y para crear una red con dos capas ocultas, y 12 y 6 neuronas en la primera y segunda capa oculta respectivamente, se podría escribir lo siguiente:

```
rna = feedforwardnet([12 6]);
```

De esta manera, se crea una variable en memoria, llamada rna, que contiene un objeto con una estructura con distintos campos que configura la red. De hecho, si se escribe el nombre de la variable rna para visualizar su contenido, aparecerá lo siguiente:

```
rna =

    Neural Network

               name: 'Feed-Forward Neural Network'
         efficiency: .cacheDelayedInputs, .flattenTime,
                     .memoryReduction
           userdata: (your custom info)

    dimensions:

          numInputs: 1
          numLayers: 2
         numOutputs: 1
    numInputDelays: 0
    numLayerDelays: 0
 numFeedbackDelays: 0
 numWeightElements: 121
        sampleTime: 1

    connections:

       biasConnect: [1; 1]
      inputConnect: [1; 0]
      layerConnect: [0 0; 1 0]
     outputConnect: [0 1]

    subobjects:

            inputs: {1x1 cell array of 1 input}
            layers: {2x1 cell array of 2 layers}
```

```
        outputs: {1x2 cell array of 1 output}
         biases: {2x1 cell array of 2 biases}
   inputWeights: {2x1 cell array of 1 weight}
   layerWeights: {2x2 cell array of 1 weight}

functions:

       adaptFcn: 'adaptwb'
     adaptParam: (none)
       derivFcn: 'defaultderiv'
      divideFcn: 'dividerand'
    divideParam: .trainRatio, .valRatio, .testRatio
     divideMode: 'sample'
        initFcn: 'initlay'
     performFcn: 'mse'
   performParam: .regularization, .normalization, .squaredWeighting
       plotFcns: {'plotperform', plottrainstate, ploterrhist,
                 plotregression}
     plotParams: {1x4 cell array of 1 param}
       trainFcn: 'trainlm'
     trainParam: .showWindow, .showCommandLine, .show, .epochs,
                 .time, .goal, .min_grad, .max_fail, .mu, .mu_dec,
                 .mu_inc, .mu_max

weight and bias values:

              IW: {2x1 cell} containing 1 input weight matrix
              LW: {2x2 cell} containing 1 layer weight matrix
               b: {2x1 cell} containing 2 bias vectors

methods:

           adapt: Learn while in continuous use
       configure: Configure inputs & outputs
          gensim: Generate Simulink model
            init: Initialize weights & biases
         perform: Calculate performance
             sim: Evaluate network outputs given inputs
           train: Train network with examples
            view: View diagram
     unconfigure: Unconfigure inputs & outputs

evaluate:       outputs = rna(inputs)
```

De esta forma, se puede modificar los valores que toman los distintos campos para configurar la red de una forma u otra. Algunos de estos campos más interesantes son:

- rna.IW: matrices de pesos de la capa de entrada a cada una de las restantes capas.
- rna.LW: pesos entre las capas que no son de entrada.
- rna.b: matrices de bias de cada capa.

- rna.trainParam: Este campo, a su vez, contiene nuevos campos con distintos parámetros que controlarán el entrenamiento de la red. Si se visualiza:

```
>> rna.trainParam

ans =


    Function Parameters for 'trainlm'

    Show Training Window Feedback   showWindow: true
    Show Command Line Feedback showCommandLine: false
    Command Line Frequency                show: 25
    Maximum Epochs                      epochs: 1000
    Maximum Training Time                 time: Inf
    Performance Goal                      goal: 0
    Minimum Gradient                   min_grad: 1e-005
    Maximum Validation Checks          max_fail: 6
    Mu                                       mu: 0.001
    Mu Decrease Ratio                    mu_dec: 0.1
    Mu Increase Ratio                    mu_inc: 10
    Maximum mu                           mu_max: 10000000000
```

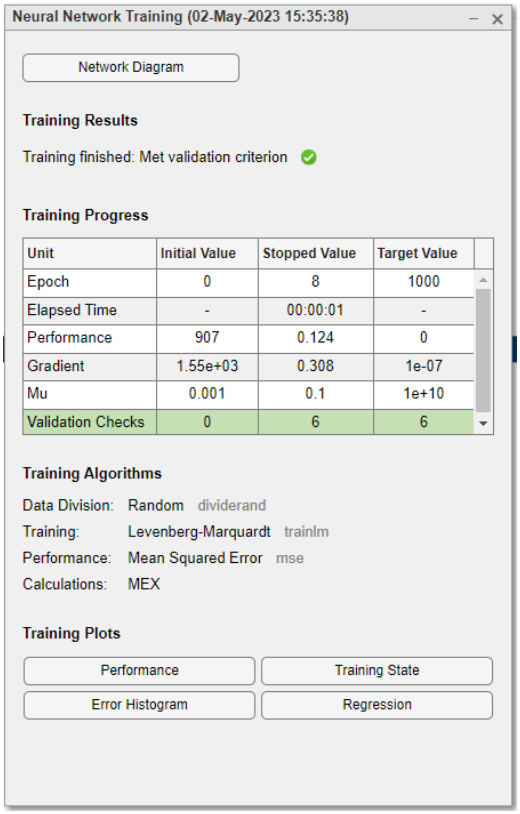
  Algunos de los parámetros más importantes son los que permiten establecer el número máximo de ciclos del algoritmo de backpropagation (epochs), el tiempo máximo que puede estar ejecutándose (time), el error que se quiere alcanzar (goal), el hecho de querer o no que se muestre la ventana de entrenamiento (showWindow), o el número de ciclos máximo que puede transcurrir sin mejorar el error de validación antes de parar el entrenamiento (max_fail).

- rna.divideParam: Este campo contiene 3 campos en los que se establece el ratio de división de los patrones en los conjuntos de entrenamiento, validación y test: rna.divideParam.trainRatio,rna.divideParam.valRatio y rna.divideParam.testRatio. Antes de realizar el entrenamiento, se dividen los patrones aleatoriamente según esos valores y cada uno de esos nuevos conjuntos se utiliza con ese fin. Si se visualizan, por defecto toman valores de 60% (entrenamiento), 20% (validación) y 20% (test).

Para entrenar una red una red se tiene la función train, que recibe como parámetros la red a entrenar y los conjuntos de entradas y salidas deseadas. El resto de parámetros del proceso de entrenamiento los toma de los campos correspondientes de la red pasada como parámetro. Como resultado, devuelve la misma red entrenada. Dado que Matlab no acepta el paso de parámetros por referencia, se deberá asignar a la misma variable para actualizar su valor con la red entrenada. Por ejemplo, si se ejecutan las líneas:

```
load building_dataset
rna = feedforwardnet([8]);
rna = train(rna, buildingInputs, buildingTargets);
```

se carga primero la base de datos con los patrones sobre predicción del consumo de energía en edificios, con lo que se crean las variables con los patrones de entrada (buildingInputs) y salidas deseadas (buildingTargets), posteriormente se crea la red y se entrena. Al escribir esto en Matlab aparecerá una ventana similar a la siguiente:



En esta ventana se puede distinguir 5 zonas distintas. En primer lugar, en la parte superior, aparece un botón que, si se pulsa, hace aparecer una ventana con un esquema con la estructura de la red. Inmediatamente debajo, bajo el título *Training Results*, aparecen descritos los resultados del entrenamiento, principalmente el criterio utilizado para parar el entrenamiento. Debajo de esto aparece otra zona denominada *Training Progress*, donde se puede observar una tabla. Cada una de las filas representa un criterio distinto de parada y, cuando se cumpla uno de los mismos, la fila se pone de color verde y el entrenamiento se para. Los criterios de parada indicados en cada fila son los siguientes:

- Se ha realizado un número máximo de ciclos del algoritmo de entrenamiento (por defecto, 1000 ciclos).
- Se ha entrenado hasta llegar a un periodo de tiempo máximo (por defecto,

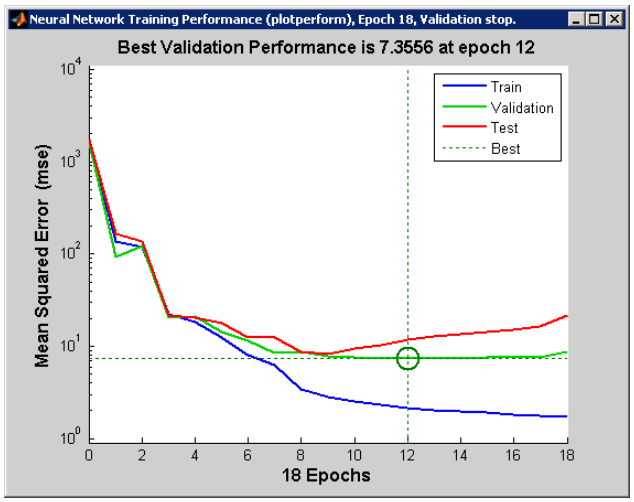
deshabilitado).

- Se ha alcanzado un error aceptable (por defecto, 0).
- Se ha alcanzado un valor de gradiente muy bajo (por defecto, 10^{-7}).
- Se ha alcanzado un valor muy alto del parámetro mu (por defecto, 10^{10}).
- Se ha alcanzado un número prefijado de ciclos en los cuales no se ha mejorado el mejor error obtenido en el conjunto de validación (*validation checks*, por defecto 6). Este suele ser el criterio de parada más común, para evitar el sobreajuste.

Debajo, en la siguiente zona, se pueden ver cuatro de los algoritmos más importantes utilizados en el entrenamiento de la red. Los tres primeros que aparecen, los más importantes, son:

- Método de división del conjunto de patrones en entrenamiento/validación/test: Por defecto: división aleatoria (*dividerand*).
- Algoritmo de entrenamiento. Por defecto, la variante del algoritmo de *backpropagation* de Levenberg-Marquardt (*trainlm*).
- Función de medida de error. Por defecto, error cuadrático medio (*mse*).

Por último, debajo de todo hay una zona (*Plots*) donde se pueden ver distintas gráficas que se pueden sacar como resultado del proceso de entrenamiento. Entre ellas, destaca la gráfica *Performance*. En esta gráfica aparece la evolución del error en cada ciclo del algoritmo de entrenamiento para los conjuntos de entrenamiento, validación y test. Aparece resaltado el ciclo en el que se obtuvo el mejor error en el conjunto de validación, y, una vez terminado el proceso de entrenamiento, se devuelve la red con los valores de los pesos en ese ciclo (con menor error de validación). De esta forma se evita el efecto del sobreajuste.



Una vez entrenada una red, para utilizarla se puede emplear la función sim, que recibe como parámetros la propia red y un conjunto de patrones. Por ejemplo, si se quiere emplear la red para generar salidas ante el conjunto de patrones, y calcular el error cuadrático medio con respecto a las salidas deseadas se podría escribir lo siguiente:

```
>> buildingOutputs = sim(rna, buildingInputs);
>> mse(buildingOutputs-buildingTargets)

ans =

    0.0024
```

1.2.- Problemas de clasificación

Para resolver un problema de clasificación, se puede utilizar la función patternnet, que se utiliza de una forma similar a la función feedforwardnet. Por ejemplo, para cargar los patrones del problema de flores iris y crear una red que clasifique con una sola capa oculta y 8 neuronas en la misma, se podría escribir lo siguiente:

```
load iris_dataset;
rna = patternnet([8]);
```

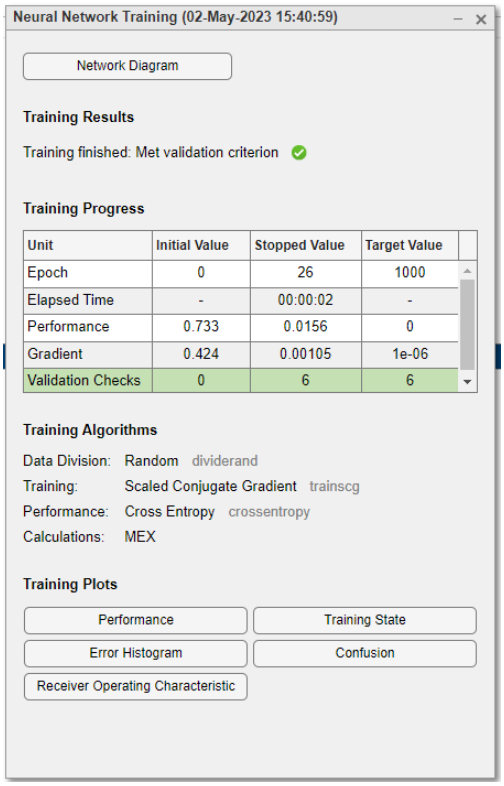
La arquitectura de una capa oculta con 8 neuronas en la misma se puede ver en el parámetro [8] que se le pasa a la función patternnet, y podría ser cualquier otra arquitectura que el usuario desee utilizar.

El problema de flores iris, descrito por Fisher en 1936, es uno de los problemas de clasificación más famosos y utilizados en la bibliografía. Consta de 150 patrones repartidos en 3 clases distintas. Mientras que una de las clases es fácilmente separable, en las otras dos hay una zona en la que unos cuantos patrones se entremezclan, con lo que no son linealmente separables.

De esta manera, se crea una estructura similar a la creada para problemas de regresión pero con valores distintos en ciertos campos, preparando la red para resolver problemas de clasificación, entre los que destacan, en primer lugar, que las funciones de transferencia son sigmoidales en todas las capas, incluidas las de salida. Otro valor distinto es la función de entrenamiento, que para resolver problemas de clasificación es una variante del algoritmo de *backpropagation* con gradiente de escala conjugada (trainscg).

Además, al entrenar una red para resolver problemas de clasificación la ventana que sale es ligeramente distinta, para reflejar estos cambios. Este entrenamiento se realiza de forma similar, mediante una llamada a la función train.

```
rna = train(rna, irisInputs, irisTargets);
```



Como se puede ver, además de dos de las gráficas que se podían representar en problemas de regresión (*Performance*, *Training State* y *Error Histogram*), es posible representar, en problemas de clasificación, otras dos distintas, entre la que destaca la llamada *Confusion*. Esta gráfica muestra las matrices de confusión para los conjuntos de entrenamiento, validación y test, y para todos los patrones reunidos. Cada matriz de confusión representa, para cada columna, cada una de las clases verdaderas de tal forma que los patrones se reparten en las columnas según la clase a la que realmente pertenecen (salidas deseadas). Por su parte, en las filas se representan las clases en las que se clasifican los patrones (salidas obtenidas). De esta manera, los patrones correctamente clasificados se muestran en la diagonal de la matriz, y cualquier elemento incorrectamente clasificado estaría fuera de la diagonal. Por ejemplo, un valor de 2 en la fila 3, columna 2 de la matriz de confusión de test indica que la red, en el conjunto de test, clasificó dos patrones en la tercera clase, cuando en realidad pertenecían a la segunda. Adicionalmente, cada matriz de confusión tiene una fila y columna extra con estadísticos, siendo el más importante el último, que muestra, en la parte inferior derecha, la precisión en la clasificación. En el caso de la base de datos utilizada en este

ejemplo, existen 3 clases distintas, y las matrices de confusión podrían ser las siguientes:

Al igual que en el caso anterior, para utilizar una red entrenada se puede usar la función sim. Por ejemplo, para calcular las salidas que la red emite cuando se utilizan como entrada los patrones de entrenamiento, y el error cuadrático medio con respecto a las salidas deseadas se podría escribir lo siguiente:

```
>> irisOutputs = sim(rna, irisInputs);
>> mse(irisOutputs-irisTargets)

ans =

    0.0091
```

Sin embargo, en un problema de clasificación la precisión en la clasificación, es decir, el porcentaje de patrones bien clasificados, es más importante que el error cometido. Como ya se ha dicho antes, esto se puede ver en la matriz de confusión, en la casilla de abajo a la derecha. Para sacar este valor para que pueda ser utilizado, la función *confusion*, que recibe un conjunto de salidas deseadas y de salidas obtenidas y devuelve la tasa de error. La precisión será 1 – tasa de error:

```
load iris_dataset;
rna = patternnet([8]);
rna = train(rna, irisInputs, irisTargets);
irisOutputs = sim(rna, irisInputs);
precision = 1-confusion(irisTargets, irisOutputs);
```

1.3.- Problemas de ajuste de curvas

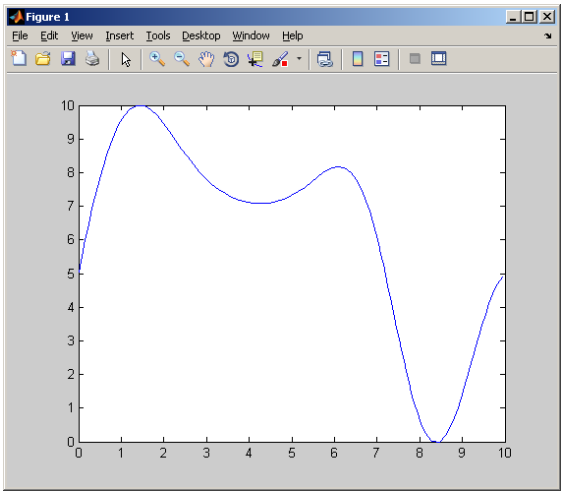
Para resolver un problema de ajuste de curvas, se puede utilizar la función fitnet, que se utiliza exactamente igual que las anteriores. Por ejemplo, un sencillo problema de ajuste de curvas se podría cargar con la siguiente línea:

```
load simplefit_dataset
```

Este problema tiene una sola entrada y una salida deseada, es decir, se desea desarrollar una RNA que reproduzca una función desconocida del tipo y=f(x). Se puede representar gráficamente la función a modelizar, mediante la siguiente línea:

```
plot(simplefitInputs, simplefitTargets);
```

De esta forma, se representan todos los patrones con la entrada en el eje x y la salida deseada en el eje y:



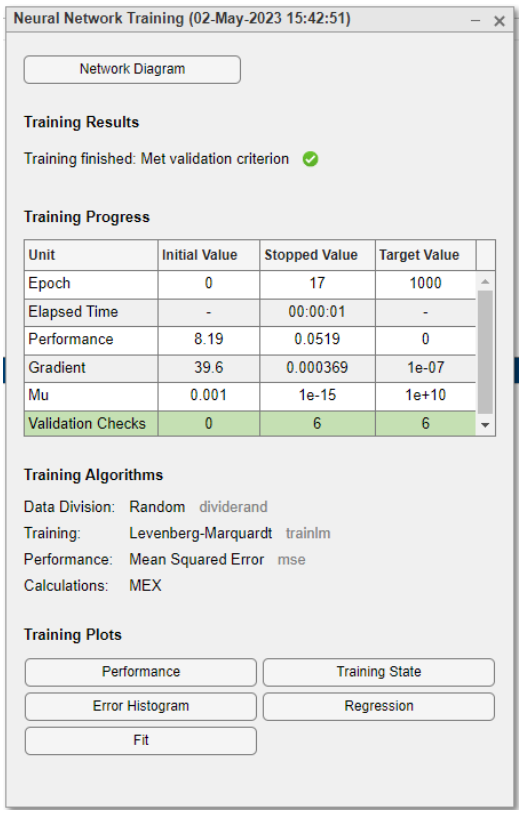
Se puede crear una RNA que resuelva este problema mediante estas líneas:

```
rna = fitnet([6]);
rna = train(rna, simplefitInputs, simplefitTargets);
```

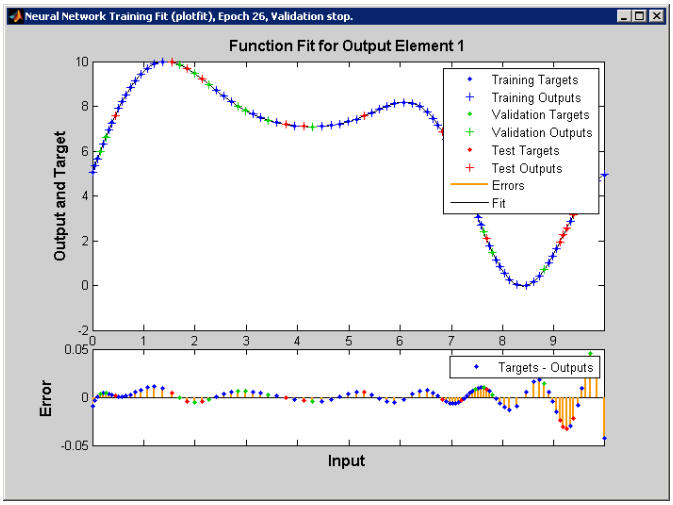
De esta manera, se crea una estructura similar a la creada para problemas de regresión y clasificación pero con valores distintos en ciertos campos, preparando la red para resolver problemas de ajuste de curvas, entre los que destacan, en primer lugar, que las funciones de transferencia son sigmoidales en las capas ocultas, y lineales en la capa de salida, es decir, igual que en problemas de regresión. Otro valor distinto es la función de entrenamiento, que para resolver problemas de ajuste de curvas

es también la misma que para problemas de regresión.

Por lo tanto, una red para resolver problemas de ajuste de curvas es muy similar a una para resolver un problema de regresión. Algunos de los cambios más perceptibles se refieren a las gráficas que se pueden representar en el proceso de entrenamiento.



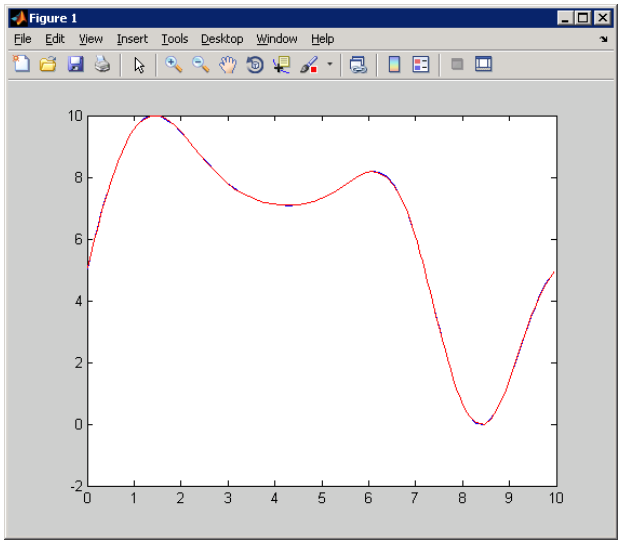
Como se puede ver, es posible representar las cuatro gráficas para los problemas de regresión, más una nueva, llamada *Fit*, para problemas de ajuste de curvas:



Esta gráfica representa, en la parte superior, cada uno de los patrones, en los que se puede ver, para cada uno, la salida obtenida comparada con la salida deseada, en azul para los patrones de entrenamiento, verde para validación, y rojo para test. En la parte inferior se puede ver, para esos mismos patrones, el error cometido por la red, calculado como salidas deseadas – salidas obtenidas por la red.

Igual que antes, para utilizar una red se podría usar la función sim. Por ejemplo, para calcular las salidas que emite la red para los patrones y representar en una misma gráfica las salidas deseadas en azul y las salidas obtenidas en rojo se podría hacer lo siguiente:

```
plot(simplefitInputs, simplefitTargets,'b');
simplefitOutputs = sim(rna, simplefitInputs);
hold on;
plot(simplefitInputs, simplefitOutputs, 'r');
```



2.- Conjuntos de entrenamiento, validación y test

Como ya se ha estudiado en clase de teoría, a la hora de entrenar una RNA son necesarios 3 conjuntos: entrenamiento y validación y test. La mayoría de las ocasiones se comienza con un solo conjunto, que se divide en estos 3 subconjuntos de forma aleatoria. Como pone más arriba, los parámetros que dicen qué porcentajes se destinarán a cada conjunto son rna.divideParam.trainRatio, rna.divideParam.valRatio y rna.divideParam.testRatio.

Para valorar correctamente la bondad de una RNA, esto debe realizarse con el conjunto de test, así

que el valor final de error (para un problema de regresión) o precisión (para un problema de clasificación) que se mire debe ser calculado en el conjunto de test. Por tanto, es necesario saber qué patrones han sido destinados para el conjunto de test durante el proceso de entrenamiento.

Para ello, después de realizar el entrenamiento, además de devolver la red entrenada, la función *train* devuelve una segunda variable, que contiene un resumen del proceso de entrenamiento de la red, y que puede asignarse a una variable en el entorno. Por ejemplo:

```
load building_dataset
rna = feedforwardnet([8]);
[rna, tr] = train(rna, buildingInputs, buildingTargets);
```

De esta manera, se crearía la variable tr (*training record*) donde se guarda el resumen del proceso de entrenamiento de la red. El contenido de esta variable podría ser el siguiente:

```
>> tr

tr =

        trainFcn: 'trainlm'
      trainParam: [1x1 nnetParam]
      performFcn: 'mse'
    performParam: [1x1 nnetParam]
       derivFcn: 'defaultderiv'
      divideFcn: 'dividerand'
      divideMode: 'sample'
     divideParam: [1x1 nnetParam]
        trainInd: [1x2946 double]
          valInd: [1x631 double]
         testInd: [1x631 double]
            stop: 'Training finished: Met validation criterion'
      num_epochs: 16
       trainMask: {[1x4208 double]}
         valMask: {[1x4208 double]}
        testMask: {[1x4208 double]}
      best_epoch: 10
            goal: 0
          states: {1x8 cell}
           epoch: [0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16]
            time: [1x17 double]
            perf: [1x17 double]
           vperf: [1x17 double]
           tperf: [1x17 double]
              mu: [1x17 double]
        gradient: [1x17 double]
        val_fail: [0 0 0 0 0 1 0 0 0 1 0 1 2 3 4 5 6]
       best_perf: 0.0025
      best_vperf: 0.0022
      best_tperf: 0.0023
```

En estos valores concretos de esta variable se puede ver, por ejemplo, que la red se ha entrenado

durante 16 ciclos (tr.num_epochs), comenzando en el ciclo 0, pero que la red que se devuelve no es la correspondiente al último ciclo, sino la que corresponde al ciclo 10 (tr.best_epoch), puesto que es la que tiene menor error de validación, y que tiene un error en entrenamiento de 0.0025 (tr.best_perf), en validación de 0.0022 (tr.best_vperf) y en test de 0.0023 (tr.best_tperf). Estos valores se corresponden a los valores de los vectores que contienen los errores durante el entrenamiento en los conjuntos de entrenamiento, validación y test (tr.perf, tr.vperf y tr.tperf respectivamente), en el índice tr.best_epoch+1, puesto que el primer elemento de estos vectores corresponde con el ciclo 0. Otra información importante son los índices de los patrones utilizados para entrenamiento (tr.trainInd), validación (tr.valInd) y test (tr.testInd). Esta última información resulta muy útil para calcular el error en el conjunto de test, que se puede realizar de la siguiente manera:

```
load building_dataset
rna = feedforwardnet([8]);
[rna, tr] = train(rna, buildingInputs, buildingTargets);
buildingInputsTest = buildingInputs(:,tr.testInd);
buildingTargetsTest = buildingTargets(:,tr.testInd);
buildingOutputsTest = sim(rna, buildingInputsTest);
errorTest = mse(buildingOutputsTest - buildingTargetsTest);
```

Para un problema de clasificación, en el que lo importante no es el error sino la precisión, podría hacerse de la siguiente manera:

```
load iris_dataset;
rna = patternnet([8]);
[rna, tr] = train(rna, irisInputs, irisTargets);
irisInputsTest = irisInputs(:,tr.testInd);
irisTargetsTest = irisTargets(:,tr.testInd);
irisOutputsTest = sim(rna, irisInputsTest);
precisionTest = 1-confusion(irisTargetsTest, irisOutputsTest);
```

Sin embargo, en algunas ocasiones los creadores de la base de datos especifican qué patrones deben ser utilizados como test. Si estos patrones se encuentran en una variable aparte, será necesario especificar a la RNA que en el conjunto de patrones que usen para el entrenamiento no se separará ninguno para test, por ejemplo, para el problema de los edificios, suponiendo que se tienen los patrones de test en las variables buildingInputsTest y buildingTargetsTest:

```
rna = feedforwardnet([8]);
rna.divideParam.trainRatio = 0.7; % 70% para entrenamiento
rna.divideParam.valRatio = 0.3; % 30% para validacion
rna.divideParam.testRatio = 0; % 0% para test
[rna, tr] = train(rna, buildingInputs, buildingTargets);
```

```
buildingOutputsTest = sim(rna, buildingInputsTest);
errorTest = mse(buildingOutputsTest - buildingTargetsTest);
```

Para un problema de clasificación el código sería similar.

3.- Estudio de los parámetros de una RNA

El entrenamiento de una RNA es un proceso no determinístico, es decir, cada vez que se entrena da un resultado distinto. Por ello, para ver si un determinado parámetro es apropiado para resolver un problema es necesario entrenarla varias veces y ofrecer la media de los errores en el conjunto de test para todos los entrenamientos realizados. De toda la gran cantidad de parámetros que tiene una RNA, el más importante es la arquitectura, puesto que es el que limita la complejidad de los problemas a resolver.

Por tanto, para estudiar cómo de buena es una arquitectura, es necesario realizar un bucle en el que, dentro del mismo, se cree una red (siempre con la misma arquitectura), se entrene y se almacenen los resultados de entrenamiento, validación y test para, posteriormente, fuera del bucle, se muestren la media de estos resultados para todas las veces que se crearon y entrenaron estas redes.

Un valor típico de iteraciones de este bucle sería entre 30 y 50. Además, dado que se va a realizar un gran número de entrenamientos, para el proceso vaya más rápido se puede configurar la red antes de entrenarla para que no aparezca la ventana de entrenamiento de la siguiente manera:

```
rna.trainParam.showWindow = false;
```

Para cada entrenamiento distinto, los errores en entrenamiento, validación y test se encuentran almacenados en tr.best_perf, tr.best_vperf y tr.best_tperf. Por lo tanto, lo que hay que hacer es almacenar, después de cada entrenamiento, al final de cada iteración del bucle, cada uno de esos tres valores en tres distintos arrays, uno para los errores de entrenamiento, otro para los de validación y otro para los de test, y, después del bucle, calcular el valor promedio de cada uno de esos arrays con la función mean.

Por lo tanto, suponiendo que los errores en entrenamiento, validación y test se almacenen en las variables errorEntrenamiento, errorValidacion y errorTest, el bucle podría ser similar a la siguiente (para el problema building_dataset):

```
errorEntrenamiento = [];
errorValidacion = [];
errorTest = [];

for i=1:50,
```

```
      rna = feedforwardnet(arquitectura);
      rna.trainParam.showWindow = false;
      [rna tr] = train(rna, buildingInputs, buildingTargets);

      errorEntrenamiento(end+1) = tr.best_perf;
      errorValidacion(end+1)    = tr.best_vperf;
      errorTest(end+1)          = tr.best_tperf;
end;
```

De esta manera, se almacenan los errores en los conjuntos de entrenamiento, validación y test. De estos 3 conjuntos, el que más interesa es el de test, y se puede ver el promedio de valores obtenidos y su desviación típica con:

```
mediaErrorTest = mean(errorTest);
desviacionTipicaErrorTest = std(errorTest);
```

Este es el valor que indica la bondad de esa arquitectura.

En un problema de clasificación no se almacenaría el error cometido sino la precisión, que, como se ha indicado anteriormente, se calcula mediante la función *confusion*. Para saber qué patrones se utilizaron en cada conjunto, en tr.trainInd, tr.valInd y tr.testInd están almacenados los índices de los patrones utilizadas para entrenamiento, validación y test respectivamente. Por lo tanto, el código del bucle podría ser el siguiente:

```
precisionEntrenamiento = [];
precisionValidacion = [];
precisionTest = [];
for i=1:50,
       rna = patternnet(arquitectura);
       rna.trainParam.showWindow = false;
       [rna tr] = train(rna, irisInputs, irisTargets);
       irisOuputs = sim(rna, irisInputs);

       precisionEntrenamiento(end+1) = 1-confusion(irisTargets(:,tr.trainInd),
                                          irisOuputs(:,tr.trainInd));
       precisionValidacion(end+1) = 1-confusion(irisTargets(:,tr.valInd),irisOuputs(:,tr.valInd));
       precisionTest(end+1) = 1-confusion(irisTargets(:,tr.testInd), irisOuputs(:,tr.testInd));
end;
```

De esta manera, se almacenan las precisiones en los conjuntos de entrenamiento, validación y test. De estos 3 conjuntos, el que más interesa es el de test, y se puede ver el promedio de valores obtenidos y su desviación típica con:

```
mediaPrecisionTest = mean(precisionTest);
desviacionTipicaPrecisionTest = std(precisionTest);
```

Este es el valor que indica la bondad de esa arquitectura.

En ambos casos, regresión y clasificación, si se quiere evaluar la bondad de distintas arquitecturas, es necesario incluir este bucle dentro de otro, en el que se itere sobre esas arquitecturas.

4.- Ejercicios

Coger la base de datos escogida en la práctica de la semana anterior, cargar los datos en Matlab, y realizar los siguientes ejercicios:

- Crear un script con el código para entrenar una RNA con una capa oculta, con distinto número de neuronas. Entrenar cada RNA hasta que los resultados sean satisfactorios. Pasar posteriormente a dos capas ocultas, y probar distintas configuraciones de neuronas en las capas. Entrenar también repetidas veces hasta que los resultados sean satisfactorios.
    - Un script es un archivo de texto con extensión .m
    - Es necesario tener en cuenta que entrenar varias veces no implica llamar a la función train varias veces de forma repetida, puesto que, de esta manera, se está entrenando una red que ya ha sido entrenada. En su lugar, para cada vez que se quiera entrenar, será necesario crear la red (mediante feedforardnet o patternnet) y llamar a la función train para inicializar la red y entrenarla.
    - ¿Cuándo y por qué se para el entrenamiento?
    - Una vez que el entrenamiento se para, ¿cuáles son los resultados que se obtienen? ¿A qué ciclo corresponden? ¿Por qué son los de ese ciclo y no los de otro?
- Crear un script con el código necesario para que evalúe la bondad de distintas arquitecturas y muestre los resultados en pantalla (media y desviación típica). Copiar estos valores y elaborar una tabla de resultados que muestre una comparativa de las distintas arquitecturas.
    - ¿Qué arquitectura es mejor para resolver este problema concreto?

Estos 2 ejercicios tendrán que ser defendidos en la siguiente clase de prácticas.