PRÁCTICA 1

Patricia Cuesta Ruiz

Víctor Gamonal Sánchez

ID CONTROL NEURONAL (I)

Ejercicio 1. Perceptron

Se desea clasificar un conjunto de datos pertenecientes a cuatro clases diferentes. Los datos y las clases a las que pertenecen con los que se muestra a continuación:

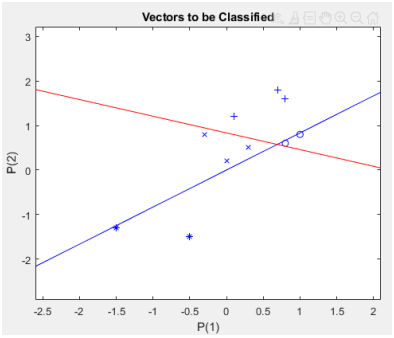
x_1	x_0	Clase
0.1	1.2	2
0.7	1.8	2
0.8	1.6	2
0.8	0.6	0
1.0	0.8	0
0.3	0.5	3
0.0	0.2	3
-0.3	0.8	3
-0.5	-1.5	1
-1.5	-1.3	1

Se desea diseñar un clasificador neuronal mediante un perceptrón simple que clasifique estos datos. Diseñe el clasificador, visualice los parámetros de la red y dibuje los datos junto con las superficies que los separan.

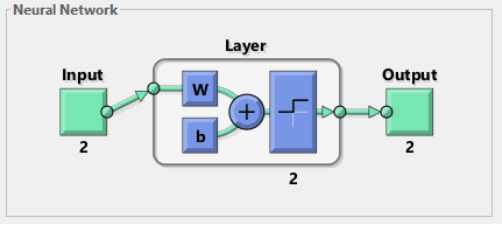
¿Consigue la red separar los datos?, ¿cuántas neuronas tiene la capa de salida?, ¿por qué?

¿Qué ocurre si se incorpora al conjunto un nuevo dato: [0.0 -1.5] de la clase 3?

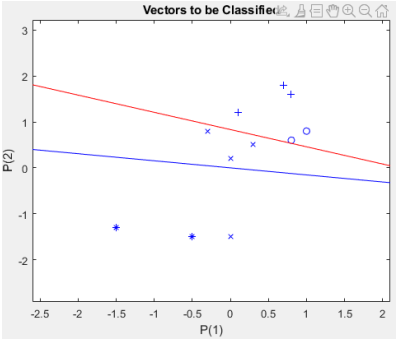
La red consigue separar los datos.



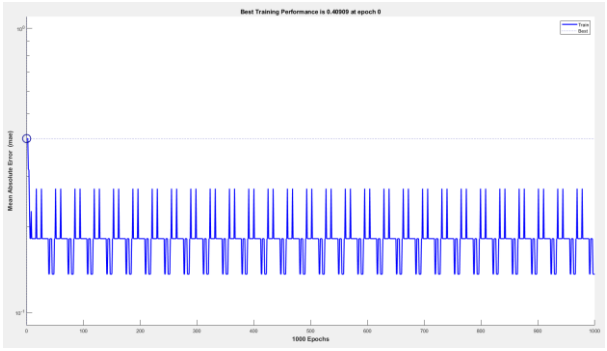
La capa de salida tiene dos neuronas como se puede ver en el esquema de la red.



Si añadimos el nuevo dato la red no es capaz de hallar las rectas que dividen los puntos.



Y como podemos observar, por mucho que se intente reducir el error se puede observar que el mínimo error solo se obtiene con esas rectas.



Ejercicio 2. Aproximación de funciones

Una de las aplicaciones inmediatas de las redes neuronales es la aproximación de funciones. Para ello, Matlab dispone de una red optimizada, fitnet, con la que se trabajará en este ejercicio. El objetivo en este caso es aproximar la función $f = sinc(t)$ tal y como se muestra a continuación:

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% APROXIMACIÓN DE FUNCIONES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all; close all;

%    DEFINICIÓN DE LOS VECTORES DE ENTRADA-SALIDA
%    ============================================

t = -3:.1:3; % eje de tiempo
F=sinc(t)+.001*randn(size(t)); % función que se desea aproximar

plot(t,F,'+');
title('Vectores de entrenamiento');
xlabel('Vector de entrada P');
ylabel('Vector Target T');

%    DISEÑO DE LA RED
%    ================

hiddenLayerSize = 4;
net = fitnet(hiddenLayerSize,'trainrp');

net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

net = train(net,t,F);

Y=net(t);

plot(t,F,'+'); hold on;
plot(t,Y,'-r'); hold off;
title('Vectores de entrenamiento');
xlabel('Vector de entrada P');
ylabel('Vector Target T');
```

Estudie los efectos sobre la solución final de modificar el método de entrenamiento (consulte la ayuda de Matlab y pruebe 4 métodos diferentes) y el número de neuronas de la capa oculta.

En este ejercicio vamos a utilizar cuatro distintos métodos de entrenamiento, incluyendo el predeterminado y que hemos visto en las diapositivas de clase (trainrp):

- trainlm
- trainbr
- traingd

También vamos a modificar el número de neuronas de la capa oculta dependiendo del método utilizado por valores bajos, medios (y normales) y altos para ver los distintos resultados que nuestra red consigue darnos y si es capaz o no de aprender.

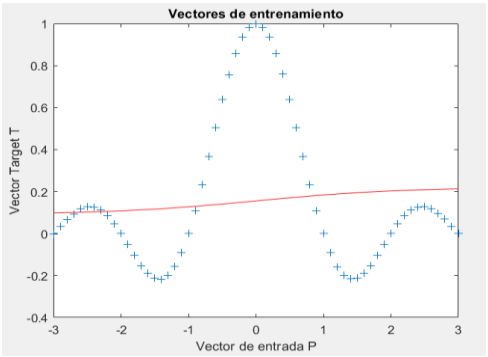
Todo ello lo modificaremos en la línea de código **net = fitnet(hiddenLayerSize, 'trainrp')** donde el primer parámetro representa el número de neuronas de la capa oculta y el segundo, el método de entrenamiento escogido.

1. Método de entrenamiento TRAINRP (Resilient Backpropagation)

Para este método utilizaremos el rango del número de neuronas [1, 5, 35] como valores bajo, medio y alto, respectivamente.

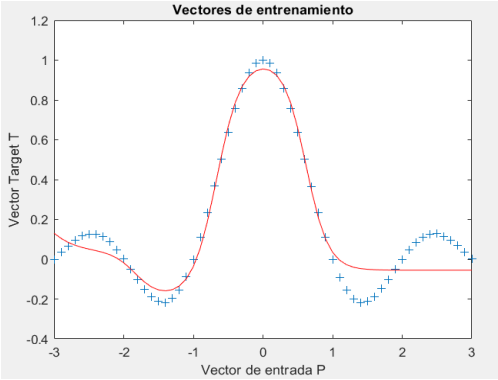
a. Neuronas de capa oculta = 1

A continuación, en la figura, se puede ver cómo la red obtiene y muestra un resultado que no se ajusta para nada a la función esperada al tratarse de un número de neuronas demasiado bajo.



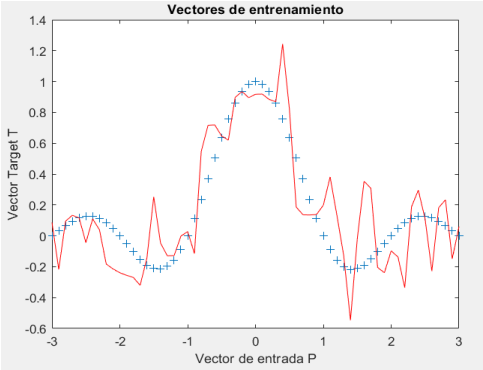
b. Neuronas de capa oculta = 5

A continuación, en la figura, se puede ver cómo la red aprende y muestra un resultado bastante más acertado y bueno que antes, pero que aun así no es capaz de ajustarse totalmente a la función que queremos aproximar.

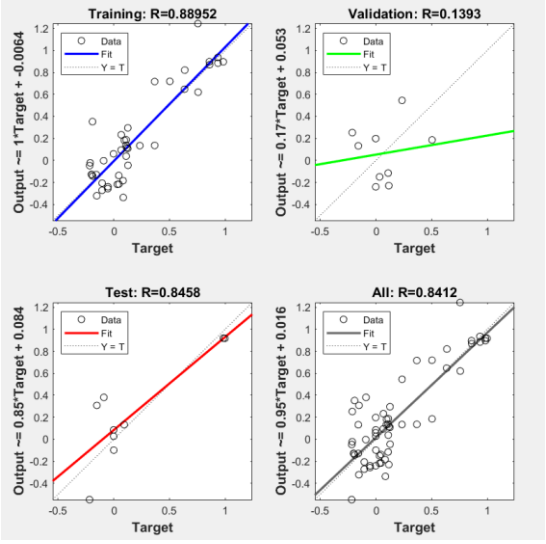


c. Neuronas de capa oculta = 35

A continuación, en la figura, se puede ver cómo esta vez la red aprende demasiado todos los datos de entrenamiento mostrando resultados con muchos picos y nada ajustados a la función que queremos aproximar.



Además, es evidente que con este resultado tampoco la red podrá adaptarse a la función con los datos del test y de la validación, algo que se ve más claramente usando las gráficas de regresión (plotregression).

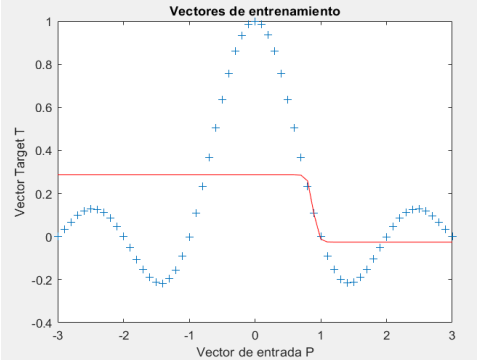


2. Método de entrenamiento TRAINLM (Levenberg-Marquardt)

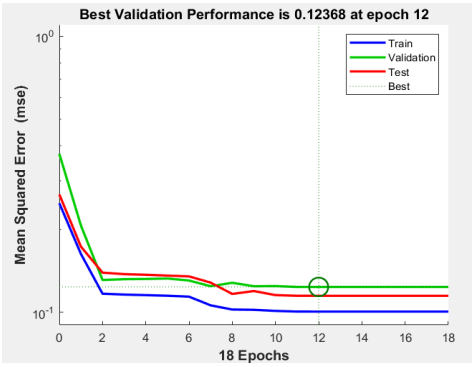
Para este método utilizaremos el rango del número de neuronas [1, 8, 35] como valores bajo, medio y alto, respectivamente.

a. Neuronas de capa oculta = 1

A continuación, en la figura, se puede ver cómo la red obtiene y muestra un resultado que al principio se mantiene constante y que no se ajusta a la función pero que llega a un punto final en el que parece aprender un poco.

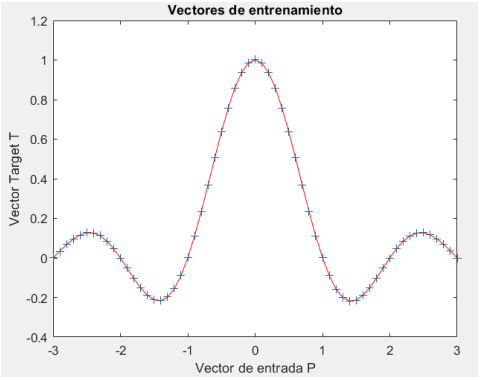


Sin embargo, para los datos de test y validación el error cae al principio pero finalmente se mantiene constante, aun así, está claro que la red no ha aprendido.



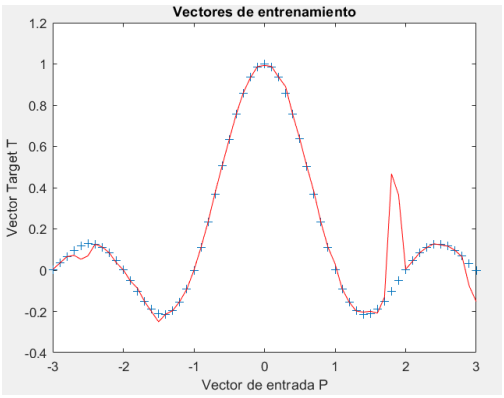
b. Neuronas de capa oculta = 8

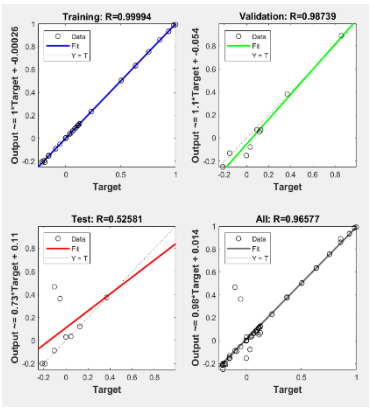
A continuación, en la figura, mediante el uso de un valor medio de neuronas se puede ver cómo la red obtiene y muestra un resultado que se ajusta exactamente a la función que queremos aproximar, a la perfección.



c. Neuronas de capa oculta = 35

A continuación, en la figura, mediante el uso de un valor muy alto de neuronas se puede ver cómo la red aprende de más y se muestran picos al intentar aproximarse a la función que queremos aproximar; esto quiere decir que se ha aprendido demasiado los valores de entrenamiento y que no da buenos resultados para la validación y el test como se ve en la recta de regresión.



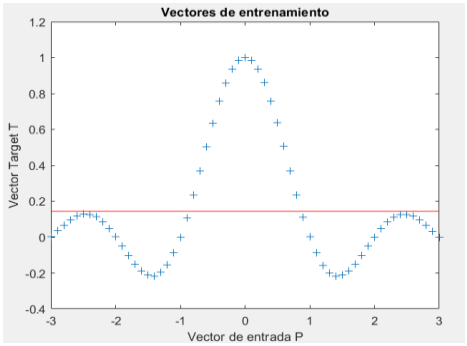


3. Método de entrenamiento TRAINBR (Bayesian Regularization)

Para este método utilizaremos el rango del número de neuronas [1, 5, 150] como valores bajo, medio y alto, respectivamente.

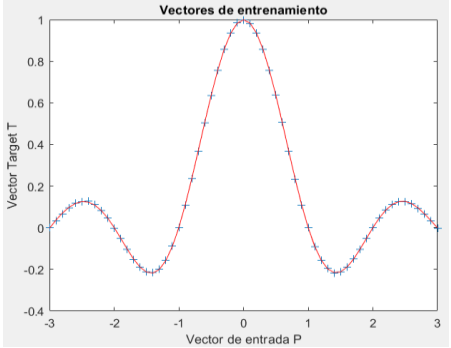
a. Neuronas de capa oculta = 1

A continuación, en la figura, se puede ver cómo la red obtiene y muestra un resultado que es una línea recta y que no se ajusta para nada a la función que queremos aproximar; la red no aprende.



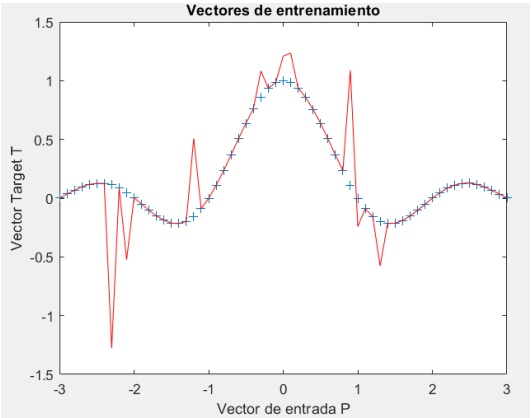
b. Neuronas de capa oculta = 5

A continuación, en la figura, se puede ver cómo la red obtiene, para un valor normal de neuronas, un resultado que es totalmente el esperado y ajustado a la perfección a la función que queremos aproximar; por tanto, la red aprende.



c. Neuronas de capa oculta = 150

A continuación, en la figura, se puede ver cómo la red obtiene, para un valor muy elevado de neuronas, picos durante toda la ejecución del resultado; como mencionamos ya anteriormente, esto se debe a que la red aprende demasiado los datos de entrenamiento, lo que conlleva obtener malos resultados para la fase de validación y test.

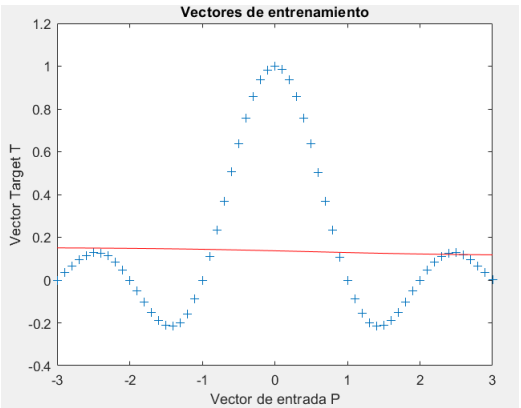


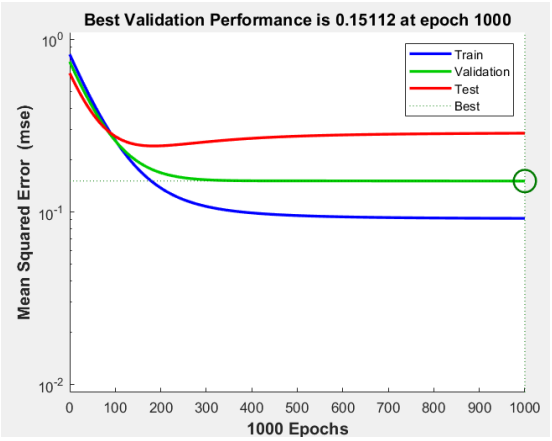
4. Método de entrenamiento TRAINGD (Gradient Descent)

Para este método utilizaremos el rango del número de neuronas [1, 5, 20] como valores bajo, medio y alto, respectivamente.

a. Neuronas de capa oculta = 1

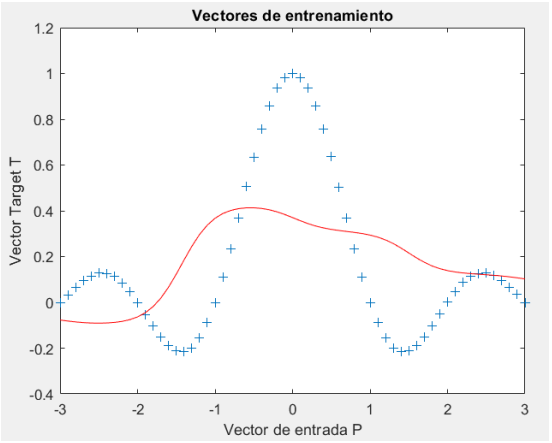
A continuación, en la primera figura, se puede ver cómo la red obtiene y muestra un resultado que es una línea recta y que no se ajusta para nada a la función que queremos aproximar; la red no aprende. Además, se puede observar en la segunda figura cómo el error decrece poco al principio pero que finalmente consiguió mantenerse, lo que apoya este hecho de que la red no ha aprendido.

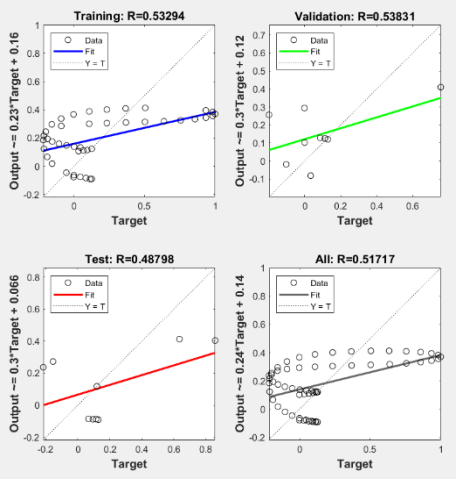




b. Neuronas de capa oculta = 5

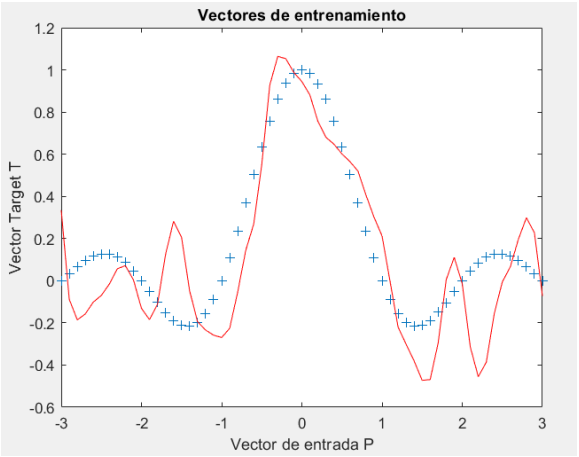
A continuación, en la primera figura, se puede ver cómo la red obtiene y muestra un resultado que, a diferencia del resto que ya hemos visto para valores medios de neuronas en la capa oculta, no se ajusta correctamente a la función que queremos aproximar y por tanto la red no ha podido aprender de forma óptima. En la segunda figura, en la recta de regresión, se aprecia también cómo los datos no se han clasificado bien para ninguna de las tres fases de entrenamiento, validación y test.





c. Neuronas de capa oculta = 20

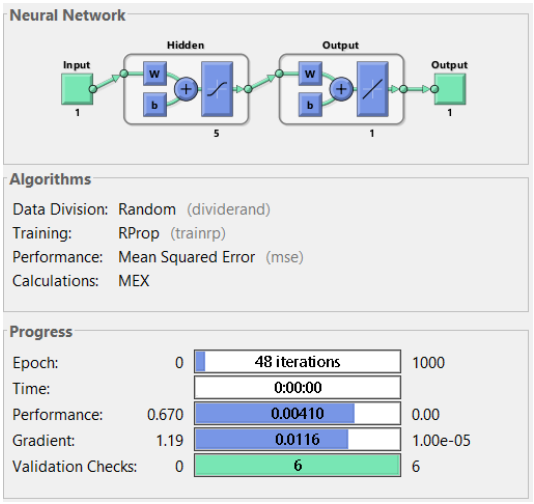
A continuación, en la figura, se puede ver cómo la red obtiene y muestra un resultado que denota que ha aprendido demasiado los resultados de entrenamiento debido a la cantidad de picos que hay y que por tanto, este resultado no es correcto.



De estos resultados obtenidos hasta ahora para cada uno de los métodos de entrenamiento utilizados, podemos deducir que siempre que introducimos un número muy bajo o muy alto de neuronas, la red no es capaz de aprender o por el contrario, aprende demasiado los valores de entrenamiento y no es capaz de ajustarse bien a la función de aproximación. Siempre es bueno optar por un valor de neuronas medio, que por nuestra experiencia hemos deducido que va entre 5 y 20.

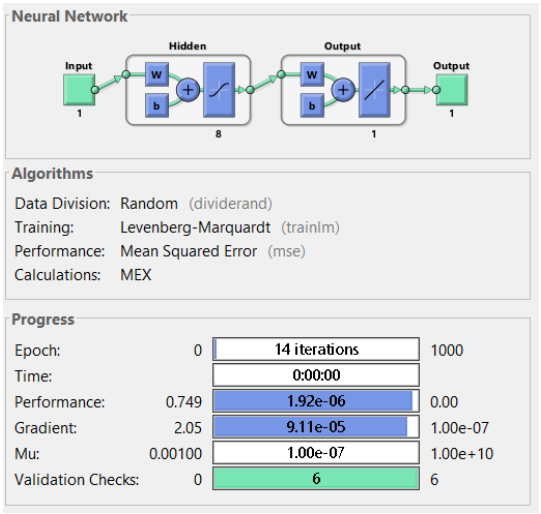
Para cada método de entrenamiento, además, hemos observado lo siguiente en cuanto al rendimiento para los valores óptimos (número de iteraciones agotadas, performance y tiempo medio):

5. Método de entrenamiento TRAINRP



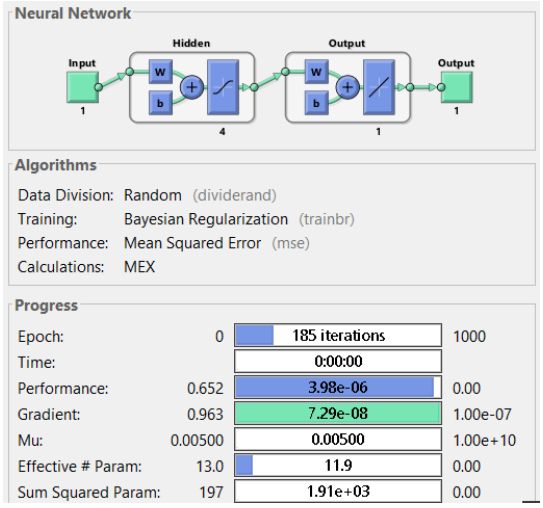
Ofrece un buen rendimiento, ya que como se puede observar el número de iteraciones es muy bajo y el tiempo requerido para entrenar la red ha sido casi nulo, instantáneo.

6. Método de entrenamiento TRAINLM



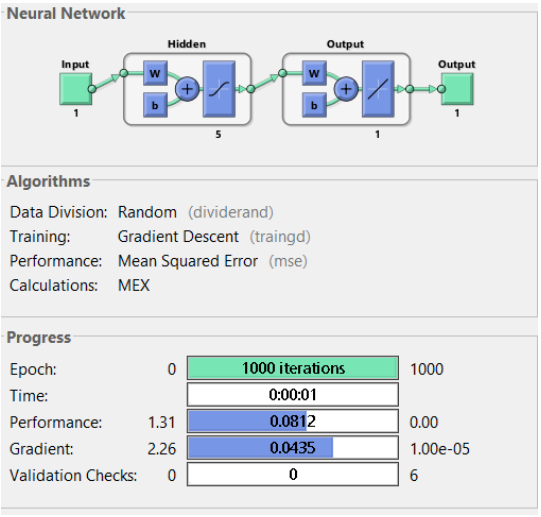
Desde nuestra experiencia ha sido este el método que mejores resultados ha ofrecido en cuanto a tiempo, iteraciones y performance (siendo los más bajos).

7. Método de entrenamiento TRAINBR



Este método tiene datos bastante buenos en general, aunque el performance ligeramente más alto que el anterior.

8. Método de entrenamiento TRAINGD



Este método de entrenamiento agota todas las iteraciones y además tarda más que los anteriores.

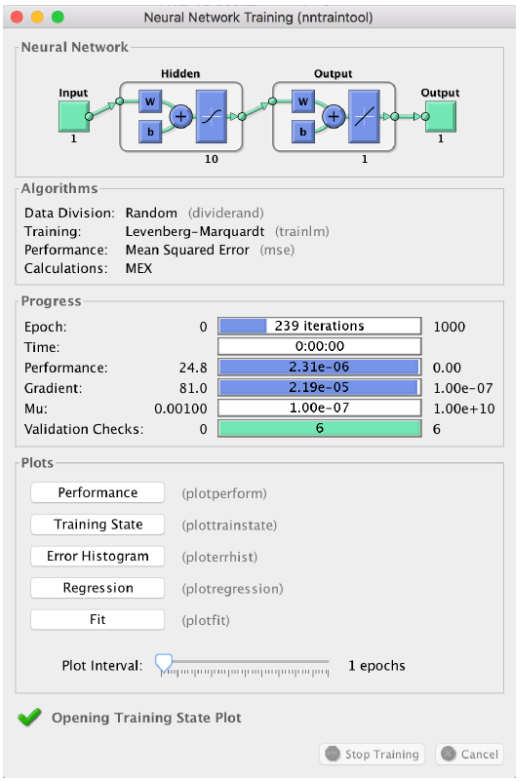
Como conclusión podemos decir que el método de entrenamiento que mejor rendimiento ofrece es **TRAINLM** (en cuanto a número de iteraciones, performance y tiempo), aunque el que más se ha aproximado sin duda a la función y con la mayor precisión ha sido **TRAINBR**.

Ejercicio 3. Aproximación de funciones (II)

En este ejercicio, se estudiarán en detalle las herramientas que facilita Matlab para el diseño y prueba de redes neuronales ejecutando el siguiente código de ejemplo:

```
% Carga de datos de ejemplo disponibles en la toolbox
[inputs,targets] = simplefit_dataset;

% Creación de la red
hiddenLayerSize = 10;
net = fitnet(hiddenLayerSize);

% División del conjunto de datos para entrenamiento, validación y test
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Entrenamiento de la red
[net,tr] = train(net,inputs,targets);

% Prueba
outputs = net(inputs);
errors = gsubtract(outputs,targets);
performance = perform(net,targets,outputs)

% Visualización de la red
view(net)
```

Al ejecutar el script, se muestra la siguiente ventana:



Esquema de la red

Detalles de los algoritmos

Progreso del proceso de entrenamiento y test

Gráficas con diversos resultados

Explore las gráficas disponibles:

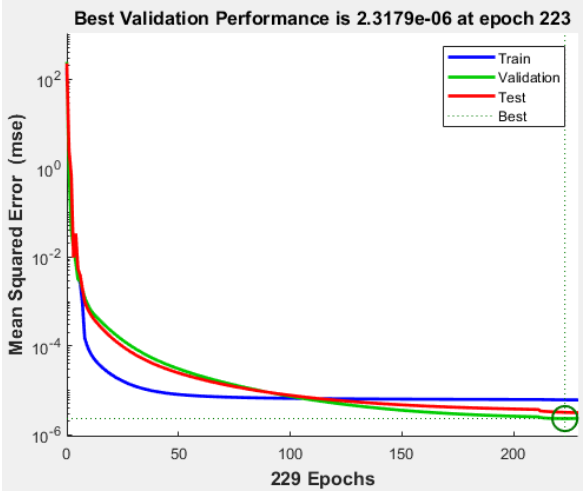
- **plotperform: gráfica que representa el error en función del número de épocas para los datos de entrenamiento, validación y test.**
- **plottrainstate: evolución del entrenamiento.**
- **plotrrhist: histograma del error.**
- **plotregression y plotfit: ajuste de los datos de entrenamiento, validación y test.**

Pruebe este mismo script con el conjunto de datos bodyfat_dataset, y evalúe sus resultados. Estudie la mejora que supone utilizar distintos métodos de entrenamiento y una división diferente de los datos (entrenamiento, validación y test).

Realizando la ejecución de la red propuesta sobre **simplefit_dataset** se obtienen los siguientes resultados.

1. Plotperform

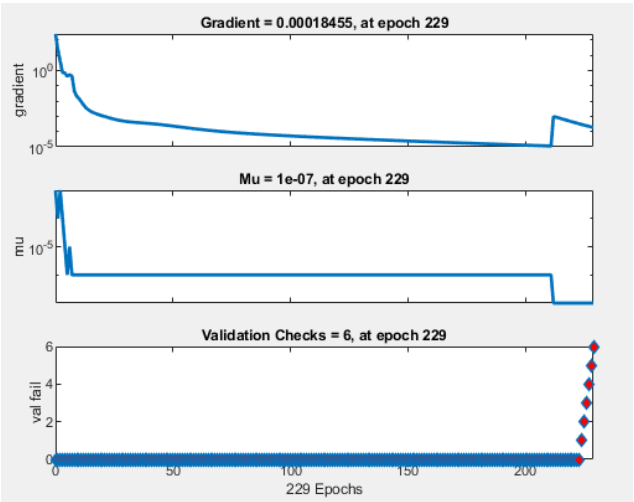
En esta gráfica vemos el error de cada conjunto de datos (entrenamiento, validación y test) para la red en cada época. Podemos ver como el error cae con el tiempo según avanza el entrenamiento reduciéndose para cada uno de los conjuntos de datos.



2. Plottrainstate

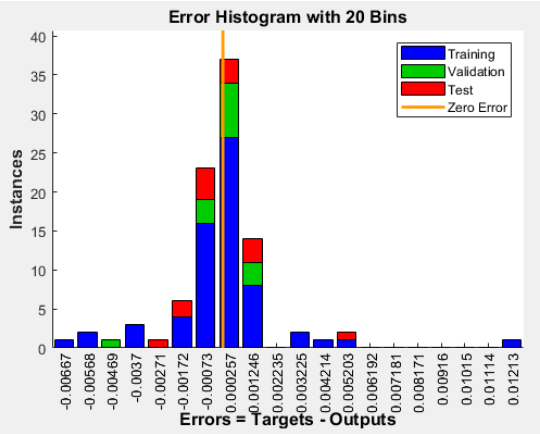
La gráfica se divide en tres gráficas, la primera de ellas nos muestra el gradiente, la segunda la ganancia de entrenamiento y la última los incrementos de error consecutivos en los datos de validación.

Se puede observar como el entrenamiento termina al llegar a seis incrementos seguidos. El gradiente nos informa sobre lo rápidos o lentos que son los cambios que se realizarán en le proceso de backpropagation en la siguiente iteración.



3. Ploterrhist

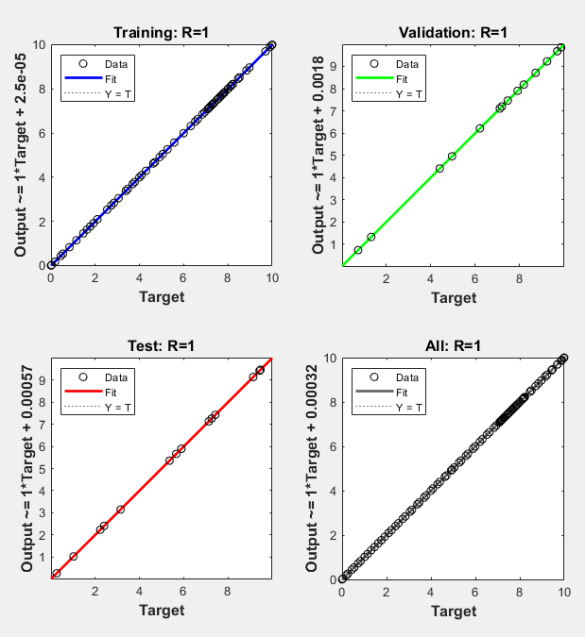
El histograma de error nos indica la cantidad de error ocurrida para cada dato de los datos de entrada de la red. En su gran mayoría para este caso concreto se han distribuido sobre errores muy bajos y de forma más o menos uniforme a lo largo de los datos de entrenamiento, test y validación.



4. Plotregression

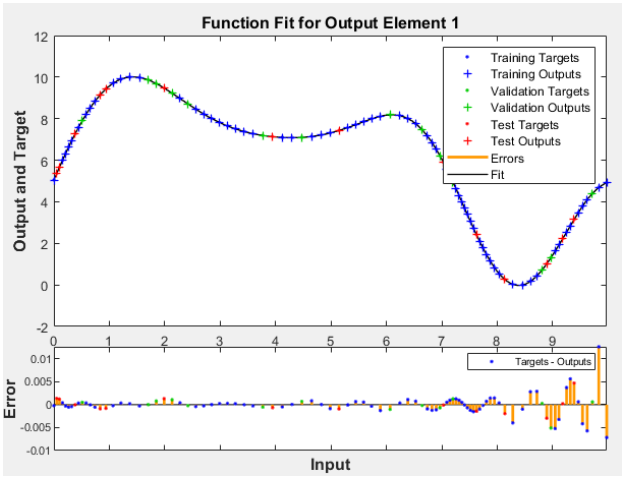
Estas gráficas nos representan el estado final de la red tras haber sido entrenada. En ellas se muestra la relación entre las salidas obtenidas y el resultado esperado sobre cada uno de los conjuntos de datos que manejamos.

La inclinación y posición de la recta de regresión obtenida sobre los puntos de la gráfica representará la desviación media entre los resultados obtenidos y los esperados.

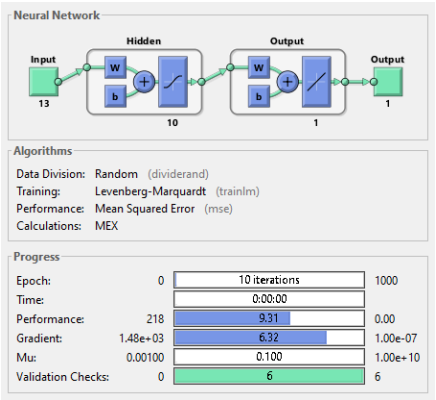


5. Plotfit

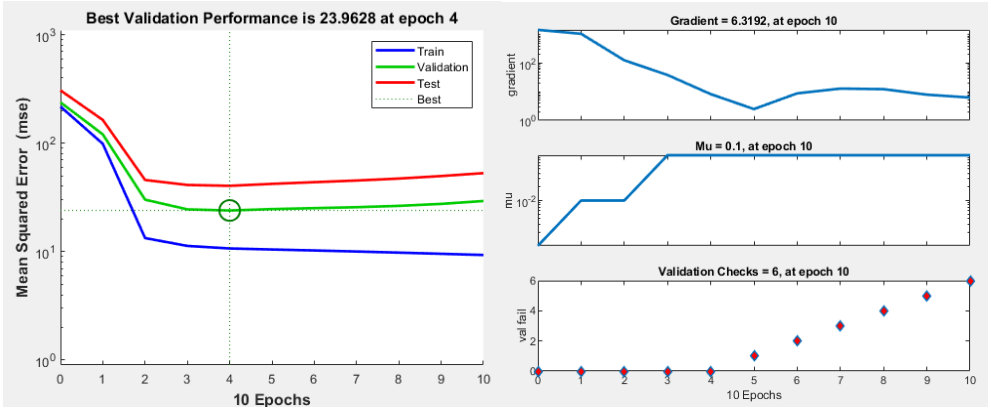
Nos genera una representación visual de la función a la cual la salida de la red se ajusta. Cuanto más se aproxime la función generada respecto a la función ideal que representan los resultados esperados mejor habrá sido el entrenamiento y menor será el error.



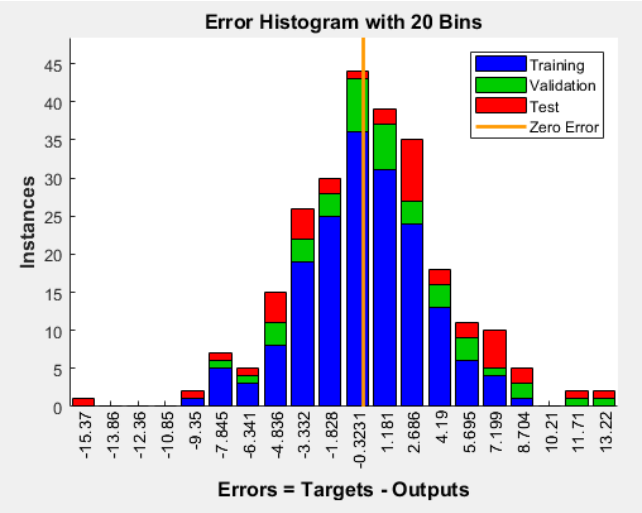
A continuación analizaremos los datos del **bodyfat_dataset**. En este caso debido al tipo de datos utilizados tendremos 13 neuronas de entrada y 1 de salida.



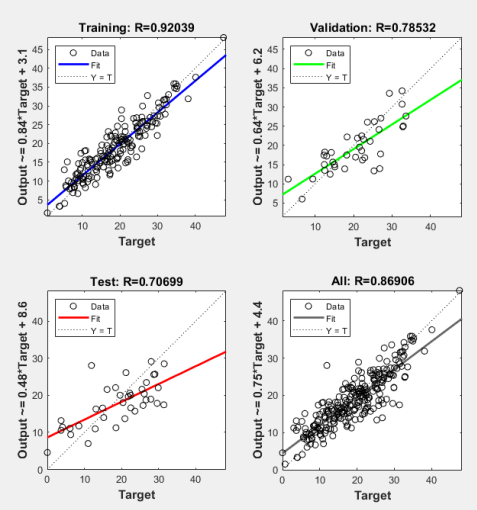
Podemos ver que no se han realizado todas las iteraciones que podrían haberse hecho de modo que los errores de los datos de validación han cortado el aprendizaje con prontitud. Se puede obvervar como en la primera época la red no ha sido capaz de aprender nada de los datos de entrada de modo que su error ha aumentado. En respuesta la red ha aumentado el learning rate (mu) logrando en este punto aprender de los datos. Hasta la época 3 sigue aprendiendo pero a partir de ella a pesar del error de entrenamiento cae en los datos de validación aumenta.



En el histograma se puede ver como los errores son mayores que en el caso anterior, También se puede observar como los errores están distribuidos por igual.



En las rectas de regresión vemos como el entrenamiento no ha sido perfecto. Los mejores datos se obtienen con los datos del entrenamiento y los peores con los datos del test.

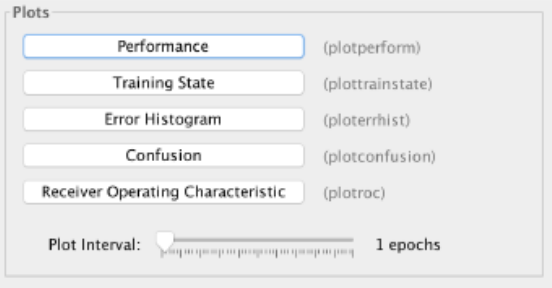


Ejercicio 4. Clasificación.

La clasificación de patrones es una de las aplicaciones que dieron origen a las redes neuronales artificiales. Como en el caso anterior, la toolbox de redes neuronales de Matlab dispone de una red optimizada para la clasificación, patternnet, que analizaremos en este ejemplo.

```
% Carga de datos de ejemplo disponibles en la toolbox
[inputs,targets] = simpleclass_dataset;

% Creción de una red neuronal para el reconocimiento de patrones
hiddenLayerSize = 10;
net = patternnet(hiddenLayerSize);

% División del conjunto de datos para entrenamiento, validación y test
net.divideParam.trainRatio = 70/100;
net.divideParam.valRatio = 15/100;
net.divideParam.testRatio = 15/100;

% Entrenamiento de la red
[net,tr] = train(net,inputs,targets);

% Prueba
outputs = net(inputs);
errors = gsubtract(targets,outputs);
performance = perform(net,targets,outputs)

% Visualización
view(net)
```

La ejecución del script muestra una ventana similar a la anterior en la que cambian algunas de las gráficas disponibles para mostrar los resultados como se ve en la siguiente figura:



En lugar de las gráficas específicamente relacionadas con la aproximación de una función, en el caso de una tarea de clasificación, se ofrecen:

- **plotconfusion: matrices de confusión de los resultados.**
- **plotroc: curvas roc (característica operativa del receptor).**

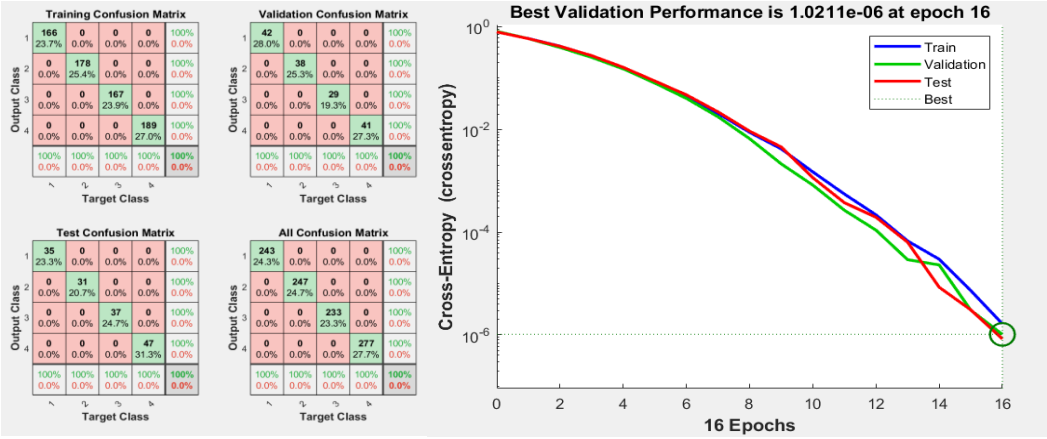
Pruebe este mismo script con el conjunto de datos cancer_dataset, y evalúe sus resultados. Estudie de nuevo la mejora que supone utilizar distintos métodos de entrenamiento y una división diferente de los datos (entrenamiento, validación y test).

Para la realización de este ejercicio es necesario llevar a cabo una clasificación de datos mediante el uso de patternet y del conjunto de datos simpleclass_dataset para ver el funcionamiento. Más adelante cambiaremos este conjunto por cancer_dataset como establece el enunciado del ejercicio.

Además, las herramientas de las cuales haremos más uso serán Confusion (plotconfusion) y Receiving Operating Characteristic (plotroc) para analizar los resultados.

1. Conjunto de datos SIMPLECLASS_DATASET

Para este primer ejemplo con estos datos de entrada y con la división establecida con valores predeterminados (**70% entrenamiento / 15% validación / 15% test**) obtenemos lo siguiente:



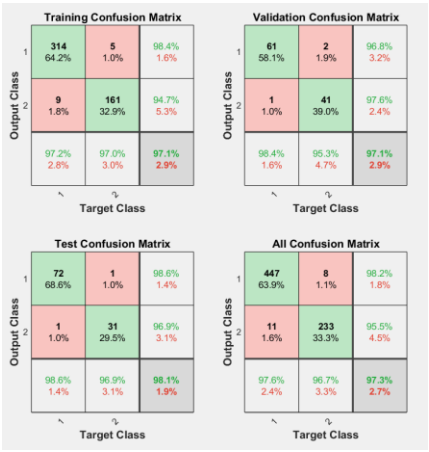
No obtenemos ningún error a la hora de predecir las clases dándole los inputs; además, la gráfica del error muestra cómo no deja de descender hasta haber llegado a un mínimo, es decir, hasta que la red no ha terminado de aprender. Es por esto por lo que los resultados obtenidos son los mejores que se podían obtener.

2. Conjunto de datos CANCER_DATASET

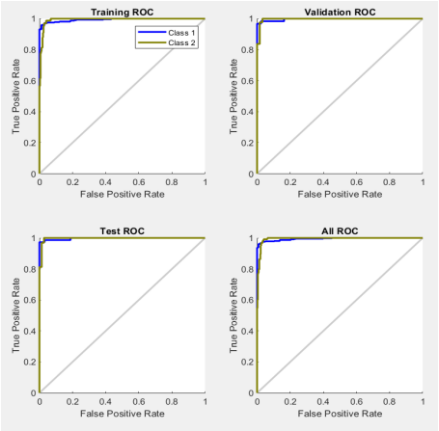
En primer lugar vamos a realizar la división (**70% entrenamiento / 15% validación / 15% test**) y vamos a utilizar el método de entrenamiento trainrnp, aunque lo iremos variando y viendo distintos de ellos para poder compararlos.

a. TTRAINRNP

Se puede observar que la clasificación es muy buena ya que existen pocos datos mal clasificados (en concreto 2,7%). Esto se debe a que este conjunto de datos es más complejo y por tanto es más difícil encontrar patrones.

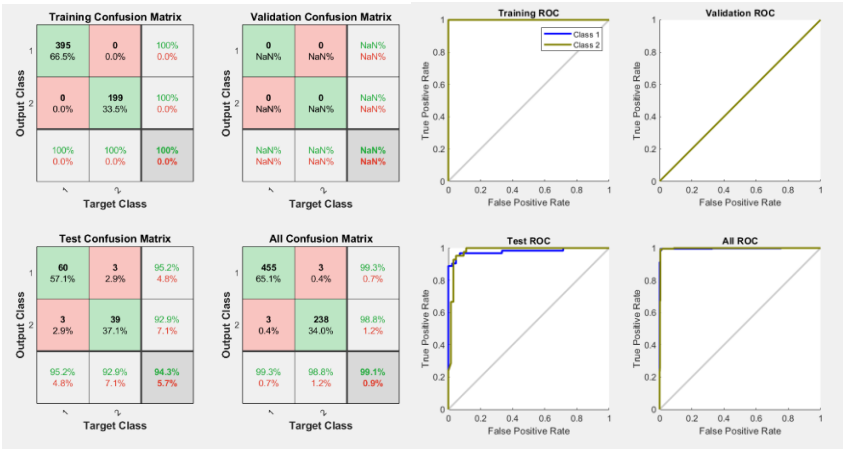


En cuanto a la gráfica de Receiver Operating Characteristic (ROC), es la encargada de relacionar la ratio de los verdaderos positivos con los falsos positivos al entrenar la red para los datos de entrenamiento, validación y test. A continuación, en la figura, se puede comprobar cómo los verdaderos positivos son mayores que los falsos positivos.



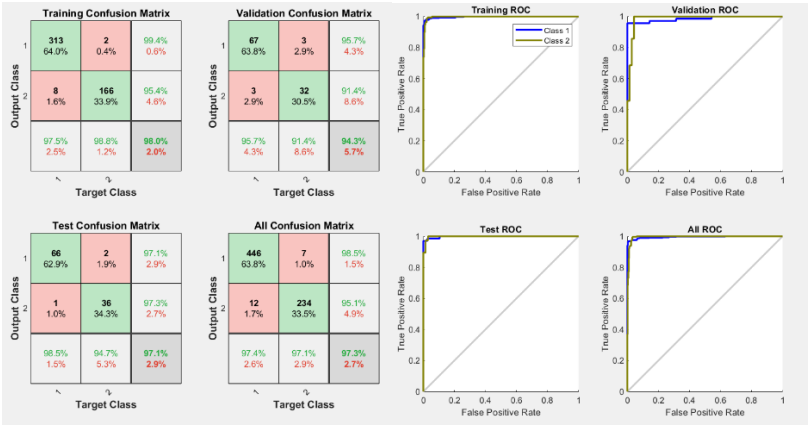
b. TRAINBR

Se puede observar que la clasificación en este caso es aún mejor, pues ofrece menos errores en los datos de entrenamiento que el anterior, pero no en el caso del test por ejemplo que el error es mucho mayor.



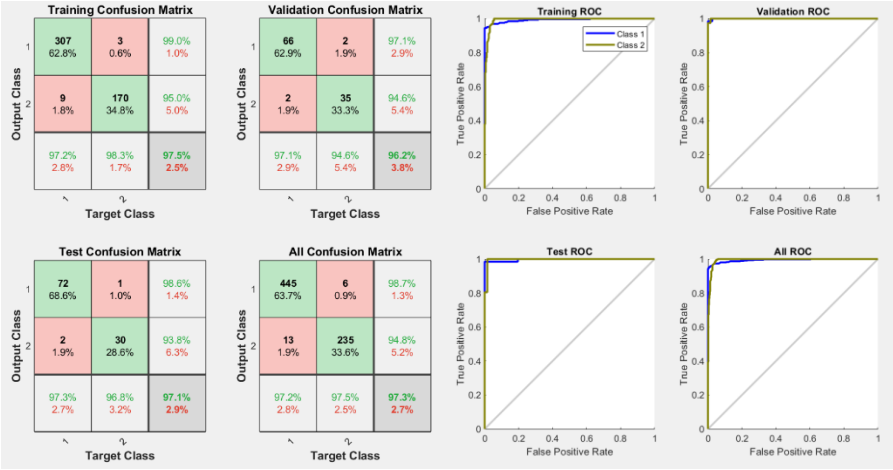
c. TRAINLM

Para este método se podría decir que tanto para entrenamiento, validación y test el error es bastante bueno y parecido a los del primer método que hemos visto antes ya que los falsos positivos y los falsos negativos son muy bajos. El error total es 2,7%.



d. TRAINBFG

Este último método de entrenamiento ofrece peores resultados que los anteriores, pero aun así no son exactamente malos; se puede apreciar cómo los porcentajes de error para los datos de validación son menores, pero sólo para este caso.

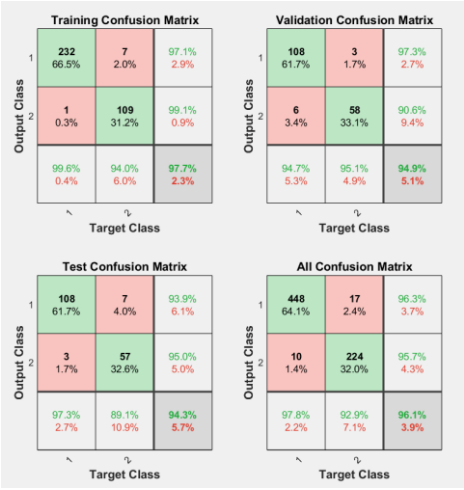


Finalmente, para este conjunto de datos y esta división realizada, consideramos que el método que ofrece mejores resultados es trainrp junto con trainbfg.

A continuación, vamos a realizar distintos cambios en la división de porcentajes utilizando el método de entrenamiento inicial, es decir, trainrp.

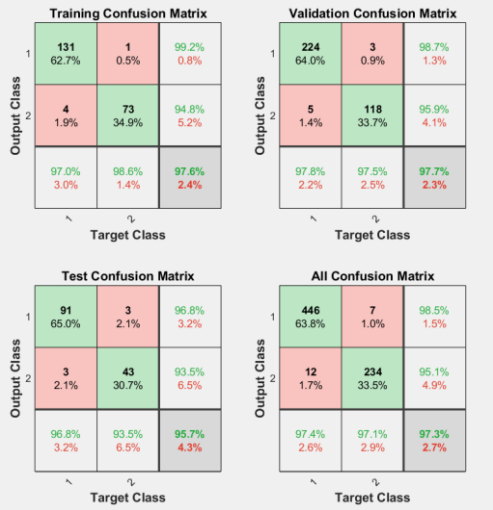
i. División de datos (50% entrenamiento | 25% validación | 25% test)

Se puede apreciar cómo en general el error aumenta, especialmente para el caso de la validación y el test pero no de forma excesiva; la razón es que la cantidad de datos que la red debe entrenar es la misma respecto a los datos que tiene que comprobar si predice correctamente.



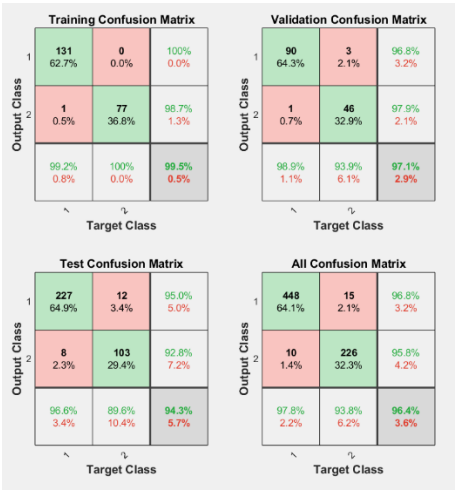
ii. División de datos (30% entrenamiento | 50% validación | 20% test)

Los errores son muy parecidos a los de la división anterior exceptuando los de test, debido a que los datos que la red debe entrenar siguen siendo muy pocos y el entrenamiento no llega a ser el mejor.



iii. División de datos (30% entrenamiento | 20% validación | 50% test)

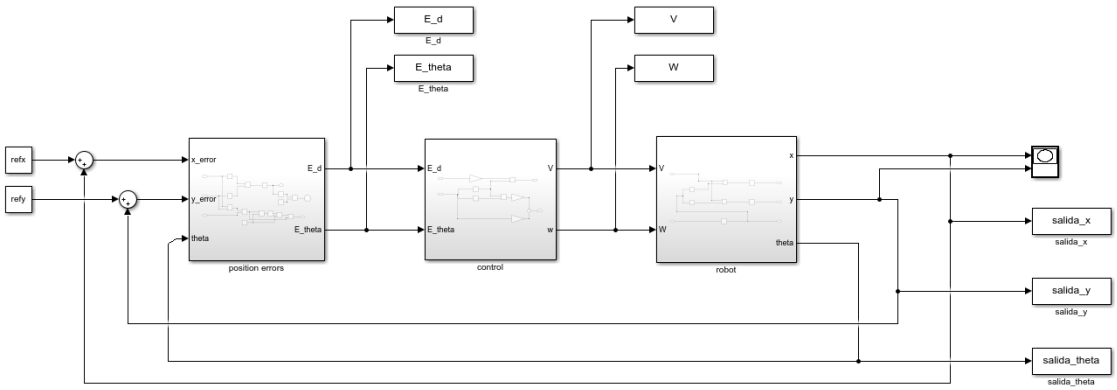
En esta división se puede apreciar cómo el porcentaje de error del test es muy grande comparado con el de entrenamiento y validación; esto es debido a que los datos para el entrenamiento siguen siendo muy pocos y la red no es capaz de encontrar más patrones.



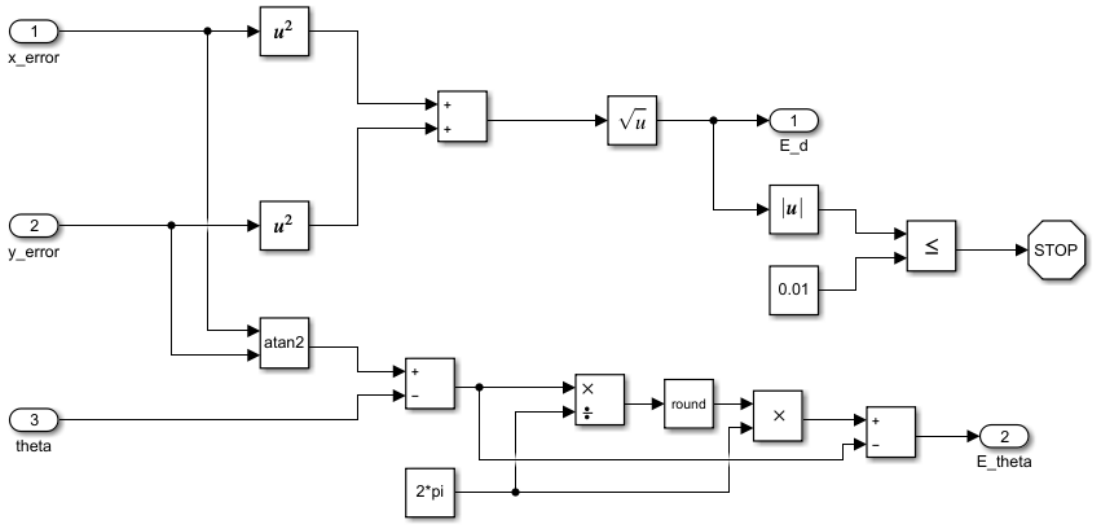
Como conclusión, nos hemos fijado en que la mejor forma de dividir los datos es 70% entrenamiento, 15% validación y 15% test ya que así la red tiene muchos más datos para entrenar y poder identificar así más patrones para poder aplicarlos posteriormente a los datos de test y validación y obtener buenos resultados.

ID CONTROL NEURONAL (II)

Creamos el sistema de comportamiento del robot a partir del creado en la práctica 0 y de la caja negra proporcionada, además creamos un módulo que controla el error de la posición del robot.



El módulo de control del error de la posición controla que se detenga el robot si se encuentra a una distancia menor a 0.01 y que los ángulos con los que se trabaja se mantengan entre [-π,π] ya que se trabaja en radianes.



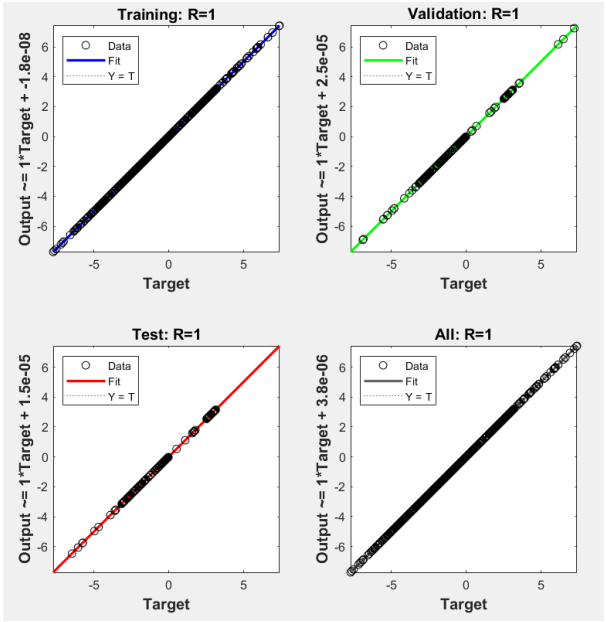
Tras la realización de todo esto, conectamos el modelo de Simulink *'PositionControl.slx'* con el archivo .m de MATLAB para crear el dataset y entrenar la red.

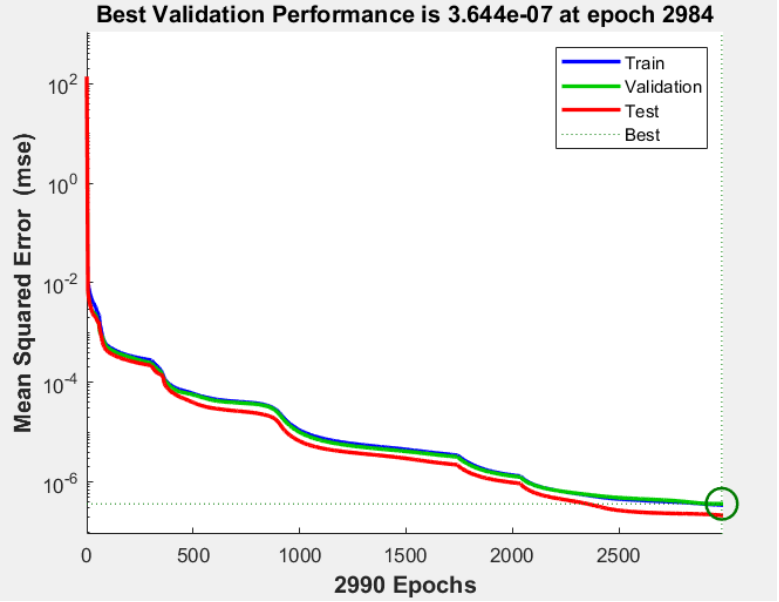
Una vez conectados, hemos generado el dataset mediante el uso de los siguientes parámetros:

- Los valores **rexf**, **rexy** están comprendidos en un rango de valores [-10, 10].
- La **división de datos** que hemos establecido ha sido 80% entrenamiento,

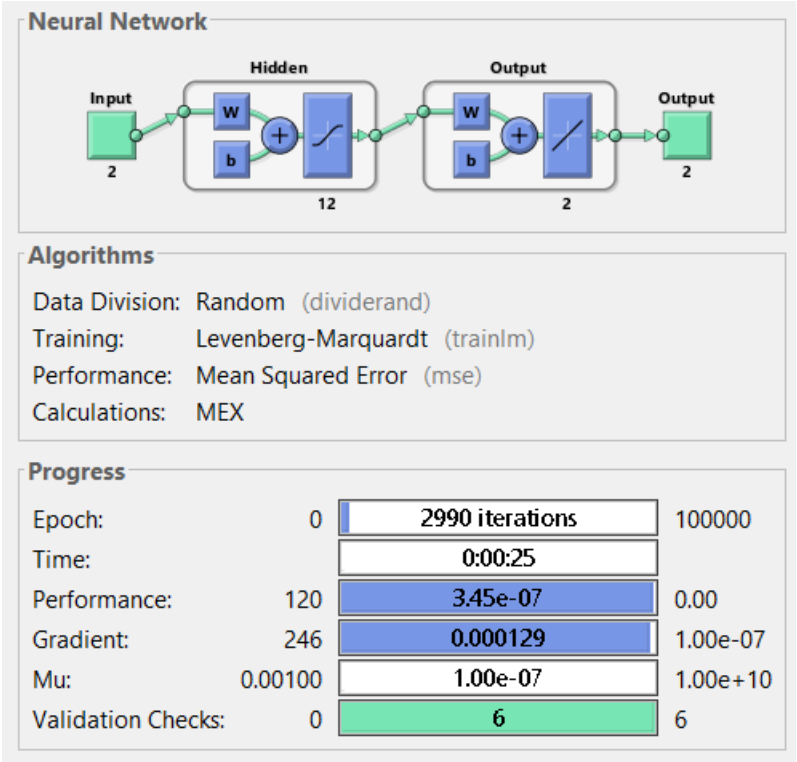
  10% validación y 10% test ya que consideramos que es una división más que eficiente para que la red cuente con una gran cantidad de datos para poder entrenar y aprender de forma correcta.

- El total de **neuronas en capa oculta** que hemos establecido ha sido finalmente **12**, tras realizar varios experimentos entre un rango de 10 y 18 ya que los resultados que nos proporciona son más que satisfactorios como se puede ver a continuación en las siguientes gráficas:

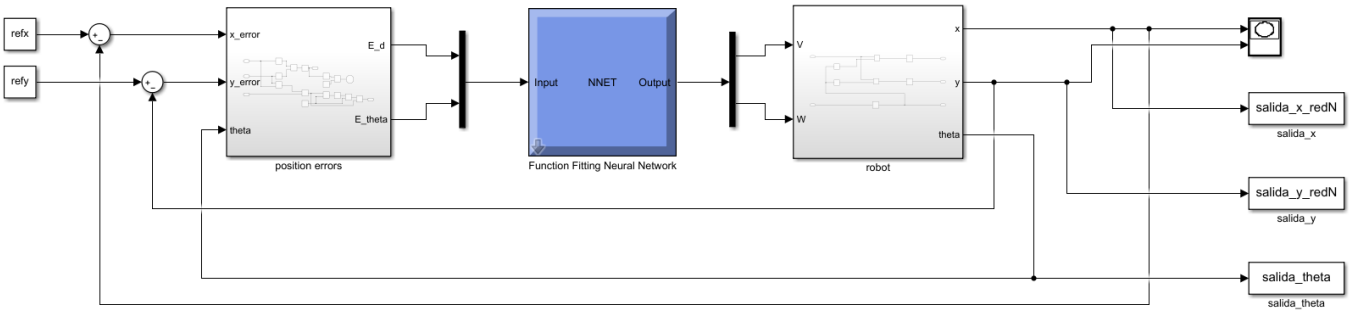




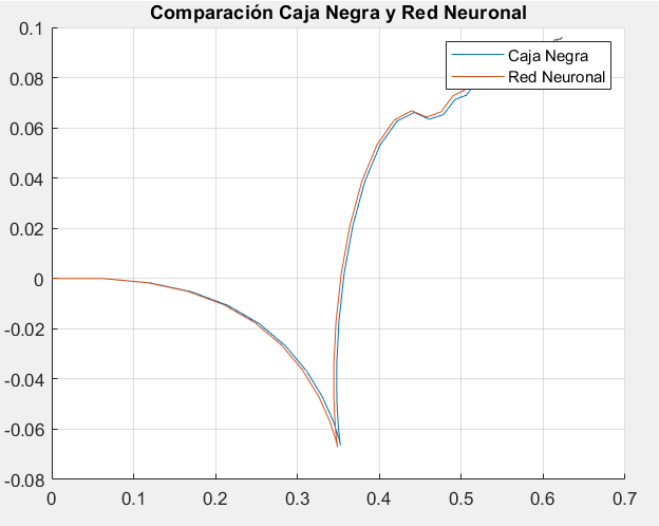
La recta de regresión obtenida es síntoma de un muy buen resultado como adelantábamos y de que la red ha podido aprender, y la performance nos muestra cómo siempre obtengo un error menor de $10^{6.}$

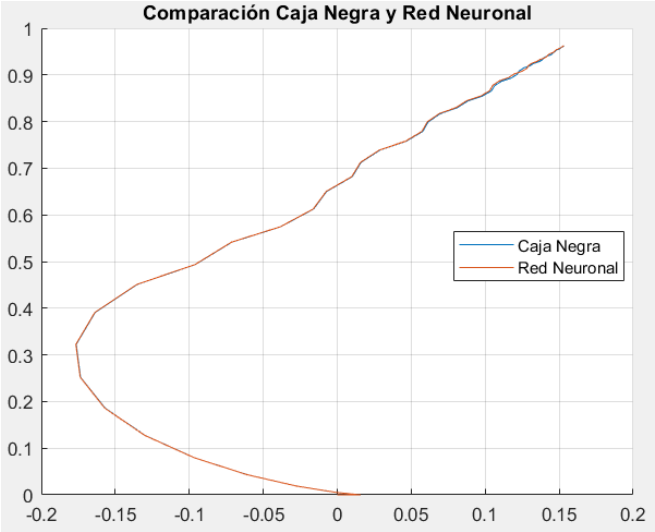


A continuación y como indica el enunciado de la práctica, hemos creado un nuevo archivo ‘PositionControlNet.slx’ en el que hemos utilizado la red neuronal creada mediante la orden gensim(net,Ts) en el archivo .m de MATLAB en lugar del bloque controlador que teníamos anteriormente.



Por último, hemos creado un archivo .m de MATLAB ‘CompararResultados.m’ que nos permita comparar el resultado obtenido tanto por el modelo original como por el de la red neuronal recién creada; para ello simplemente hemos realizado una simulación de cada una de ellas atribuyéndole valores aleatorios para refx y refy. A continuación, se pueden observar estos resultados en las siguientes dos gráficas:

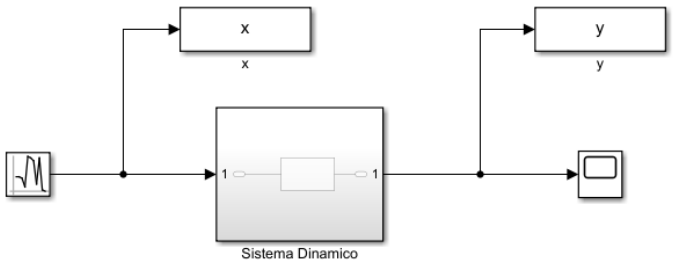




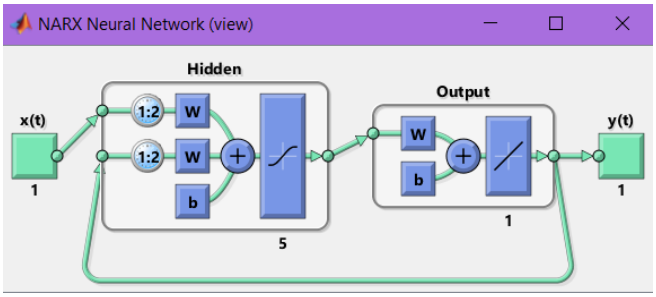
El comportamiento de ambas es prácticamente similar.

ID CONTROL NEURONAL (III)

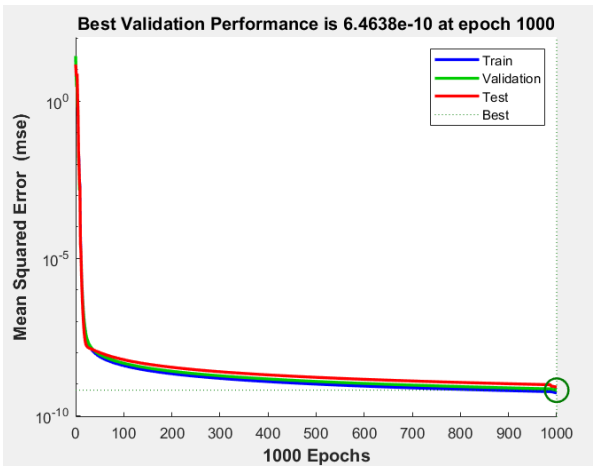
En primer lugar hemos creado el sistema en Simulink utilizando el archivo de caja negra proporcionado en el enunciado de la práctica utilizando las especificaciones requeridas, como el uso de un bloque de entrada de datos aleatoria con los parámetros detallados también en el enunciado (Ts = 0.1, Solver con tiempo discreto, variables X e Y accesibles desde el Workspace con estructura “Structure With Time”), quedando tal que así:

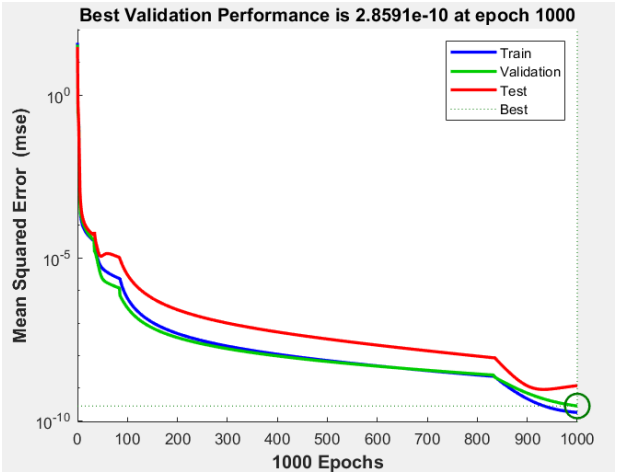


Siguiendo los pasos que se indican en el enunciado, hemos configurado nuestra red NARX de tal manera que hemos usado los valores obtenidos en X e Y para entrenarla, correspondiendo estos valores a las entradas y salidas de la red, respectivamente; además, tiene 5 neuronas en la capa oculta. Tras entrenar la red hemos usado el comando closeloop() para convertirla en recursiva.

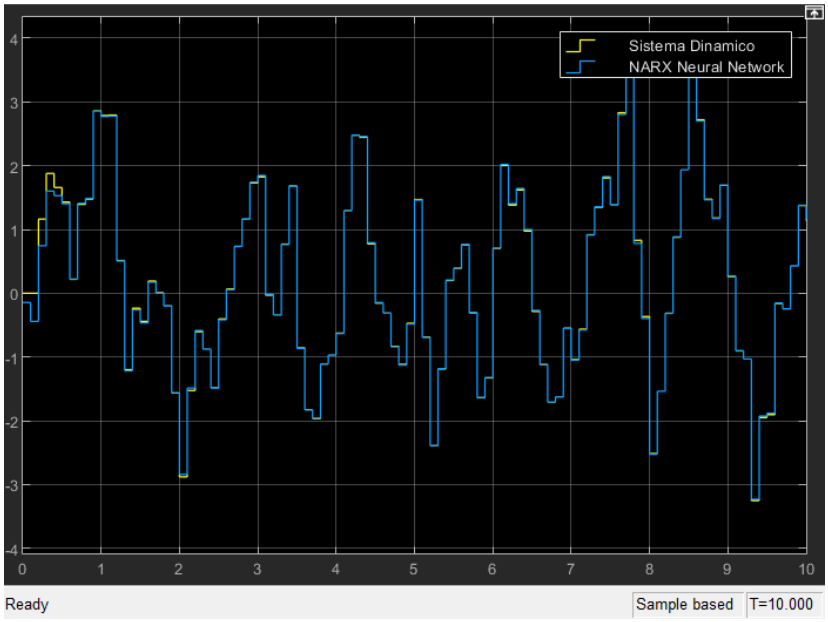


Tras haber realizado varias ejecuciones, hemos obtenido que los resultados para el entrenamiento son muy buenos, obteniendo un error por debajo de 10^{-10}.

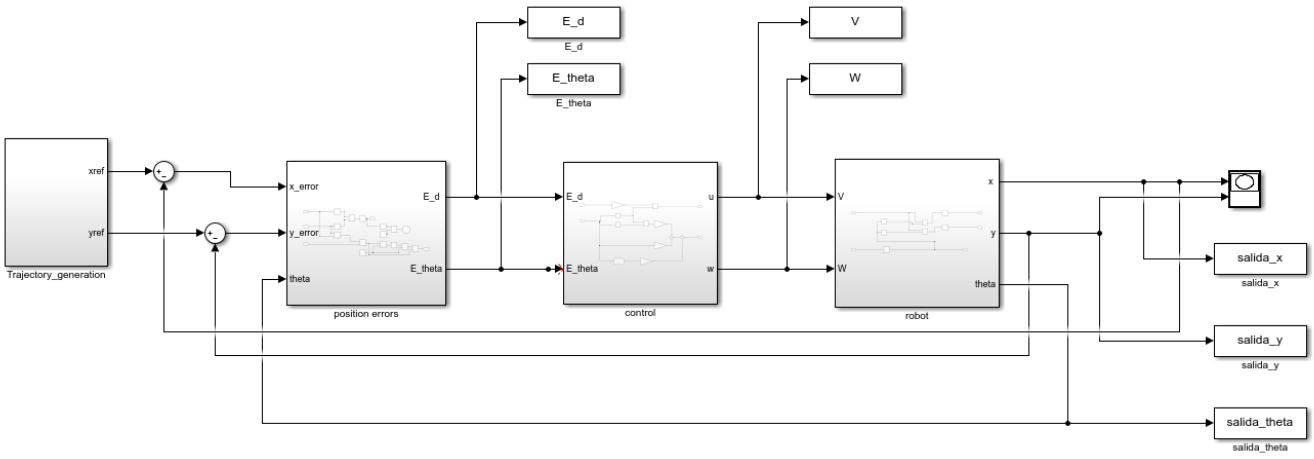




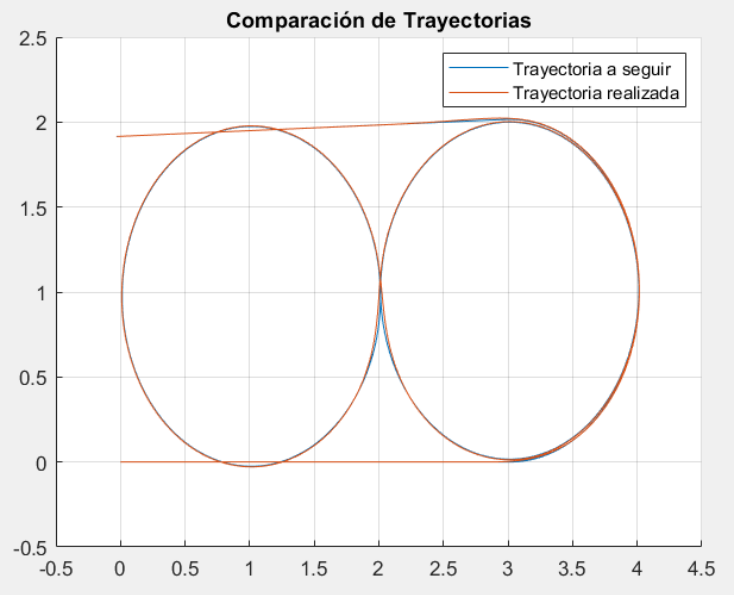
Finalmente, el elemento scope nos permite ver gráficamente el resultado de dos canales que en este caso son el generado por el bloque original y el generado por el bloque de la red. A continuación, se muestran los resultados, donde se puede ver que las dos líneas coinciden prácticamente durante toda la simulación excepto durante el primer segundo, por lo que es bastante satisfactorio.



Para la realización del **Ejercicio 2** de esta parte de la práctica, hemos empezado por modificar el robot utilizado anteriormente tal y como pide el enunciado: hemos cambiado los bloques de posición refx y refy por un subsistema que genera la trayectoria que debe seguir el robot móvil y que se nos facilitaba en los archivos de BlackBoard. Además, hemos sustituido el controlador por uno nuevo de tipo caja negra especial para el seguimiento de trayectorias. El resultado queda así:



Para ejecutarlo, hemos generado un script en MATLAB el cual se encarga de iniciar los parámetros de trayectoria, y de dibujar por tanto una gráfica con la trayectoria que se debe seguir teóricamente y la que sigue realmente el robot. Como se puede ver a continuación y como era de esperar, el robot es capaz de imitar a la perfección la trayectoria del generador.



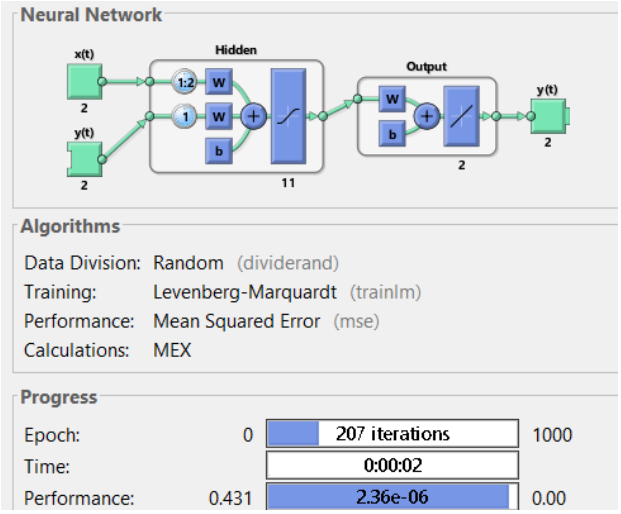
A continuación, se nos pide encontrar el número de neuronas para la capa oculta de la red neuronal NARX para obtener un comportamiento óptimo. Para ello, hemos creado la estructura de la red con 1 capa oculta, probando con diferentes números de neuronas y a la vez observando el resultando que ofrecía al aplicar la trayectoria sobre la red generada. Esta red tiene 1 retenedor para la realimentación de la salida y 2 retenedores para la entrada de datos.

```
net = narxnet(1:2, 1, [i]);

[x,xi,ai,t] = preparets(net,inputsc,{},outputsc);

net = train(net,x,t,xi,ai);
```

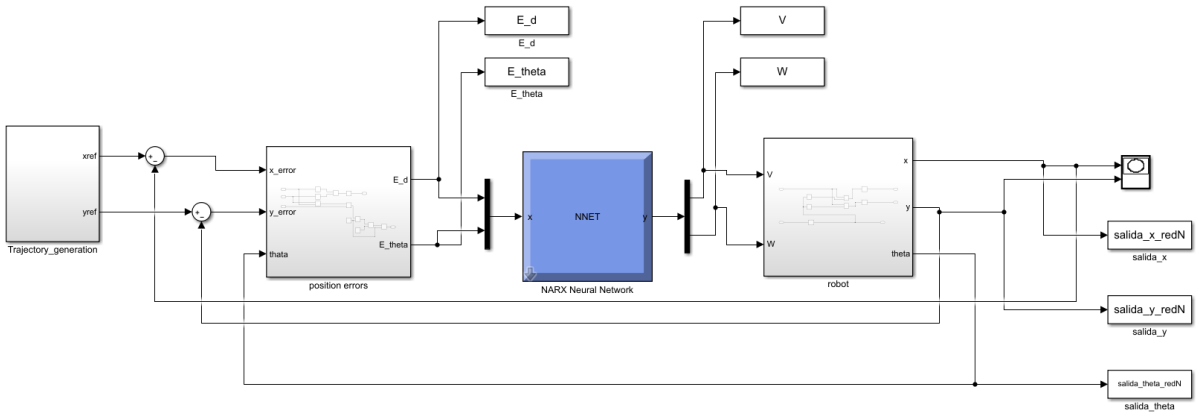
Después de entrenar la red para probarla y ejecutarla con otros datos es necesario cerrar el lazo.

```
net = closeloop(net);
```

Tras experimentar con un rango de valores entre 5 y 20 para el número de neuronas de la capa oculta, hemos obtenido que el mejor comportamiento de la red ha sido con 11 neuronas, con un performance de 0,431 como se puede apreciar en la siguiente imagen.

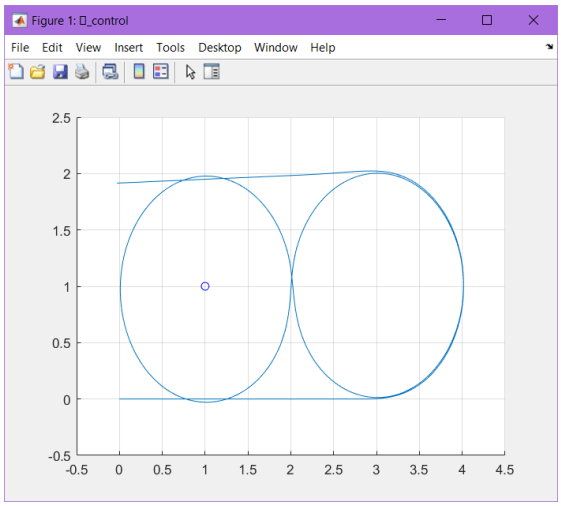


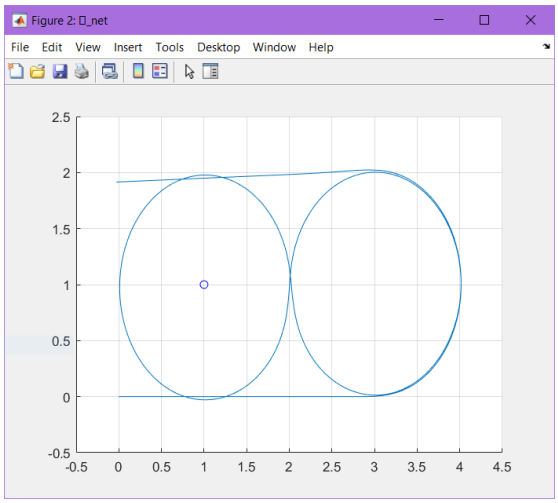
Tras ello, hemos ejecutado el comando gensim para obtener un bloque con la red neuronal para poder utilizarlo de aquí en adelante en las simulaciones con Simulink; después de obtenerlo, lo sustituimos en el sistema que teníamos y que hacía uso de un controlador. El resultado es el siguiente, guardado en *'SeguimientoTrayectoriasRed.slx'*:

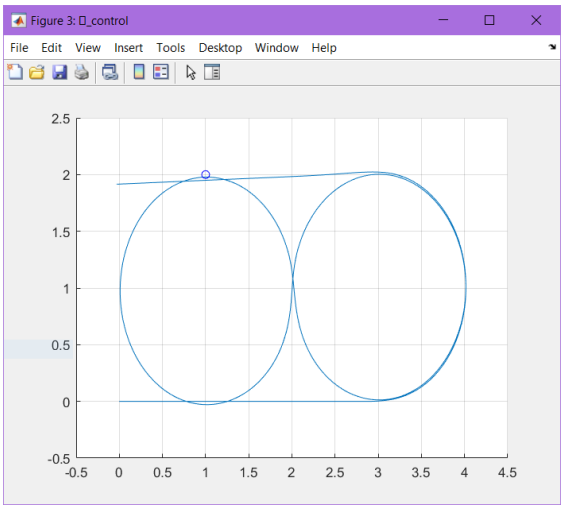


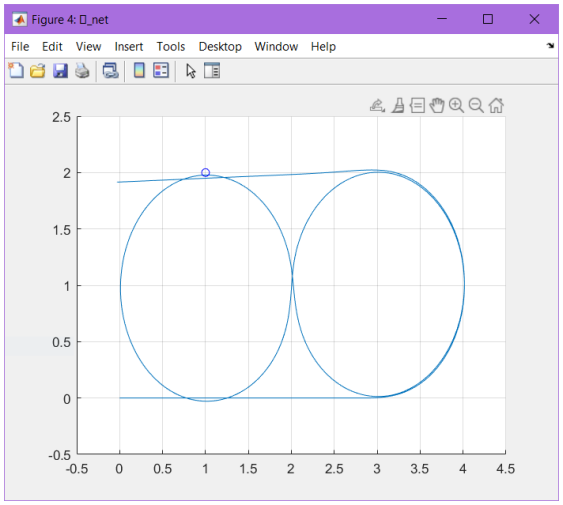
Después de crear la nueva red, la probamos y simulamos su comportamiento con diferentes puntos de entrada; para ello hemos realizado un nuevo script en MATLAB que nos permita variar los valores de **x_0** e **y_0** hasta N = 2 para ver distintas combinaciones.

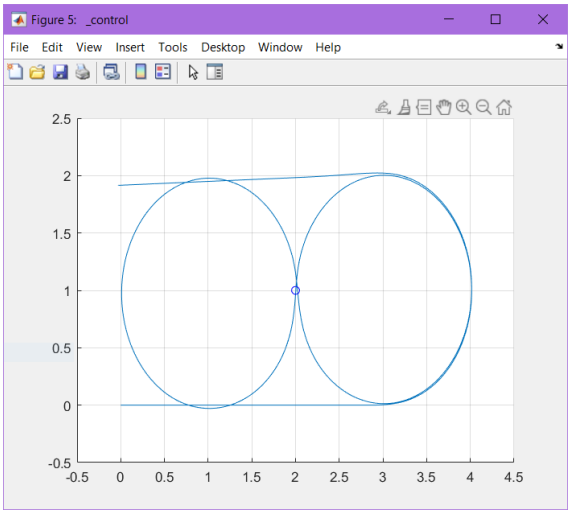
Este es el resultado obtenido:

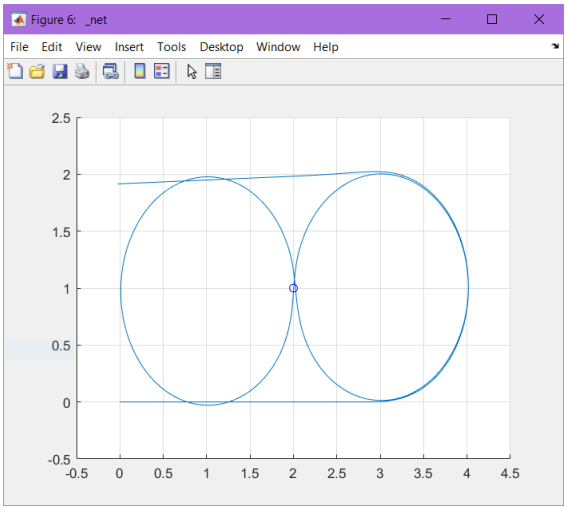


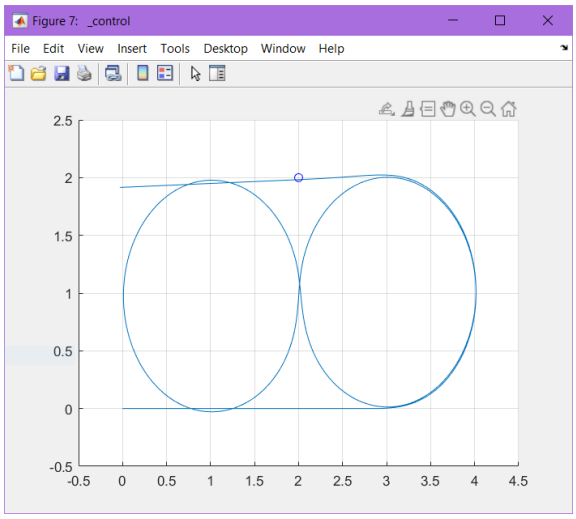


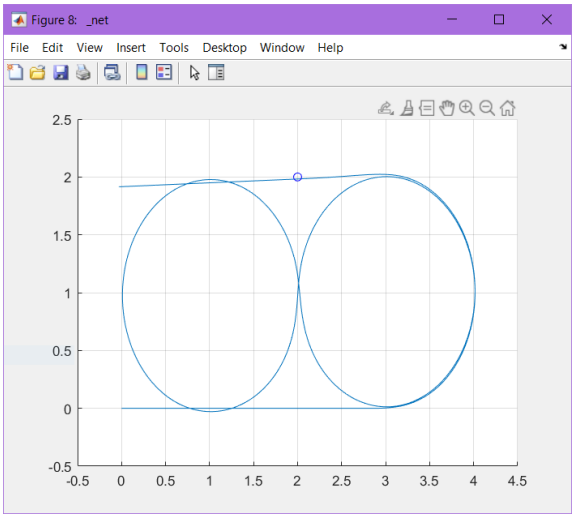












Por lo que, como se puede ver, el comportamiento de la red neuronal es similar al del controlador; podemos concluir que la red NARX creada funciona de forma correcta.