



Universidad
de Alcalá

Práctica 1. Identificación y control neuronal (I)

Sistemas de Control Inteligente

Grado en Ingeniería Computadores Grado en Ingeniería
Informática Grado en Sistemas de Información

Universidad de Alcalá

Toolbox necesarias para esta práctica:

- **Redes neuronales:**
 - o **Deep Learning Toolbox** (versiones R2018b y posteriores) o **Neural Network Toolbox** (versiones R2018a y anteriores)
- **Signal Processing Toolbox**

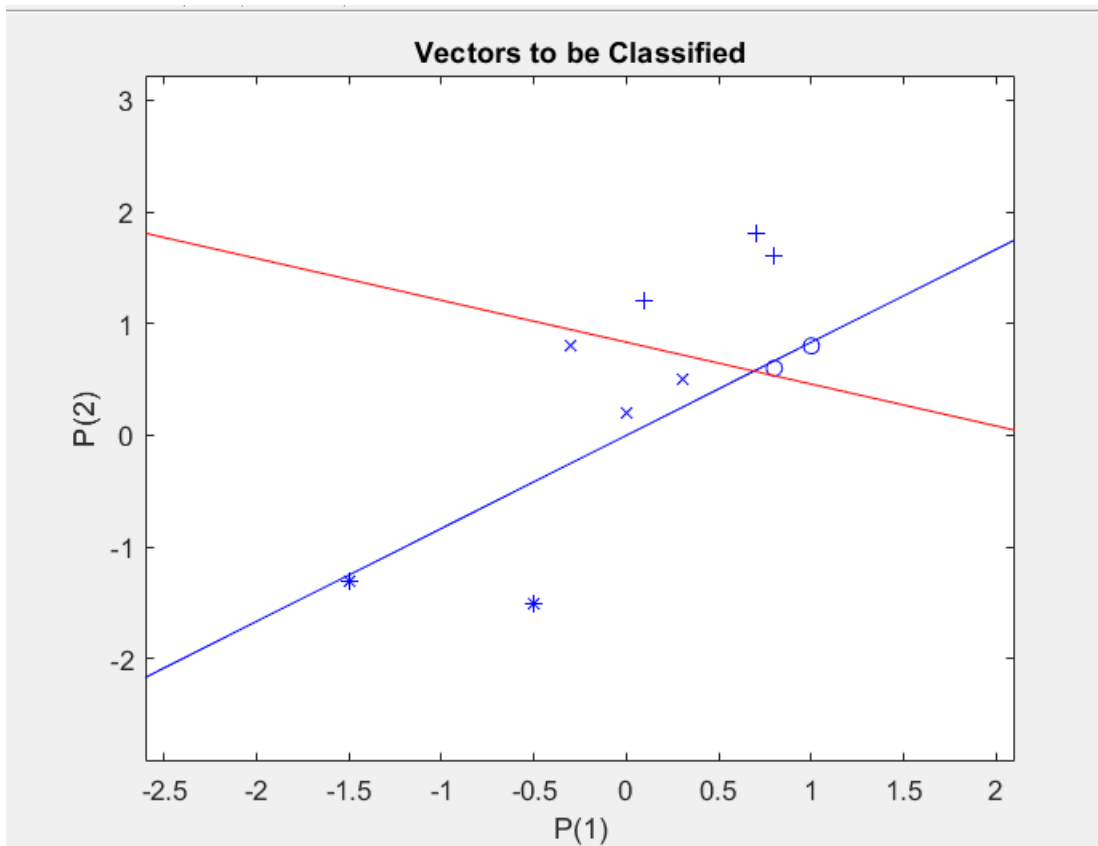
Ejercicio 1. Perceptrón

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

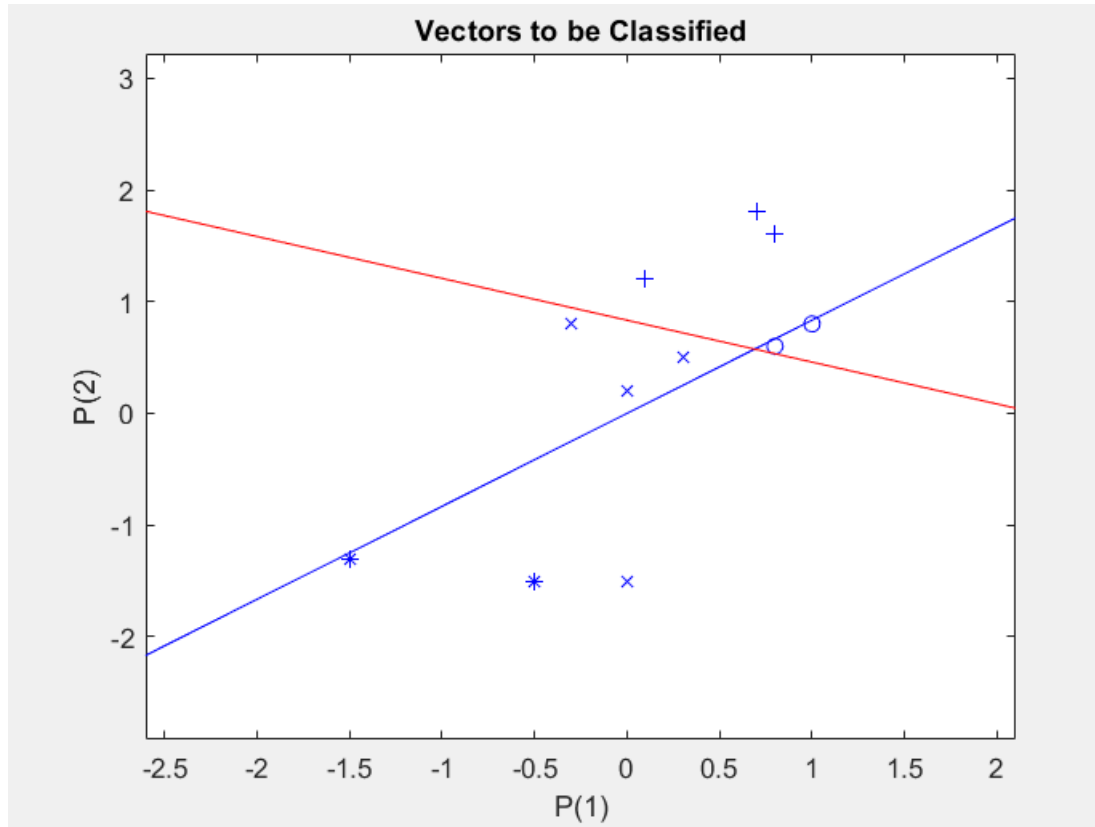
Se desea diseñar un clasificador neuronal mediante un perceptrón simple que clasifique estos datos. Diseñe el clasificador, visualice los parámetros de la red y dibuje los datos junto con las superficies que los separan.

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



La red sí consigue separar los datos correctamente como se puede apreciar en la imagen. La capa de salida tiene que tener 2 neuronas, con una sola como dice el enunciado no es posible clasificar, porque, al tratarse de un perceptrón, los perceptrones sólo pueden clasificar entre 2 valores con su hardlim. Al tener 2 neuronas, ya puede clasificar entre 4 valores.

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



El último dato $[0.0 \ -1.5]$ es clasificado erróneamente porque el perceptrón simple es un modelo lineal que se basa en la creación de una línea recta de separación en el espacio de entrada. Este modelo no puede capturar relaciones de separación no lineales o regiones de decisión complejas.

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 $f = \text{sinc}(t)$ 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.

Antes que nada, el código proporcionado para este ejercicio nos daba un error (Error in ejer2 (line 8) F=sinc(t)+.001*randn(size(t)); % función que se desea aproximar), por lo que lo hemos modificado y realizado el ejercicio con el nuestro.

El error nos daba constantemente en torno a la función sinc(), por lo que finalmente la única solución que hemos podido conseguir es, investigar qué hace dicha función en la ayuda de Matlab.

▼ sinc

La función sinc viene definida por

$$\text{sinc } t = \begin{cases} \frac{\sin \pi t}{\pi t} & t \neq 0, \\ 1 & t = 0. \end{cases}$$

Y a partir de esto, hemos recreado la función nosotros mismos.

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% APROXIMACIÓN DE FUNCIONES
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
clear all; close all;
% DEFINICIÓN DE LOS VECTORES DE ENTRADA-SALIDA
% =====
t = -3:1:3; % eje de tiempo
```

```
F = zeros(size(t));

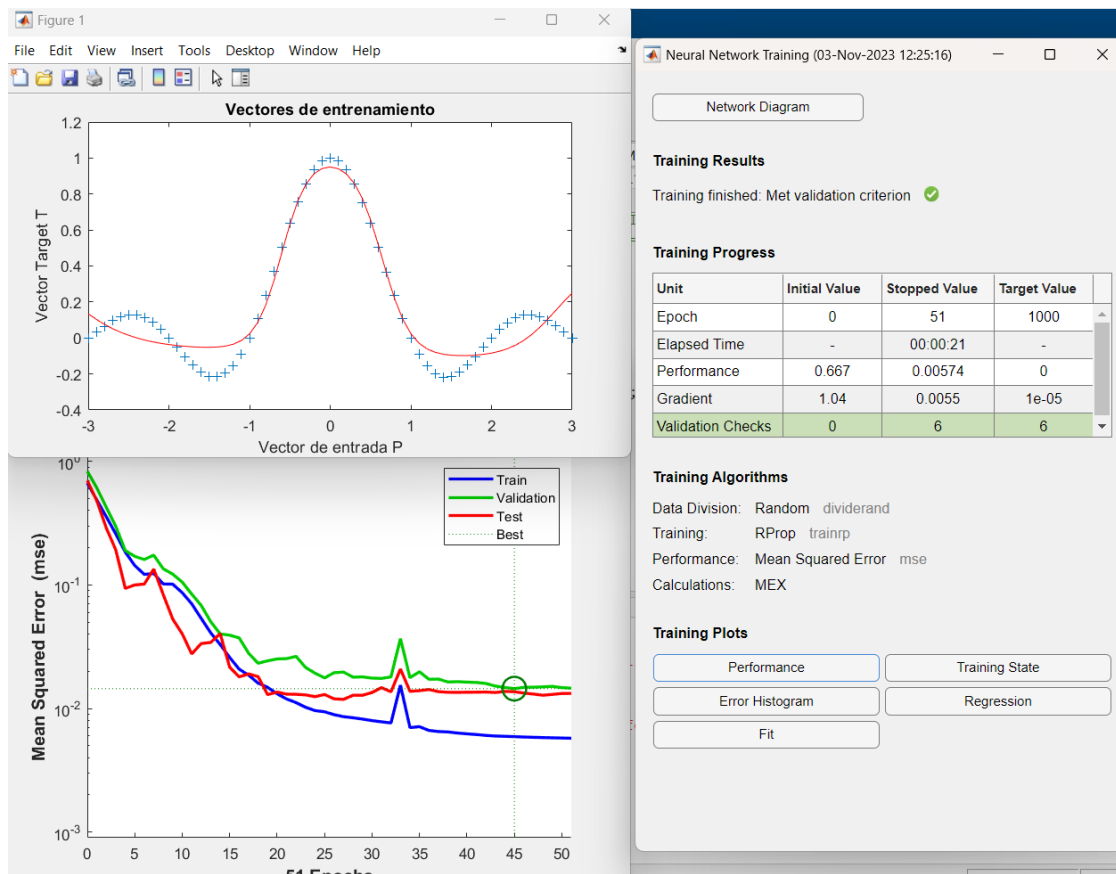
for i = 1:length(t)
    if t(i) == 0
        F(i) = 1;
    else
        F(i) = sin(pi * t(i)) / (pi * t(i));
    end
end

F = F + 0.001 * randn(size(t));

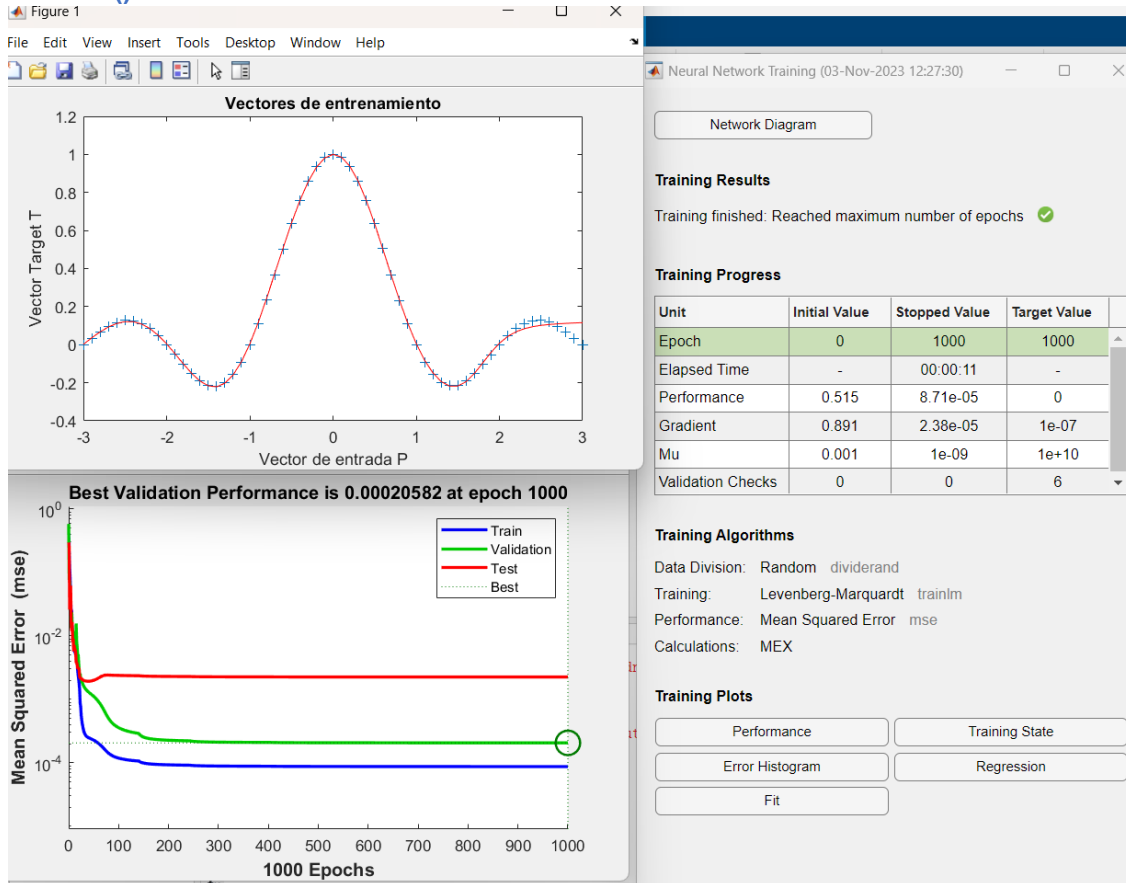
plot(t, F, '+');
title('Vectores de entrenamiento');
xlabel('Vector de entrada P');
ylabel('Vector Target T');

% 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');
```

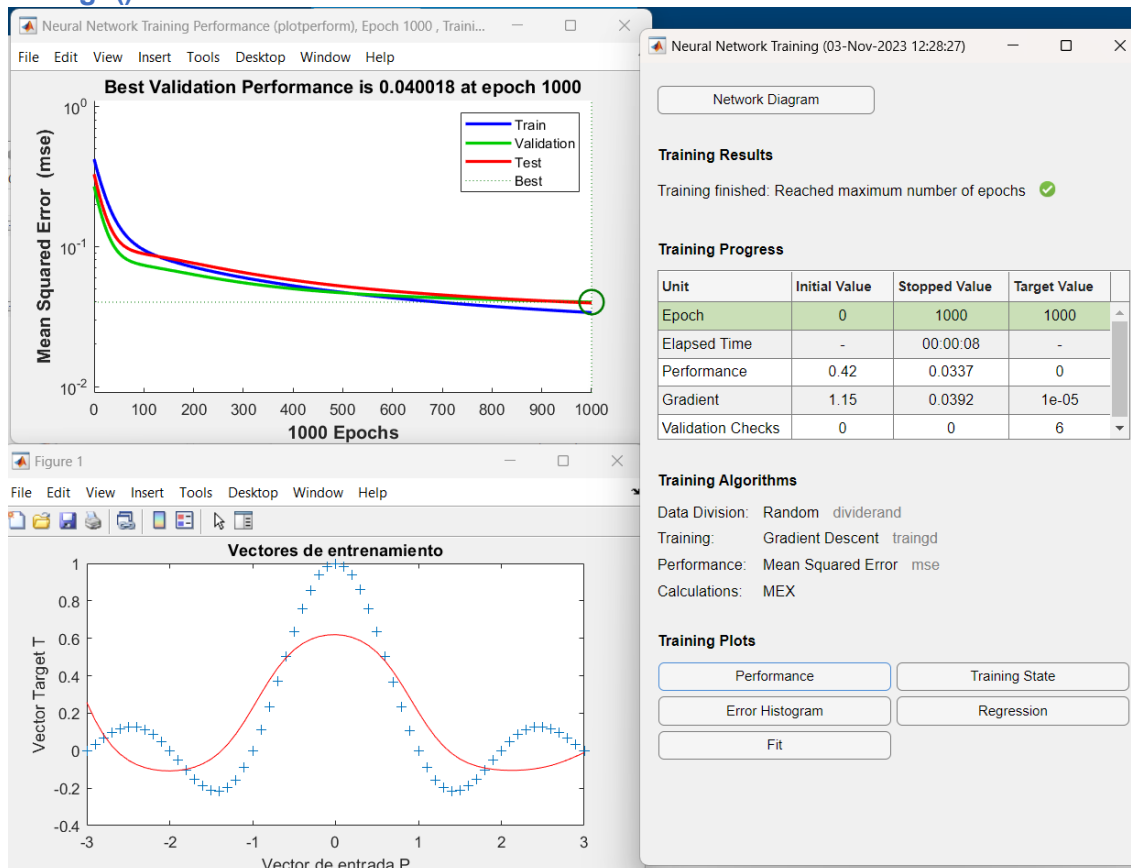
Método:
train()



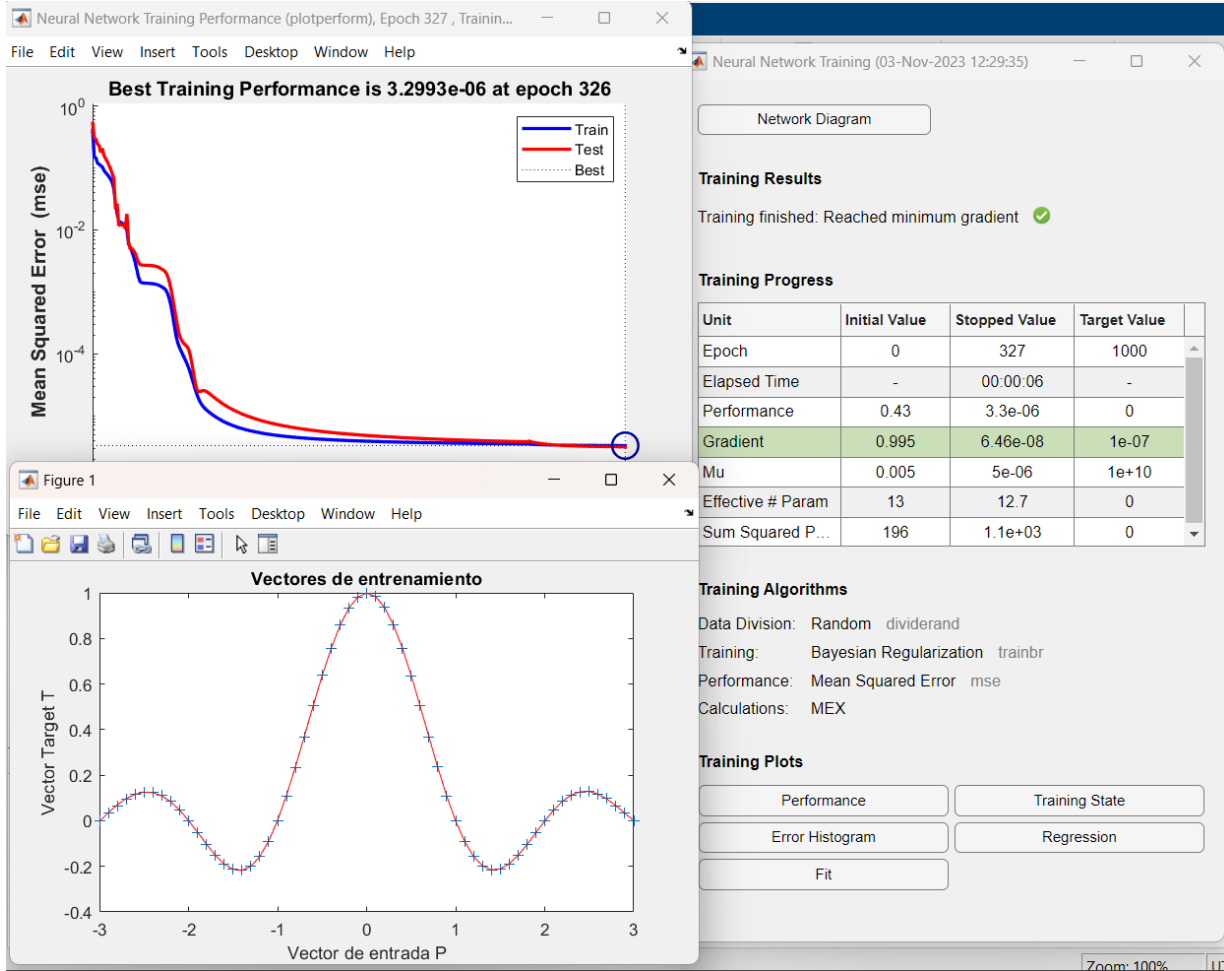
Trainlm()



Traingd()



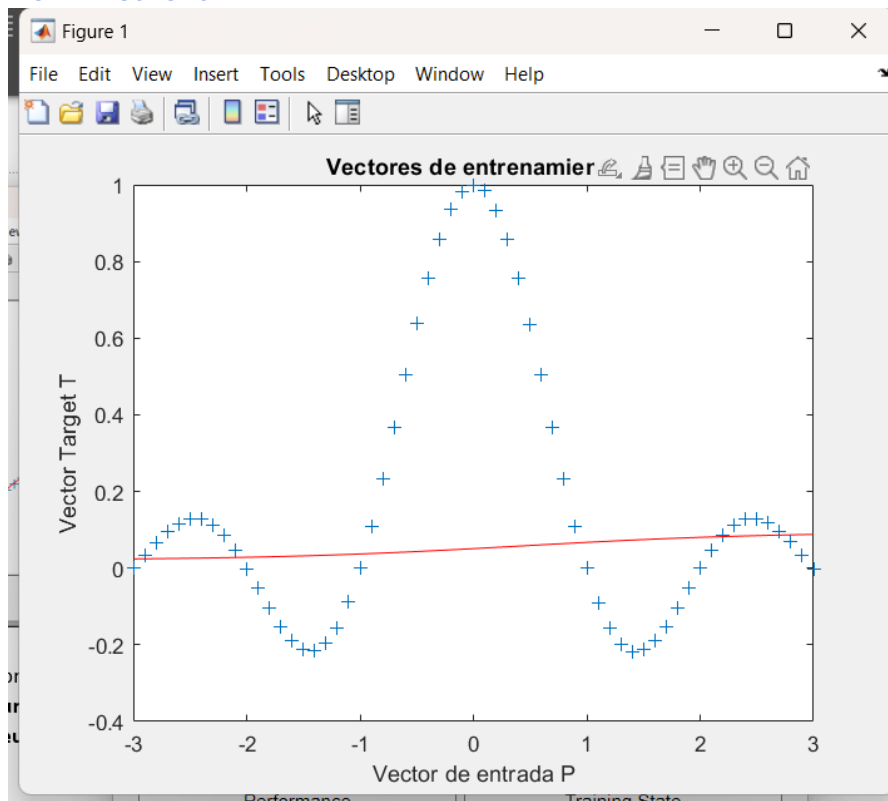
Trainbr()



Hemos podido ver que cada uno de los 4 métodos aproxima la función de una forma distinta y que, el que mayor éxito tiene es este último. Es el que más fielmente aproxima la función.

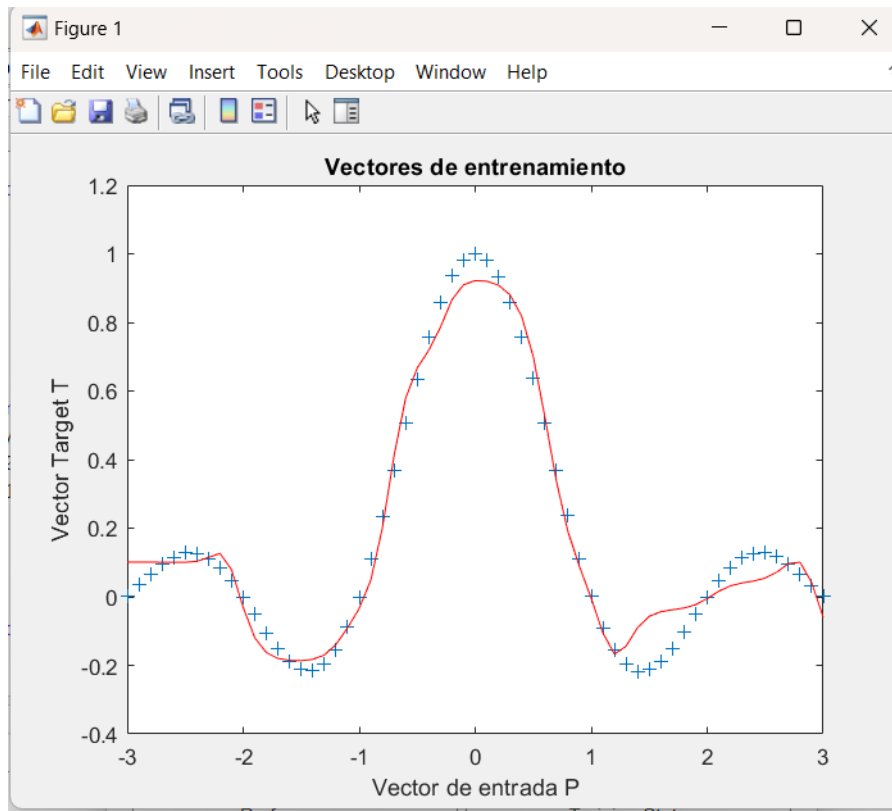
Vamos a probar ahora con la función del principio, `train()`, pero cambiando el número de neuronas.

Con 1 neurona:



Podemos ver que aproxima fatal.

Con 12 neuronas:



En algunas áreas un poco mejor que con 4 neuronas, pero en otras algo peor.

Conclusiones:

El método de entrenamiento y el número de neuronas en la capa oculta son dos aspectos clave para ajustar y optimizar una red neuronal. Experimentar con diferentes configuraciones te permitirá encontrar la combinación que genere la mejor aproximación de la función $f=\text{sinc}(t)$ para tu problema específico.

Cada uno de los 4 métodos probados tiene sus propias características y puede afectar la convergencia y la precisión de la red. Al probar diferentes métodos de entrenamiento, puedes encontrar el que mejor se ajuste a tu problema y obtener resultados más precisos.

Efectos del número de neuronas en la capa oculta:

El código proporciona una capa oculta con 4 neuronas. Modificar el número de neuronas en la capa oculta puede afectar significativamente la capacidad de la red para aproximar la función. Un número menor de neuronas puede llevar a una subaproximación, mientras que un número mayor puede llevar a una sobreaproximación. Debes ajustar el número de neuronas según la complejidad de la función que deseas aproximar.

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;

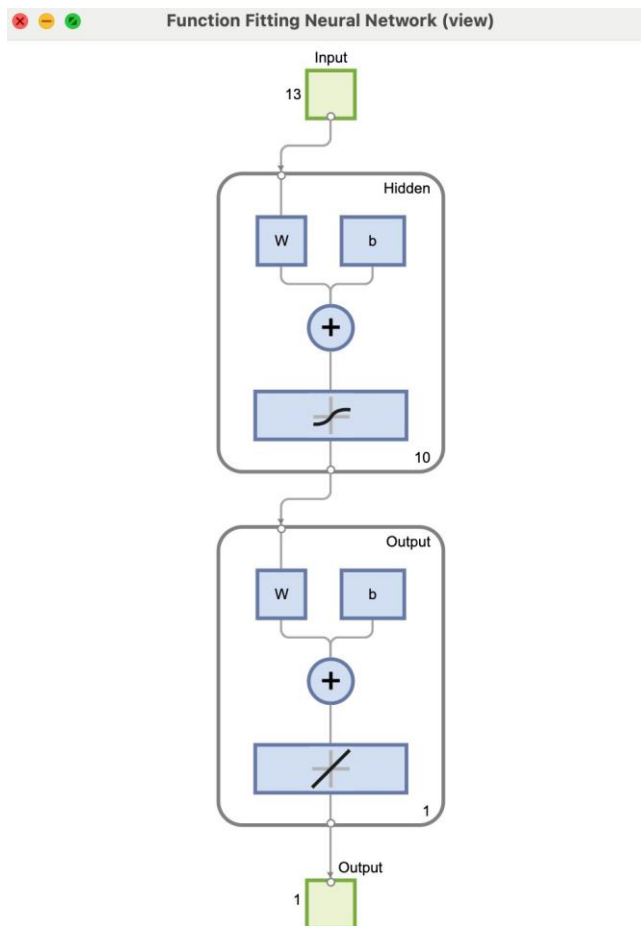
% Creación de la red
hiddenLayerSize = 10;
Net =
fitnet(hiddenLayerSize);

% 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 muestran las siguientes ventanas:



← Esquema de la red

Network Diagram

Training Results

Training finished: Met validation criterion ✓

Training Progress

Unit	Initial Value	Stopped Value	Target Value
Epoch	0	11	1000
Elapsed Time	-	00:00:03	-
Performance	105	9.8	0
Gradient	292	12.5	1e-07
Mu	0.001	0.1	1e+10
Validation Checks	0	6	6

Training Algorithms

Data Division: Random dividerand
Training: Levenberg-Marquardt trainlm
Performance: Mean Squared Error mse
Calculations: MEX

Training Plots

Performance Training State
Error Histogram Regression
Fit

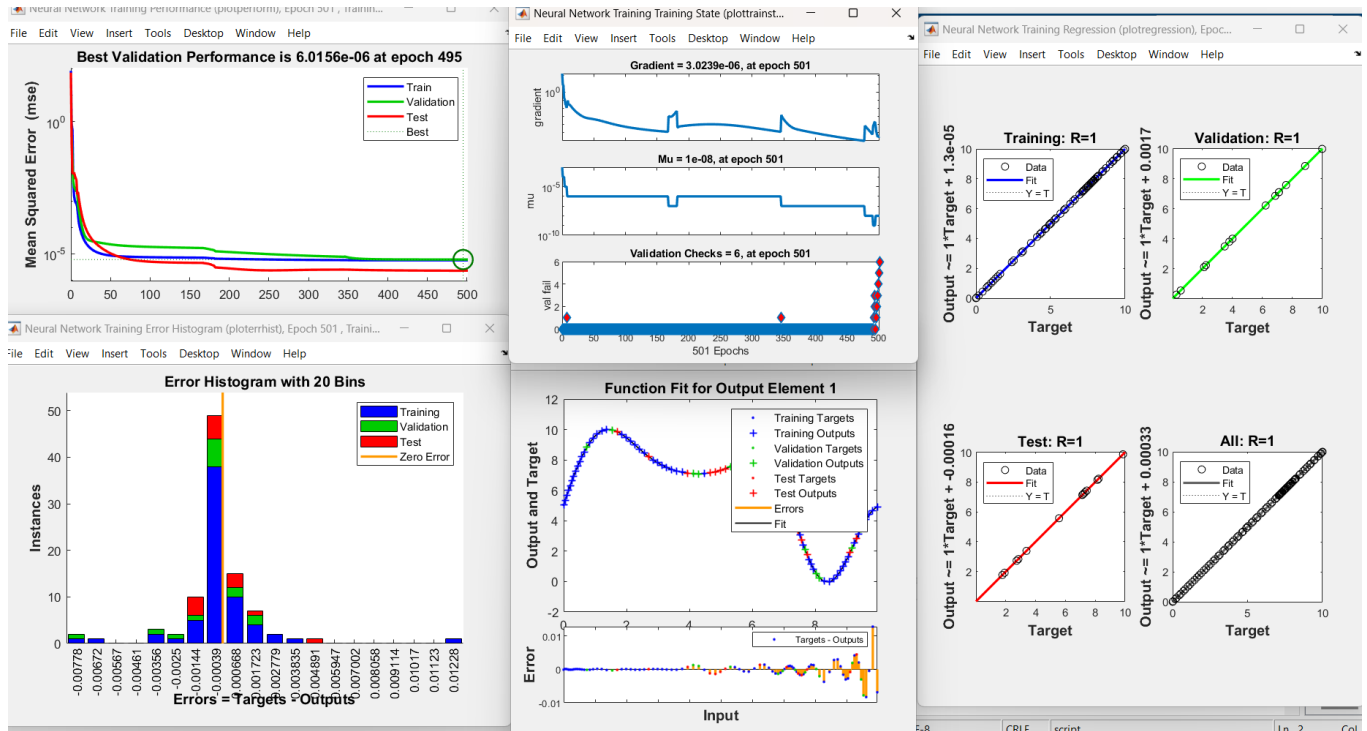
← Progreso del proceso de
entrenamiento y test

← Detalles de los algoritmos

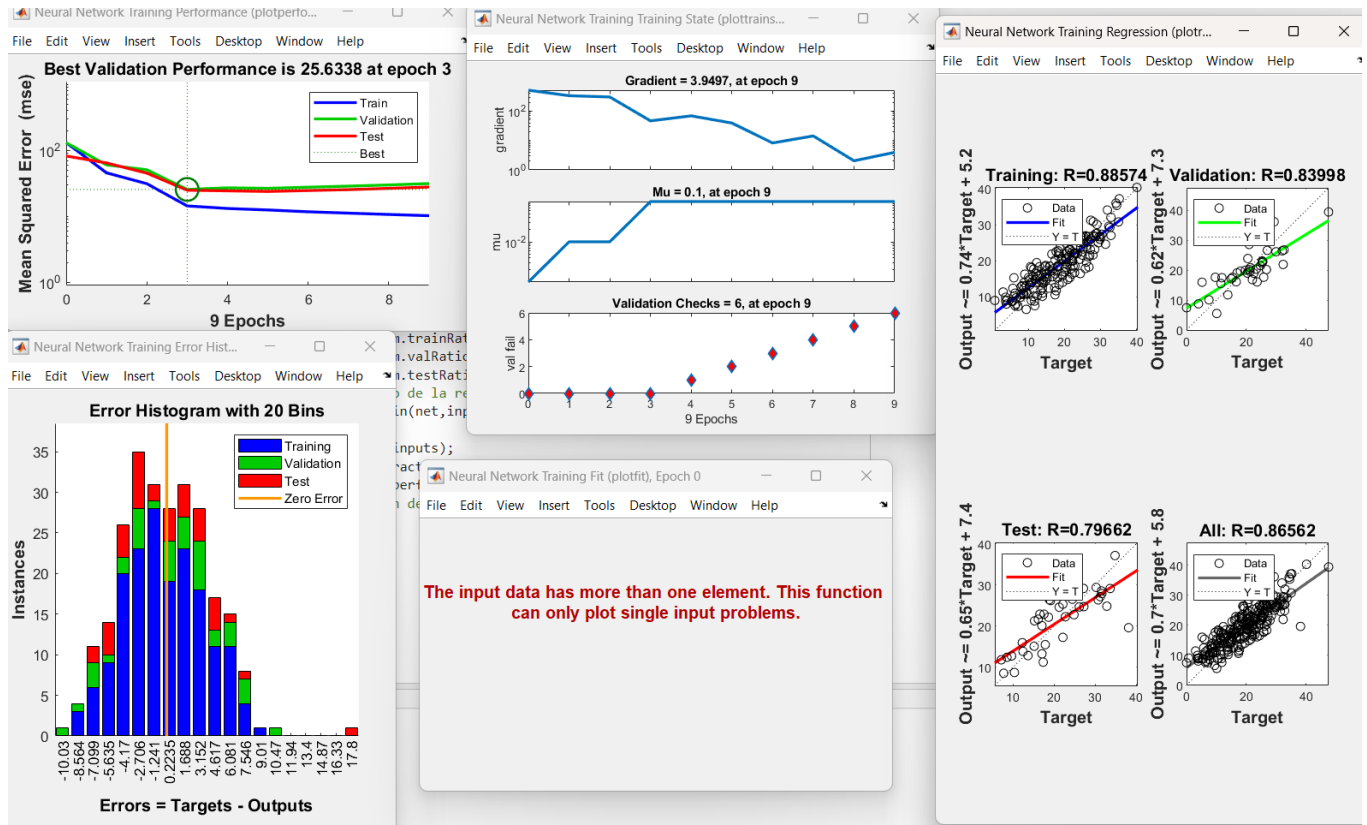
← Gráficas con diversos
resultados

Explore las gráficas disponibles:

- **Performance:** gráfica que representa el error en función del número de épocas para los datos de entrenamiento, validación y test.
- **Training State:** evolución del entrenamiento.
- **Error Histogram:** histograma del error.
- **Regression and Fit:** 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).



Comparando los resultados de las gráficas de `simplefit_dataset` y `bodyfat_dataset`, podemos ver varias cosas.

Comparando las gráficas de regresión, vemos que en el primer caso los resultados, se adaptan perfectamente al objetivo mientras que en el segundo caso tienen cierta desviación. Esto se debe a que el `simplefit_dataset`, es un dataset con datos muy simples y sin outliers.

Sin embargo, si nos fijamos en el training performance, podemos ver que, en el segundo conjunto de datos, ha encontrado el mejor desempeño en el 3, mientras que en el `simplefit_dataset`, para en el epoch 495, tarda mucho más y esto en términos de tiempo, obviamente, si el error es el mismo, es mucho menos óptimo.

Pasemos ahora a comparar los resultados de distintas funciones. Para ello, ejecutaremos cada función 3 veces y haremos una media de sus resultados, para intentar evitar en cierta medida un mal resultado puntual, ya que funciona con datos aleatorios.

Trainlm:

1.4639e-04

2.0935e-04

6.2111e-06

Media: 1.1861e-04

Traingd:

103.0901

116.2749

26.2994

Media: 81.8881

Trainbr:

1.8643e-11

6.7619e-10

4.6991e-12

Media: 2.3118e-10

Trainrp:

2.3649

0.0488

0.0245

Media: 0.8127

La función que mejor resultados aporta, es la que menor performance ofrezca, ya que en esta te saca el error. Comparando los valores obtenidos, podemos ver que, para este conjunto de datos y estos porcentajes de entrenamiento, test y validación, el mejor método de entrenamiento ha sido trainbr.

Pasemos ahora a analizar esta función con una división diferente de sus datos de entrenamiento, test y validación.

La prueba anterior que tan buen resultado nos dio fue con una distribución de 70/15/15.

Probemos con 40/30/30.

De nuevo haremos 3 pruebas y realizaremos la media de sus resultados.

Trainbr con 40/30/30

1.7954e-10

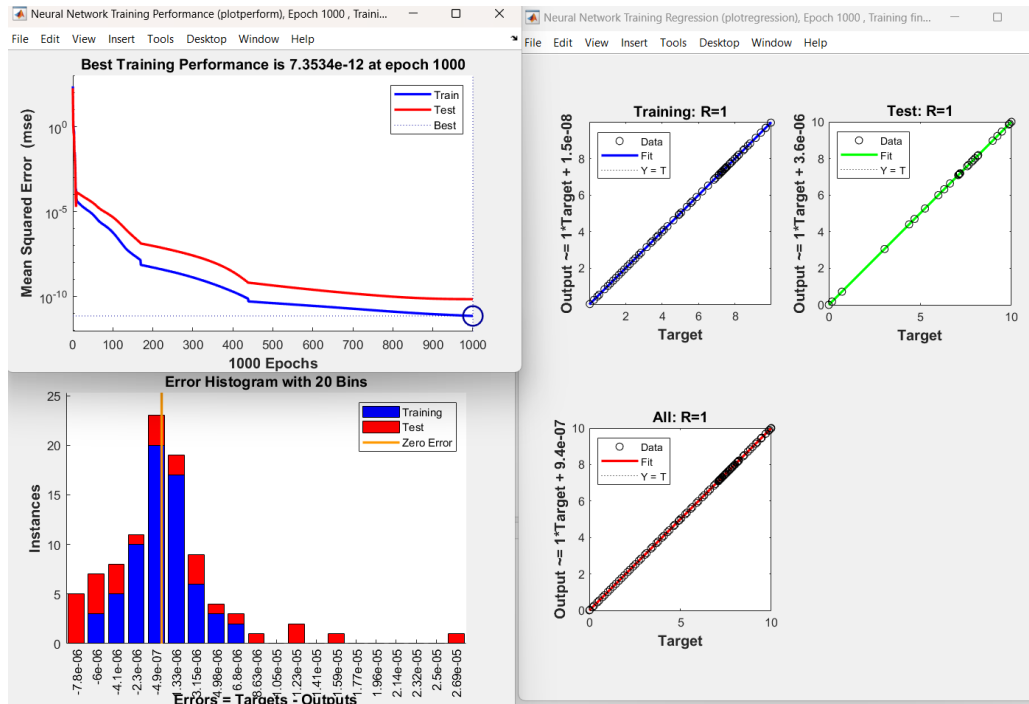
1.1996e-11

2.9994e-11

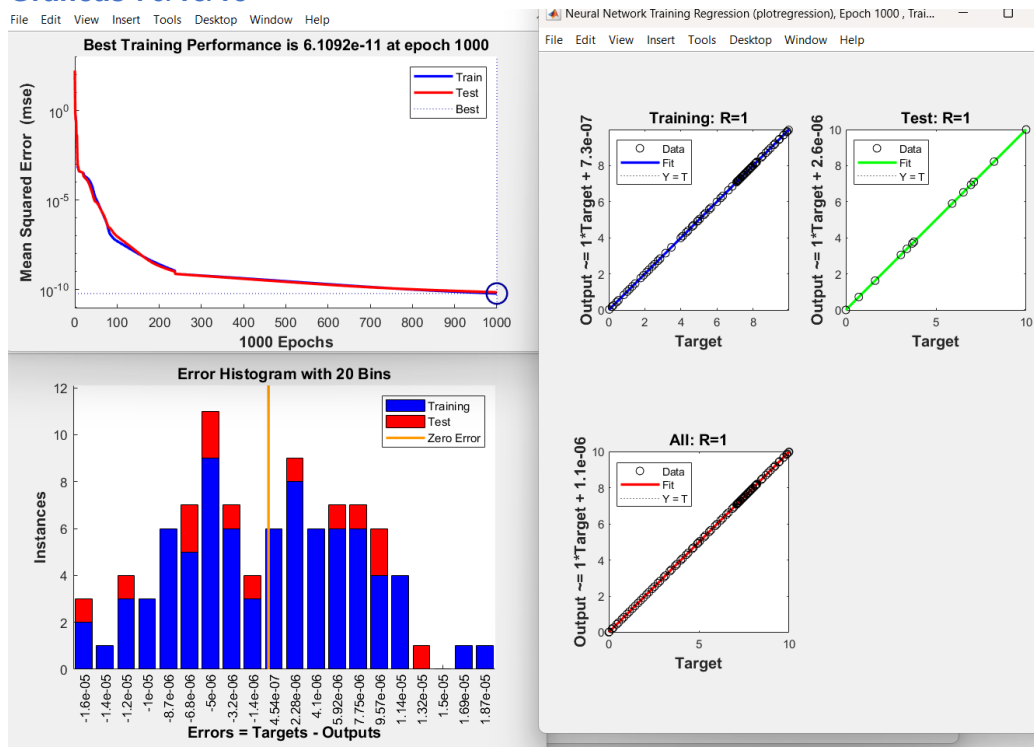
Media: 1.5981e-10

En este caso, podemos ver que el error obtenido ahora (1.5981e-10) es ligeramente menor que el obtenido con la primera distribución (2.3118e-10). Esto puede deberse a las características del conjunto de datos, como la variabilidad, la cantidad de ruido y la calidad de los datos. Un conjunto de datos ruidoso o con alta variabilidad puede beneficiarse de una mayor proporción de validación y prueba para evaluar mejor el rendimiento del modelo. En nuestro caso, como estamos utilizando simplefit_dataset, que es un conjunto de datos muy simples y sin outliers, a la red neuronal le está sirviendo con un menor número de datos para entrenarse.

Gráficas 40/30/30



Gráficas 70/15/15



Comparando las gráficas, podemos ver fácilmente que la de regresión es idéntica.

En cuanto a la de training performance, vemos que las dos paran en el límite establecido de 1000 epochs pero, que con la segunda distribución (70/15/15), los datos de entrenamiento y test son mucho más parejos.

Al observar el histograma, puedes identificar si hay problemas específicos con la calidad del modelo. Por ejemplo, si ves que la mayoría de los errores se concentran en un rango particular, podría indicar que el modelo tiene dificultades para ajustarse a esa región de los datos. Esto es exactamente lo que pasa en el caso de 40/30/30.

Algo que nos ha llamado poderosamente la atención es que, a pesar de que una porción de los datos está dedicada a la validación, esto no se refleja en algunas gráficas, cuando en otros ejemplos anteriores sí pasaba.

Tras investigar un poco, hemos descubierto que algunos algoritmos de entrenamiento pueden no mostrar explícitamente métricas de validación durante el entrenamiento.

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;
```

% Creación 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);
```

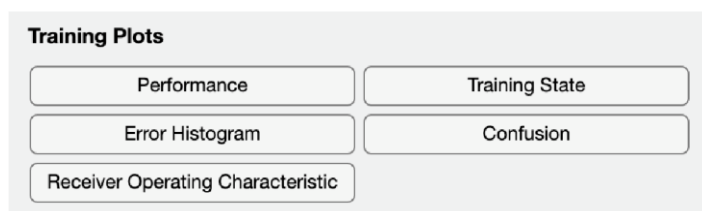
```
errors = gsubtract(targets,outputs);
```

```
performance = perform(net,targets,outputs)
```

% Visualización

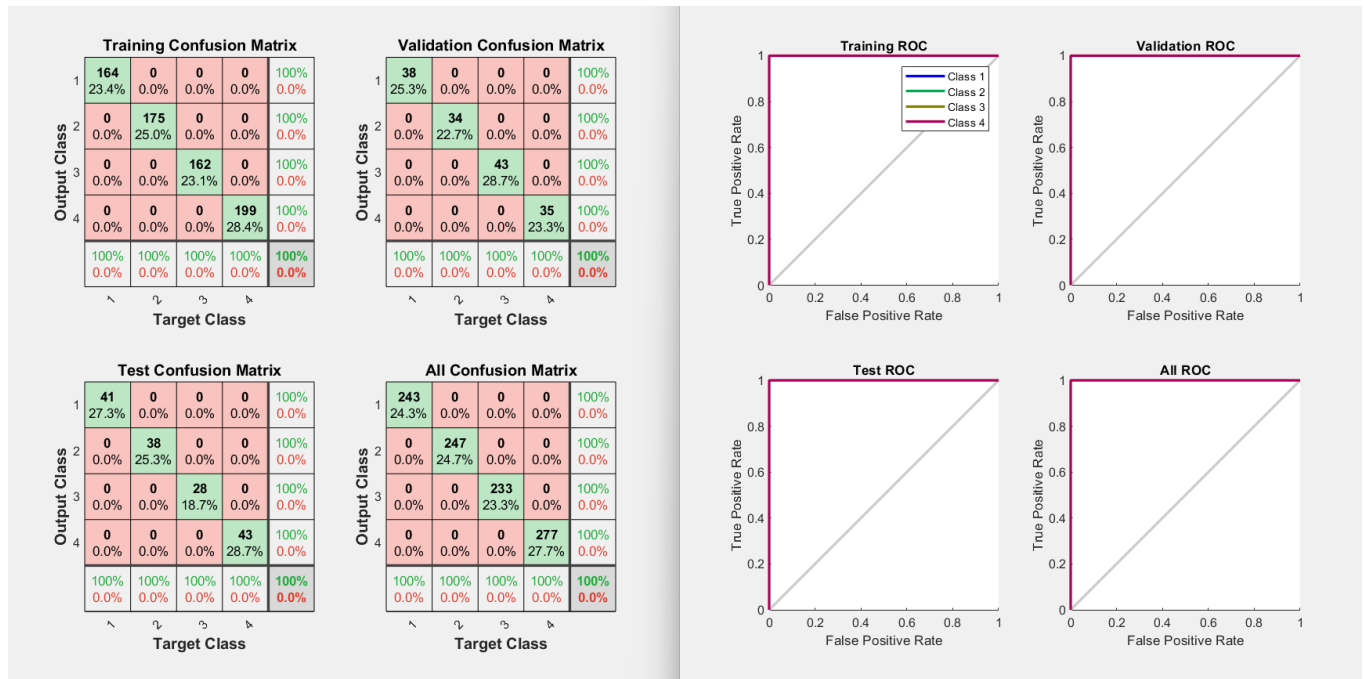
```
view(net)
```

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:

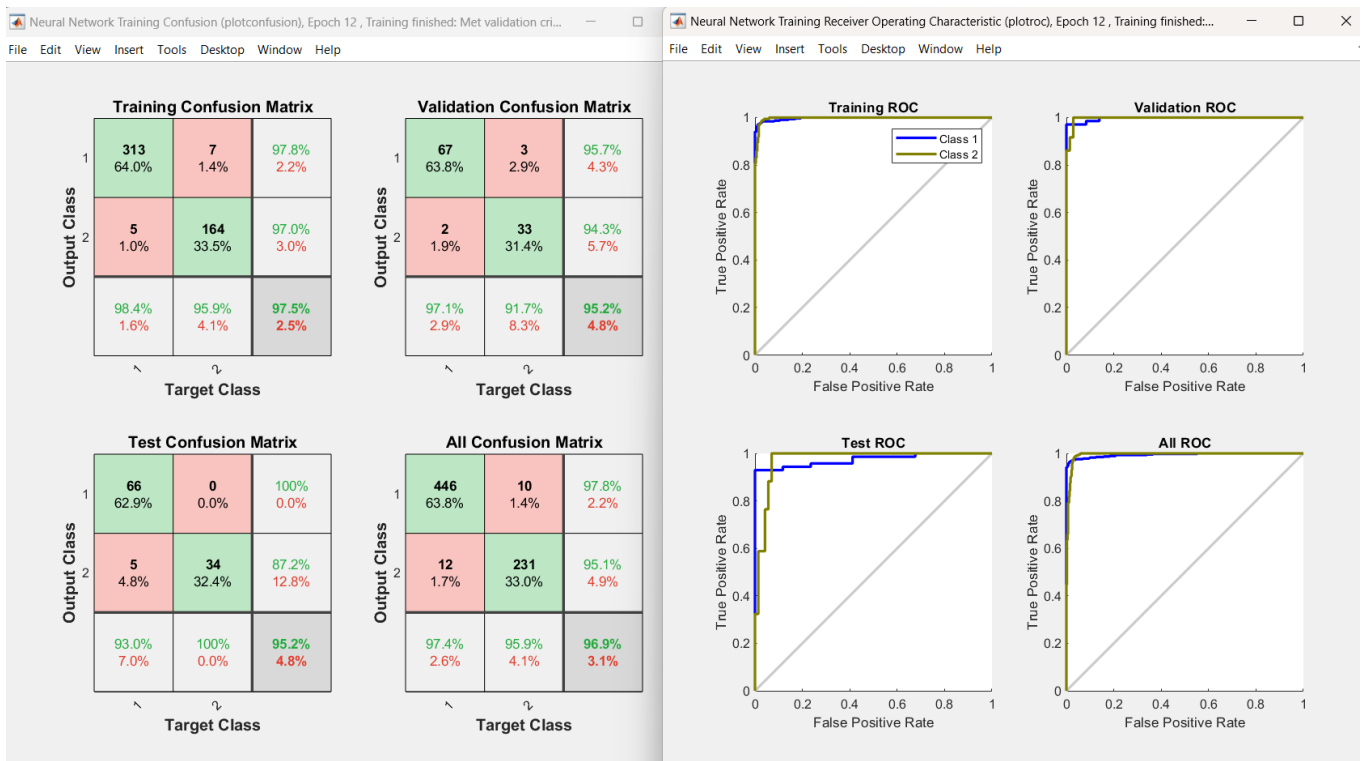


En lugar de las gráficas específicamente relacionadas con la aproximación de una función, en el caso de una tarea de clasificación, se ofrecen:

- **Confusion:** matrices de confusión de los resultados.
- **Receiver Operating Characteristic:** 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).



En el primero de los dos casos, los resultados son perfectos. En la matriz de confusión podemos ver que clasifica correctamente el 100% de los casos y, en las gráficas training ROC, también podemos observar la perfección viendo que las 4 clases se amoldan perfectamente a la esquina superior izquierda.

En la gráfica ROC, un modelo perfecto que distingue perfectamente entre las clases tendría una curva ROC que se eleva verticalmente hacia (0,1) y luego se extiende horizontalmente a lo largo del eje y (100% de Sensibilidad y 0% de Falsos Positivos). Cuanto más se acerque la curva ROC de un modelo a esta esquina superior izquierda, mejor será su rendimiento.

Con el dataset de cancer, sin embargo, podemos ver que a pesar de seguir teniendo un ratio bastante alto de aciertos, baja de la perfección anterior a un 95%, lo que queda reflejado en las dos gráficas.

Pasemos ahora a probar con distintos modelos para comprobar su desempeño. De nuevo haremos 3 pruebas con cada modelo y sacaremos la media del error obtenido.

Trainlm:

0.0296

0.0128

0.0277

Media: 0.0234

Traingd:

0.0797

0.0844

0.0595

Media: 0.0745

Trainbr:

0.0034

0.0042

0.0084

Media: 0.0053

Trainrp:

0.0424

0.0441

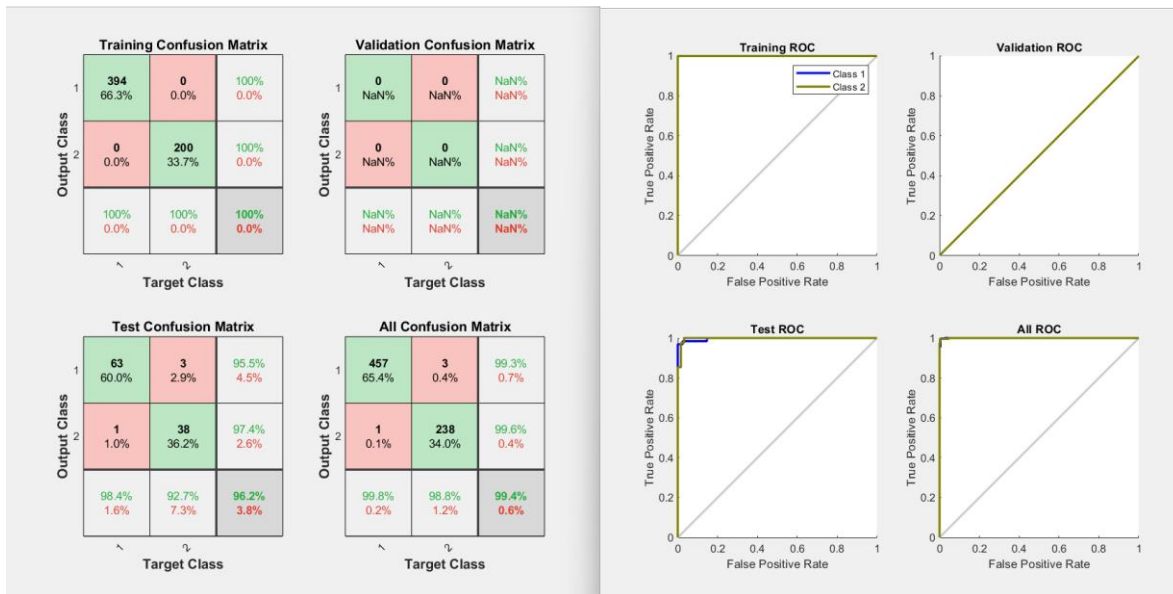
0.0553

Media: 0.0473

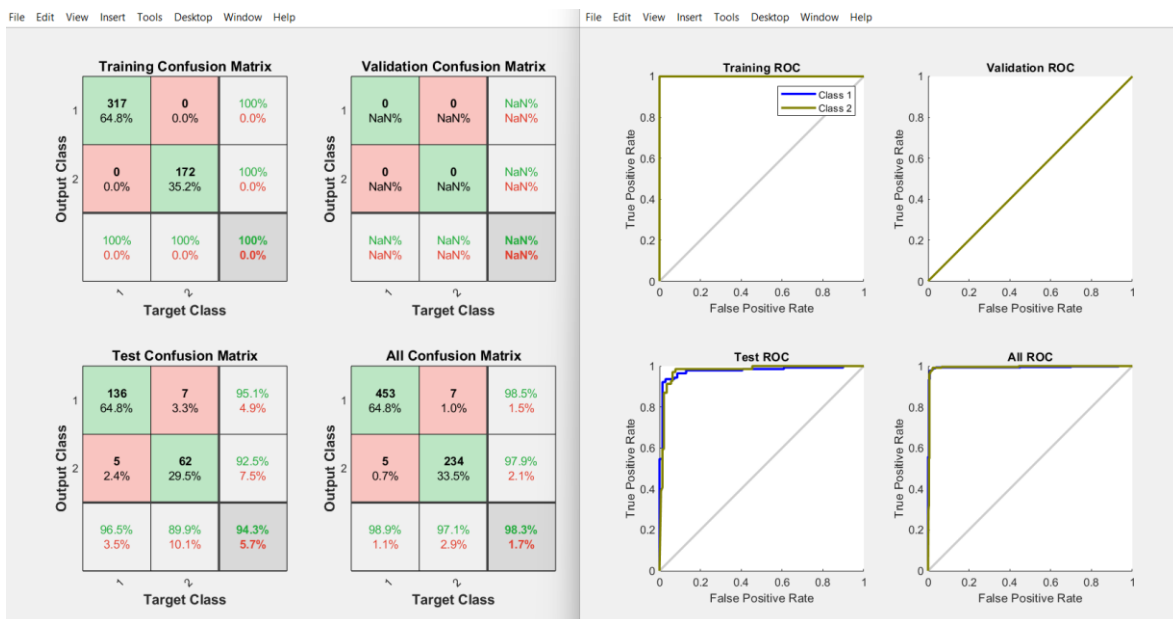
De nuevo, obtenemos un mejor resultado con el modelo trainbr.

Comparemos ahora, con este modelo, entre la distribución actual de datos y, otra distinta.

Gráficas 70/15/15



Gráficas 40/30/30



Lo primero en lo que nos hemos fijado es, que como antes, el modelo trainbr, no muestra los datos de validación en ninguna de estas dos gráficas tampoco.

En el caso de estas dos gráficas, están muy relacionadas y básicamente podemos ver los mismos resultados en las dos.

En ambos casos, el training es perfecto, lo que podemos ver en cualquiera de las dos gráficas, mientras que, a la hora del test, la primera distribución mejora la efectividad de la segunda en un 2%. Esto también se puede apreciar en que las líneas azules están más pegadas a la esquina superior izquierda en la gráfica training ROC, en el caso de 70/15/15. Por lo que esta vez, nos ofrece un mejor desempeño.



Universidad
de Alcalá

Práctica 1. Identificación y control neuronal (II)

Sistemas de Control Inteligente

Grado en Ingeniería Computadores
Grado en Ingeniería Informática Grado
en Sistemas de Información

Universidad de Alcalá

Diseño de un control de posición mediante una red neuronal no recursiva.

En esta práctica, se estudiará el diseño de controladores neuronales para el control de posición y seguimiento de trayectorias de robots móviles. Se hará uso del entorno Matlab para el entrenamiento de las redes neuronales y el entorno Simulink para la simulación del robot móvil y los controladores propuestos. Afrontaremos en esta sesión el diseño de un control de posición mediante una red neuronal no recursiva para abordar en la siguiente el diseño de un control para seguimiento de trayectorias mediante redes recurrentes.

1. Objetivo y descripción del sistema.

Para la correcta realización de la práctica, se necesita crear el entorno de Simulink correcto, para ello vamos a copiar el robot realizado en la práctica anterior y vamos a modificarlo para que quede de la siguiente manera:

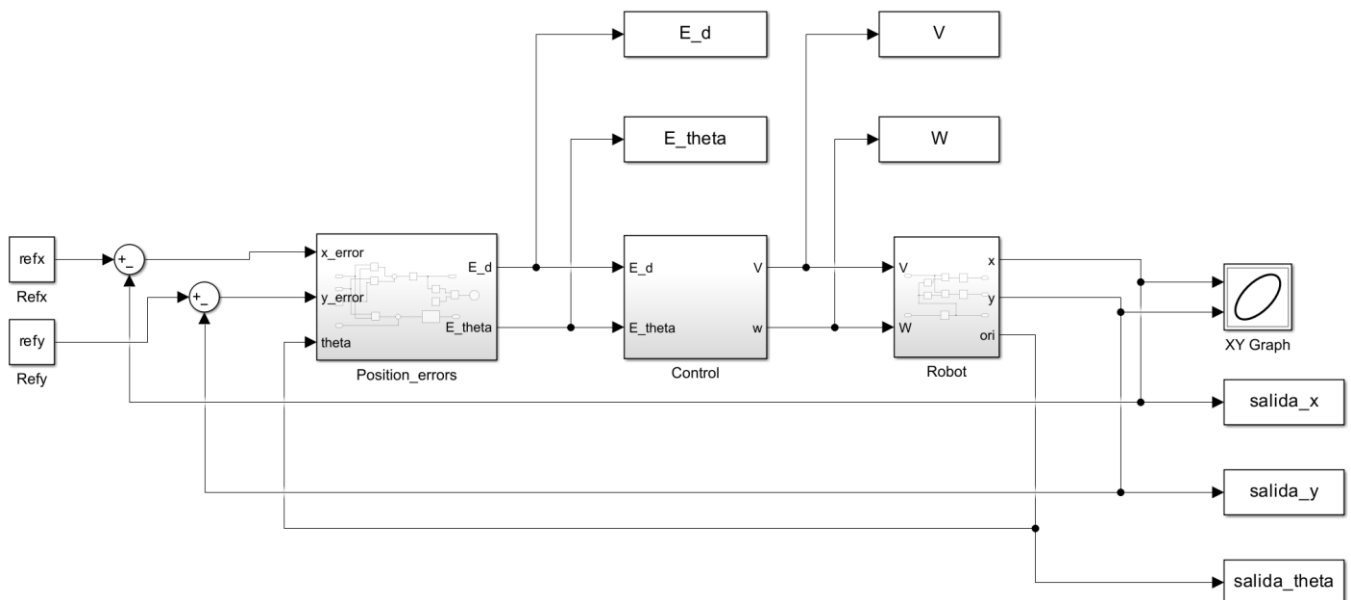


Figura 1. PositionControl.slx

Este entorno ha sido modificado añadiendo el subsistema “control”, el cual se puede descargar e blackboard, los bloques “To Workspace” y el subsistema “Position_errors” el cual es de la siguiente manera:

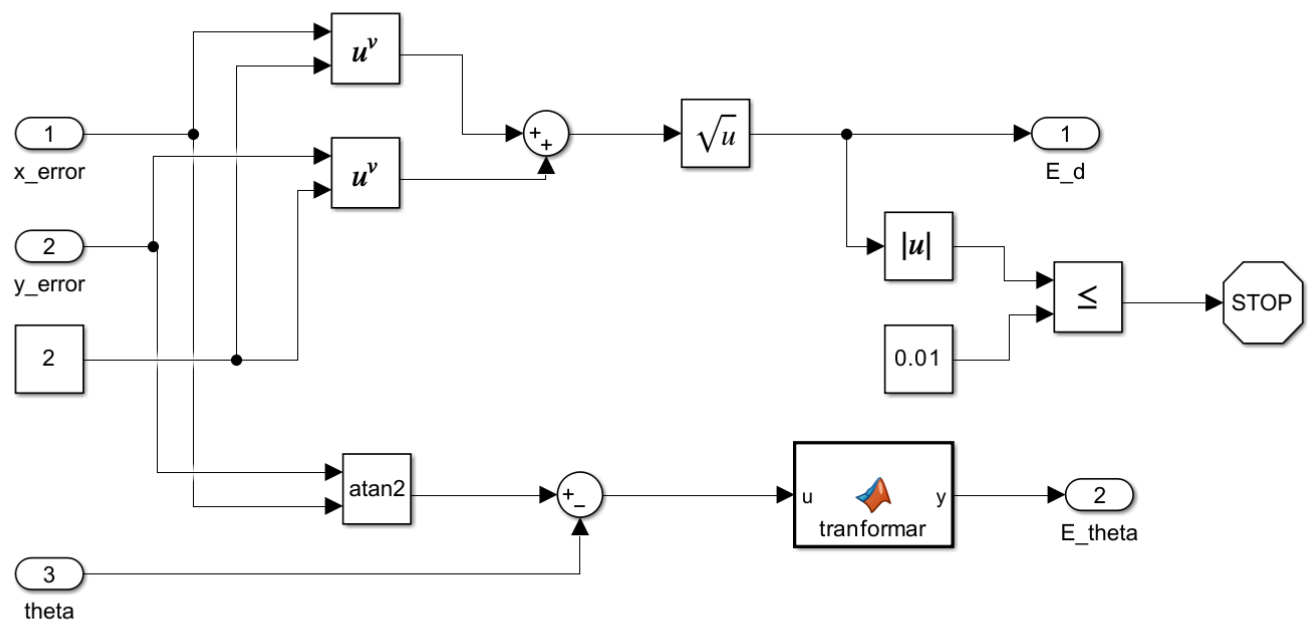


Figura 2. Subsistema "Position_error"

Se puede observar una comparación para ver si el resultado de $E_d = \sqrt{(\text{ref}_x - x_k)^2 + (\text{ref}_y - y_k)^2}$ es mayor de 0.01, en cuyo caso pararía la ejecución. Por otro lado, podemos observar una función con el siguiente código:

```
function y = transformar(u)
y = wrapTo180(u);
```

El cual asegura que el resultado de $E_{\text{theta}} = \text{atan2}(\text{ref}_y - y_k, \text{ref}_x - x_k) - \text{theta}_k$ este entre $[-\pi, \pi]$.

Tras esto y las especificaciones indicadas en el enunciado de la práctica esta todo preparado para el desarrollo de esta.

2. Desarrollo de la práctica

Como se indicó al comienzo del documento, el objetivo de esta primera parte de la práctica es diseñar un controlador basado en una red neuronal que emule el comportamiento del controlador caja negra ("controlblackbox.slx"). Para ello se piden los siguientes apartados:

A,b,c,d,e) A continuación, se ha creado un script en Matlab llamado "RunPositionControl" el cual simulará el entorno simulink creado anteriormente con el siguiente código:

```
%Tiempo de muestreo
Ts = 100e-3;
% Referencia x-y de posicion
refx=2.0;
```

```

refy=2.0;
% Ejecutar Simulacion
sim('PositionControl.slx')

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

```

De esta manera, se inicializan los valores “Time stop” (Ts) y la referencia de la posición X e Y (refx y refy). Al ejecutar podemos observar 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 simulación (salida_x, salida_y, salida_theta).

Name ^	Value
E_d	1x1 struct
E_theta	1x1 struct
refx	2
refy	2
salida_theta	1x1 struct
salida_x	1x1 struct
salida_y	1x1 struct
tout	[0;0.1000]
Ts	0.1000
V	1x1 struct
W	1x1 struct
x	[0;0.2000]
y	[0;0]

Figura 3. Captura de pantalla de las variables de “PositionControl”

Por otro lado, podemos observar la trayectoria del robot gracias a la función plot. Esta trayectoria se vería de la siguiente manera:

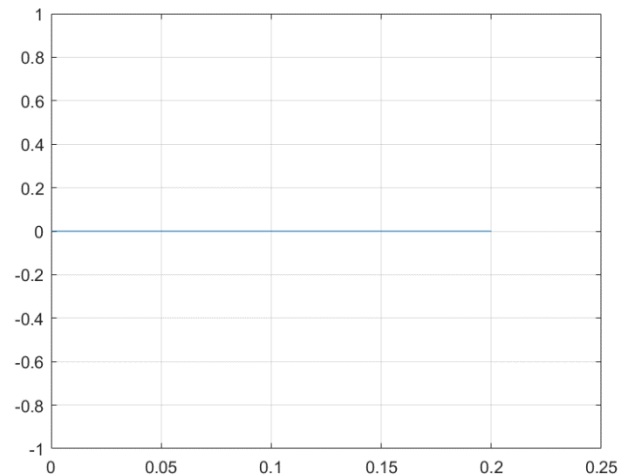


Figura 4. Trayectoria del robot ($T_s = 0.1$)

Se ve de esta manera debido a que el stop time es de tan solo 0.1 segundos, si dejamos que se termine la ejecución se ve de la siguiente manera:

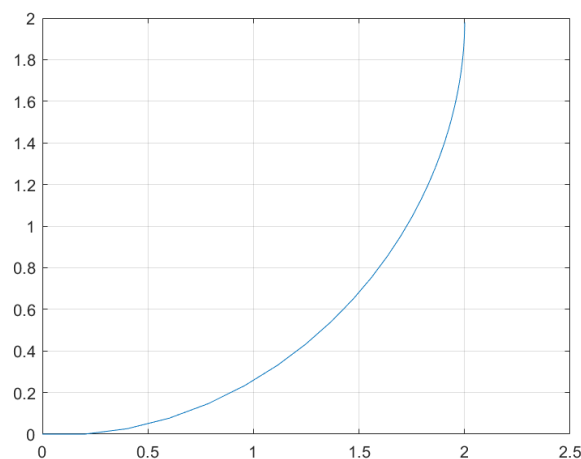


Figura 5. Trayectoria del robot ($T_s = 5$)

Como se puede observar, el recorrido final acaba en las posiciones inicializadas al principio del código.

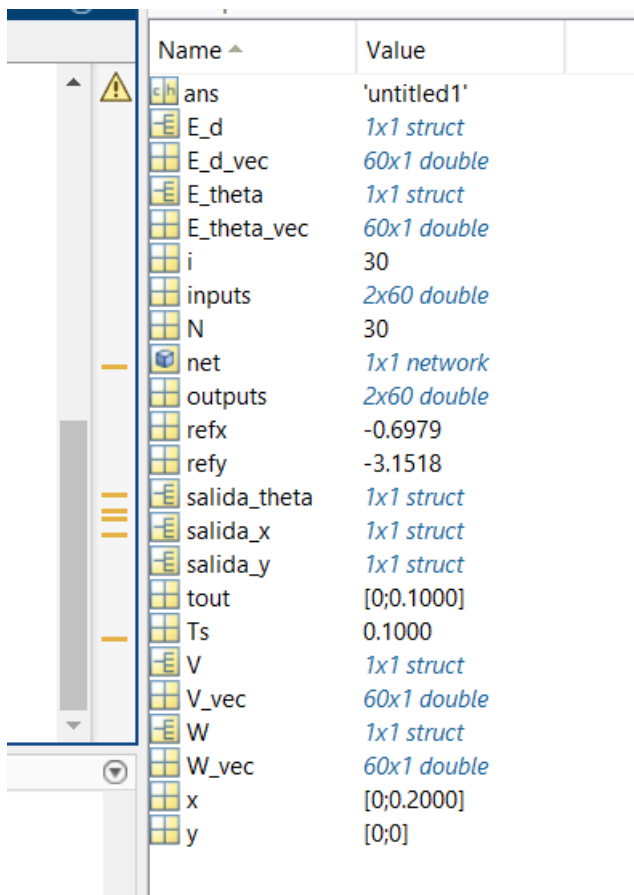
- e) Realice $N=30$ simulaciones del controlador proporcionado mediante un bucle donde se varían los valores de *refx* y *refy* de manera aleatoria dentro del entorno de 10x10 metros. En cada simulación se deberá guardar el valor a lo largo del tiempo de las entradas (E_d y E_{theta}) y salidas (V y W) del bloque controlador. Genere la matriz “*inputs*” de tamaño $2 \times N$, 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=30
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];
    disp(i);
end
inputs=[E_d_vec'; E_theta_vec'];
outputs=[V_vec'; W_vec'];

```



Name	Value
ans	'untitled1'
E_d	1x1 struct
E_d_vec	60x1 double
E_theta	1x1 struct
E_theta_vec	60x1 double
i	30
inputs	2x60 double
N	30
net	1x1 network
outputs	2x60 double
refx	-0.6979
refy	-3.1518
salida_theta	1x1 struct
salida_x	1x1 struct
salida_y	1x1 struct
tout	[0;0.1000]
Ts	0.1000
V	1x1 struct
V_vec	60x1 double
W	1x1 struct
W_vec	60x1 double
x	[0;0.2000]
y	[0;0]

- f) 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*.

```

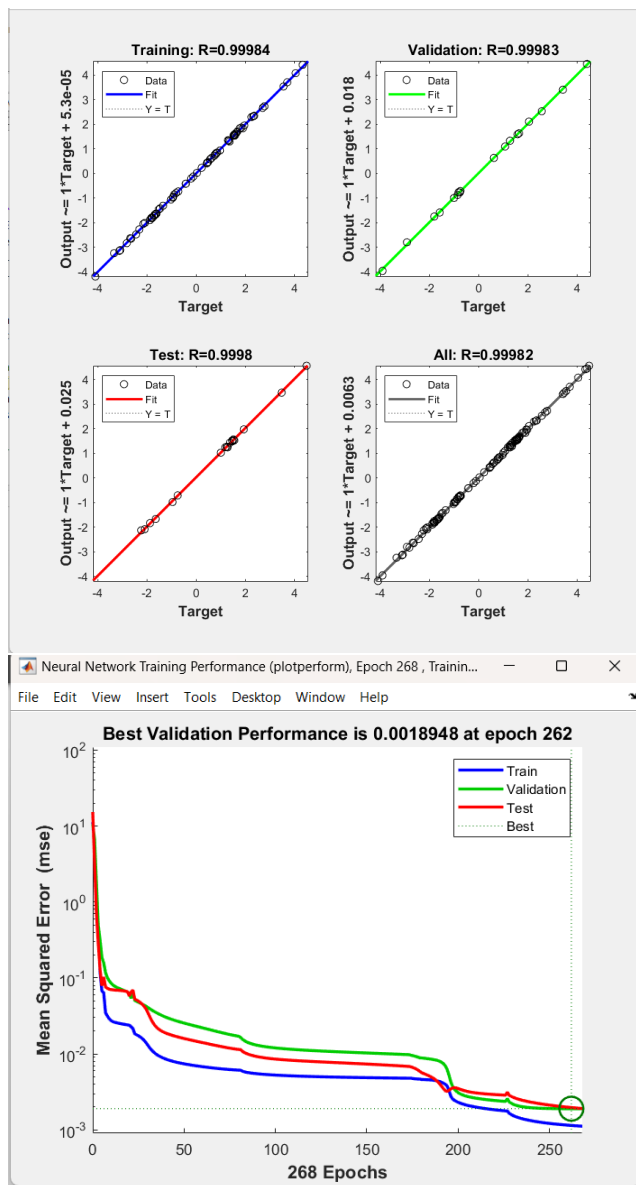
% Entrenar red neuronal con 10 neuronas en la capa oculta
net = feedforwardnet([10]);
net = configure(net,inputs,outputs);
net = train(net,inputs,outputs);

```

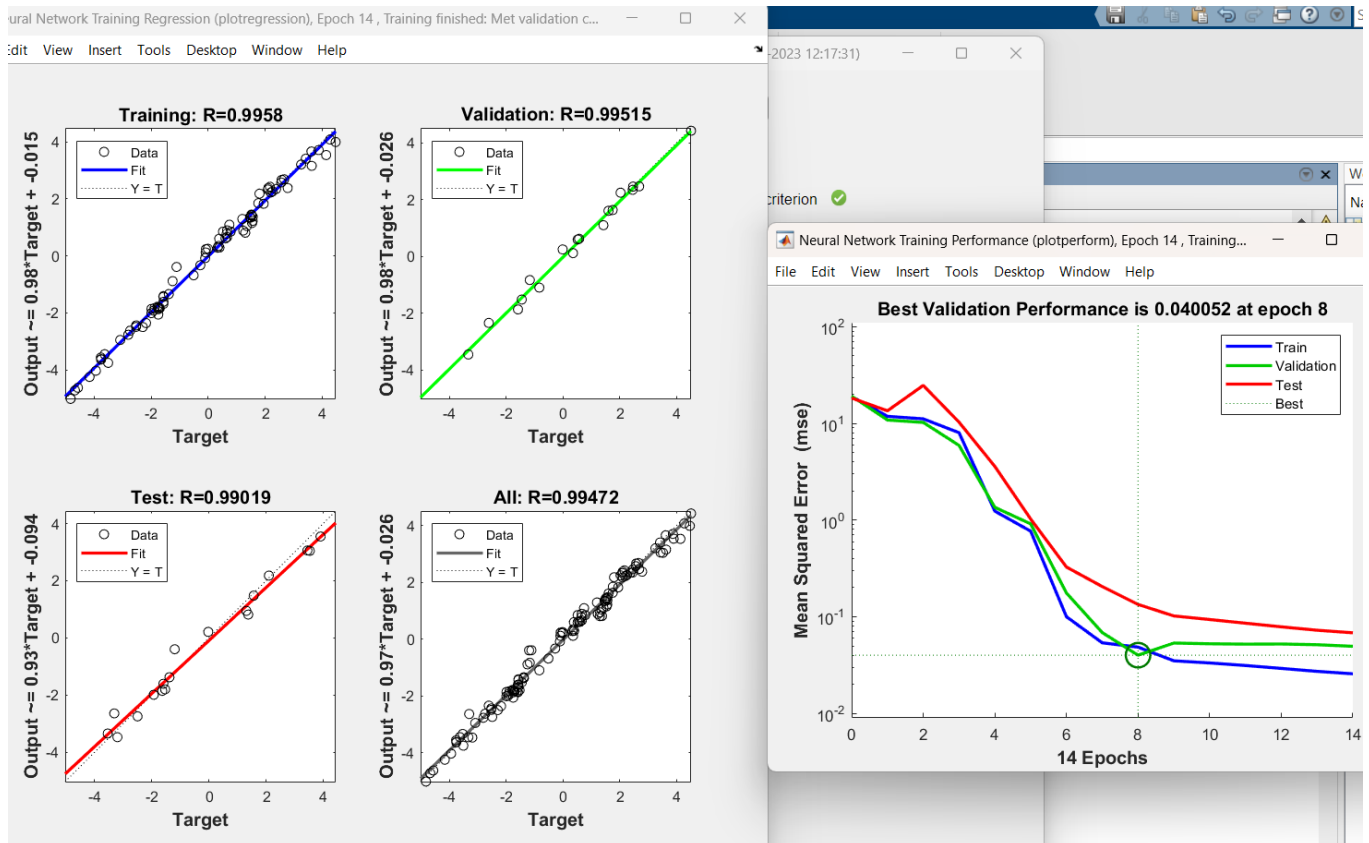
Discuta los resultados del entrenamiento y observe el comportamiento de la red en los subconjuntos de entrenamiento, test y validación.

El método para conseguir el mejor número de neuronas en la capa oculta no es otro que probando. Vamos a probar por tanto con varios números de neuronas y luego veremos cual de ellos nos da un mejor resultado.

Con 5 neuronas:

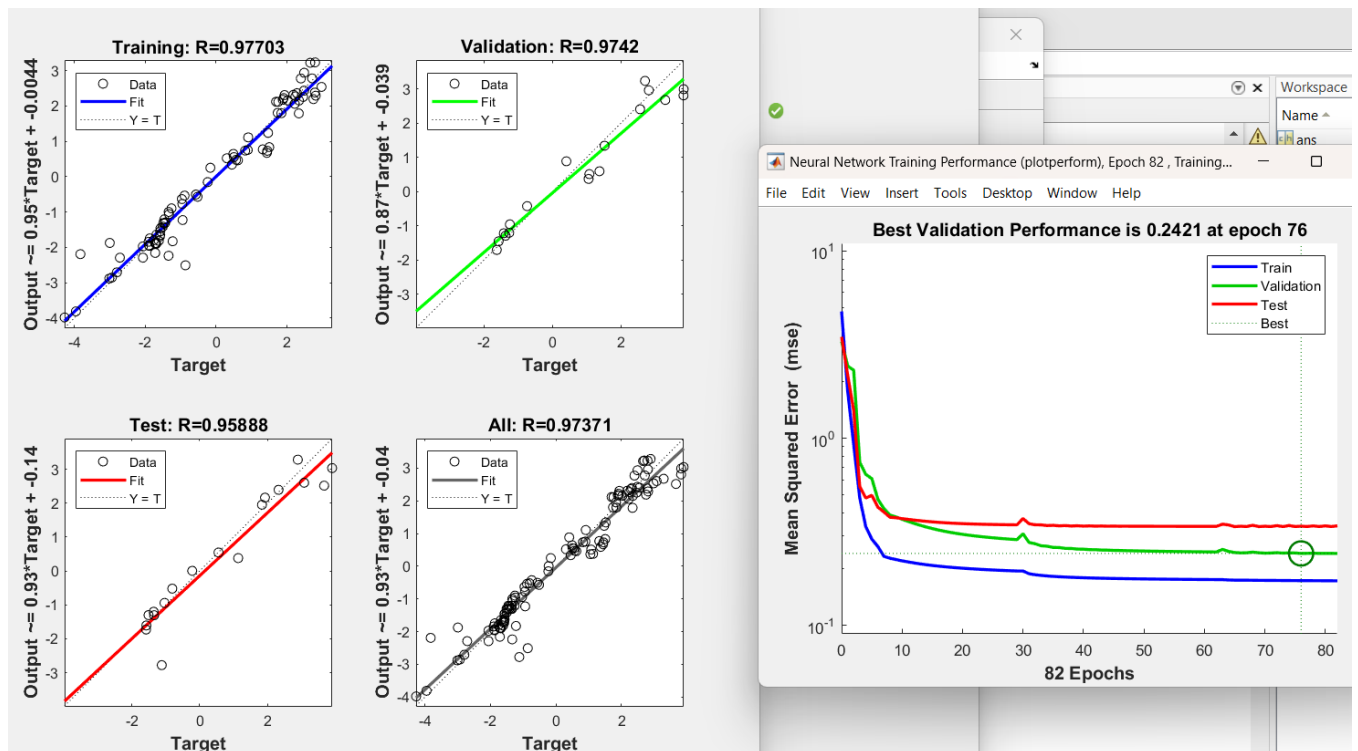


Con 10 neuronas:

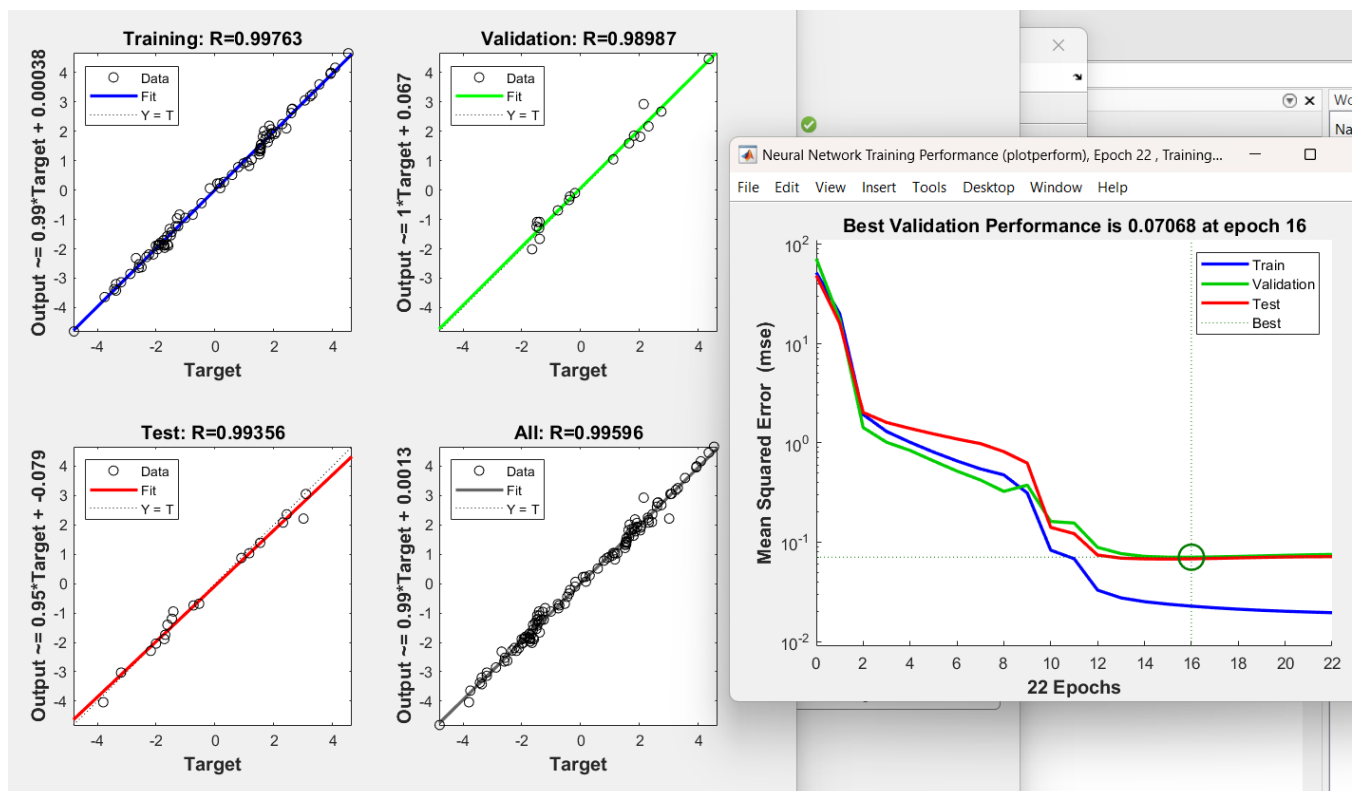


Vemos que, de estas dos pruebas, nos ha dado mejores resultados de exactitud con 5 neuronas, probemos por tanto alrededor de este número.

Con 3 neuronas:



Con 7 neuronas:



A medida que nos alejamos de 5 neuronas, los resultados sobre los conjuntos de validación, test y entrenamiento empeoran, por lo que nos da a pensar que ese número de neuronas es el óptimo.

- i) Compare el comportamiento de "PositionControlNet.slx" con "PositionControl.slx" para diferentes valores de ref_x y ref_y . Calcule el error entre las trayectorias realizadas por ambos

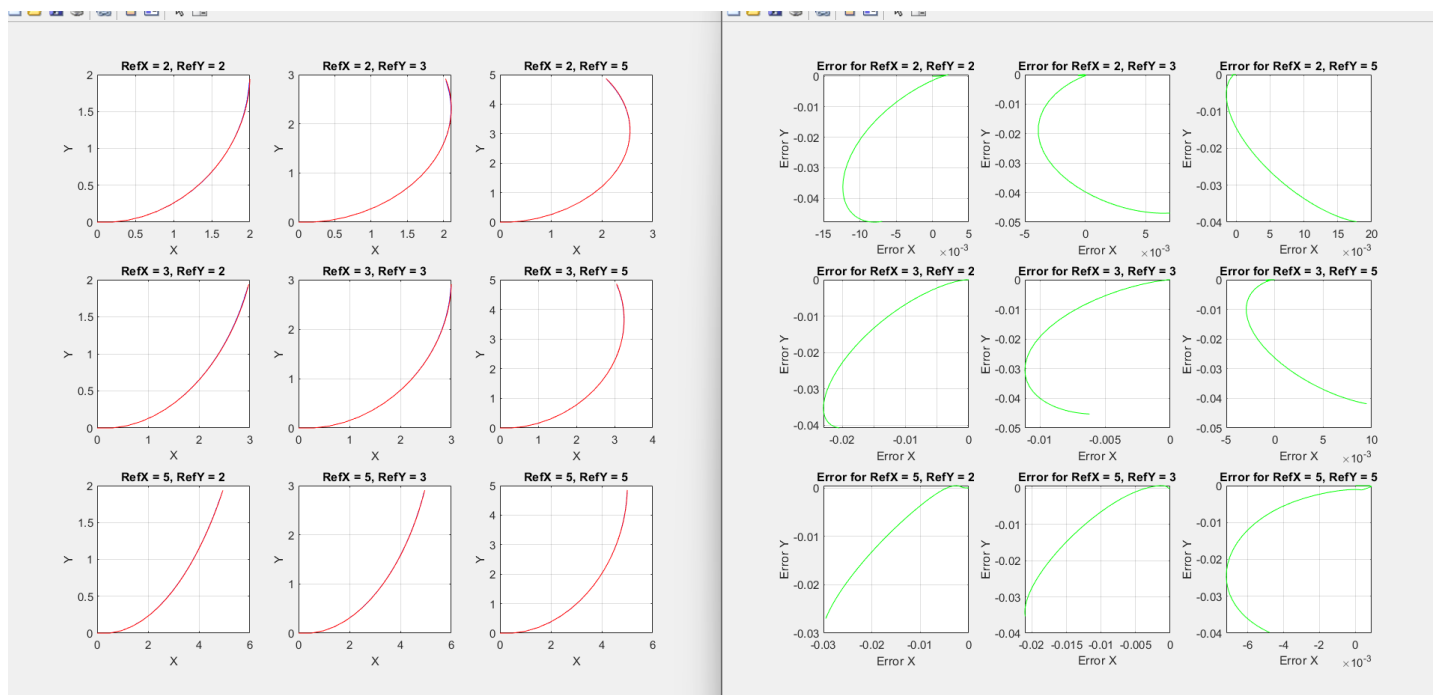
controladores. Se recomienda realizar un script de Matlab para automatizar dicha comparación y mostrar los resultados.

Para este apartado hemos creado un nuevo script donde, damos varios valores para x e y , a partir de estos, se simulan varias veces los modelos, cada vez con unos valores distintos.

Printamos estas simulaciones en una gráfica para compararlas y, en otra figura, imprimimos las gráficas del error de la trayectoria entre los dos modelos, aunque, como se puede ver a continuación, es minúsculo. Tanto que, si no agrandas mucho la imagen de la izquierda, es posible que no se perciba que hay una segunda línea azul debajo de la roja.

Podemos ver en la gráfica de la derecha que, en la mayoría de las simulaciones el error es minúsculo, del orden de 10^{-3} .

Funciona tan parecido ya que lo que estamos haciendo es, dentro del mismo sistema, cambiar un controlador tipo caja negra por una red neuronal entrenada para simular dicho comportamiento.



Para esto, el código empleado ha sido el siguiente:

```
% Tiempo de muestreo
Ts = 100e-3;

% Valores de referencia
valores_refx = [2.0, 3.0, 5.0]; % Define una serie de valores de referencia
valores_refy = [2.0, 3.0, 5.0]; % Define una serie de valores de referencia

% Bucle para probar diferentes valores de referencia
for i = 1:length(valores_refx)
    for j = 1:length(valores_refy)
        % Establecer los valores de referencia en los modelos
        refx = valores_refx(i);
        refy = valores_refy(j);

        % Simular los modelos en Simulink
```

```

simOutNet = sim('PositionControlNet.slx');
x1 = salida_x.signals.values;
y1 = salida_y.signals.values;

simOutNoNet = sim('PositionControl.slx');
x2 = salida_x.signals.values;
y2 = salida_y.signals.values;

% Calcular el error como la diferencia entre las trayectorias
error_x = x1 - x2;
error_y = y1 - y2;

% Crear subplots para las gráficas de trayectorias en la figura 1
figure(1);
subplot(length(valores_refx), length(valores_refy), (i - 1) * length(valores_refy) +
j);
plot(x1, y1, 'b');
hold on;
plot(x2, y2, 'r');
grid on;
xlabel('X');
ylabel('Y');
title(['RefX = ', num2str(refx), ', RefY = ', num2str(refy)]);
hold off;

% Crear una figura separada para las gráficas de errores en la figura 2
figure(2);
subplot(length(valores_refx), length(valores_refy), (i - 1) * length(valores_refy) +
j);
plot(error_x, error_y, 'g');
grid on;
xlabel('Error X');
ylabel('Error Y');
title(['Error for RefX = ', num2str(refx), ', RefY = ', num2str(refy)]);
end
end

```