Práctica 1: Control Neuronal

Sistemas de Control Inteligente

Daniel Lopez Moreno
Luis Alejandro Cabanillas Prudencio

PARTE 1

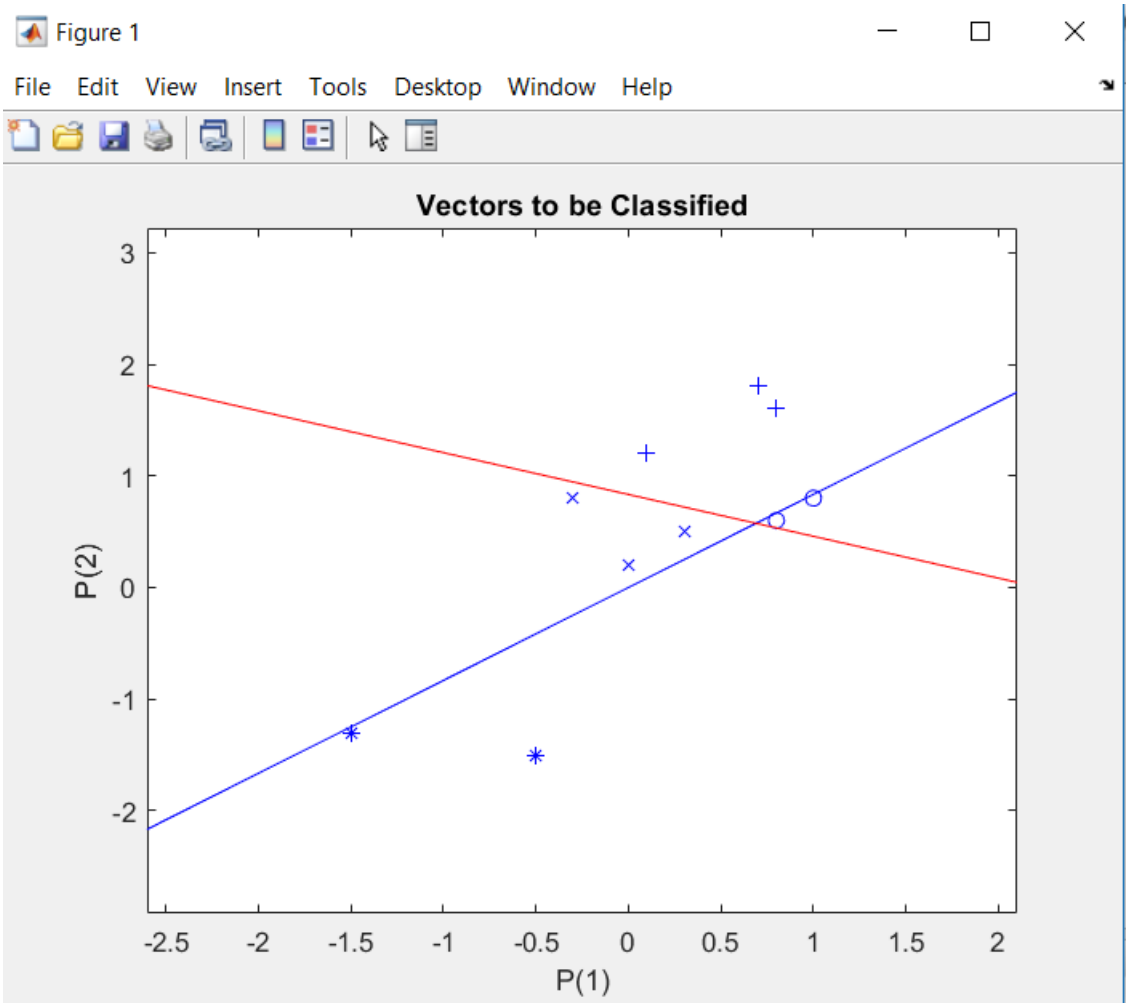
EJERCICIO 1

Generamos un Perceptron y le pasamos las distintas P de entrada y los targets T de salida para posteriormente entrenarlo y mostrar los resultados de dicho entrenamiento.

```
P=[0.1 0.7 0.8 0.8 1.0 0.3 0.0 -0.3 -0.5 -1.5;
1.2 1.8 1.6 0.6 0.8 0.5 0.2 0.8 -1.5 -1.3];
T=[1 1 1 0 0 1 1 1 0 0;
0 0 0 0 0 1 1 1 1 1];
net = perceptron;
net = train(net,P,T);
plotpv(P,T);
plotpc(net.iw{1,1},net.b{1});
```

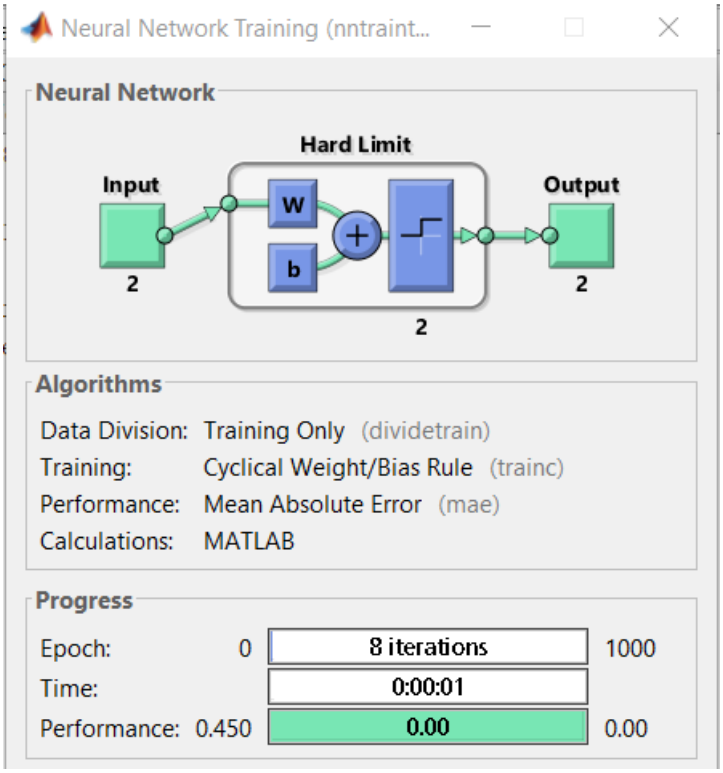
¿Consigue la red separar los datos?

Si, como se muestra en la siguiente gráfica, la red ha sido capaz de separar los datos utilizando dos líneas rectas, organizando las 4 clases (*, +, o y x).



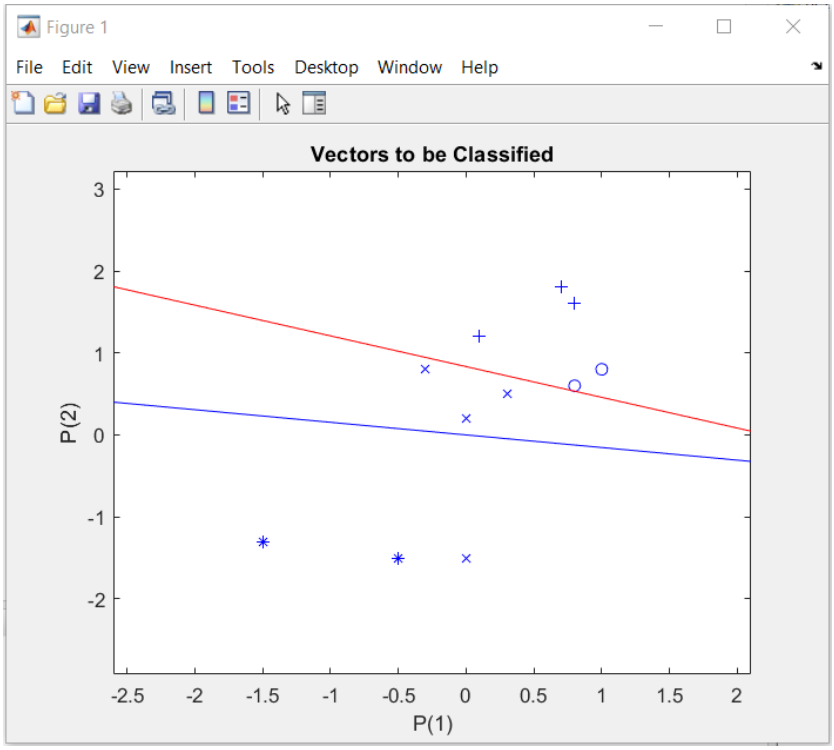
¿Cuántas neuronas tiene la capa de salida? ¿por qué?

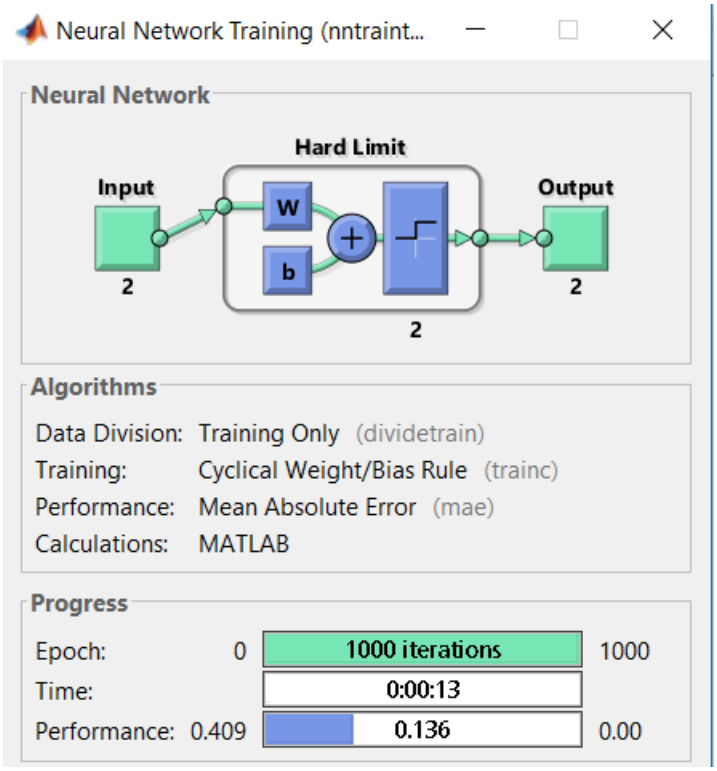
La capa de salida, como se puede observar en el siguiente esquema, tiene 2 neuronas, que a través de 0 y 1 puede diferenciar las 4 clases (00, 01, 10, 11).



¿Qué ocurre si se incorpora al conjunto un nuevo dato: [0,0 -1.5] de la clase 3?

Al introducir el nuevo dato, el conjunto de datos deja de ser linealmente separable, por lo que la red neuronal llega al límite preestablecido de iteraciones (1000) y muestra una gráfica en la que se puede observar que los datos no están separados de forma correcta.





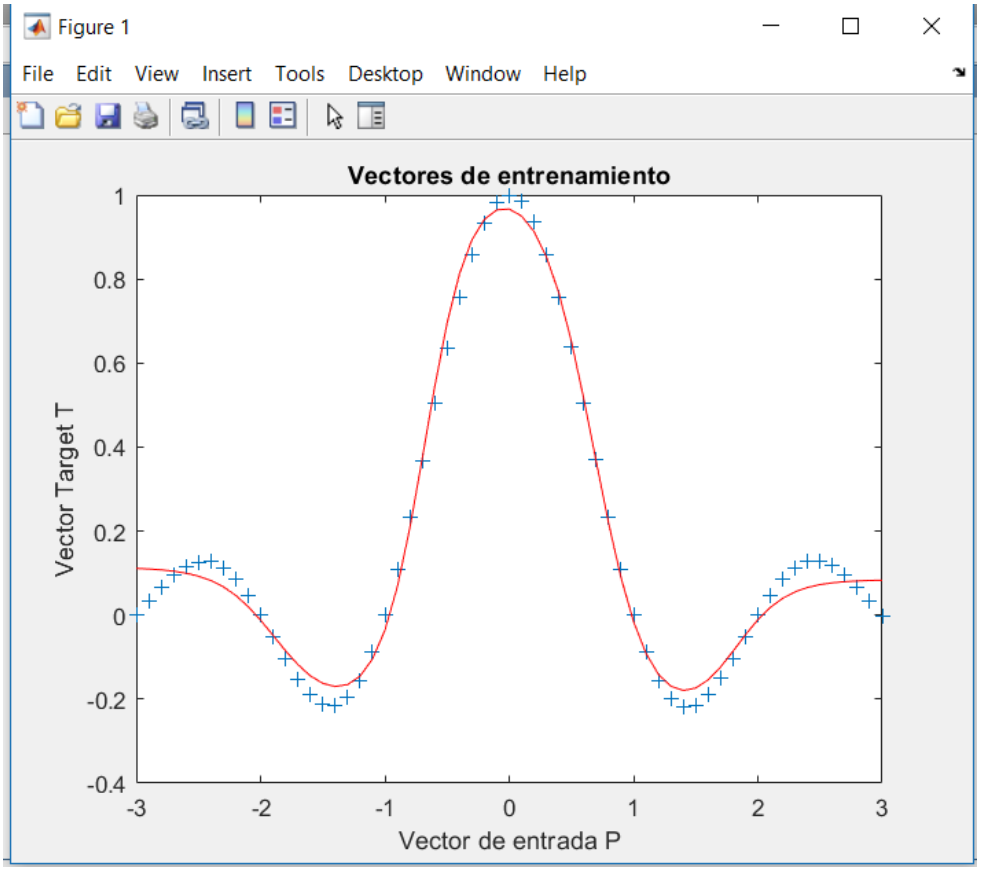
EJERCICIO 2

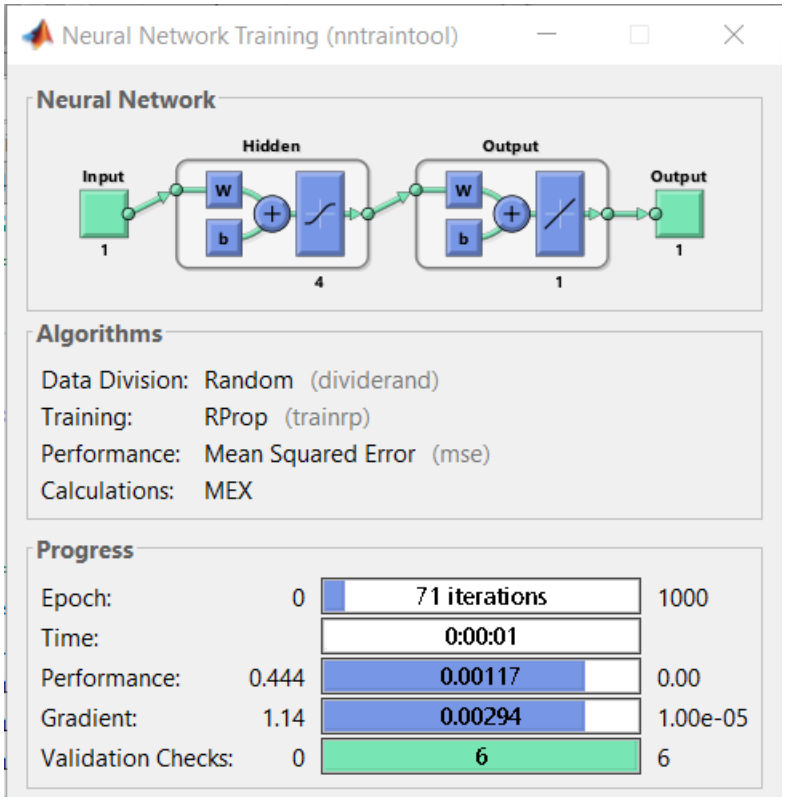
Estudie los efectos sobre la solución final de modificar el método de entrenamiento (consulte la ayuda de Matlab y pruebe 4 métodos diferentes) y el número de neuronas de la capa oculta.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% 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');
```

2.1. Entrenamiento Trainrp:

TRAINRP es un método de entrenamiento que se basa en el algoritmo de resilient backpropagation.





2.2. Entrenamiento Trainlm:

TRAINLM es un método de entrenamiento que se basa en el algoritmo de Levenberg-Marquardt backpropagation.

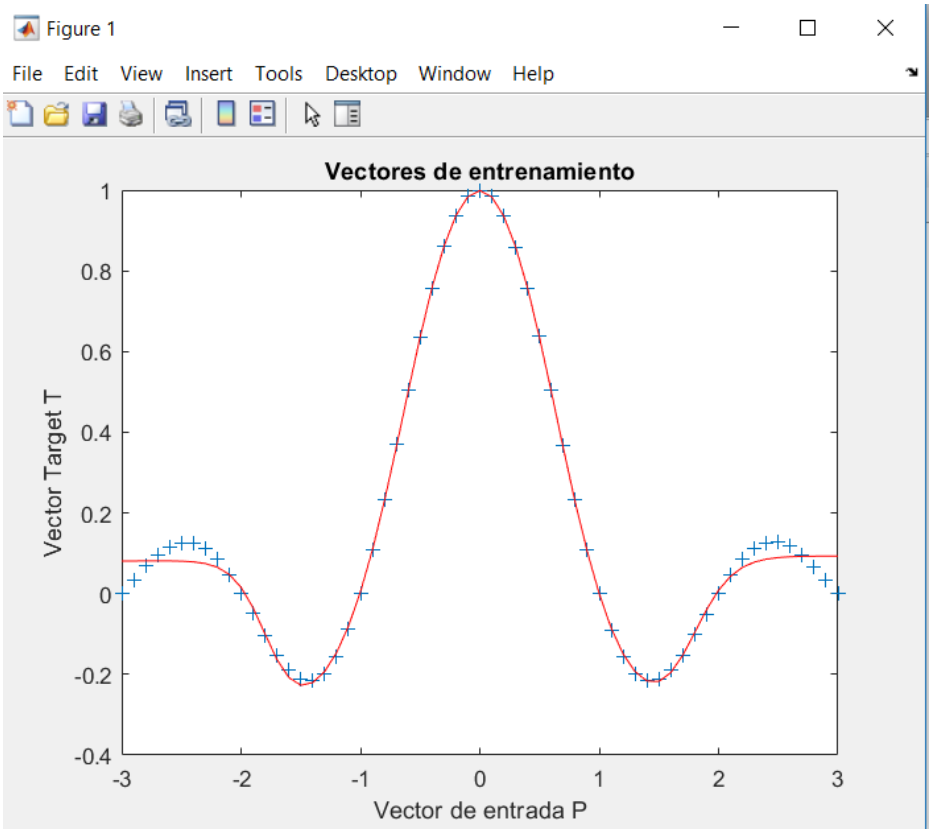
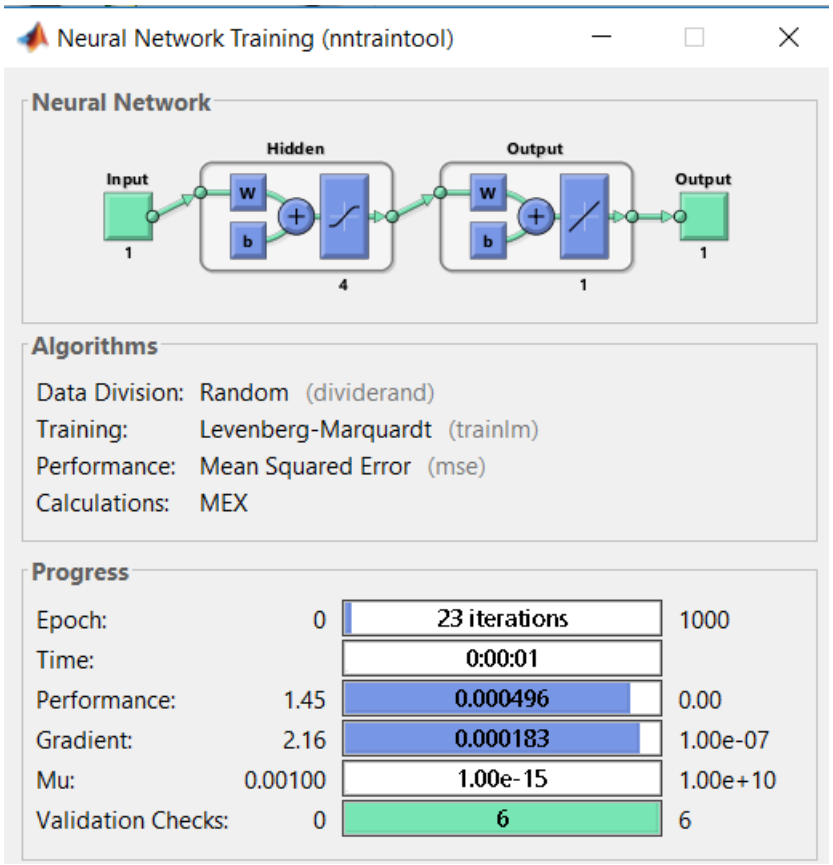
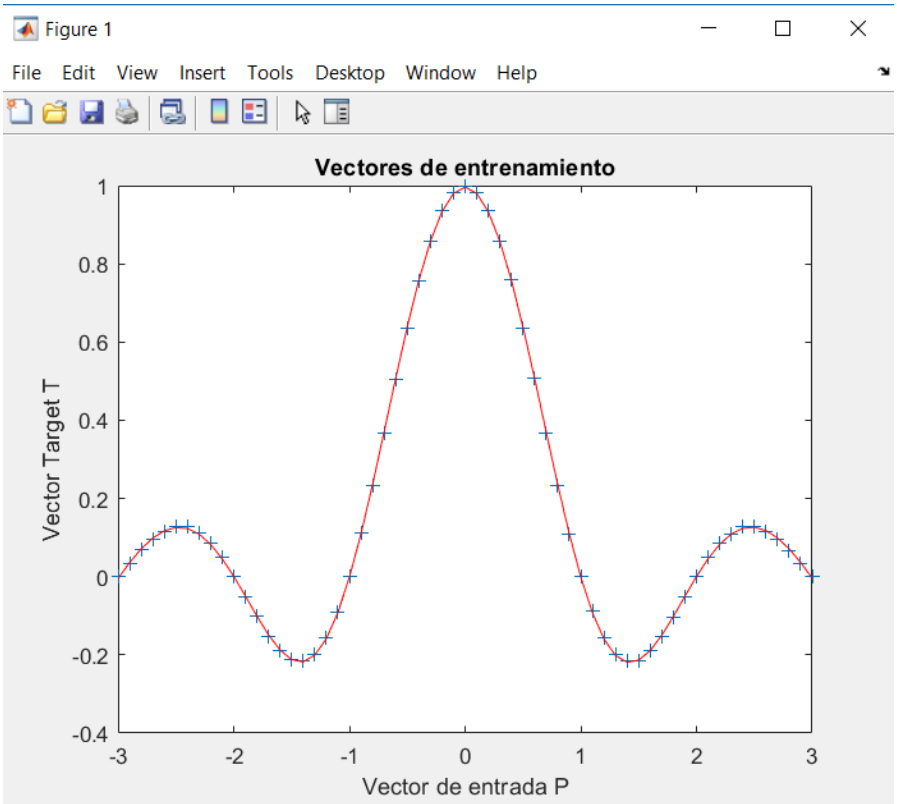


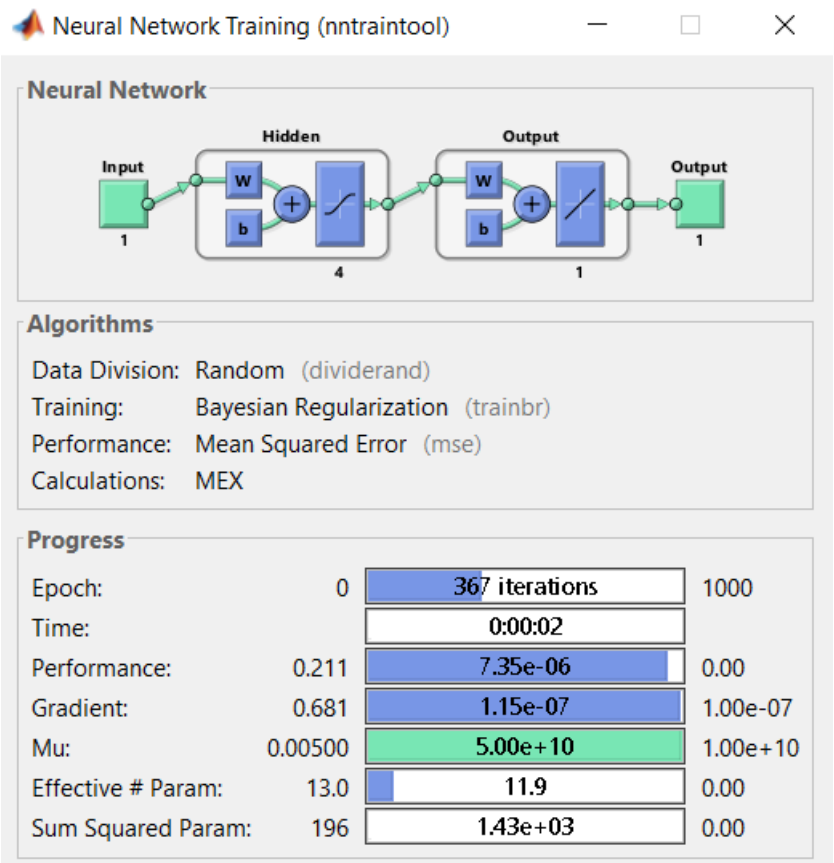
Figure 1
File Edit View Insert Tools Desktop Window Help
Vectores de entrenamiento
Vector Target T
Vector de entrada P
1
0.8
0.6
0.4
0.2
0
-0.2
-0.4
-3
-2
-1
0
1
2
3


Neural Network Training (nntraintool)
Neural Network
Input
Hidden
Output
Output
W
b
1
4
1
1
Algorithms
Data Division: Random (dividerand)
Training: Levenberg-Marquardt (trainlm)
Performance: Mean Squared Error (mse)
Calculations: MEX
Progress
Epoch: 0 23 iterations 1000
Time: 0:00:01
Performance: 1.45 0.000496 0.00
Gradient: 2.16 0.000183 1.00e-07
Mu: 0.00100 1.00e-15 1.00e+10
Validation Checks: 0 6 6

2.3. Entrenamiento Trainbr:

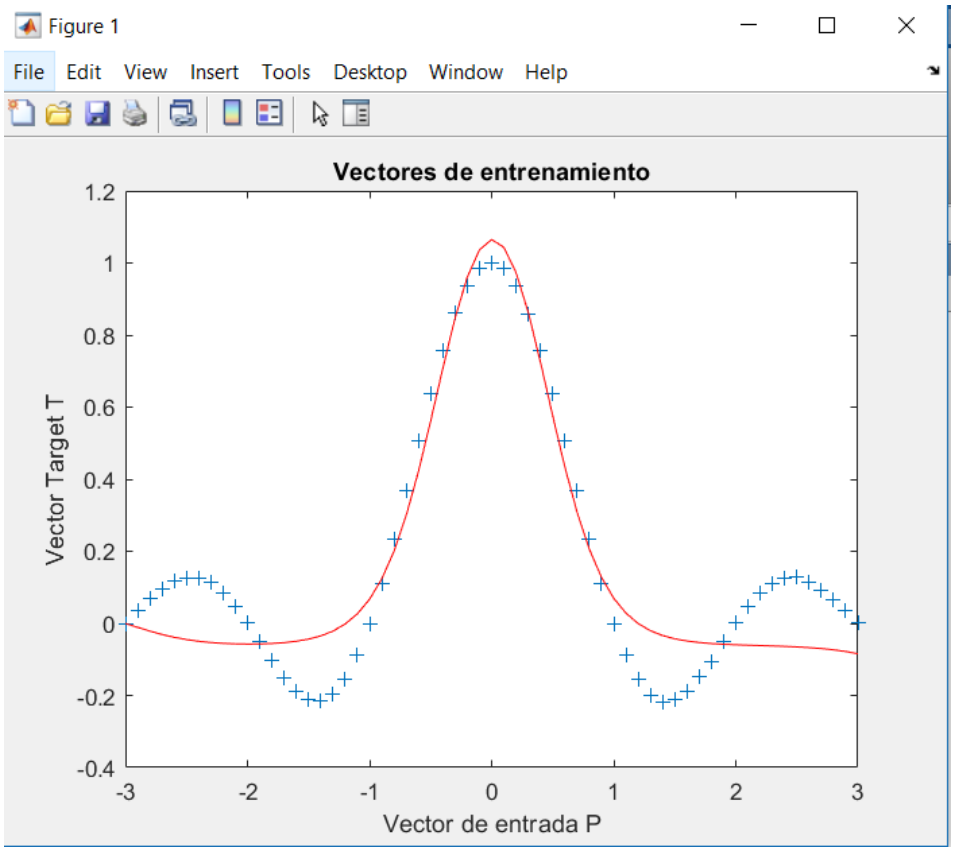
TRAINBR es un método de entrenamiento que se basa en el algoritmo de Bayesian Regularization.

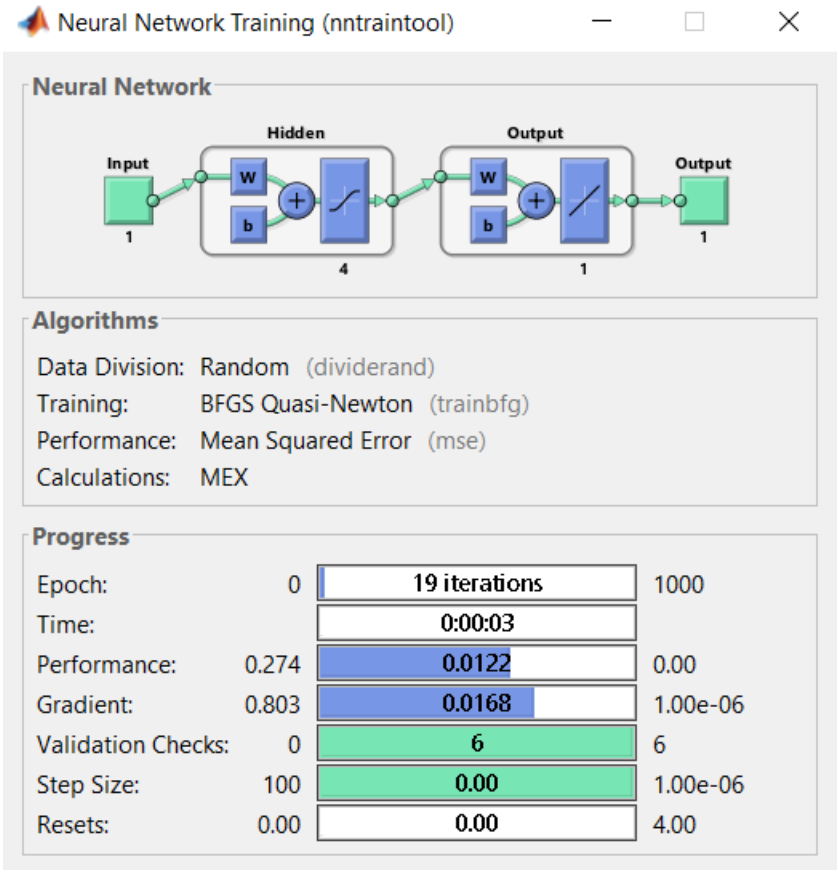




2.4. Entrenamiento Trainbfg:

TRAINBFG es un método de entrenamiento basado en el método BFGS quasi-Newton.

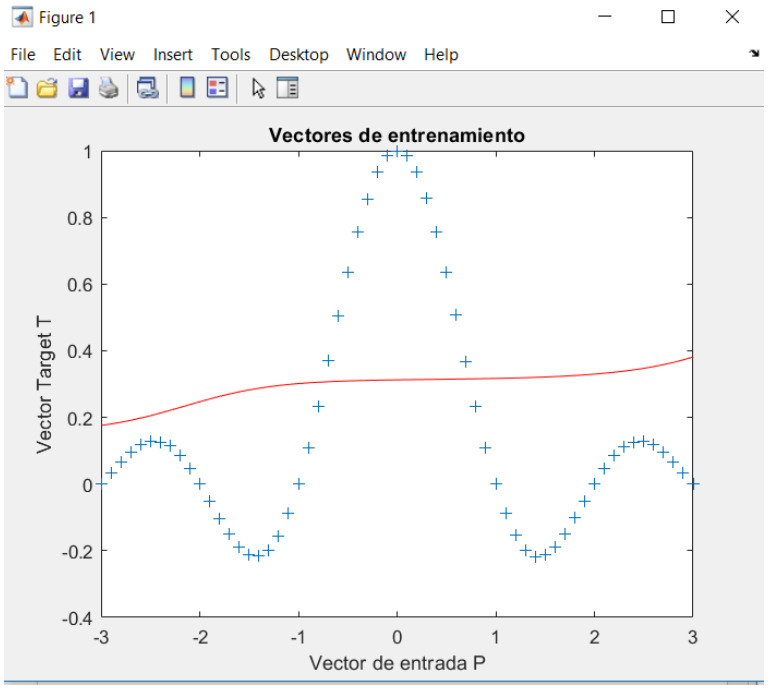


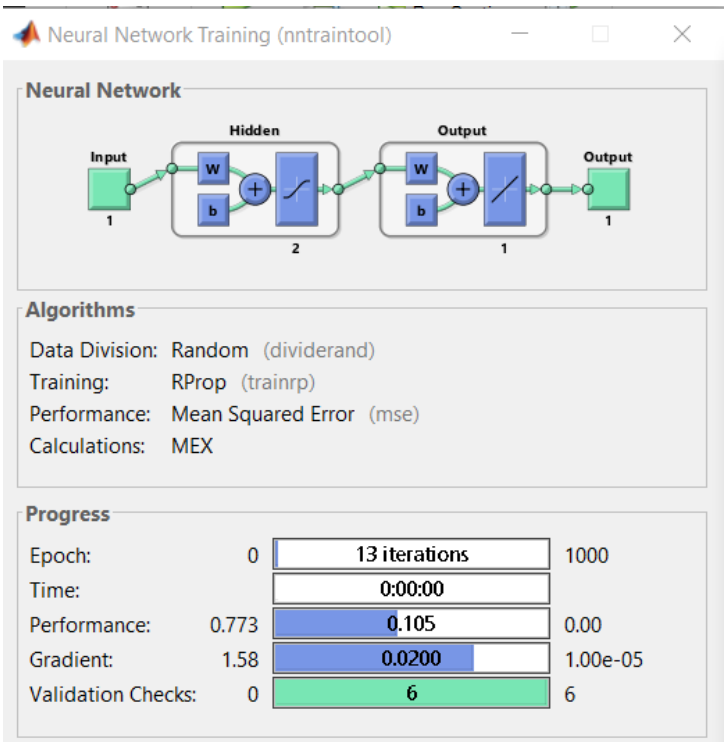


Tras utilizar los 4 distintos métodos de entrenamiento (RP, LM, BR, BFG), podemos comprobar que todos los métodos de entrenamiento finalizan de manera satisfactoria sin alcanzar el límite de 1000 iteraciones. Aunque si se puede comprobar que algunos necesitan menos iteraciones (LM:23 iteraciones y BFG: 19 iteraciones) y otros más (BFG: 367 iteraciones) en este caso en concreto.

Utilización de 2 neuronas en la capa oculta:

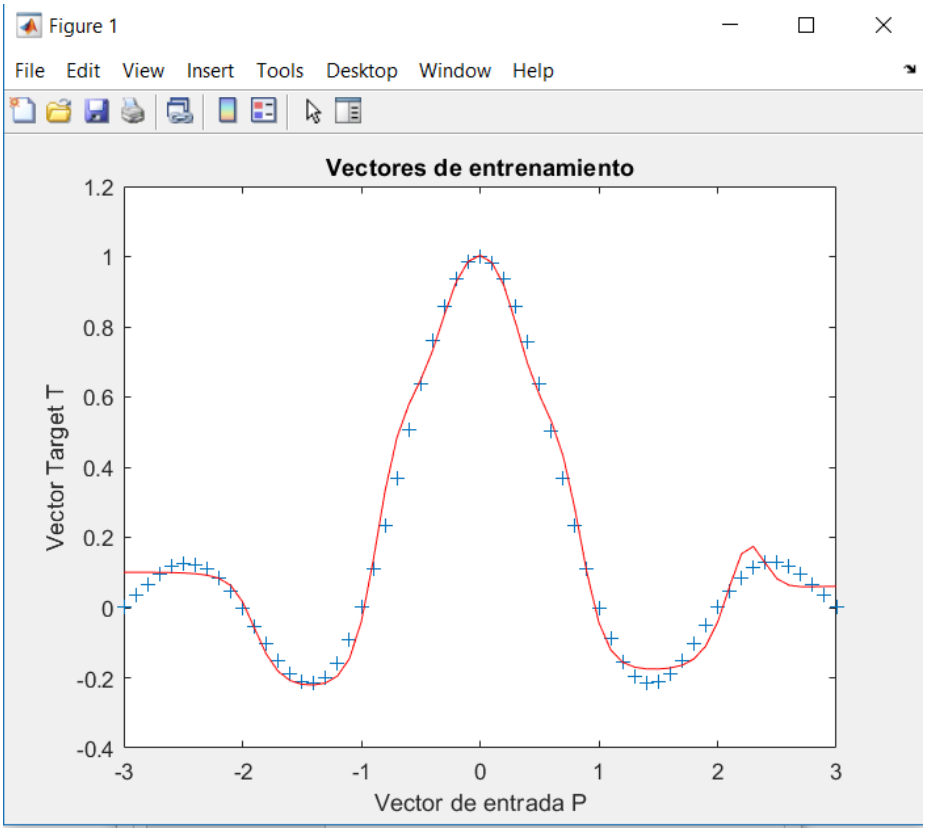
Esta vez hemos realizado el entrenamiento con el algoritmo de resilient backpropagation (TRAINRP) de nuevo, pero hemos cambiado el número de neuronas de la capa oculta de 4 a 2. Como se puede comprobar, el entrenamiento ha terminado en menos de 1000 iteraciones, pero los resultados no son satisfactorios, ya que las salidas ni siquiera se aproximan a los targets esperados, por lo que el entrenamiento ha fallado.

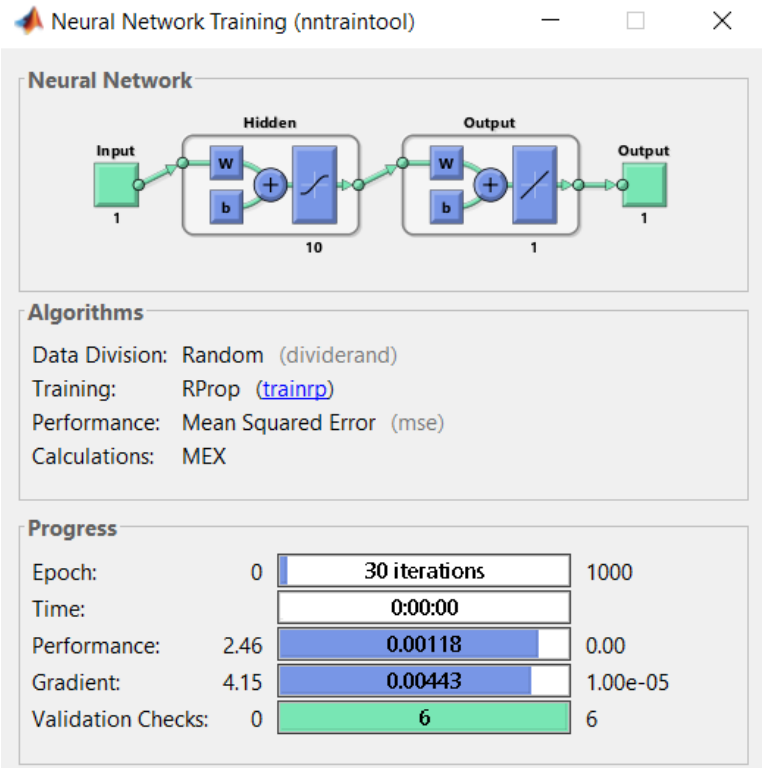




Utilización de 10 neuronas en la capa oculta:

En este caso hemos aumentado el número de neuronas de la capa oculta de 2 a 10, utilizando el mismo método de entrenamiento (TRAINRP). Ahora podemos comprobar como el entrenamiento ha finalizado en más iteraciones que con 2 neuronas, pero los resultados de salida se ajustan de mejor manera a los targets, reduciendo el error ampliamente en comparación.





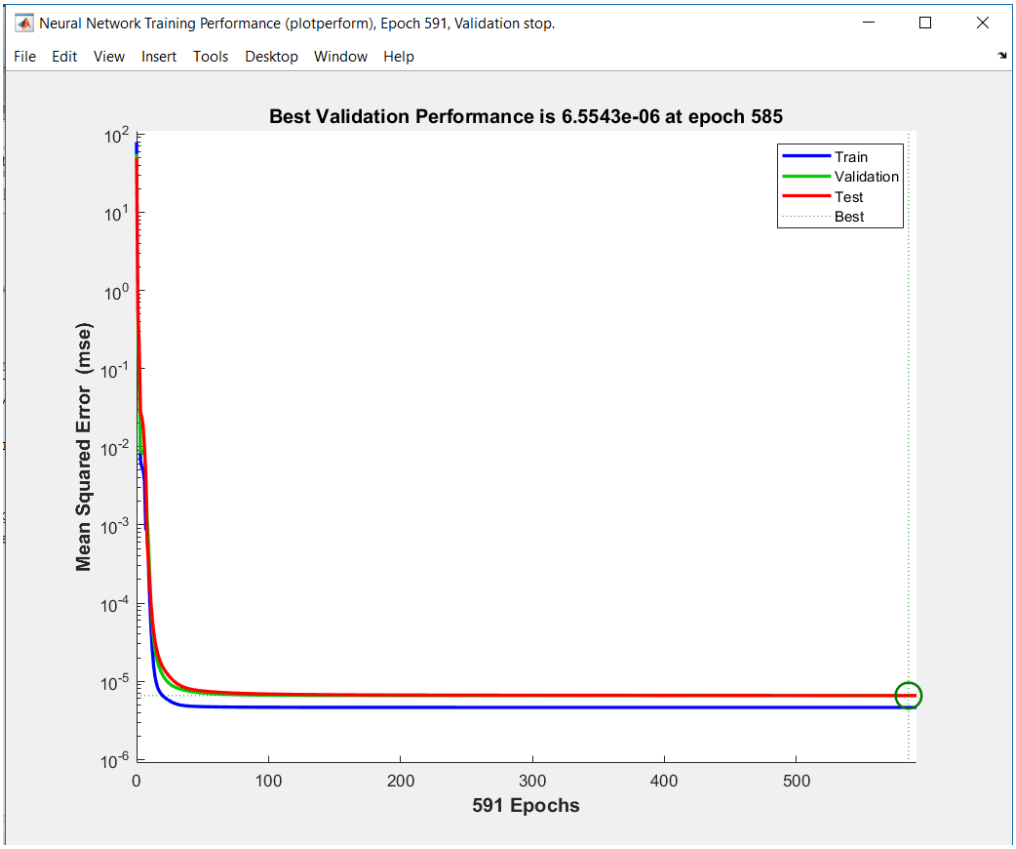
EJERCICIO 3

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)
```

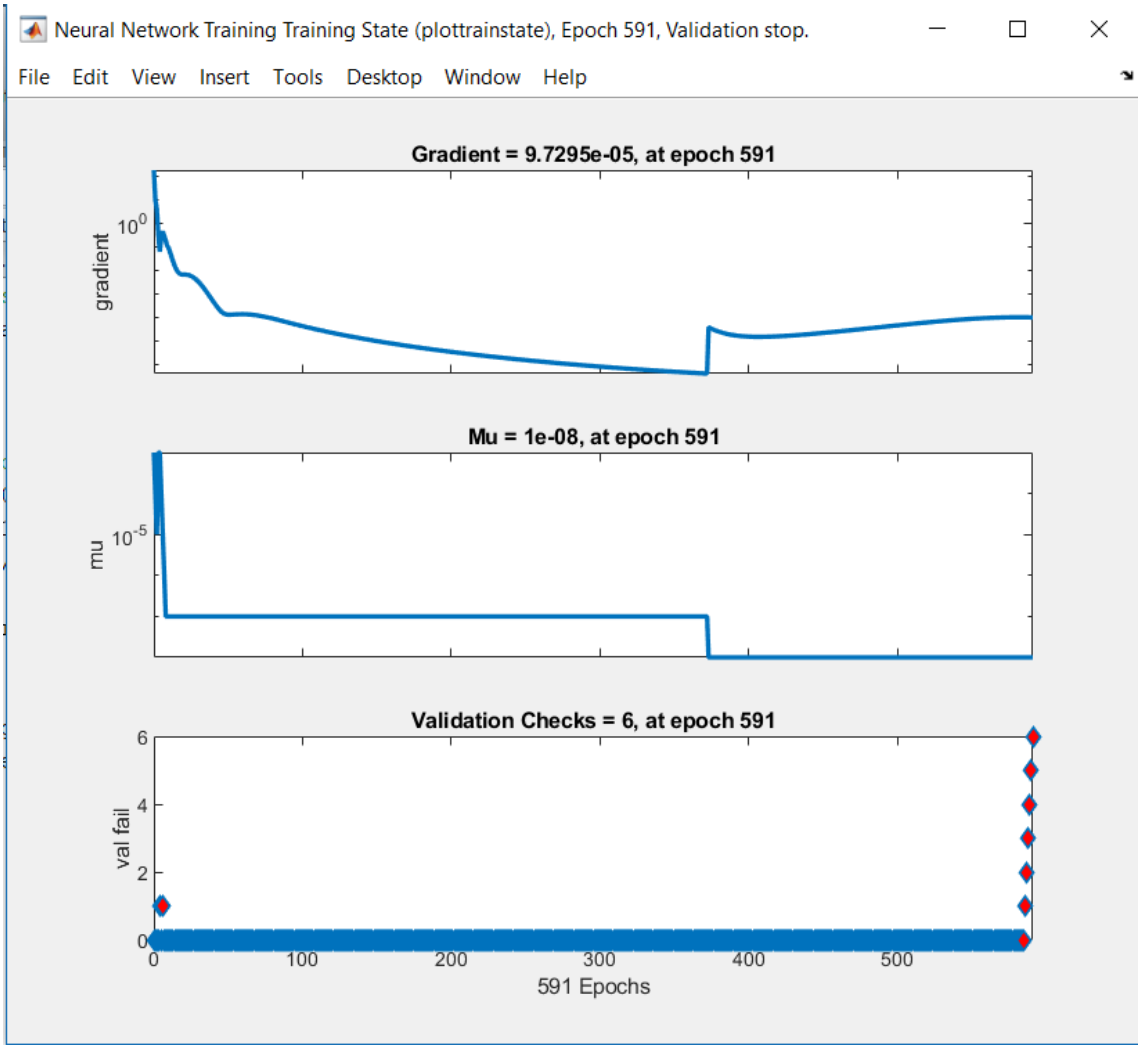
Explore las gráficas disponibles:

plotperform:



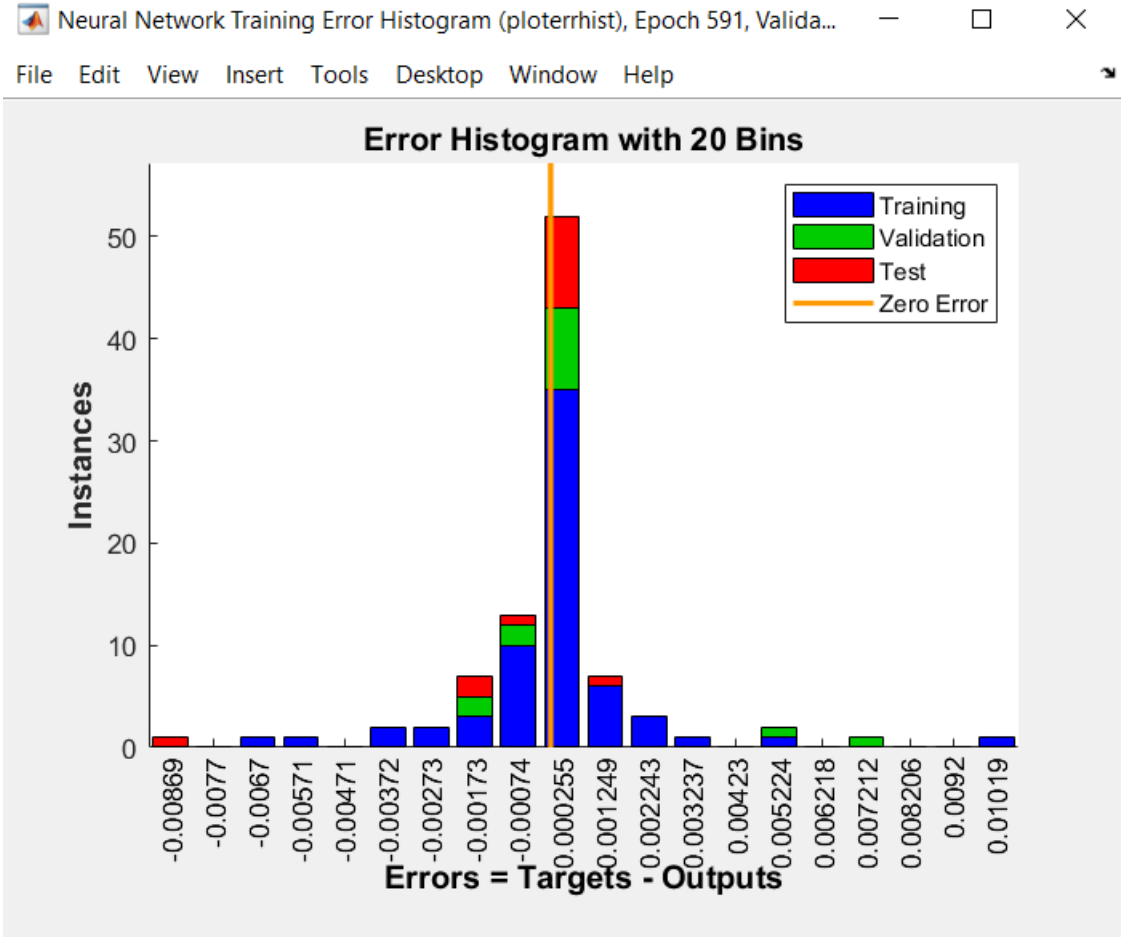
En esta gráfica del "performance" de nuestro entrenamiento podemos comprobar el progreso del error en las distintas distribuciones de datos (entrenamiento, validación y test) con el avance de las épocas. Como se puede observar, el error se reduce en cada época aproximándose cada vez más a 0, indicando que el entrenamiento está siendo efectivo.

plottrainstate:



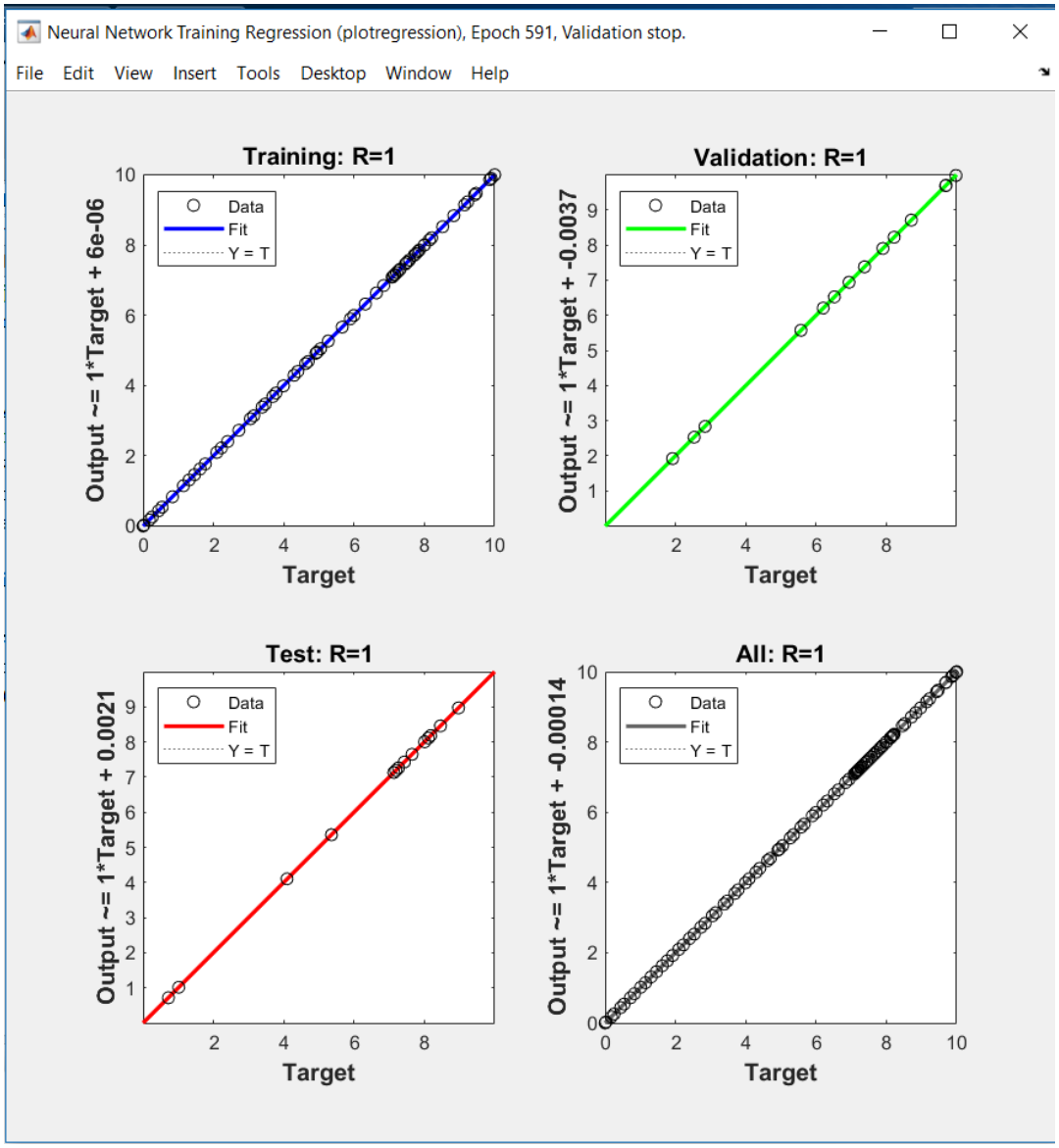
En estas gráficas podemos comprobar como se ha desarrollado el entrenamiento. Podemos comprobar como el error disminuye a lo largo de las épocas, y el número de fallos de validación se mantiene en 0. Sin embargo, cuando llegamos al final de la gráfica de validación podemos observar como el número de fallos de validación aumenta, esto indica que podemos estar sobreentrenando nuestra red, por lo que el entrenamiento termina de manera satisfactoria.

ploterrhist:

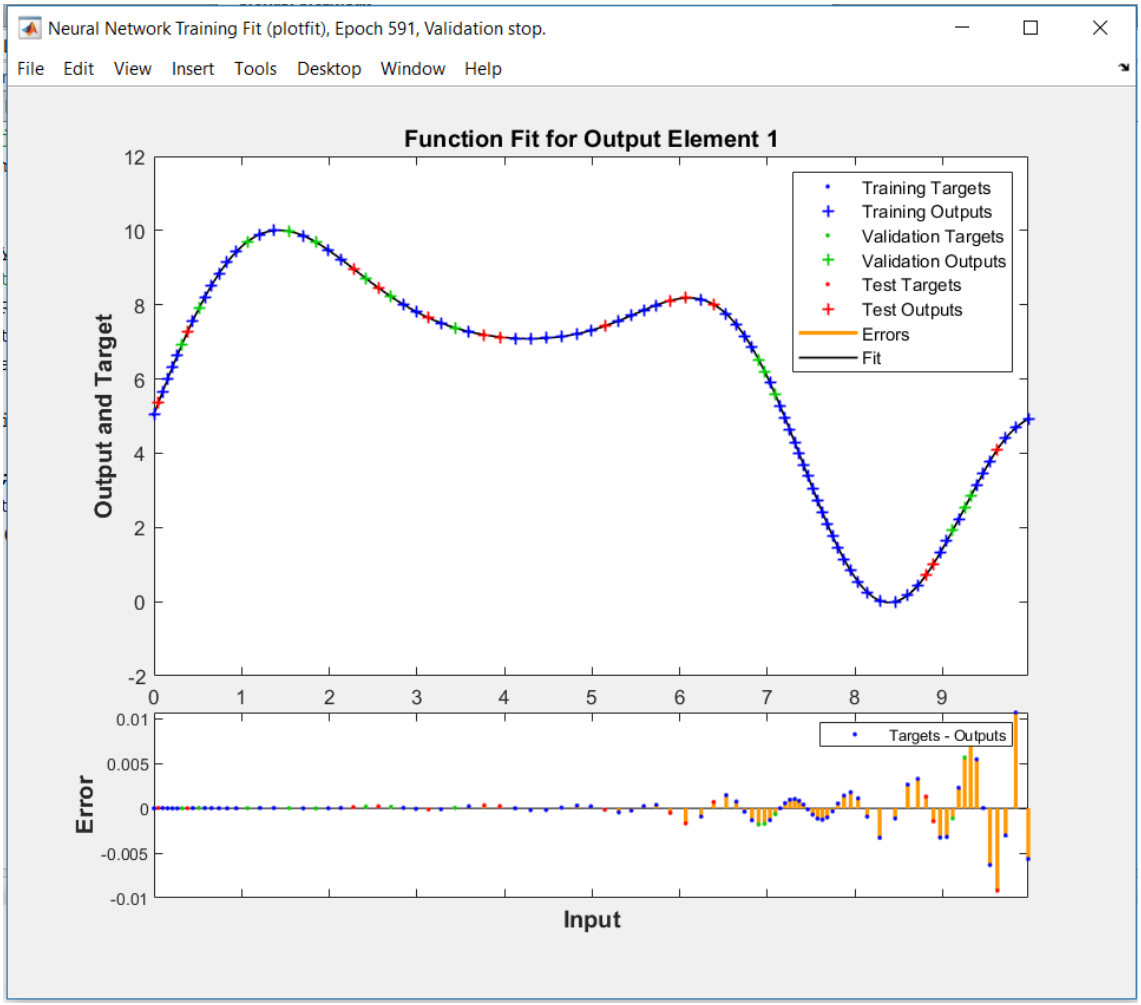


Ploterrhist nos muestra un histograma de los valores del error en las diferentes etapas (Entrenamiento, Validación y Test). Como se puede observar en la línea color Naranja es el momento en el que el error alcanza el valor más bajo y el momento en el que mayor número de instancias existen en las 3 fases, destacando las instancias de Test por encima de Validación y entrenamiento.

plotregression y plotfit:



En la gráfica de plotregression podemos ver la distribución de los datos para comprobar como se ajustan al objetivo, por eso en el eje de las Y tenemos la salida y en el eje de las X tenemos el objetivo, para comprobar cuanto se distancian el uno del otro en cada una de las fases. En nuestro caso la distribución de datos entre las fases era 70 Entrenamiento, 15 Validación y 15 Test, cada circulo en cada gráfico representa cada uno de esos datos.

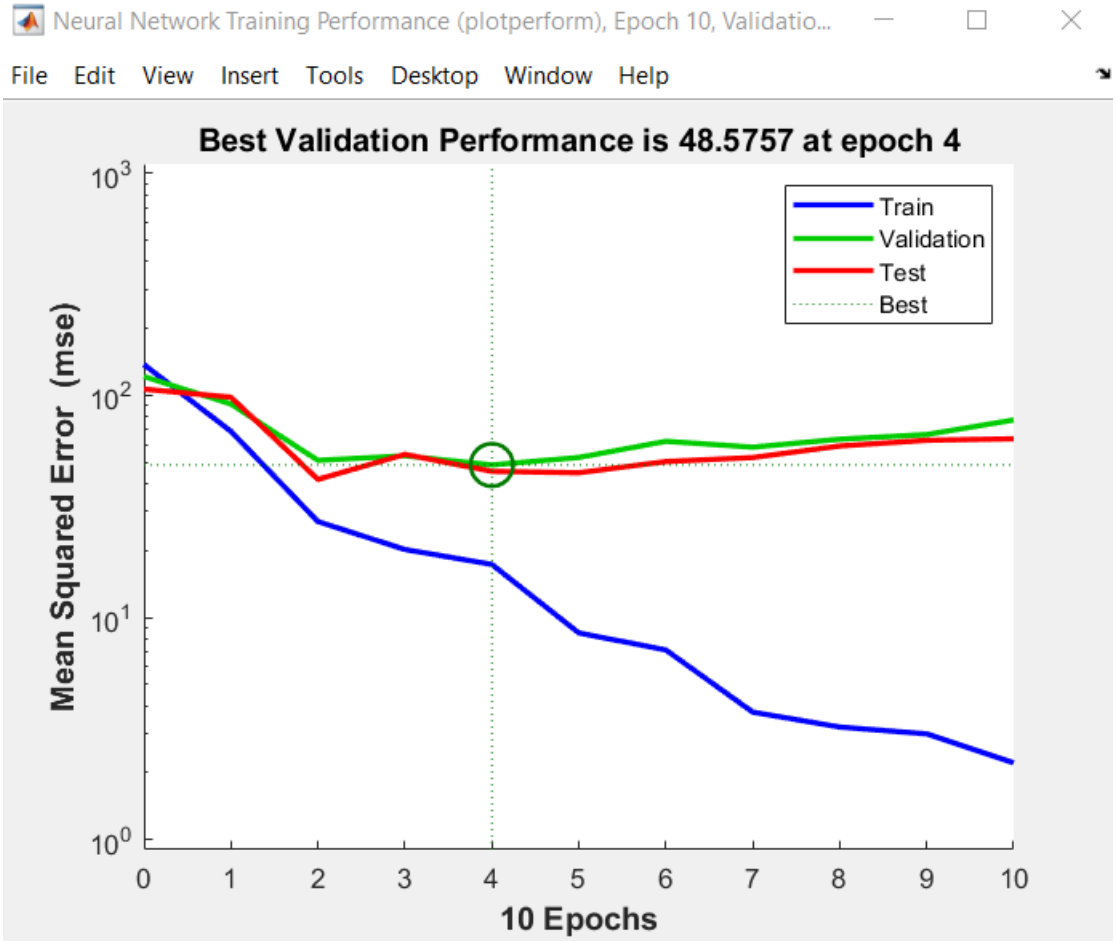


En esta gráfica plotfit podemos comprobar como se ajustan las salidas y los objetivos en función de cada entrada. La diferencia entre Output y Target en la gráfica superior es casi inapreciable debido a que el error tiene valores cercanos a 0, aunque a medida que el valor de la entrada aumenta, el error es levemente más notable.

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).

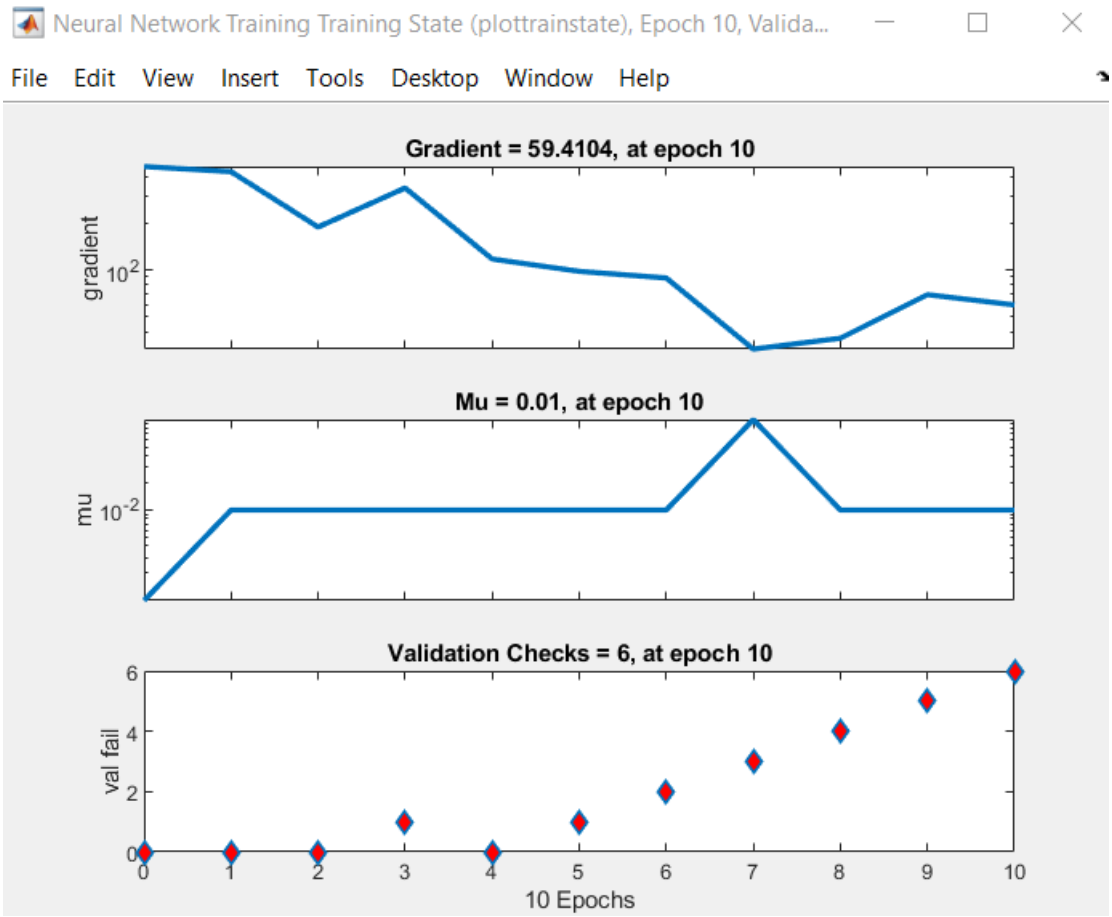
Para esta segunda parte del ejercicio procederemos a utilizar el conjunto de datos bodyfat_dataset y a cambiar la división de los datos. Anteriormente la división era 70/15/15, ahora vamos a comprobar que sucede con una división 40/30/30. También cambiaremos el método de entrenamiento a TrainLM.

plotperform:



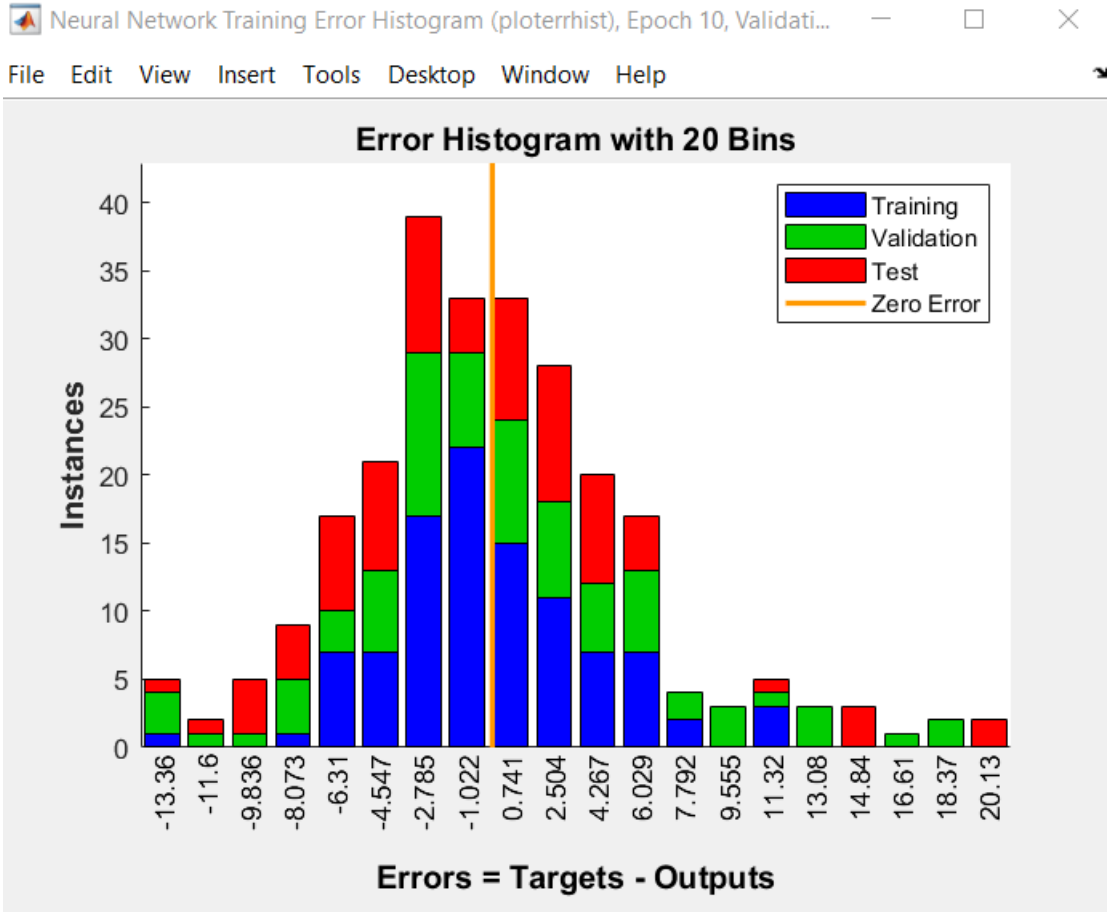
En este caso comprobamos como el error medio cuadrático no disminuye tan rápido como antes en la fase de entrenamiento, tampoco lo hace en las fases de validación y test, donde disminuye levemente. Tambíen, comprobamos que a partir de la época 4 de las 10 que ha necesitado el proceso el valor del error en la validación comienza a aumentar, por lo que estamos sobreentrenando a partir de esa época.

plottrainstate:



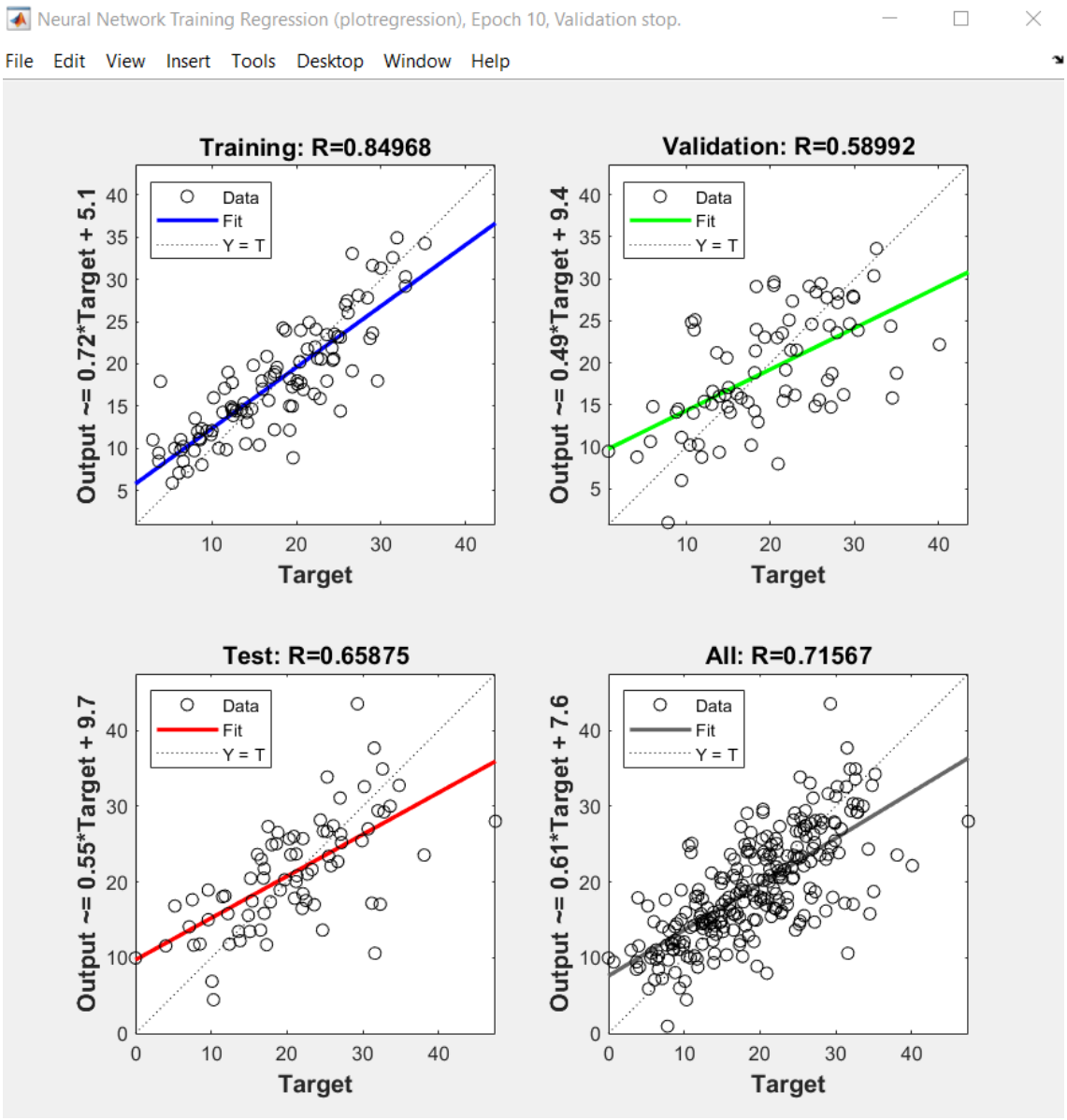
Como podemos comprobar en la gráfica de plottrainstate, a partir de la época 4 los fallos en la validación comienzan a aumentar, esto se debe a que como hemos determinado en la gráfica anterior, el mínimo global se encuentra en esa época.

plotrrhist:

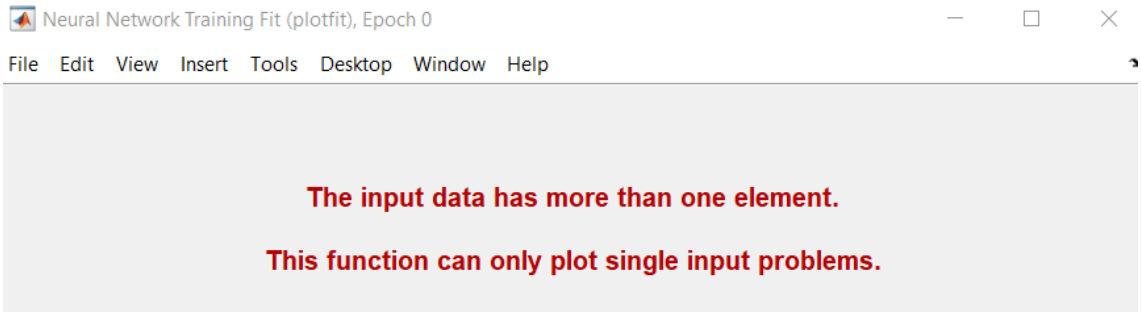


En este caso podemos comprobar que los errores no son tan cercanos a 0 como en el caso anterior, esto se debe a que hemos reducido el número de datos que se utilizan para el entrenamiento, por lo que la red no ha sido tan eficaz a la hora de reducir los errores y predecir. Además, han sido necesarias muchas más instancias por cada valor de error en comparación a el apartado anterior.

plotregression y plotfit:



Ahora podemos comprobar como los datos de salida no se ajustan tanto a los objetivos ya que como hemos dicho, hemos reducido el número de datos utilizados para entrenar, por lo que la red neuronal no ha sido capaz completamente de ajustarse correctamente. Se puede apreciar claramente una mayor dispersión de los círculos en torno a la línea de ajuste.



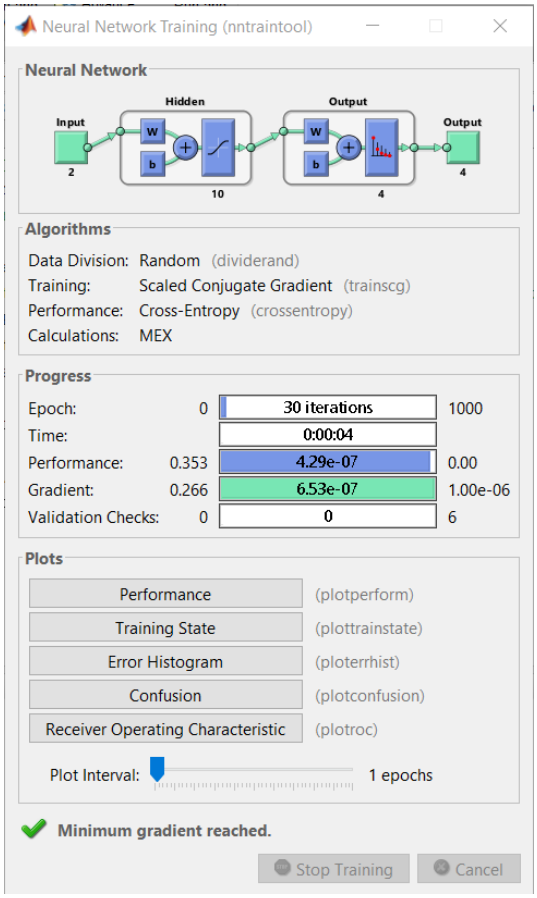
En cuanto a la gráfica de plotfit no es posible mostrarla debido a que los datos de entrada tienen más de un elemento y plotfit solo funciona con problemas de entrada simple.

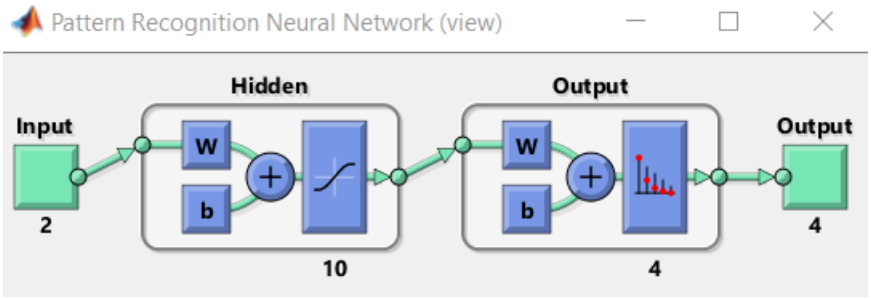
EJERCICIO 4

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)
```

Cuando ejecutamos el código anterior, obtenemos una ventana parecida a la que obteníamos con otras redes neuronales, pero con algunos cambios.





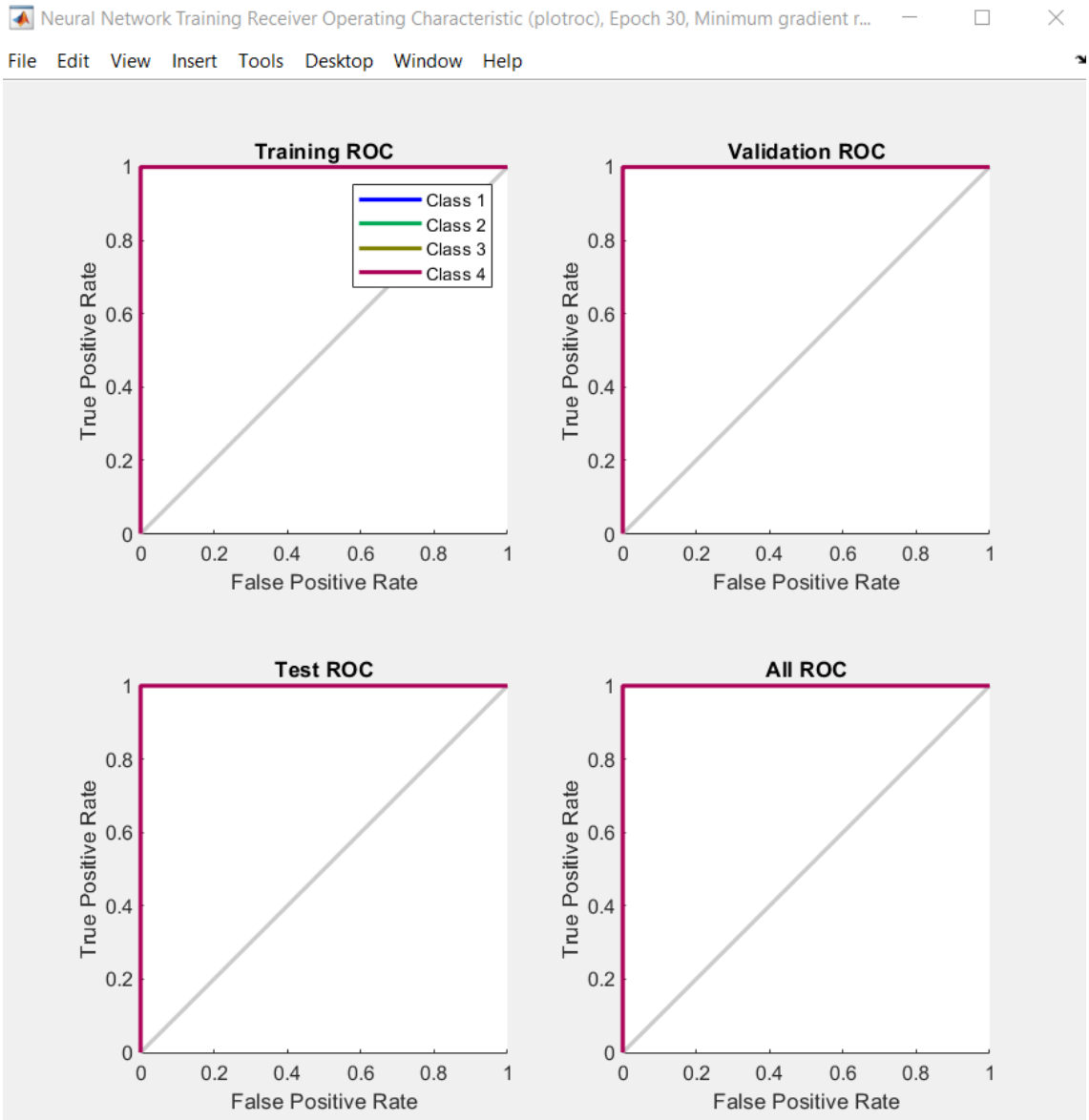
Podemos observar como la función de salida ha cambiado, ya no es lineal, el método de entrenamiento utilizado es Scaled Conjugate Gradient (trainscg), y algunas de las gráficas disponibles han cambiado.

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

Plotconfusion:

La gráfica de plotconfusion muestra las matrices de confusión de las distintas fases (Entrenamiento, Validación y Test) y posteriormente una gráfica en la que se suman todos los datos de confusión de las 3 fases. Como podemos ver, todas las clases han sido clasificadas correctamente, ya que las celdas diagonales indican aquellas clases clasificadas satisfactoriamente. Se puede observar que todas las demás celdas mantienen el valor 0 en las 4 matrices, por lo que no hay errores de clasificación.

Plotroc:

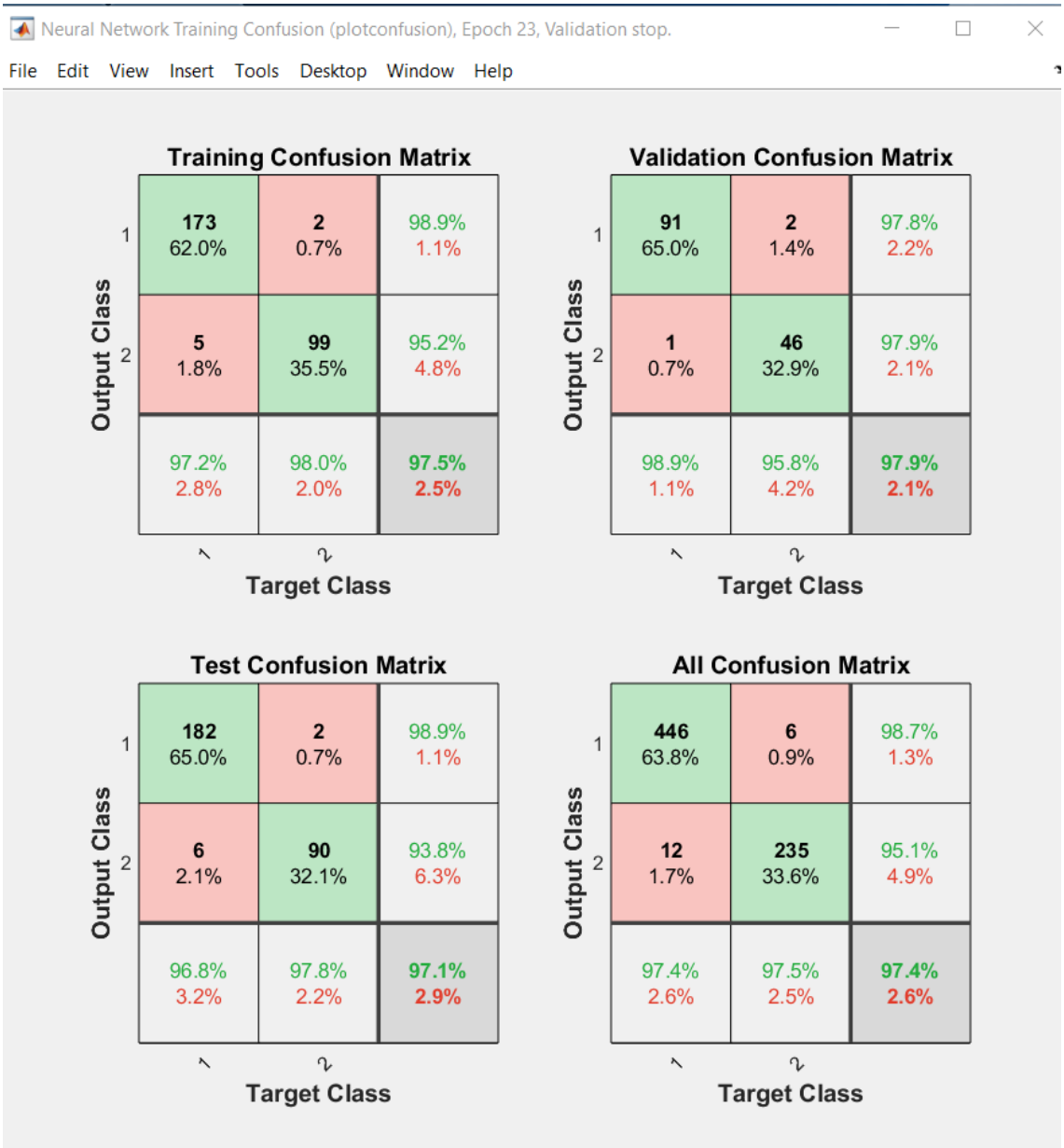


Plotroc muestra las curvas ROC (Receiver Operating Characteristic) de las 3 fases, así como una curva ROC conjunta de las 3 fases a la vez. Tras analizar las curvas de todas las fases podemos determinar que el valor del diagnóstico es perfecto ya que la curva siempre se encuentra en el valor 1 de True Positive Rate, por lo que todos los valores de comparación de patrones son Verdaderos Positivos.

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).

A continuación, vamos a realizar la ejecución utilizando el conjunto de datos cancer_dataset, además cambiaremos la división de datos a 40% Entrenamiento, 20% Validación y 40% Test, y utilizaremos "trainrp" como método de entrenamiento:

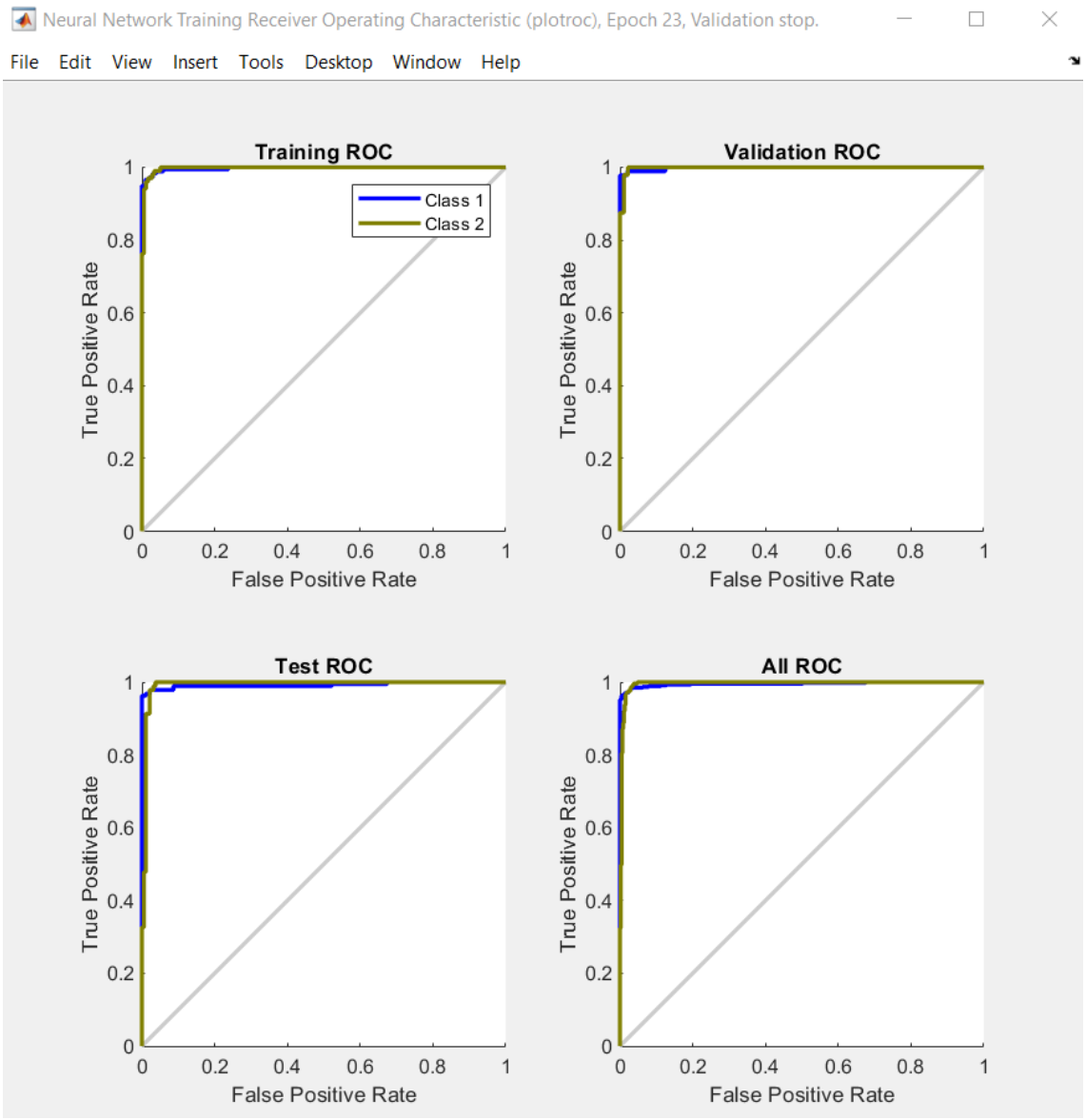
Plotconfusion:



Esta vez podemos comprobar que, si ha habido fallos de clasificación, la mayor parte de ellos en las fases de Entrenamiento y Test, aunque podemos comprobar que el porcentaje de fallo general no supera el 2.6%, por lo que el acierto de clasificación está en el 97.4%.

Por lo tanto, podemos comprobar que aunque los resultados son altamente satisfactorios, ya no encontramos una clasificación perfecta como en el ejemplo anterior con simpleclass_dataset.

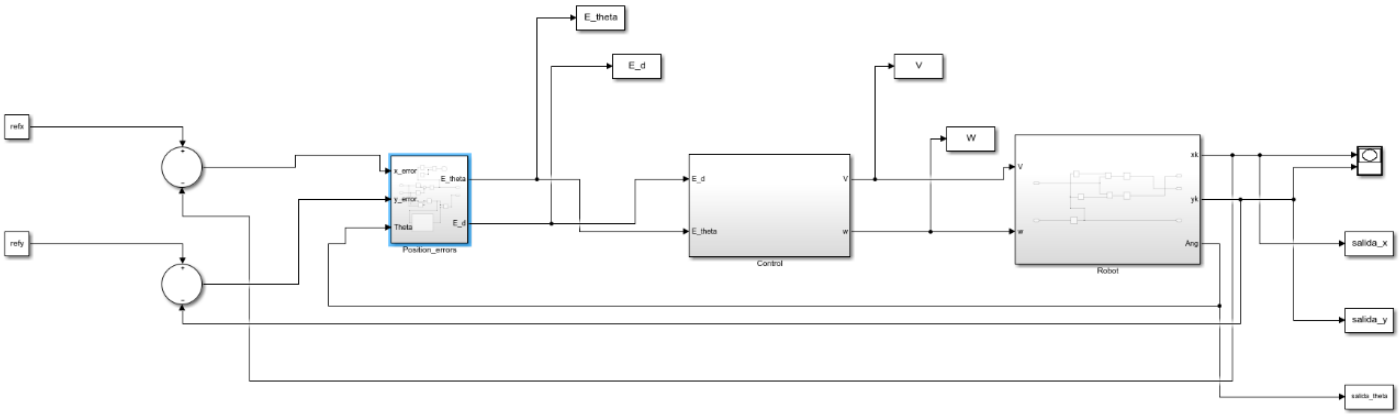
Plotroc:



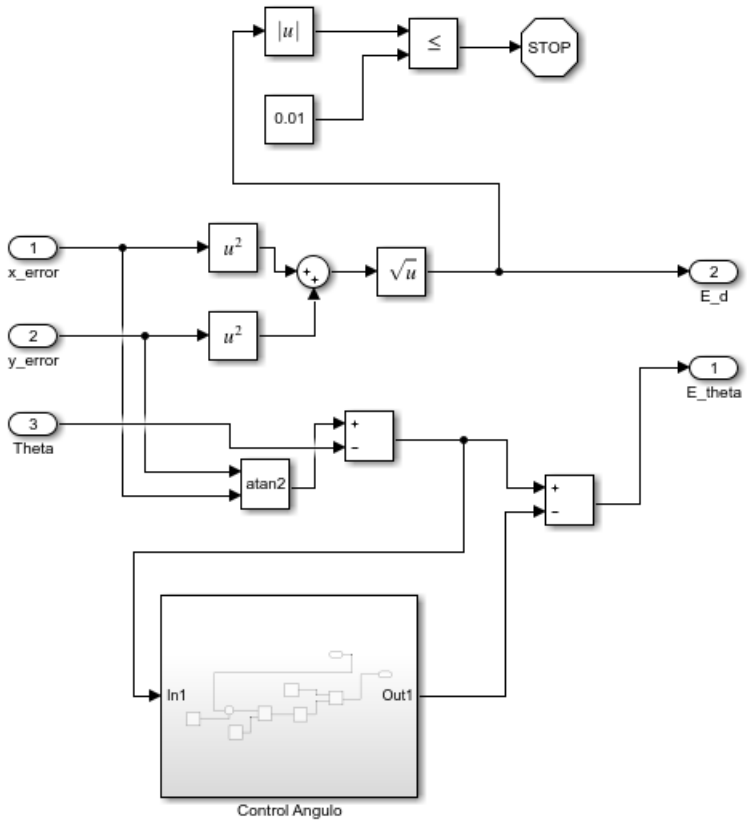
Por último, encontramos las curvas ROC de las 3 fases. Al igual que con las matrices de confusión, podemos comprobar que los resultados ya no son perfectos como en simpleclass_dataset, pero son casi perfectos ya que la curva se mantiene mayormente en un valor 1 de True Positive Rate excepto al inicio, donde es menor sin llegar a bajar de 0.9.

PARTE 2

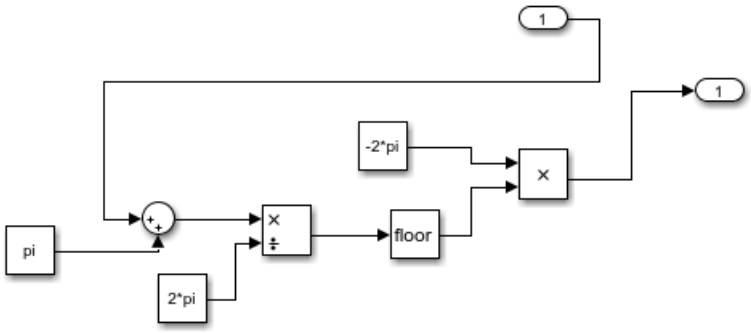
Implemente el esquema de la Figura 1, incluyendo todos los bloques principales. Utilice como controlador el proporcionado en "controlblackbox.slx". Configure los parametros de la simulación (menu "Simulation/Model Simulation Parameters") tal y como se muestra en la Figura 8. Guarde el esquema de Simulink con un nombre reconocible, por ejemplo "PositionControl.slx".



Hemos creado el siguiente esquema en Simulink utilizando el bloque "controlblackbox.slx" que se proporcionaba con la práctica, el bloque "Robot" que utilizamos en la práctica anterior y el bloque "Position_errors" que hemos creado siguiendo las instrucciones de esta práctica. También incluye una serie de entradas y salidas que utilizaremos para controlar y monitorizar el movimiento del robot.



Este es el bloque “Position_errors” visto ampliamente. Cabe destacar el bloque “Control Angulo” para que el ángulo resultante siempre se sitúe entre -pi y pi, y la condición de parada del error E_d si es inferior a 0.01.



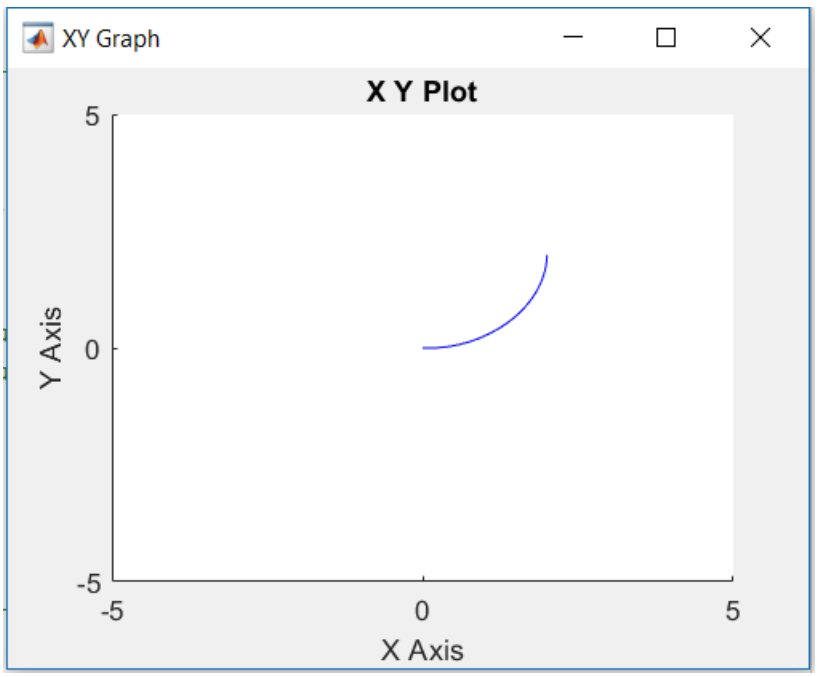
Aquí podemos echar un vistazo más a fondo a la implementación del bloque “Control Angulo”, que implementa la ecuación:

*angle_rad = angle_rad - 2*pi*floor((angle_rad+pi)/(2*pi));*

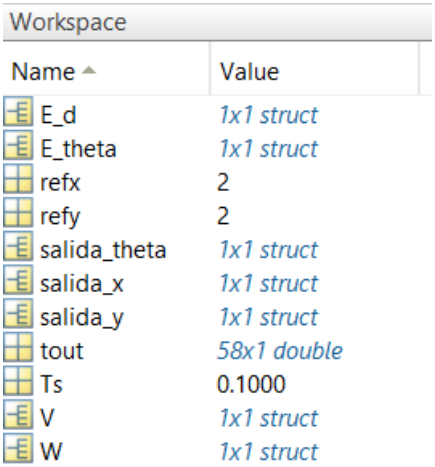
En el entorno de programacion de Matlab, cree un script nuevo con el nombre “RunPositionControl.m”. Con el siguiente codigo se puede simular el diagrama PositionControl desde el entorno de comandos de Matlab, configurando el punto destino del robot mediante las variables refx y refy y el tiempo de muestreo Ts.

```
%Tiempo de muestreo
Ts=100e-3;
%Referencia x-y de posicion
refx=2.0;
refy=2.0;
%Ejecutar simulacion
sim('PositionControl.slx')
```

Cuando ejecutamos la simulación con el punto de partida en P(2.0,2.0) se nos muestra un X Y Graph en el cual podemos ver el recorrido que ha hecho el robot para llegar hasta el punto Q(0,0).



Ejecute el script RunPositionControl y compruebe que se generan las variables que contienen las salidas y entradas del controlador (variables E_d E_theta V y W) y las salidas del robot durante la simulacion (salida_x, salida_y, salida_theta).

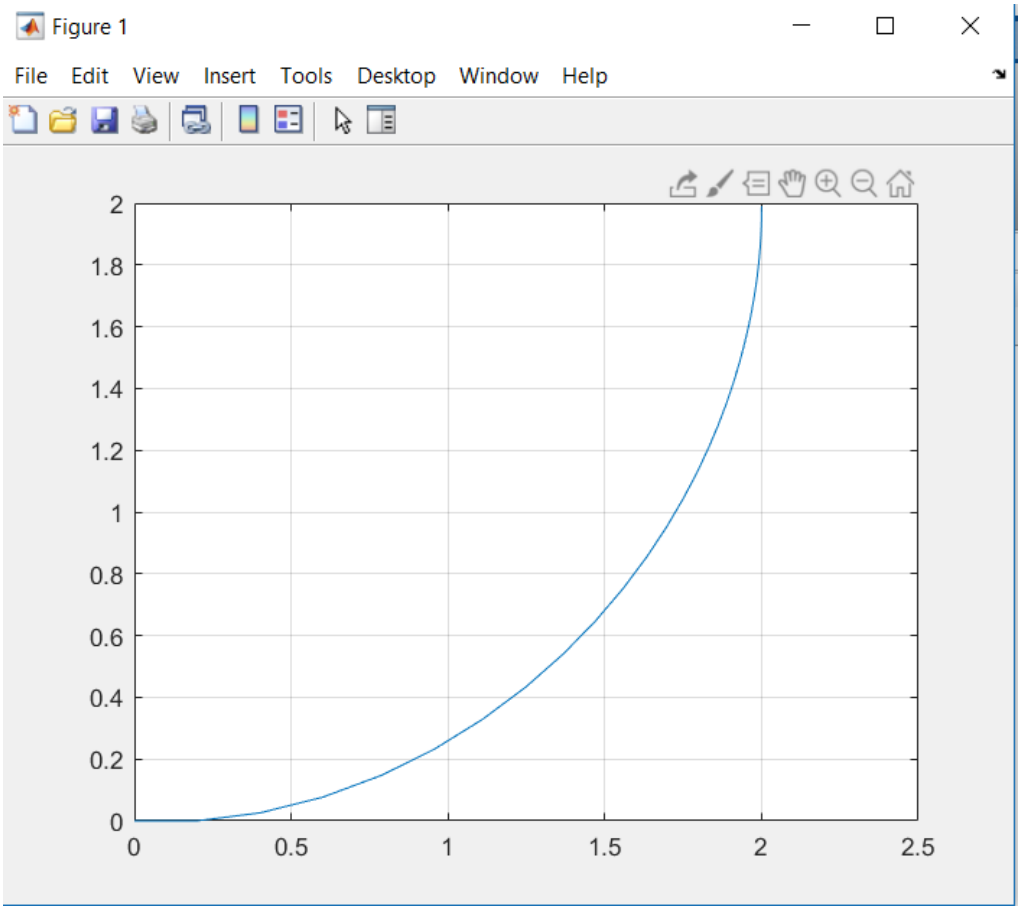


Aquí podemos ver como en nuestro Workspace se han generado todas las variables presentes en nuestro esquema de Simulink "PositionControl.slx", tanto las variables de entrada (E_d E_theta V y W) como las de salida (salida_x, salida_y, salida_theta).

Ejecute el siguiente codigo para mostrar la trayectoria del robot mediante el comando plot de Matlab.

```
% Mostrar
x=salida_x.signals.values;
y=salida_y.signals.values;
figure;
plot(x,y);
grid on;
hold on;
```

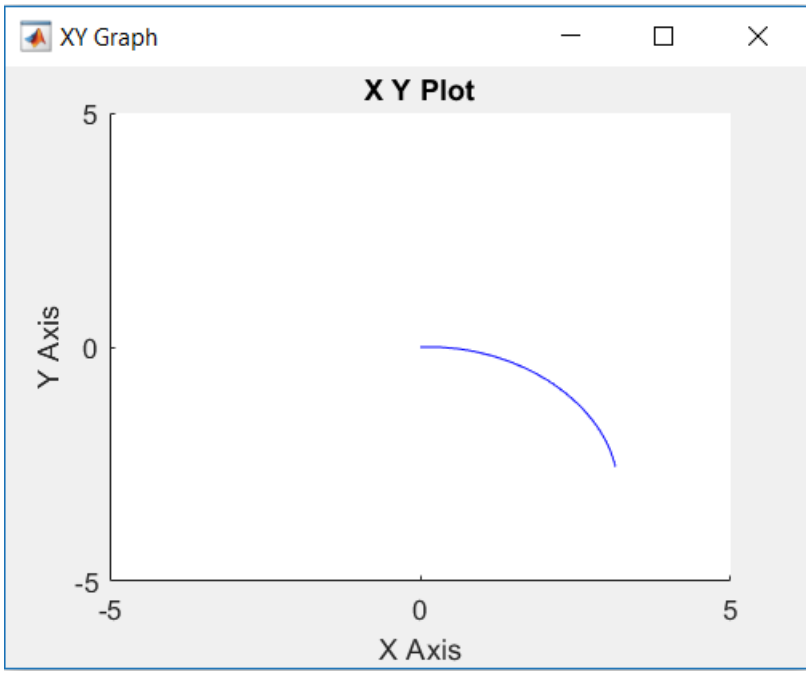
Al ejecutar el código anterior podemos ver una gráfica mejor acotada con una malla del recorrido mencionado anteriormente desde el punto de partida P(2.0,2.0) hasta Q(0,0).



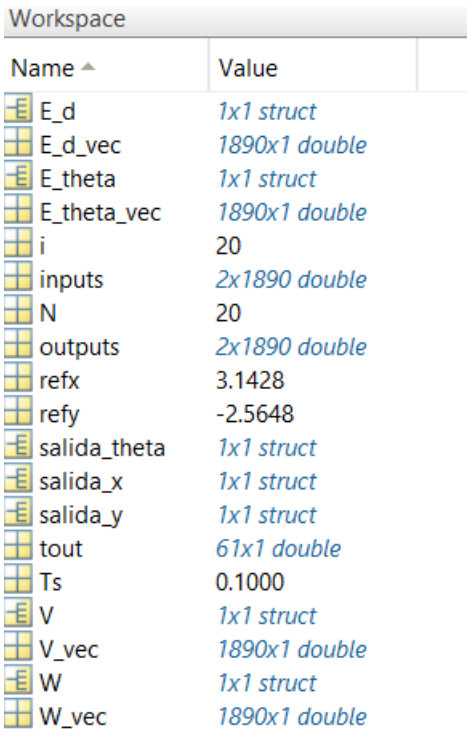
Realice N=20 simulaciones del controlador proporcionado mediante un bucle donde se varian los valores de refx y refy de manera aleatoria dentro del entorno de 10x10 metros. En cada simulación se debera guardar el valor a lo largo del tiempo de las entradas (E_d y E_theta) y salidas (V y W) del bloque controlador. Genere la matriz "inputs" de tamano 2XN, donde se acumulen los valores de E_d y E_theta. Del mismo, modo genere la matriz "outputs", donde se acumulen los valores obtenidos de las variables V y W.

```
% Generar N posiciones aleatorias, simular y guardar en variables
N=20;
E_d_vec=[];
E_theta_vec=[];
V_vec=[];
W_vec=[];
for i=1:N
    refx=10*rand-5;
    refy=10*rand-5;
    sim('PositionControl.slx')
    E_d_vec=[E_d_vec;E_d.signals.values];
    E_theta_vec=[E_theta_vec;E_theta.signals.values];
    V_vec=[V_vec; V.signals.values];
    W_vec=[W_vec; W.signals.values];
end
inputs=[E_d_vec'; E_theta_vec'];
outputs=[V_vec'; W_vec'];
```

Tras ejecutar el código anterior se han generado 20 posiciones de inicio y se ha simulado el recorrido con cada una de ellas, generando 20 distintos X Y Graph. Aquí podemos ver una captura de pantalla del último de ellos, ya que tras finalizar una simulación se sustituye la gráfica anterior por la nueva.



En el Workspace podemos comprobar que se han generado 2 matrices “outputs” e “inputs” que guardaran los valores de los datos resultantes en las variables V, W, E_d y E_theta.

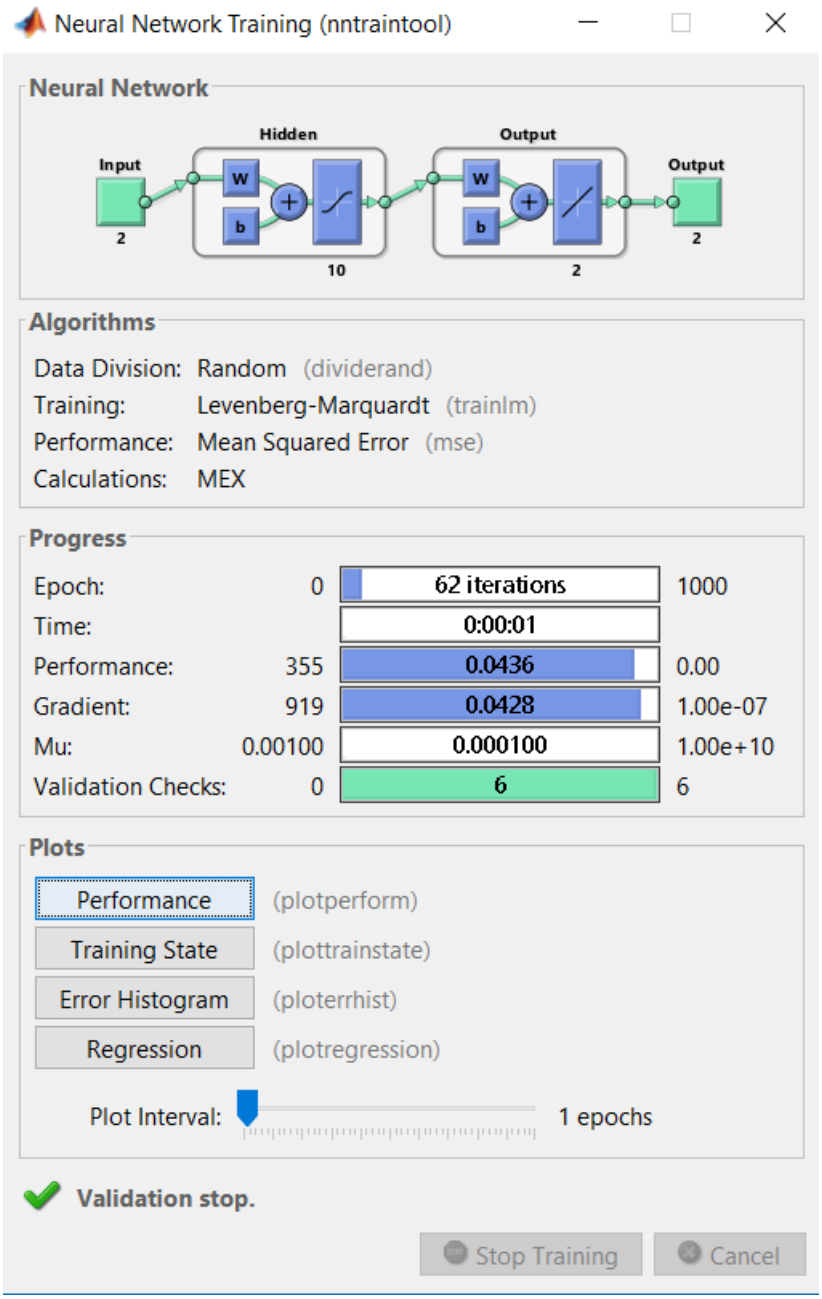


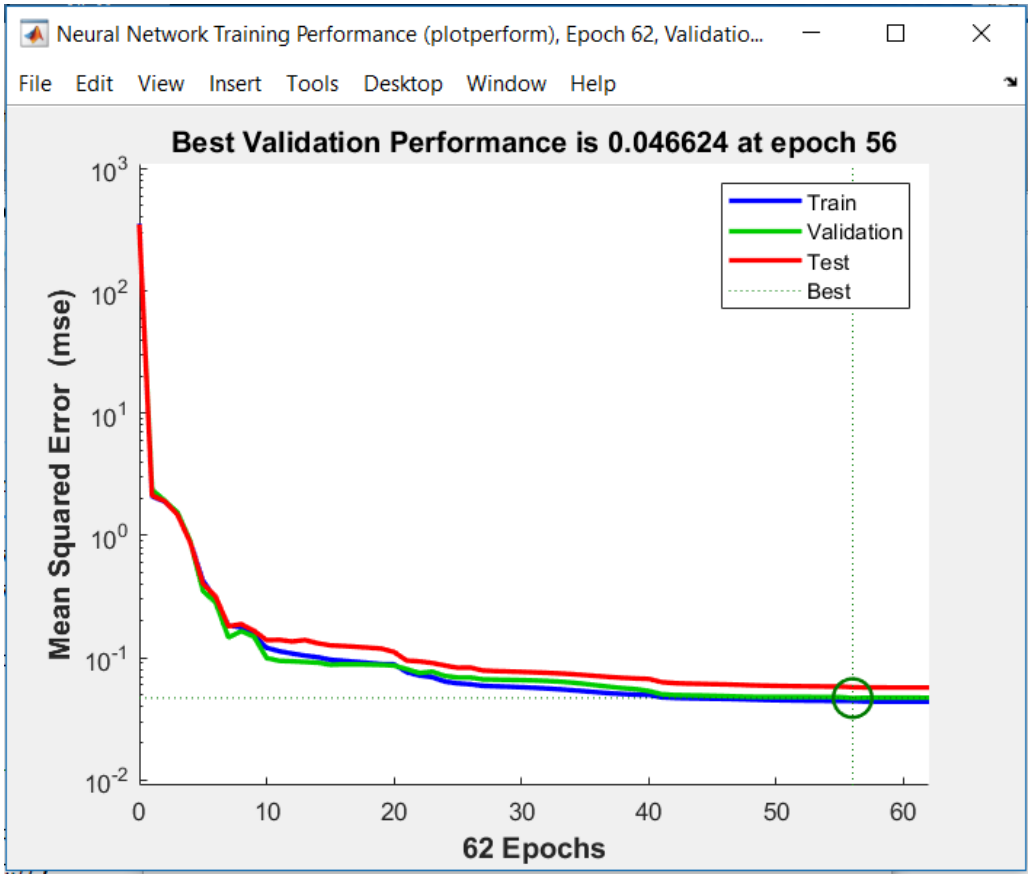
Diseñe una red neuronal con una capa oculta de tal manera que dicha red se comporte como el controlador proporcionado. El número de neuronas de la capa oculta se deberá encontrar mediante experimentación. Justifique el valor elegido. El siguiente ejemplo muestra los comandos de Matlab para realizar dicho entrenamiento a partir de los vectores inputs, outputs.

```
% Entrenar red neuronal con 10 neuronas en la capa oculta
net = feedforwardnet([10]);
net = configure(net,inputs,outputs);
net = train(net,inputs,outputs);
```

Con estas instrucciones crearemos nuestra red neuronal y la entrenaremos, probando con distinto número de neuronas en la capa oculta.

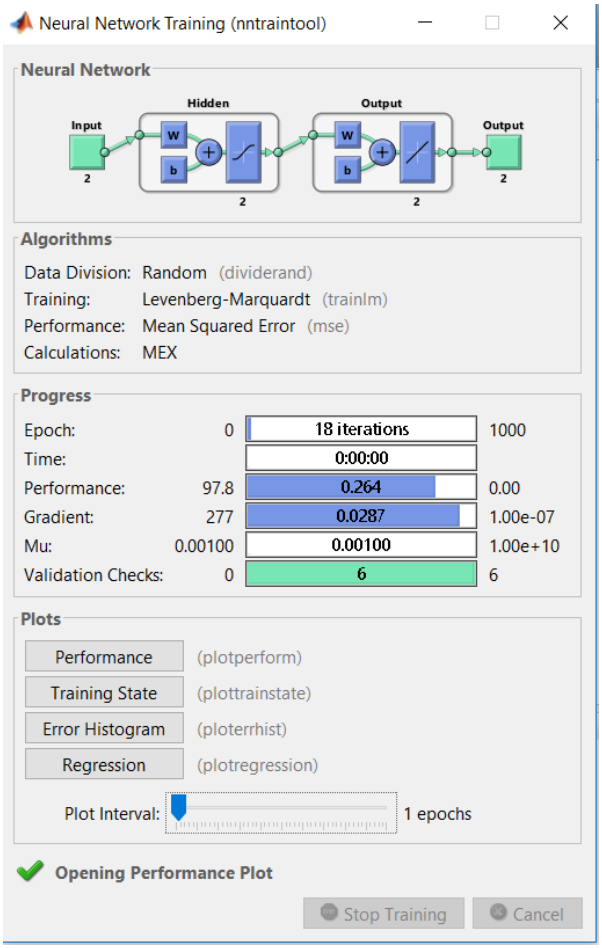
Comenzaremos con 10 Neuronas en la capa oculta, así como muestra el ejemplo:

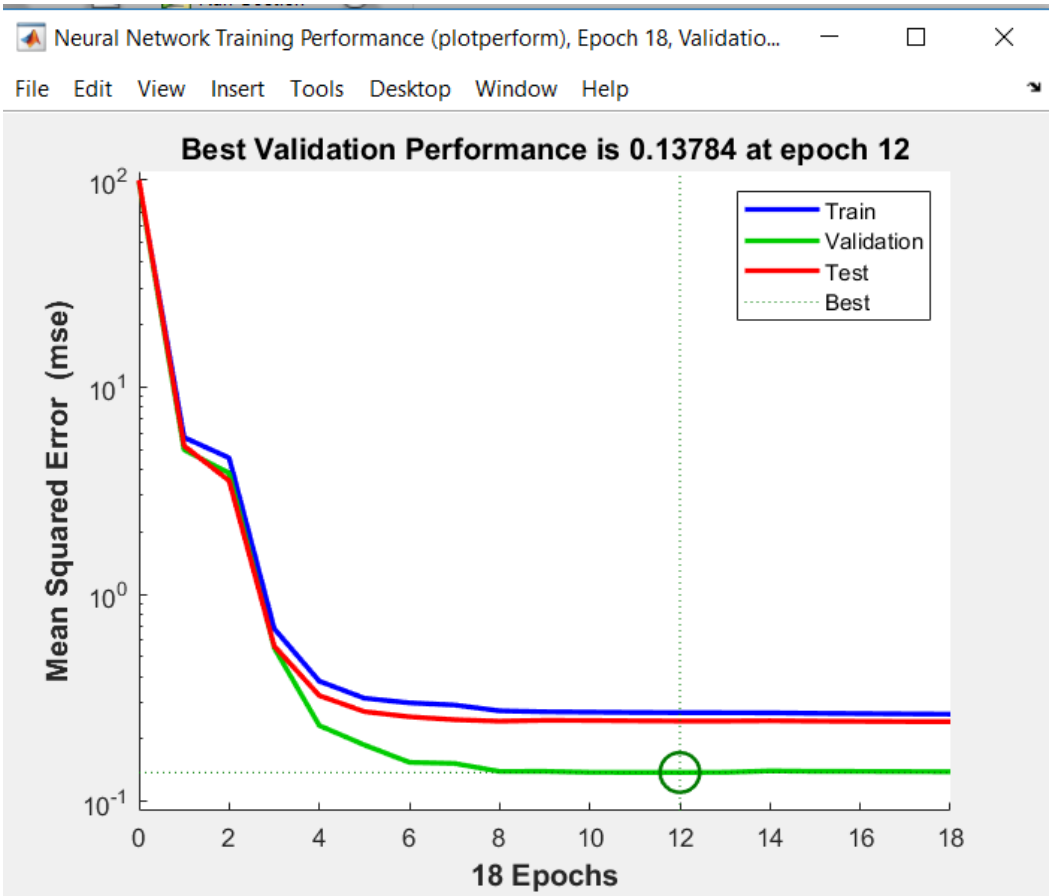




Con 10 Neuronas en la capa oculta hemos sido capaces de finalizar el entrenamiento en 62 épocas, pero, lo interesante es que el error mínimo se encuentra entre 10^{-1} y 10^{-2}, aproximándose más al primer valor.

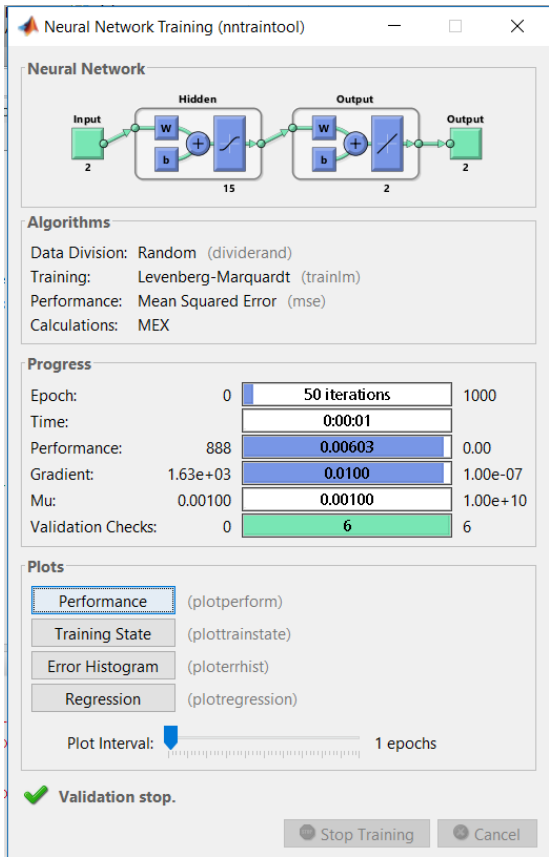
A continuación, vamos a generar nuestra red neuronal, pero esta vez solo con 2 neuronas en la capa oculta:

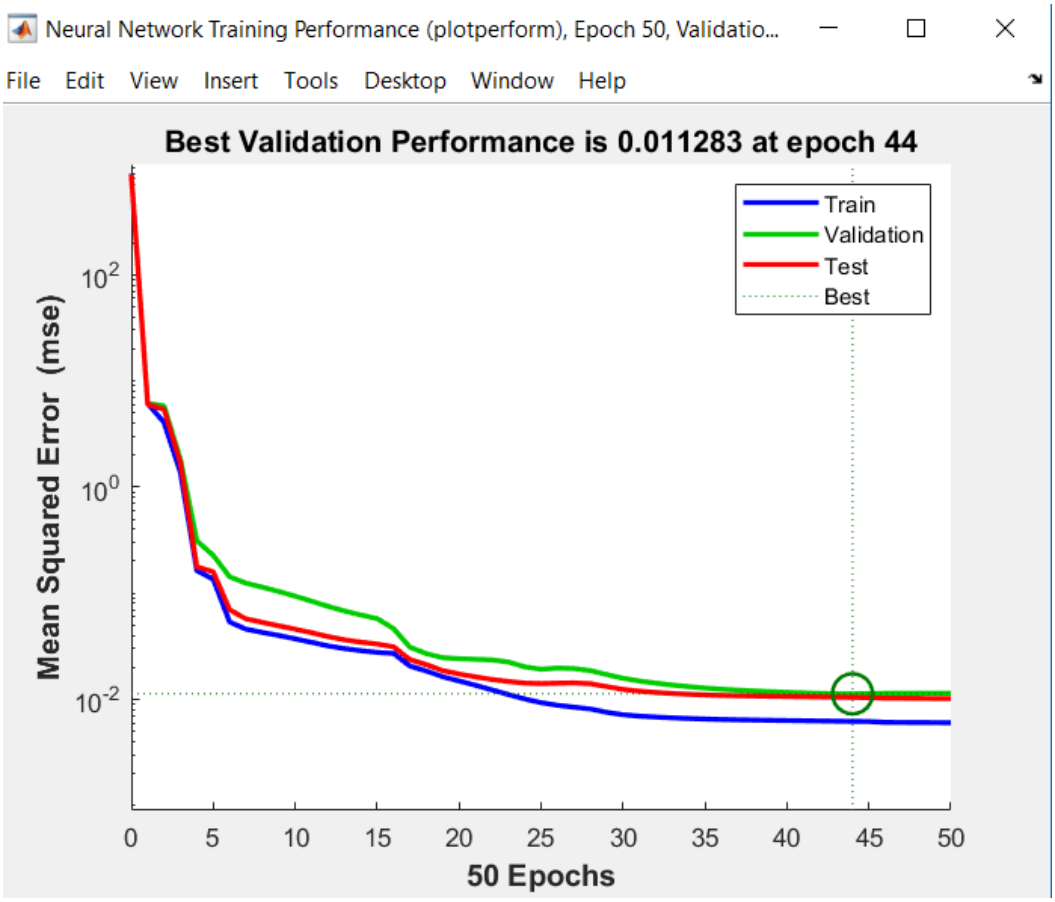




En este caso, el entrenamiento ha finalizado solamente en 18 épocas, pero el error mínimo alcanzado es mucho mayor al obtenido con 10 neuronas, ya que ni siquiera alcanza valores de orden 10^{-1}. Además, el error mínimo alcanzado en las fases de Entrenamiento y Test es aún mayor que el de la fase de Validación.

Por último, vamos a probar a aumentar el número de neuronas en la capa oculta, probaremos con 15 esta vez:

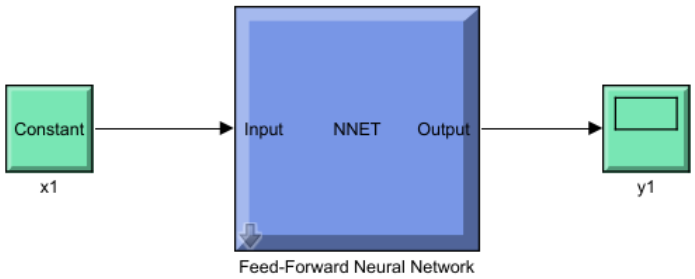




Cuando ejecutamos el entrenamiento con 15 neuronas en la capa oculta, el error mínimo alcanzado se sitúa en valores de orden 10^{-2}, el más bajo que hemos encontrado hasta el momento, y suficiente como para que el comportamiento de nuestra Red Neuronal imite el del bloque de control inicial "controlblackbox.slx".

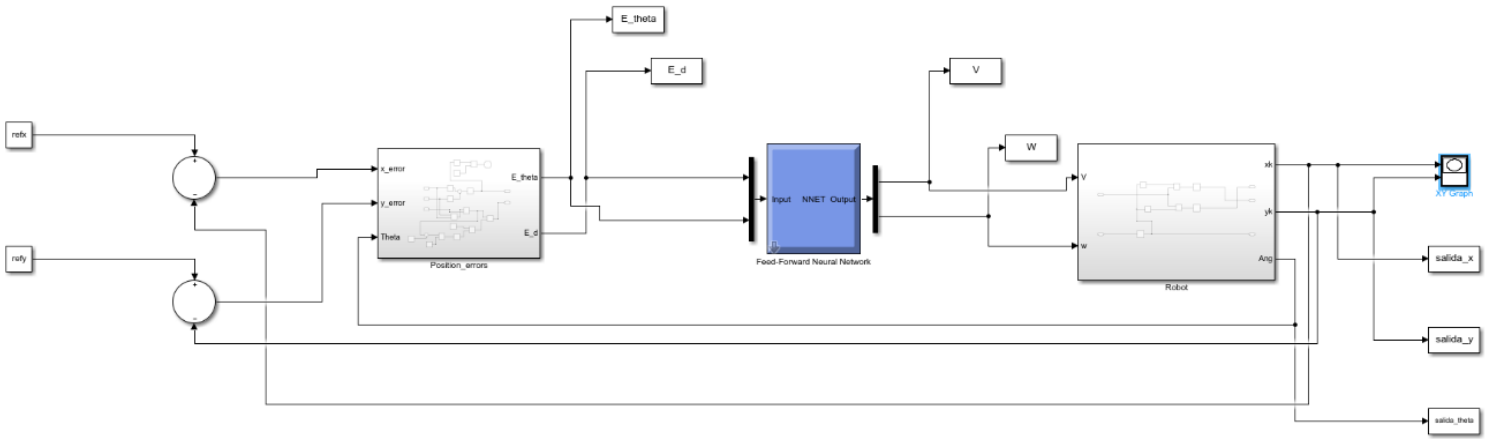
Genere, mediante la orden gensim de Matlab, un bloque con la red neuronal propuesta:

```
gensim(net,Ts);
```



Con la orden "gensim" hemos generado un bloque de Simulink con la Red Neuronal generada en el apartado anterior y ahora seremos capaces de introducirla en nuestro esquema.

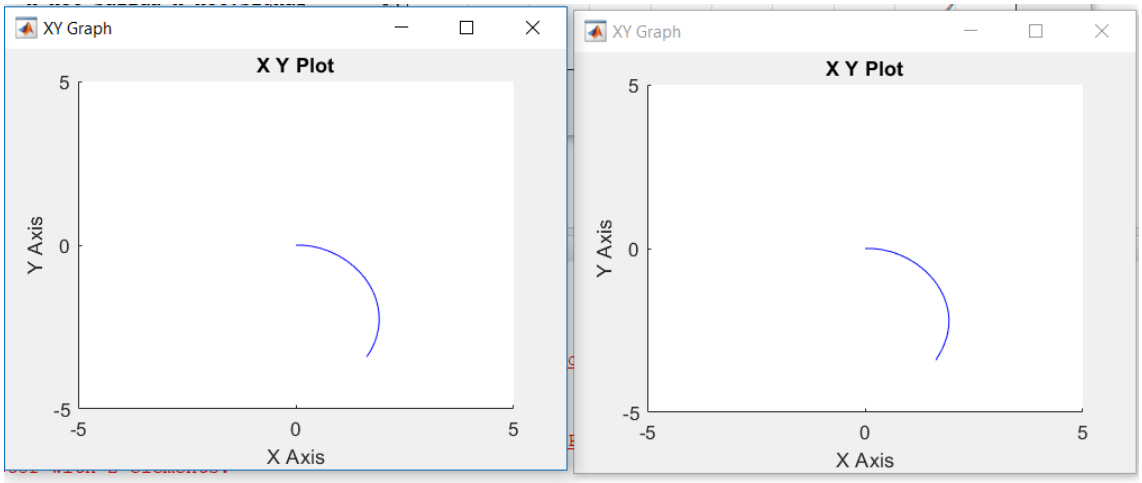
Cree un nuevo archivo de simulacion “PositionControlNet.slx” con los mismos bloques utilizados en “PositionControl.slx” y donde se utilice la red neuronal en lugar del bloque controlador (Figura 10). Se utilizan bloques multiplexores y demultiplexores para adaptar las senales de entrada y salida como se muestra en la Figura.

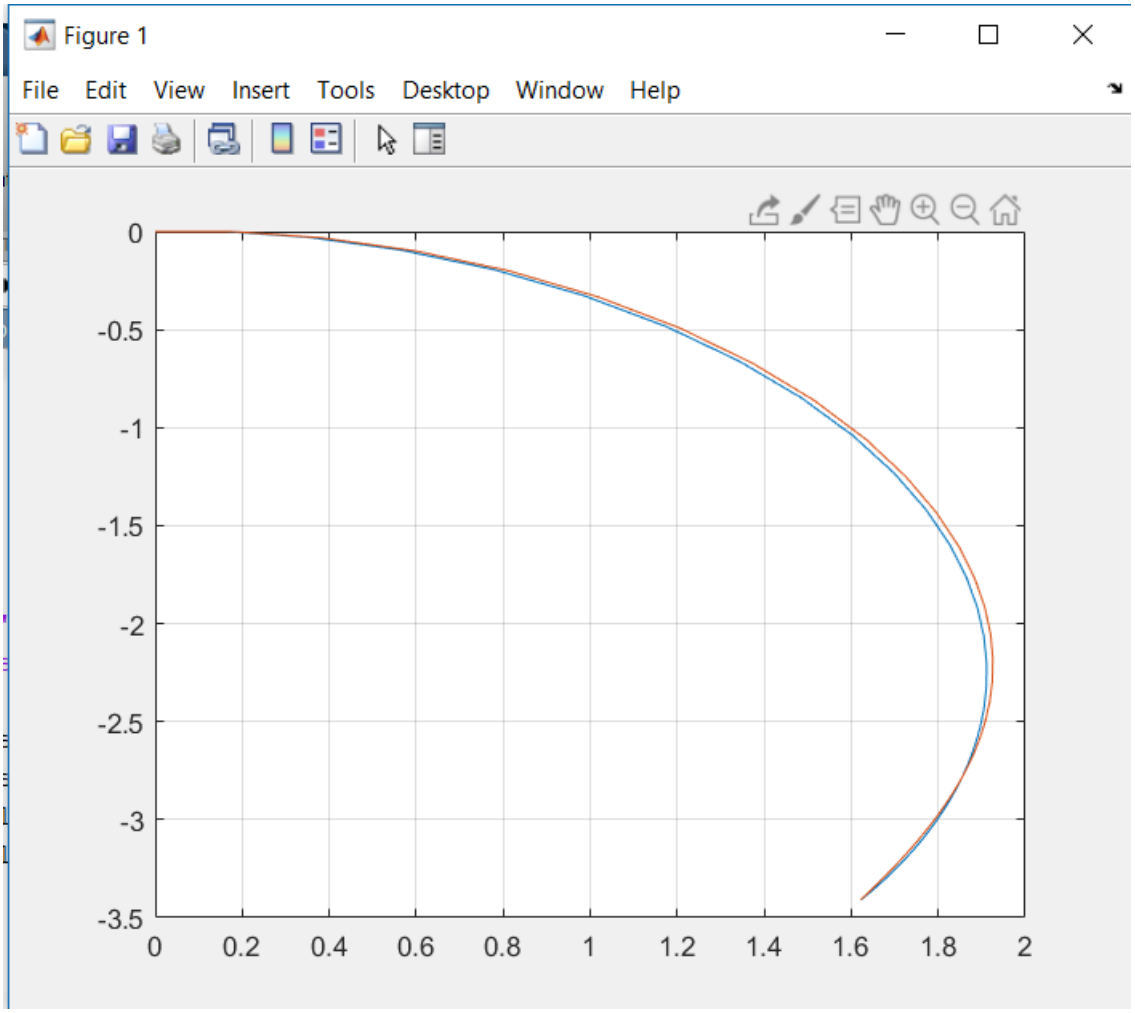


Hemos introducido nuestra nueva Red Neuronal en nuestro esquema, sustituyendo al anterior bloque de control “controlblackbox.slx” y utilizando bloques multiplexores y demultiplexores para ajustarla al resto del esquema.

Compare el comportamiento de “PositionControlNet.slx” con “PositionControl.slx” para diferentes valores de refx y refy. Calcule el error entre las trayectorias realizadas por ambos controladores. Se recomienda realizar un script de Matlab para automatizar dicha comparacion y mostrar los resultados.

Por último, generaremos 2 entradas aleatorias y simularemos el control tanto con el esquema “PositionControl.slx” como con nuestro esquema “PositionControlNet.slx” para comprobar si hemos logrado que la Red Neuronal imite el comportamiento del bloque de Control y realicen la misma trayectoria.



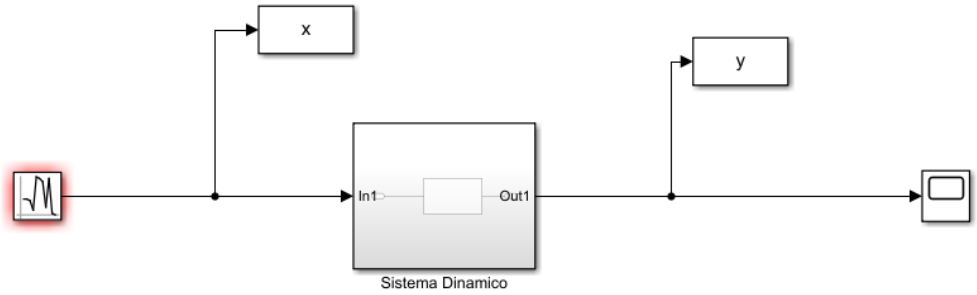


En los respectivos XY Graph de cada simulación podemos ver como el recorrido es idéntico a simple vista. Cuando juntamos ambas trayectorias en un mismo gráfico con malla podemos confirmar que son prácticamente idénticas, teniendo breves desviaciones de orden de milésimas.

Por lo tanto, podemos concluir que el desarrollo de la Red Neuronal Identificadora ha sido todo un éxito y ha logrado imitar el comportamiento del bloque de control.

PARTE 3

Se dispone de un sistema dinámico de tiempo discreto (Ts = 0.1s) con una entrada y una salida y cuyas ecuaciones son desconocidas. Es posible registrar su comportamiento (salida) ante cualquier tipo de entrada mediante un sistema de prueba o "test-bench". Se desea entrenar una red neuronal de tipo NARX que emule con exactitud la respuesta del sistema. El sistema tipo caja negra se proporciona en el archivo modelo_identificacion.slx.

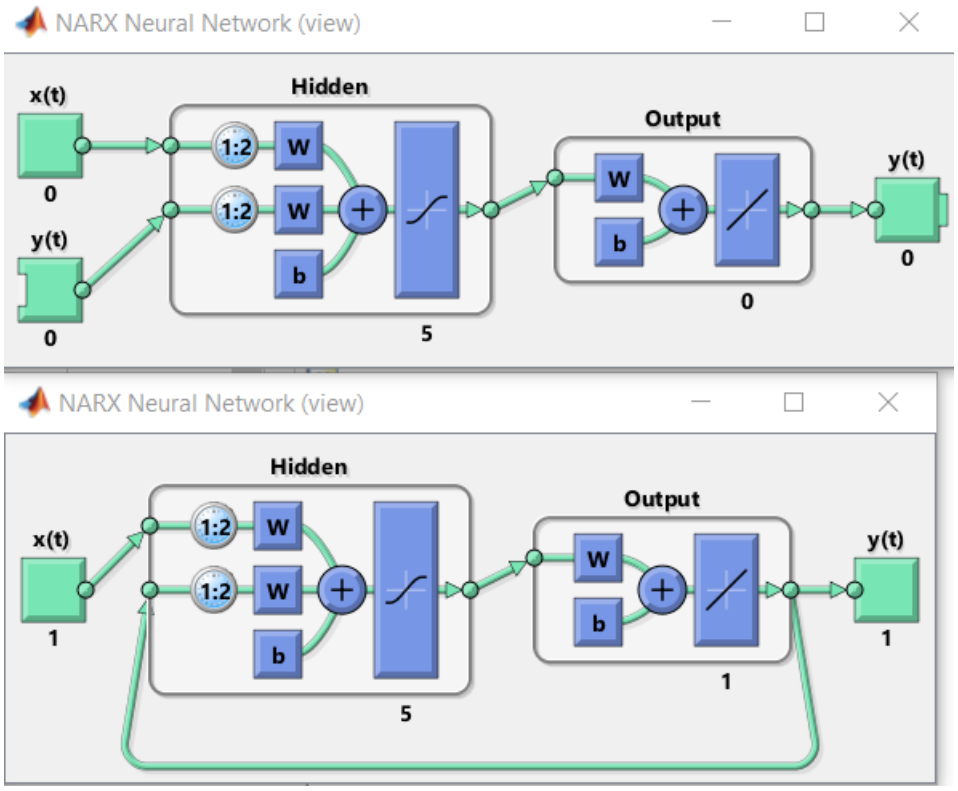



```
Ts = 0.1;
S=sim('test_bench2.slx');
inputs=S.x.signals.values';
outputs=S.y.signals.values';
```

Inicialmente contamos con un sistema dinámico que ejecutaremos con el código mostrado para realizar la simulación de nuestro archivo.

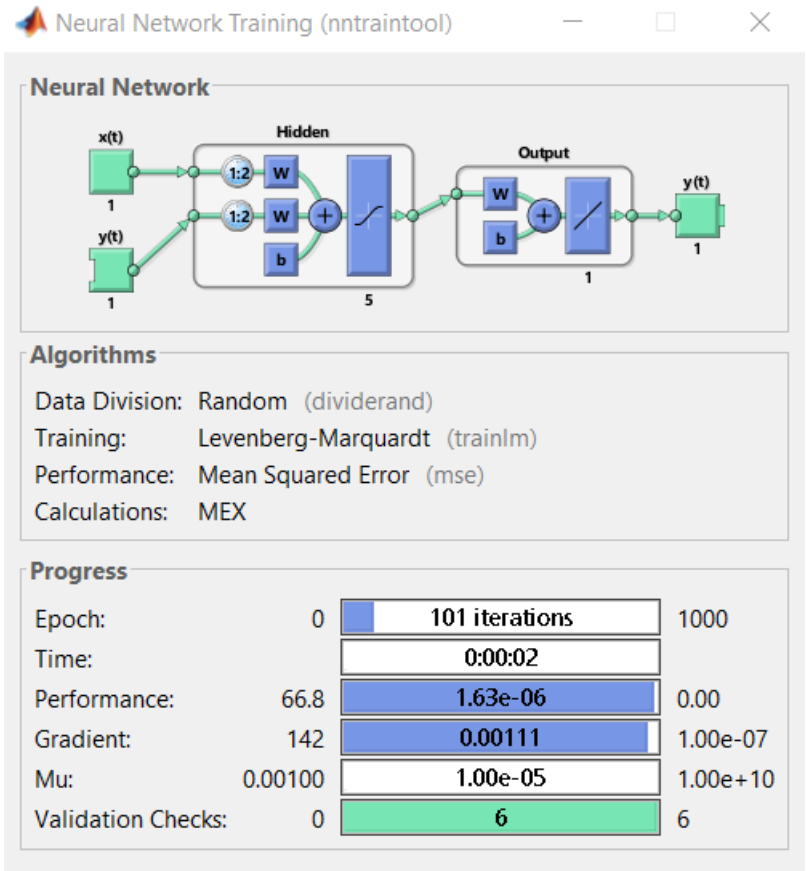
A continuación, lo que haremos será generar y entrenar una red NARX, con 5 neuronas en la capa oculta y 2 retardos en la entrada y la salida, que sea capaz de imitar el comportamiento de nuestro sistema dinámico. Para ello ejecutaremos el siguiente código:

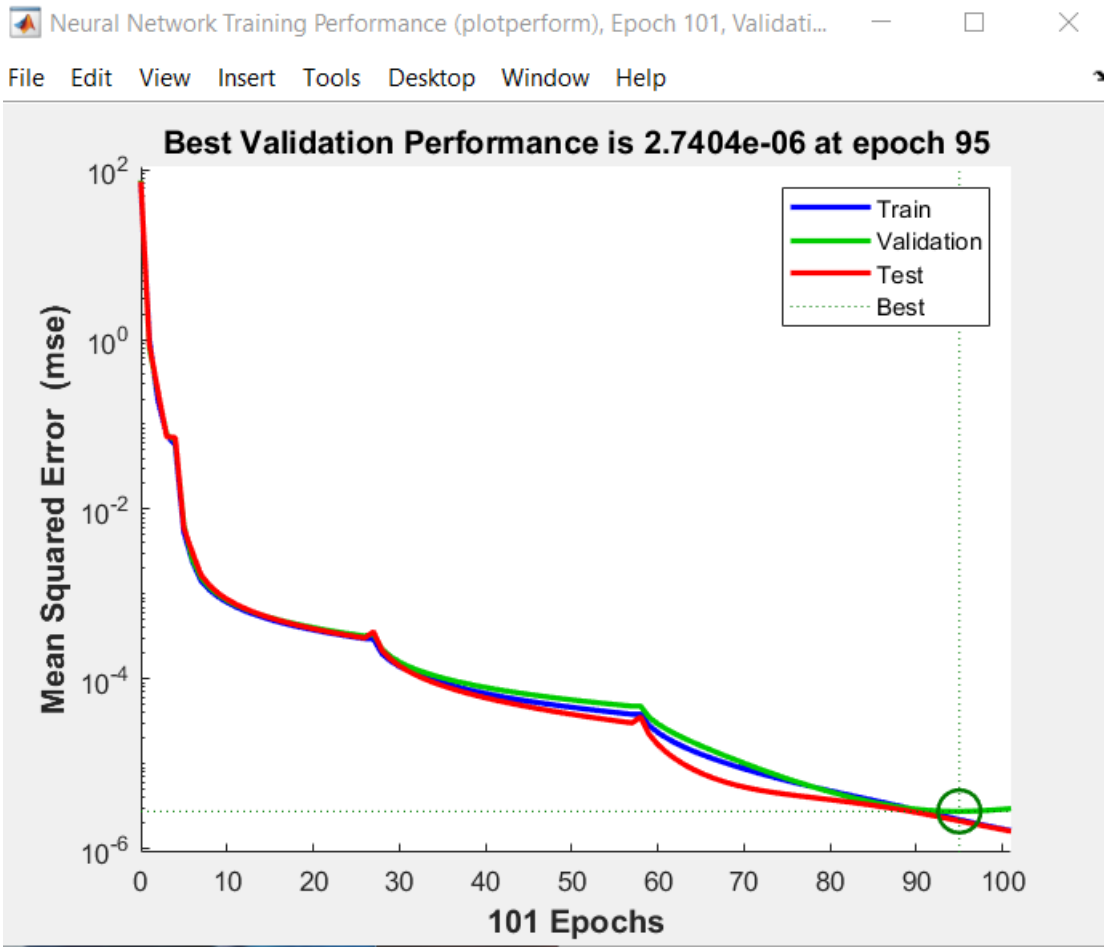
```
%Definición del modelo NARX
N=5;
net = narxnet(1:2,1:2,[N]);
view(net)

nT=size(inputs,2);
inputsc=mat2cell(inputs,1,ones(nT,1));
outputsc=mat2cell(outputs,1,ones(nT,1));

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

net = train(net,x,t,xi,ai);
net = closeloop(net);
view(net)

gensim(net,Ts);
```



Tras generar la red NARX, en el primer dibujo podemos ver como se ve nuestra red con la entrada y la salida retardadas. Abajo podemos ver cómo se realimenta la salida con dos retardos.

Compruebe el error de entrenamiento de la red en los conjuntos de train, validation y test.

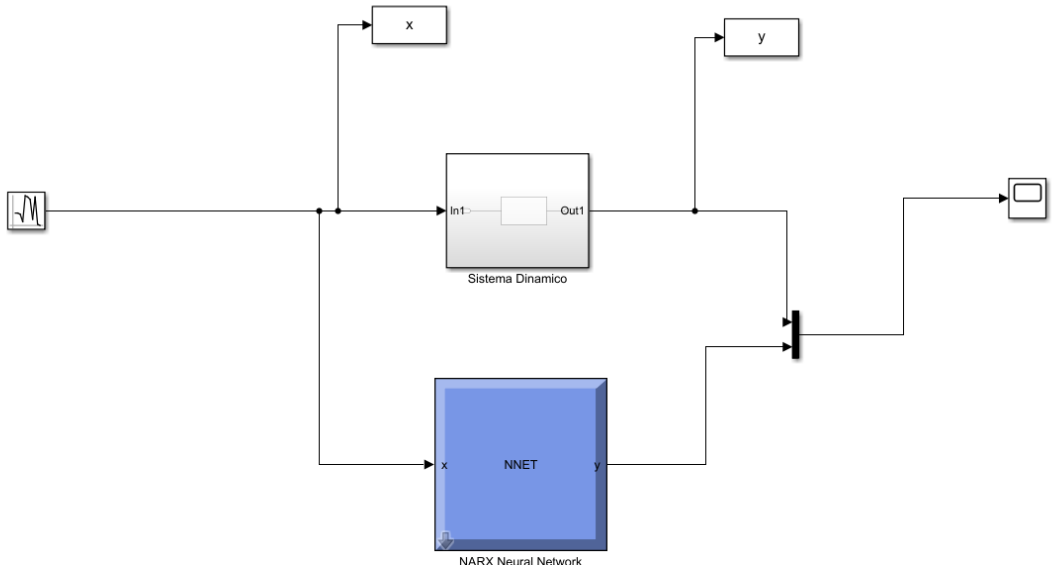




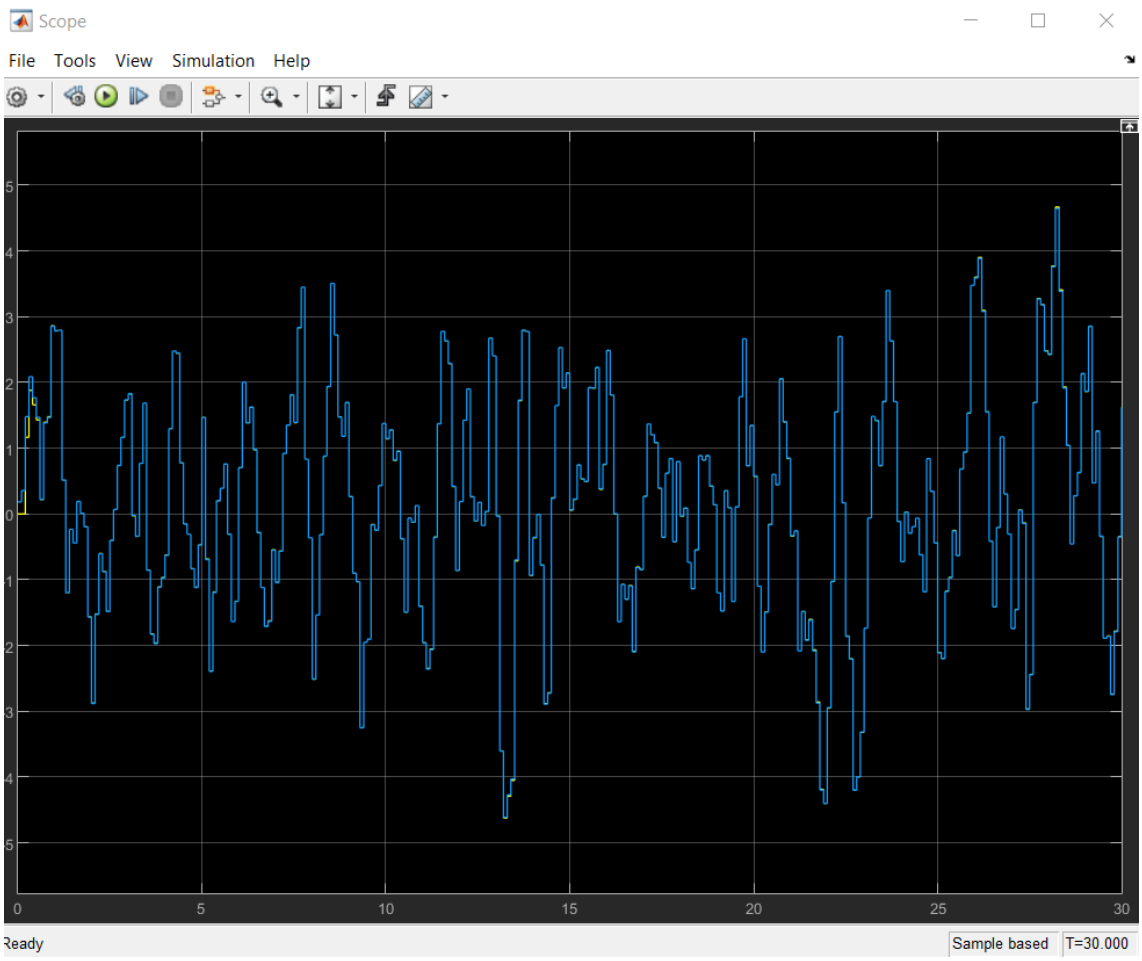
Aquí podemos ver el performance de nuestro entrenamiento y vemos como tanto en la fase de Entrenamiento, Validación y Test el error se reduce hasta llegar a valores casi de orden 10-6.

Por último, genere un modelo de Simulink de la red entrenada mediante el comando gensim y compare el desempeño de la red y el modelo, generando el s esquema de simulación test_bench2.xls mostrado en la Figura 3. Visualice el error entre la red neuronal y el sistema mediante el "scope" de dos canales.

Una vez hayamos entrenado la red NARX y generado el bloque de Simulink correspondiente a esta, vamos a añadirla a nuestro esquema de Simulink anterior para comprobar si logra imitar el comportamiento de nuestro sistema dinámico.



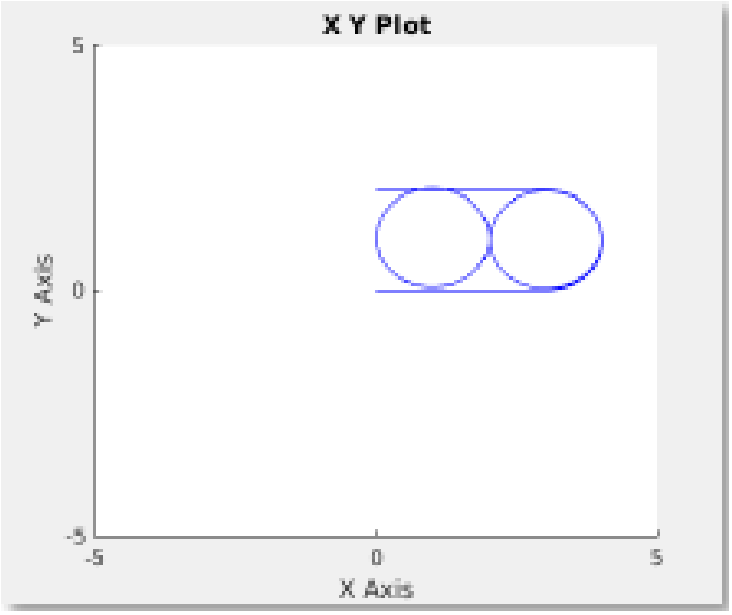
Aquí podemos ver el esquema resultante de situar en paralelo el sistema dinámico y nuestra red. A continuación, en nuestro Scope podemos comprobar si generan el mismo resultado:



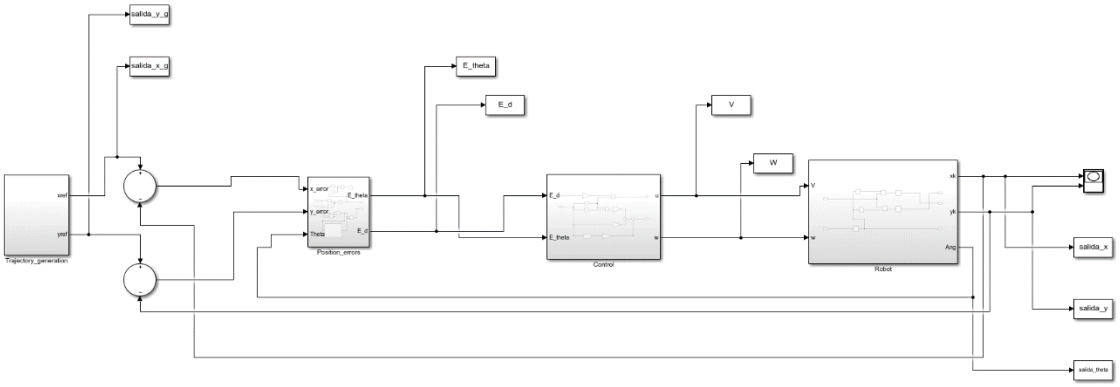
Para un tiempo de 30 segundos podemos comprobar como el resultado generado por nuestra red es idéntico al del sistema dinámico ya que las curvas se solapan en todo momento excepto levemente al inicio.

El objetivo de este ejercicio es diseñar un controlador que permita al robot, ya utilizado en sesiones anteriores de esta práctica, seguir una trayectoria preconfigurada. El esquema de control, mostrado en la Figura 4, es idéntico al utilizado en la sesión II de la práctica1, sustituyendo las referencias de posición (bloques refx y refy) por un generador de trayectorias y sustituyendo el controlador por un nuevo controlador de tipo "caja negra" diseñado especialmente para el seguimiento de trayectorias ("controlblackboxTrajectory.slx").

En esta parte se nos pide que implementemos un bloque llamado "Trajectory_generation" que realiza la siguiente trayectoria:



Implemente el esquema de la Figura 4. Utilice como controlador el proporcionado en el archivo "controlblackboxTrajectory.slx". Mantenga la configuración de los parámetros de la simulación (menú "Simulation/Model Simulation Parameters") de sesiones anteriores. Guarde el esquema de Simulink con un nombre reconocible.



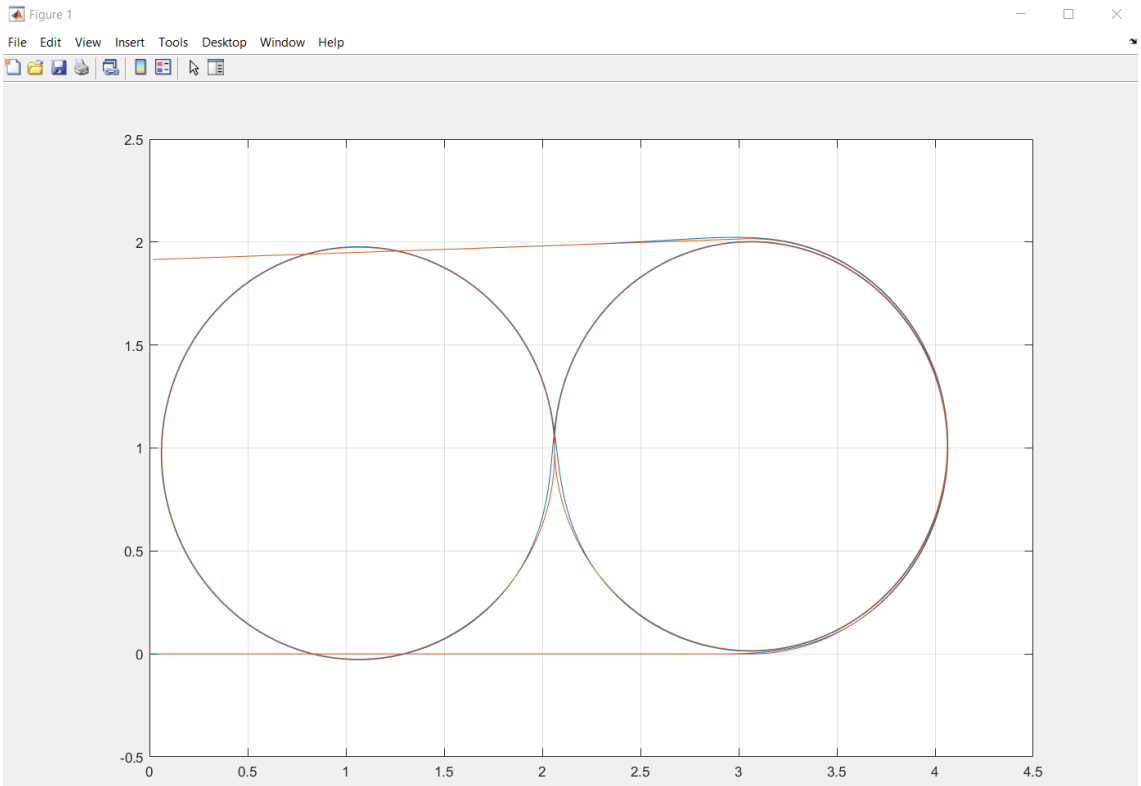
Hemos introducido el bloque Trajectory_generation dentro de nuestro esquema junto al bloque controlblackboxTrajectory.slx y lo hemos guardado en el archivo llamado "TrajectoryControl.slx".

Genere un script de Matlab, "RunTrajectoryControl.m" donde se muestre el seguimiento de la trayectoria seguida por el robot mediante el controlador proporcionado. Visualice mediante el

comando plot de Matlab la trayectoria seguida por el robot y compárela con la trayectoria generada. Para ello, añada bloques de exportación al entorno de Matlab para las referencias de posición.

```
Ts=0.1;
%Referencia x-y de posicion
x_0=0.02;
y_0=0;
th_0=0;
%Ejecutar simulacion
sim('TrajectoryControl.slx')
x=salida_x.signals.values;
y=salida_y.signals.values;
x_control=salida_x_g.signals.values;
y_control=salida_y_g.signals.values;
figure;
plot(x,y);
hold on;
plot(x_control,y_control);
hold off;
grid on;
```

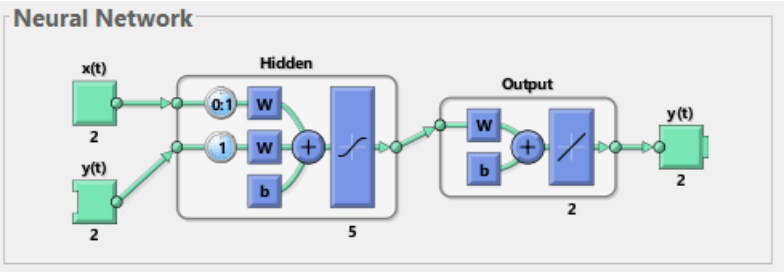
Tras ejecutar el código anterior podemos mostrar el resultado de la simulación de nuestro archivo Simulink y comprobar si las trayectorias coinciden.



Como podemos ver, el recorrido generado por el bloque Trajectory_generation y el bloque de control es idéntico pues pues las líneas se solapan y coinciden en todo momento.

Siguendo los pasos descritos en el ejercicio 1, diseñe y entrene una red neuronal recursiva de tipo "narx" para emular el comportamiento del controlador proporcionado. El entrenamiento se realizará a partir de los datos generados en el apartado b). La red debe tener una capa oculta, dos retardos en la entrada y un retardo en la realimentación de la salida. Justifique la elección del número de neuronas de la capa oculta mediante experimentación.

Como ya hemos realizado anteriormente, ahora vamos a generar una red NARX que imite el comportamiento de nuestro bloque de control y sea capaz de recorrer la misma trayectoria. Esta vez nuestra red NARX contará también con dos retardos a la entrada, pero solo uno a la salida realimentada.

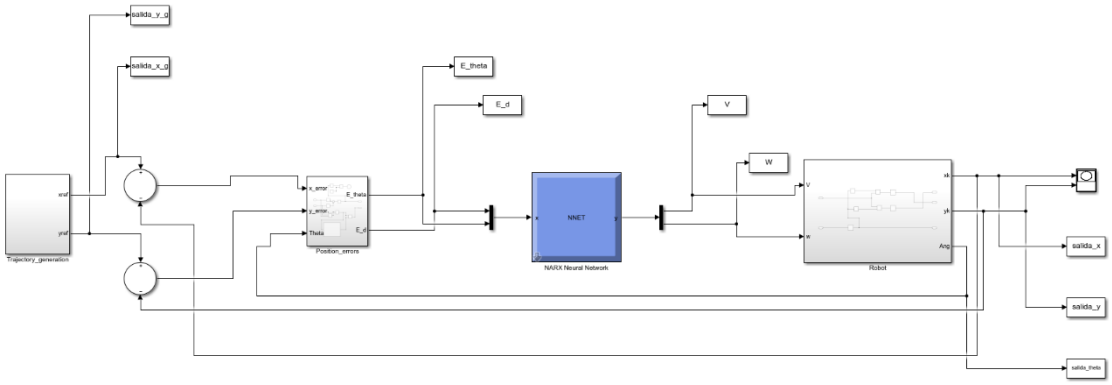


Aquí podemos ver como ahora la realimentación de la salida solo tiene 1 retardo.

```
E_d_vec=E_d.signals.values;
E_theta_vec=E_theta.signals.values;
V_vec=V.signals.values;
W_vec=W.signals.values;
inputs=[E_d_vec';E_theta_vec'];
outputs=[V_vec';W_vec'];

N=5;
net = narxnet(0:1,1,[N]);
nT=size(inputs,2);
inputsc=mat2cell(inputs,2,ones(nT,1));
outputsc=mat2cell(outputs,2,ones(nT,1));
[x,xi,ai,t] = preparets(net,inputsc,{},outputsc);
net = train(net,x,t,xi,ai);
net = closeloop(net);
% view(net);

gensim(net,Ts);
```

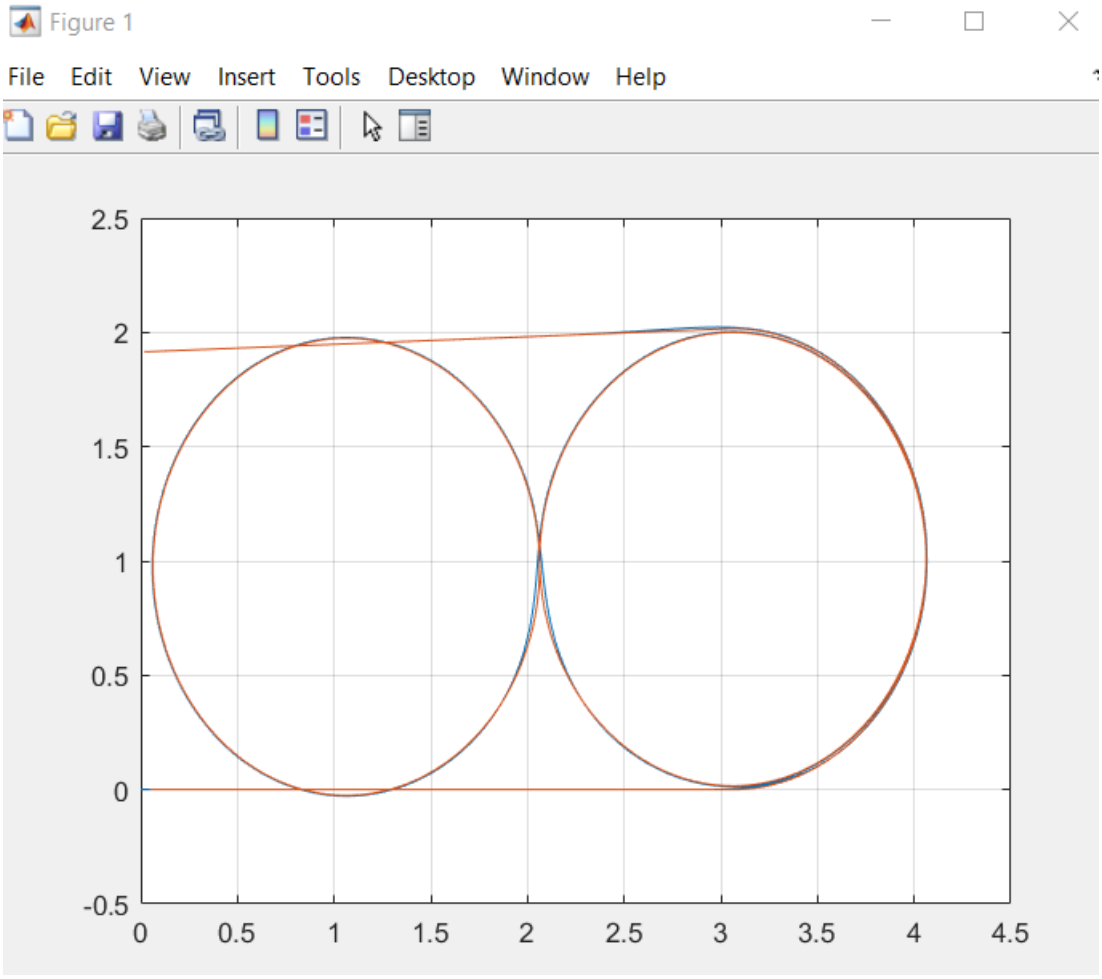
Para generar la red NARX utilizamos de parámetro de InputDelay 0:1 para que la salida y(t) sea función de tanto la entrada actual x(t), como la entrada retardada x(t-1), además, claro está, de la salida retardada y(t-1) que incluimos como parámetro de FeedbackDelay. Dando como resultado de nuestra y(t): y(t)= f(x(t-1), x(t), y(t-1)). Esto se debe a que necesitamos saber en que posición nos encontramos actualmente para saber como realizar el movimiento, por lo que necesitamos x(t).

Tras realizar diferentes pruebas con distinto número de neuronas en la capa oculta, hemos determinado que con 5 neuronas es suficiente como para que nuestra red NARX imite el comportamiento del bloque de control perfectamente, llegando a valores de error en el entrenamiento de orden 10^{-9}.



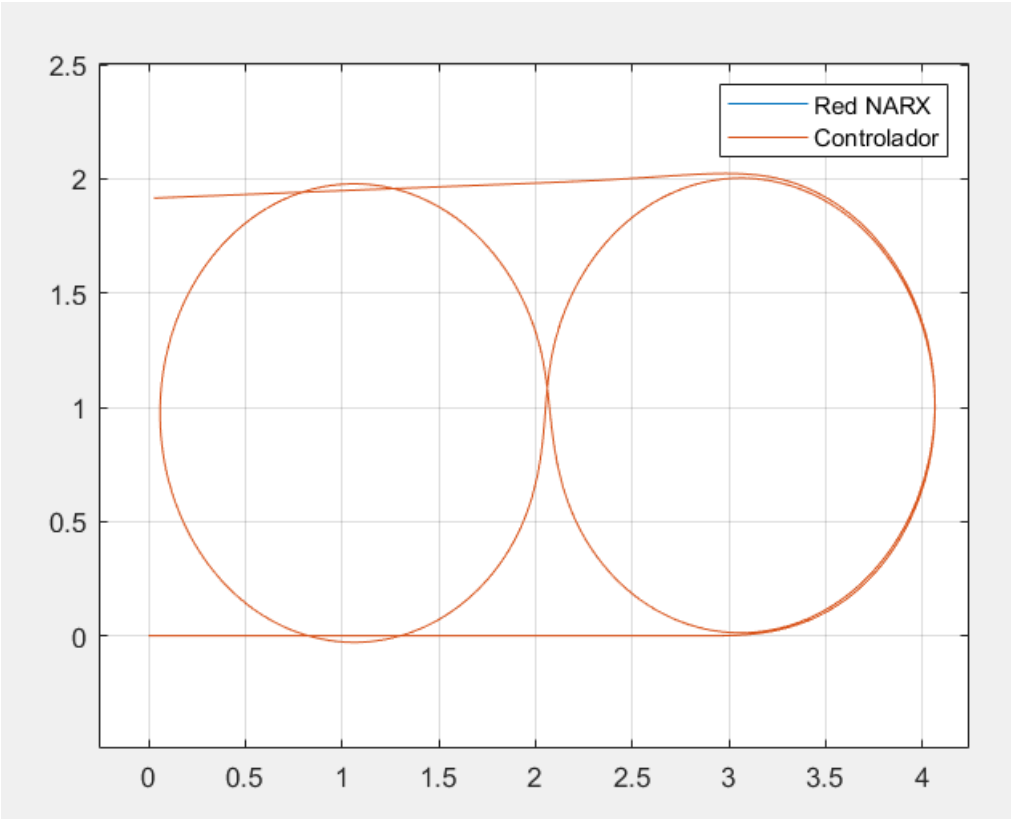
Tras entrenar y generar nuestro bloque de la red NARX lo implementaremos en nuestro esquema TrajectoryControl sustituyendo el bloque de control y cambiando el nombre del archivo de simulación a “TrajectoryControlNet.slx”.

Utilizando el comando gensim de Matlab, genere los bloques de Simulink correspondientes a las redes neuronales diseñadas en los apartados c) y d). Simule el comportamiento de dichos bloques para diferentes trayectorias generadas variando los valores de x0 e y0 y compare los resultados con el comportamiento del controlador original.



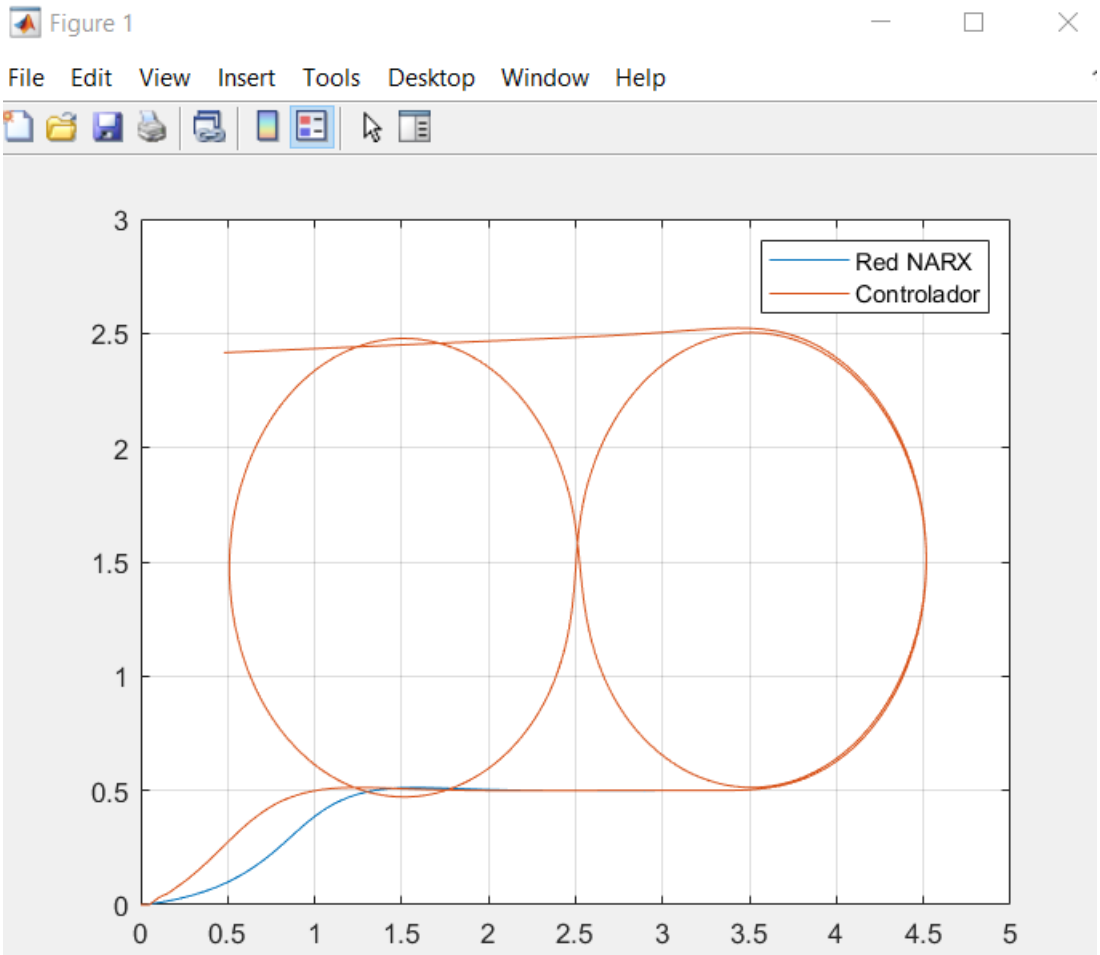
Si realizamos una comparativa de la red neuronal con la trayectoria del bloque Trajectory_generation, podemos comprobar que el gráfico resultante es idéntico al que resultaba de comparar el bloque de control con Trajectory_generation.
Por lo tanto, es de esperar que cuando comparemos el gráfico de la Red Neuronal con el del bloque de Control, el resultado sea idéntico. Probaremos con distintos puntos de inicio.

Para empezar, probaremos inicializando x_0=0.05 e y_0=0:



Como podemos ver, la trayectoria es tan parecida que solo se puede apreciar una de las líneas, la naranja, debido a que se solapan en todo momento, aunque en la leyenda podemos observar que existen 2 recorridos en el gráfico.

Por realizar otra prueba comprobaremos que sucede si ponemos x_0=0.5 e y_0= 0.5:



Como podemos ver, esta comparativa no es del todo exacta, teniendo un error de menos de 0.2 de distancia entre ambas trayectorias al principio de la simulación, sin embargo no tardan mucho en acoplarse y realizar una trayectoria idéntica a partir del punto P(1.5,0.5).

Como conclusión, hemos desarrollado una red NARX capaz de imitar el comportamiento de un bloque controlador, del que desconocíamos el funcionamiento inicialmente, a la perfección.