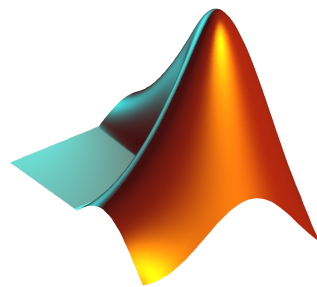




Sistemas de Control Inteligente

Tema: Práctica 1. Identificación y control neuronal



Grupo A2_P09

Integrantes

Ana Cortés Cercadillo
Carlos Javier Hellín Asensio

Grado de Ingeniería Informática

Curso: 2021-2022

Índice

Índice	2
Parte I	4
Ejercicio 1. Perceptron	4
¿Consigue la red separar los datos?	4
¿Cuántas neuronas tiene la capa de salida? ¿Por qué?	5
¿Qué ocurre si se incorpora al conjunto un nuevo dato: [0.0 -1.5] de la clase 3?	5
Ejercicio 2. Aproximación de funciones	8
Código	8
Trainrp	9
Trainbr	10
Trainbfg	11
Traingdm	12
Traingd	13
Comparativa	14
Ejercicio 3. Aproximación de funciones (II)	15
Código	15
Simplefit_dataset - trainlm	15
plotperform	17
plottrainstate	18
ploterrhist	19
plotregression	20
plotfit	21
bodyfat_dataset	22
plotperform	22
plottrainstate	23
ploterrhist	24
plotregression	25
Cambio en el método de entrenamiento	26
Modificación de la división de entrenamiento, validación y test	26
trainRatio = 100/100, valRatio = 0/100, testRatio = 0/100	26
trainRatio = 80/100, valRatio = 20/100, testRatio = 0/100	27
trainRatio = 50/100, valRatio = 0/100, testRatio = 50/100	28
trainRatio = 10/100, valRatio = 50/100, testRatio = 40/100	29
Ejercicio 4. Clasificación.	31
Código	31
simpleclass_dataset	32
plotconfusion	33
plotroc	34
cancer_dataset	35
plotconfusion	36
plotroc	37
Cambio en el método de entrenamiento	38

Modificación de la división de entrenamiento, validación y test	39
trainRatio = 50/100, valRatio = 25/100, testRatio = 25/100	39
trainRatio = 30/100, valRatio = 50/100, testRatio = 20/100	40
trainRatio = 30/100, valRatio = 20/100, testRatio = 50/100	41
Parte II	42
Código	42
Diseño de un control de posición mediante una red neuronal no recursiva.	43
Esquema general de un control de posición.	43
Simular el diagrama PositionControl.	45
Ejecutar el script RunPositionControl.	45
Trayectoria del robot.	46
Varias simulaciones del controlador.	46
Red neuronal con una capa oculta.	47
Un bloque con la red neuronal.	49
Utilizar la red neuronal.	49
Comparar el comportamiento.	50
Parte III	52
Ejercicio 1. Identificación de un sistema utilizando una red recursiva de tipo NARX	52
Ejercicio 2. Control para seguimiento de trayectorias mediante redes recurrentes.	55
Desarrollo del ejercicio.	55
Implementar el esquema	55
Generar el script	55
Diseñar y entrenar una red neuronal recursiva de tipo “narx”	57
Simular el comportamiento y comparar los resultados	60

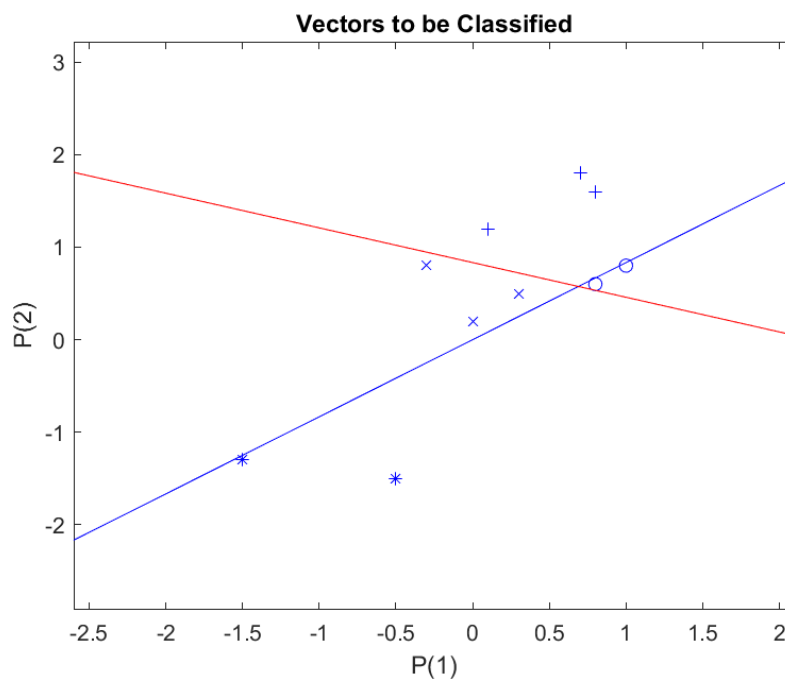
Parte I

Ejercicio 1. Perceptron

¿Consigue la red separar los datos?

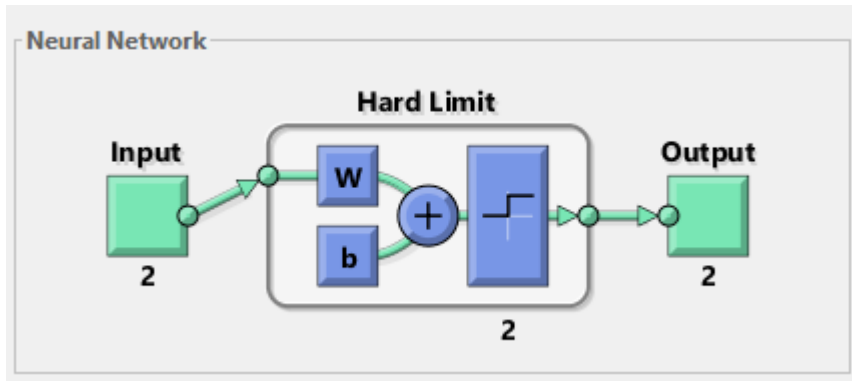
```
Ejercicio_1.m x +
1      % Ejercicio 1
2
3      close all;
4      clear all;
5
6      p = [0.1 0.7 0.8 0.8 1.0 0.3 0.0 -0.3 -0.5 -1.5;
7           1.2 1.8 1.6 0.6 0.8 0.5 0.2 0.8 -1.5 -1.3];
8
9      t = [1 1 1 0 0 1 1 1 0 0;
10          0 0 0 0 0 1 1 1 1 1];
11
12      net = perceptron;
13      net = train(net, p, t);
14
15      plotpv(p, t);
16      plotpc(net.iw{1,1}, net.b{1});
```

La red consigue separar los datos en cuatro regiones del espacio. Lo hace a través de un clasificador binario, el perceptron. Para ello lleva a cabo un aprendizaje con la función *train*. En la siguiente gráfica el símbolo "o" se identifica con la clase 0, el símbolo "*" con la clase 1, el símbolo "+" con la clase 2 y el símbolo "x" con la clase 3. Como se muestra, cada uno de los símbolos se encuentra en sus respectivas clases.



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

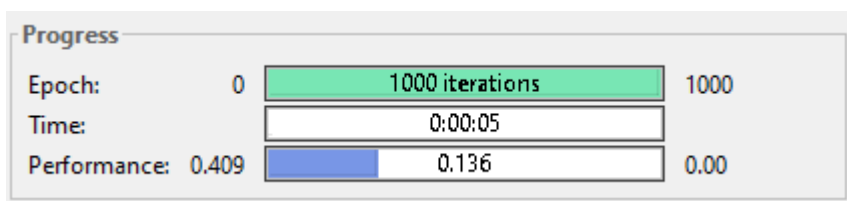
La capa de salida tiene 2 neuronas, como se observa en la imagen. Esto se debe a que como los perceptrones son binarios, se necesitan 2 salidas para representar las cuatro clases.



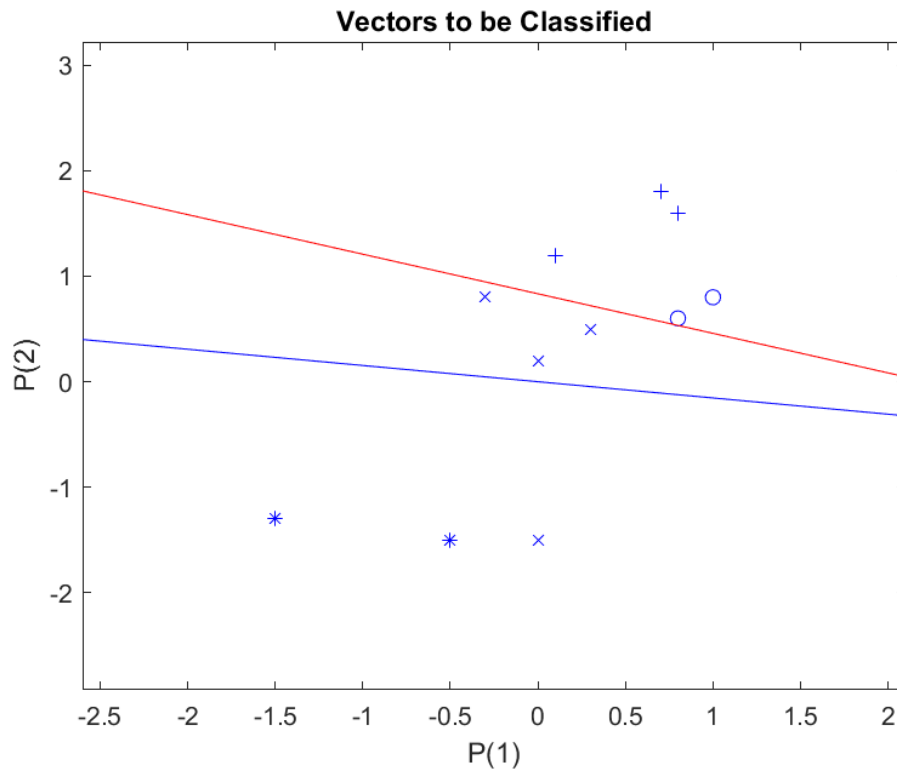
¿Qué ocurre si se incorpora al conjunto un nuevo dato: [0.0 -1.5] de la clase 3?

```
Ejercicio_1.m x +
1 % Ejercicio 1
2
3 - close all;
4 - clear all;
5
6 - p = [0.1 0.7 0.8 0.8 1.0 0.3 0.0 -0.3 -0.5 -1.5 0.0;
7       1.2 1.8 1.6 0.6 0.8 0.5 0.2 0.8 -1.5 -1.3 -1.5];
8
9 - t = [1 1 1 0 0 1 1 1 0 0 1;
10       0 0 0 0 0 1 1 1 1 1 1];
11
12 - net = perceptron;
13 - net = train(net, p, t);
14
15 - plotpv(p, t);
16 - plotpc(net.iw{1,1}, net.b{1});
```

Al ejecutar el programa y terminar el entrenamiento, se observa como las épocas han llegado a las 1000 iteraciones, el máximo permitido.

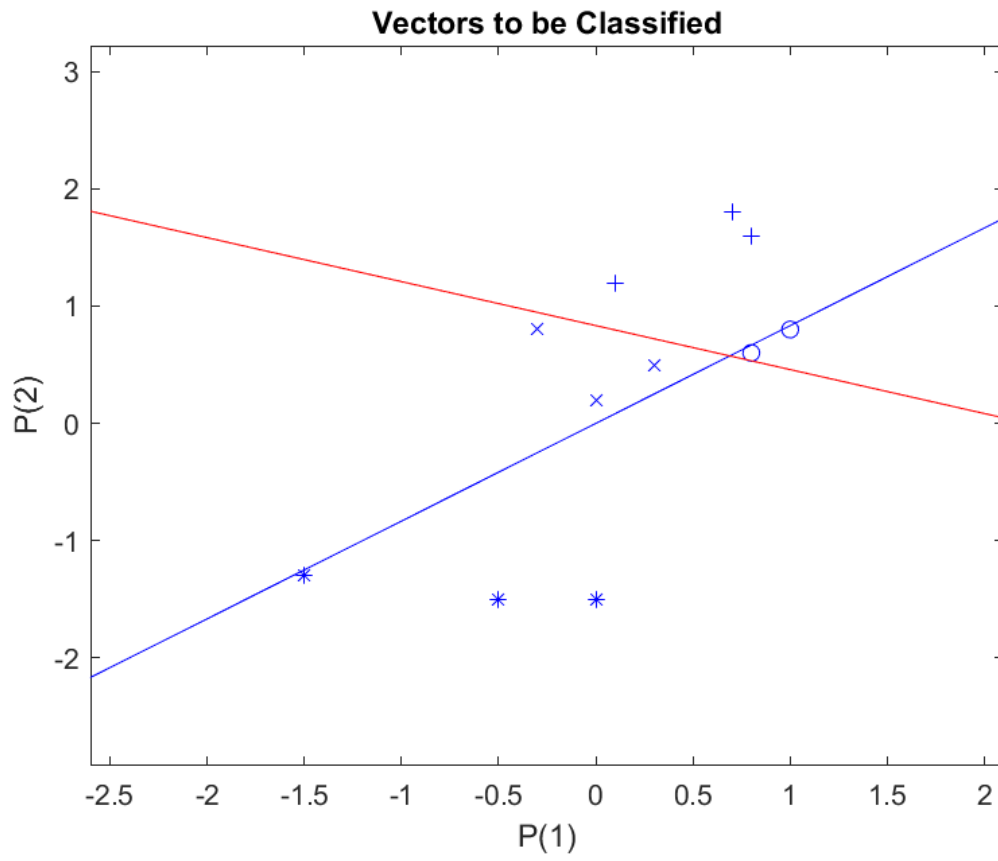


Con ello se intuye que el perceptron no ha llegado a una clasificación clara. Finalmente, en la gráfica se ve como las rectas generadas no consiguen separar los datos en sus clases.



Por último, observando el conjunto de datos nos damos cuenta de que el nuevo dato añadido [0.5 -1.5] se puede clasificar de una forma correcta si éste perteneciera a la clase 1. Haciendo este cambio obtenemos la siguiente gráfica donde se ve con claridad.

```
Ejercicio_1.m x +
1      % Ejercicio 1
2
3      close all;
4      clear all;
5
6      p = [0.1 0.7 0.8 0.8 1.0 0.3 0.0 -0.3 -0.5 -1.5 0.0;
7           1.2 1.8 1.6 0.6 0.8 0.5 0.2 0.8 -1.5 -1.3 -1.5];
8
9      t = [1 1 1 0 0 1 1 1 0 0 0;
10          0 0 0 0 0 1 1 1 1 1 1];
11
12     net = perceptron;
13     net = train(net, p, t);
14
15     plotpv(p, t);
16     plotpc(net.iw{1,1}, net.b{1});
```



Ejercicio 2. Aproximación de funciones

Código

```
1      % Ejercicio 2
2
3      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
4      % APROXIMACIÓN DE FUNCIONES
5      %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
6      clear all;
7      close all;
8
9      %     DEFINICIÓN DE LOS VECTORES DE ENTRADA-SALIDA
10     %     =====
11
12     t = -3:.1:3; % eje de tiempo
13     F=sinc(t)+.001*randn(size(t)); % función que se desea aproximar
14
15     plot(t, F, '+');
16     title('Vectores de entrenamiento');
17     xlabel('Vector de entrada P');
18     ylabel('Vector Target T');
19
20     %     DISEÑO DE LA RED
21     %     =====
22
23     hiddenLayerSize = 4; % número de neuronas en la capa oculta
24     net = fitnet(hiddenLayerSize,'trainrp'); % método de entrenamiento usado
25
26     net.divideParam.trainRatio = 70/100;
27     net.divideParam.valRatio = 15/100;
28     net.divideParam.testRatio = 15/100;
29
30     net = train(net, t, F);
31
32     Y = net(t);
33
34     plot(t, F, '+'); hold on;
35     plot(t, Y, '-r'); hold off;
36     title('Vectores de entrenamiento');
37     xlabel('Vector de entrada P');
38     ylabel('Vector Target T');
39
```


Trainrp

La función de entrenamiento *trainrp* actualiza los valores de peso y sesgo usando el algoritmo de retropropagación resistente (Rprop).

<i>hiddenLayerSize</i> = 1	<i>hiddenLayerSize</i> = 8	<i>hiddenLayerSize</i> = 20
<p>Neural Network</p> <p>Algorithms</p> <p>Data Division: Random (dividerand) Training: RProp (trainrp) Performance: Mean Squared Error (mse) Calculations: MEX</p>	<p>Neural Network</p> <p>Algorithms</p> <p>Data Division: Random (dividerand) Training: RProp (trainrp) Performance: Mean Squared Error (mse) Calculations: MEX</p>	<p>Neural Network</p> <p>Algorithms</p> <p>Data Division: Random (dividerand) Training: RProp (trainrp) Performance: Mean Squared Error (mse) Calculations: MEX</p>
<p>Vectores de entrenamiento</p>	<p>Vectores de entrenamiento</p>	<p>Vectores de entrenamiento</p>

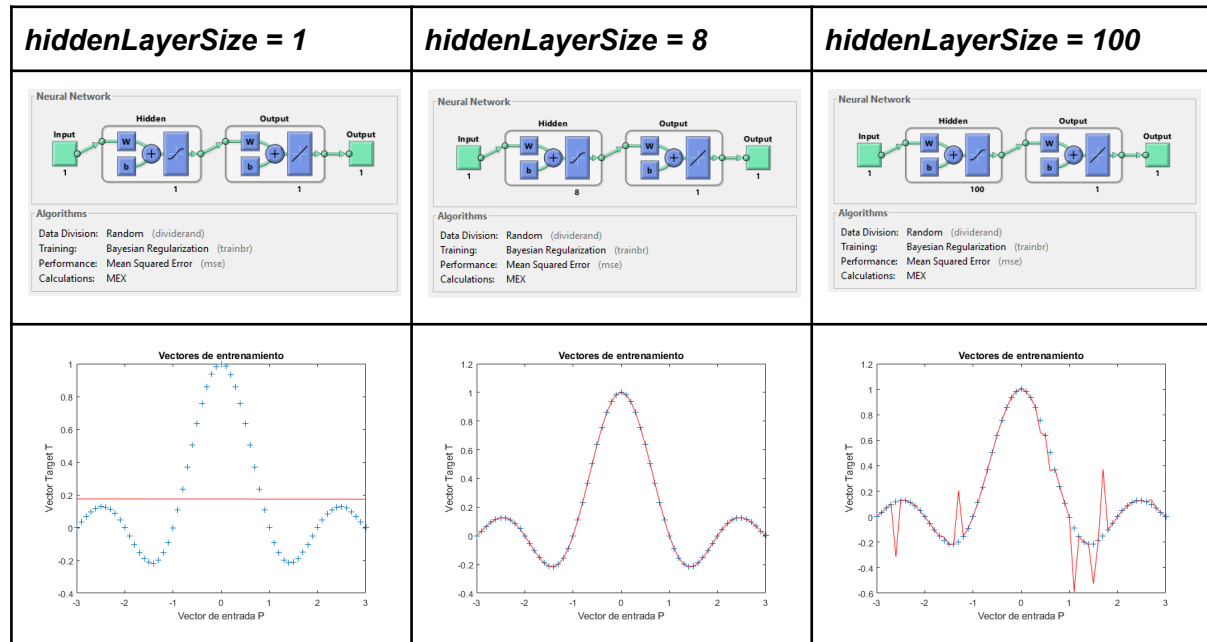
En la tabla anterior se observa que con una neurona en la capa oculta la red neuronal no aprende lo suficiente como para dar como resultado un ajuste óptimo. La función de ajuste en este caso es lineal.

Al aumentar el número de neuronas a 8 ya se obtienen mejores resultados y un mejor ajuste, sin llegar a ser perfecto.

Probamos un número más alto, el 20, y aparecen picos en la función de ajuste. Empeora su aproximación a la función que se desea adaptar.

Trainbr

La función de entrenamiento *trainbr* actualiza los valores de peso y sesgo de acuerdo con la optimización de Levenberg-Marquardt. Esta optimización se realiza minimizando una combinación de errores cuadráticos y pesos, y luego determinando la combinación correcta para producir una red que se generalice bien. El proceso se llama regularización Bayesiana.



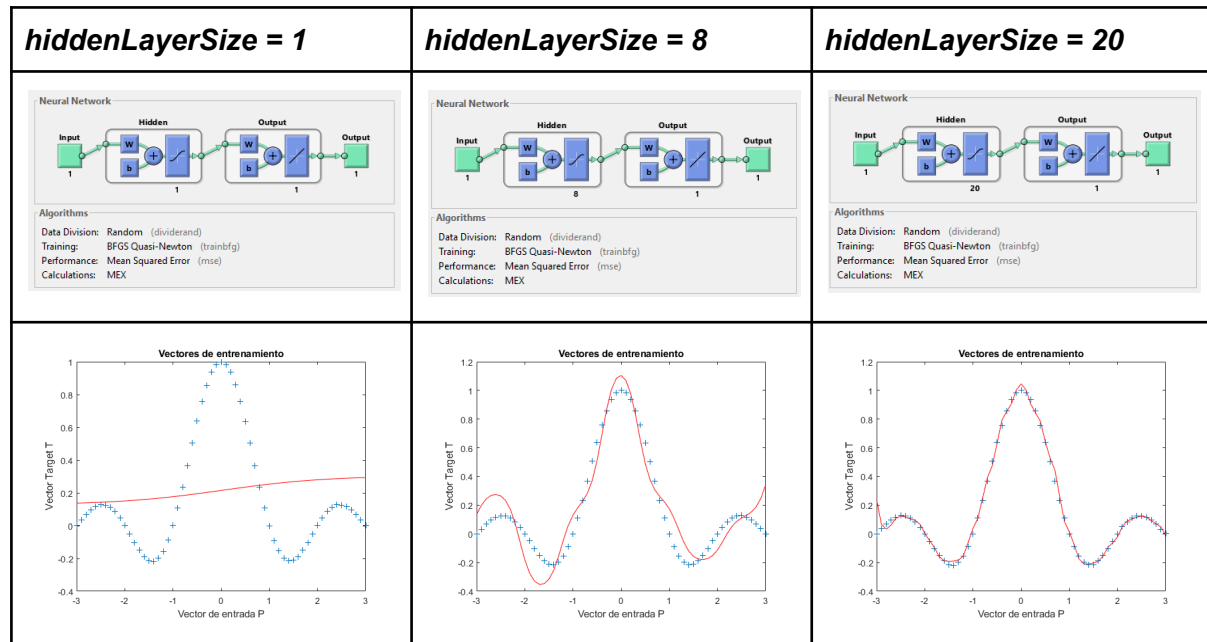
En este caso, con una neurona en la capa oculta ocurre lo mismo que en la gráfica de la función de entrenamiento *trainrp* con este mismo número de neuronas, su ajuste es una función lineal.

Con 8 neuronas, ahora sí, obtenemos un ajuste perfecto a la función que se quiere aproximar.

En el último, elegimos 100 neuronas en la capa oculta porque con 20 sigue ajustándose perfectamente a la función como en el suceso anterior. Por un mayor aprendizaje, aparecen salientes en forma puntiaguda en algunas zonas de la función que se desea ajustar, por lo tanto, se aleja de su aproximación.

Trainbfg

La función de entrenamiento *trainbfg* actualiza los valores de peso y sesgo de acuerdo con el método quasi-Newton de BFGS.



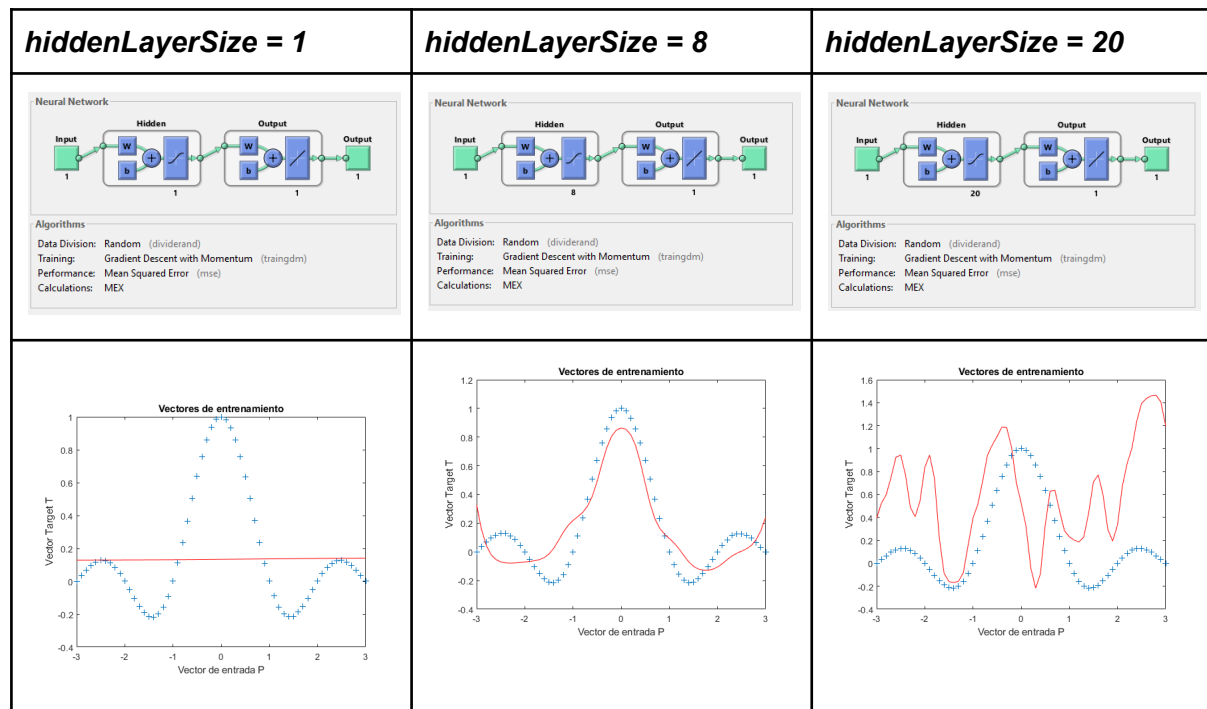
Con una sola neurona en la capa oculta el ajuste no es óptimo porque la red neuronal no aprende lo suficiente.

De la misma forma, con 8 neuronas no se llega a una aproximación deseada. Mejora el resultado anterior pero no es perfecto.

Con esta función de entrenamiento, la mejor adaptación que llegamos a obtener es con 20 neuronas. Tampoco llega a ser perfecta y si aumentamos mucho más este número, vuelve a empeorar el ajuste por el mismo motivo que en los apartados anteriores.

Trainingdm

La función de entrenamiento *trainingdm* actualiza los valores de peso y sesgo de acuerdo con el descenso del gradiente con momento.



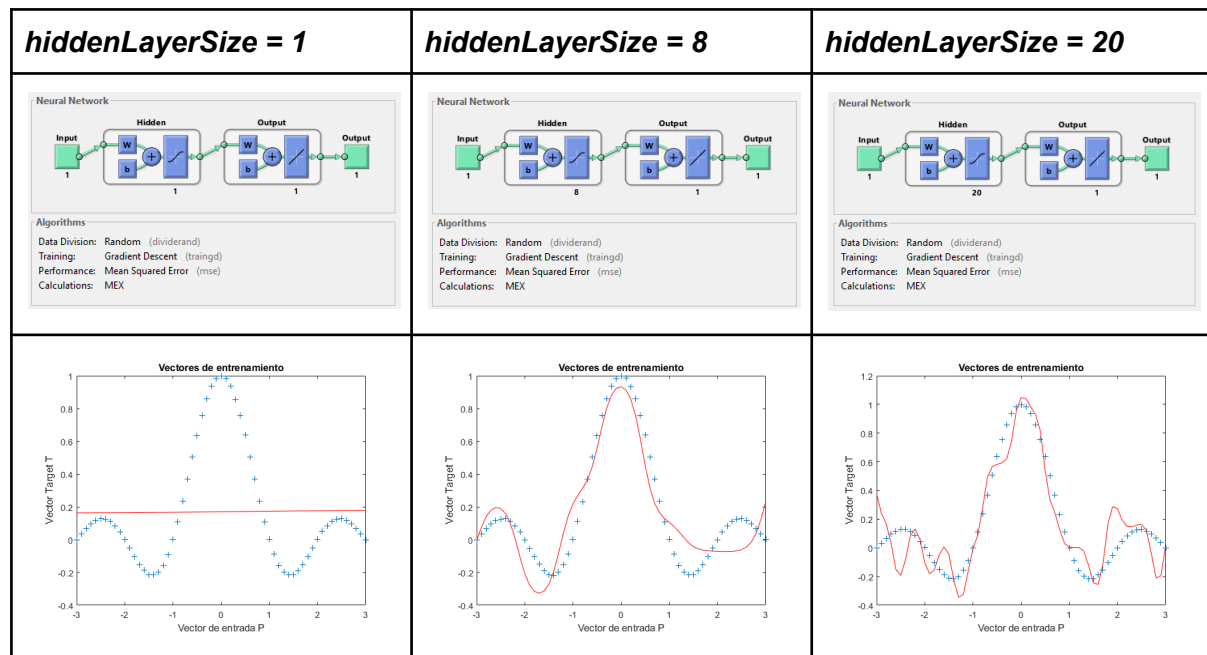
De la misma forma que anteriormente, con una neurona no ajusta la función porque no llega a aprender. Su aproximación es una función lineal.

Con 8 neuronas progresa en el ajuste pero está muy lejos de ser correcto.

Finalmente, con 20 neuronas, al contrario de mejorar el ajuste, no es capaz de generalizar los resultados.

Trainingd

La función de entrenamiento *trainingd* actualiza los valores de peso y sesgo de acuerdo con el descenso del gradient.



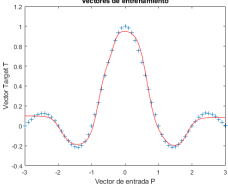
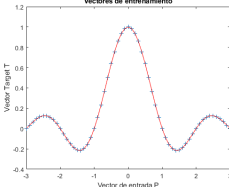
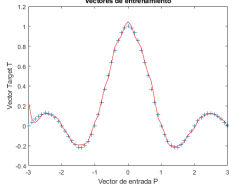
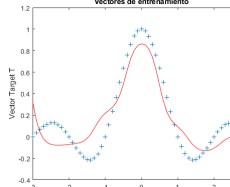
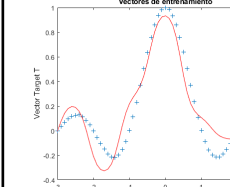
En la tabla anterior se observa que con una neurona en la capa oculta la red neuronal no aprende lo suficiente como para dar como resultado un ajuste óptimo. La función de ajuste en este caso es lineal.

Al aumentar el número de neuronas a 8 ya se obtienen mejores resultados y un mejor ajuste, con bastantes imperfecciones.

Probamos un número más alto, el 20, y como se observa en la gráfica anterior no ha sido capaz de predecir correctamente la función, es decir, la red no ha sido capaz de aprender de forma óptima debido a un sobreentrenamiento.

Comparativa

Para tener una visión más global de qué función de entrenamiento se ajusta mejor a la función que se desea aproximar, creamos la siguiente tabla con las mejores gráficas obtenidas anteriormente.

<i>Trainrp</i>	<i>Trainbr</i>	<i>Trainbfg</i>	<i>Traingdm</i>	<i>Traingd</i>
<i>hiddenLayerSize</i> = 8	<i>hiddenLayerSize</i> = 8	<i>hiddenLayerSize</i> = 20	<i>hiddenLayerSize</i> = 8	<i>hiddenLayerSize</i> = 8
				

De ello sacamos la conclusión de que, sin dudarlo, de las funciones de entrenamiento estudiadas la que se ajusta perfectamente es la *trainbr* con 8 neuronas en la capa oculta. También se puede destacar la función de *trainbfg* que se aproxima bastante aunque con el uso de más neuronas en la capa oculta y la de *trainrp*, de igual forma, con un ajuste semejante al esperado. Las dos últimas funciones analizadas, *traingdm* y *traingd*, no llegan a obtener una aproximación tan acertada como las anteriores.

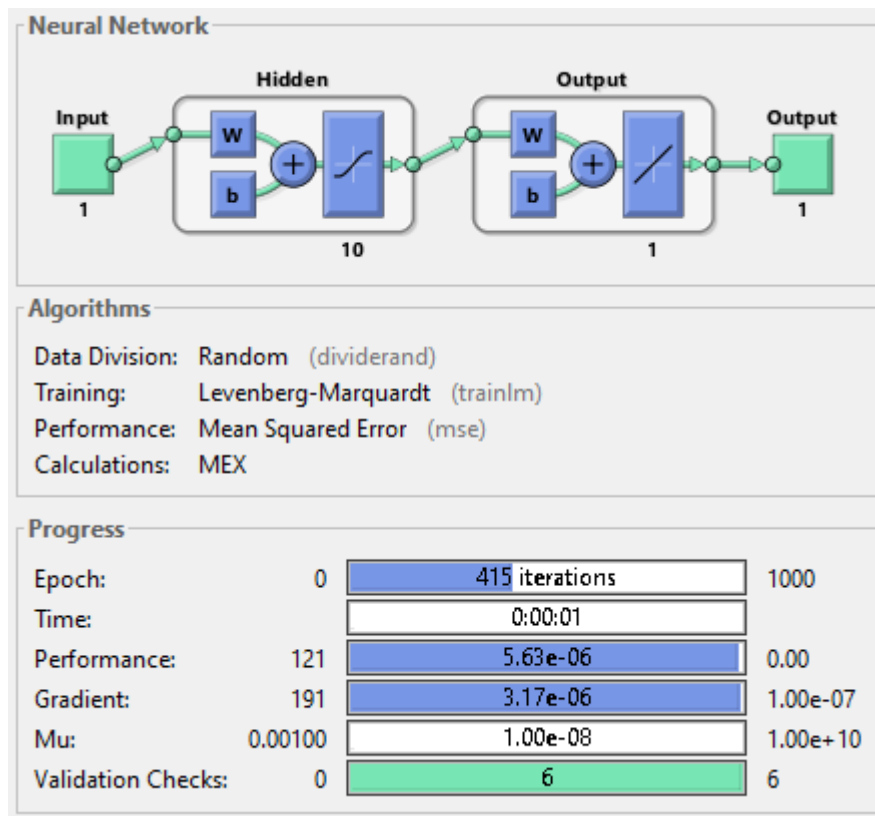
Ejercicio 3. Aproximación de funciones (II)

Código

```
1 % Ejercicio 3
2
3 clear all;
4 close all;
5 nnet.guis.closeAllViews();
6
7 % Carga de datos de ejemplo disponibles en la toolbox
8 [inputs,targets] = simplefit_dataset; % conjunto de datos usados
9
10 % Creación de la red
11 hiddenLayerSize = 10;
12 net = fitnet(hiddenLayerSize); % método de entrenamiento usado
13
14 % División del conjunto de datos para entrenamiento, validación y test
15 net.divideParam.trainRatio = 70/100;
16 net.divideParam.valRatio = 15/100;
17 net.divideParam.testRatio = 15/100;
18
19 % Entrenamiento de la red
20 [net,tr] = train(net,inputs,targets);
21
22 % Prueba
23 outputs = net(inputs);
24 errors = gsubtract(outputs,targets);
25 performance = perform(net,targets,outputs)
26
27 % Visualización de la red
28 view(net)
29
```

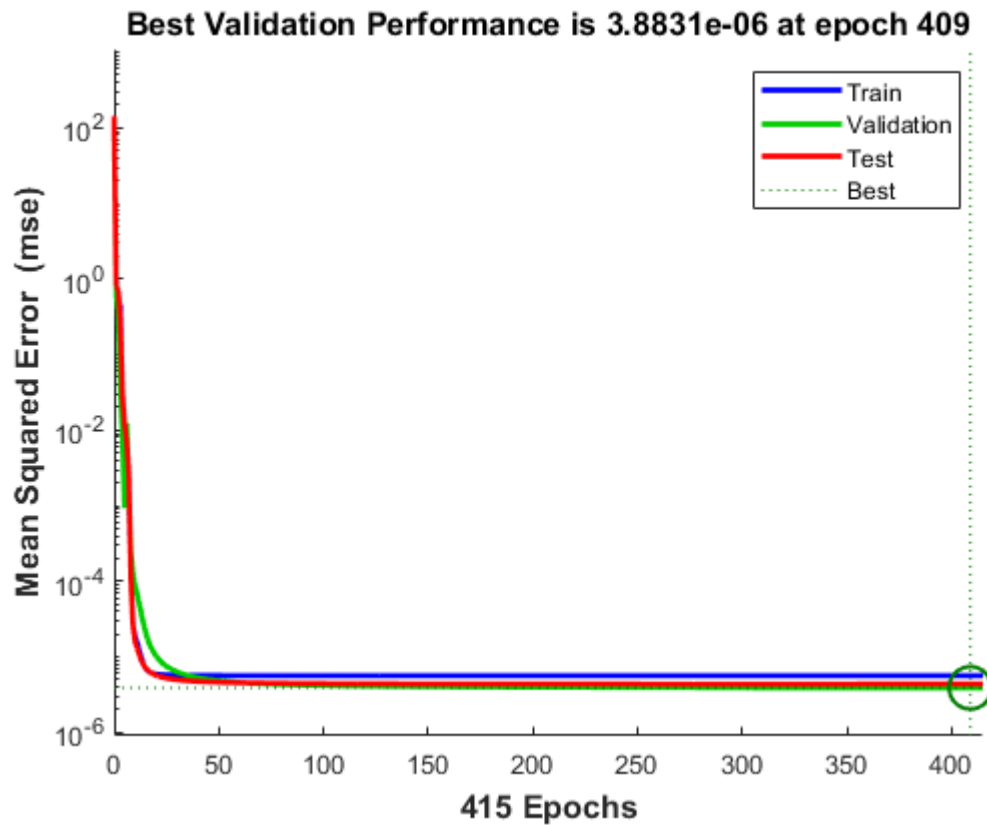
Simplefit_dataset - trainlm

En esta primera parte del estudio de las gráficas, usamos el dataset que se llama *simplefit_dataset*. El método de entrenamiento utilizado es el de *trainlm* con 10 neuronas en la capa oculta. La red neuronal está compuesta por una entrada y una salida. También podemos observar que la época se completa en 415 iteraciones. Estos datos se pueden ver en la siguiente imagen. A continuación obtenemos las diferentes gráficas disponibles.



plotperform

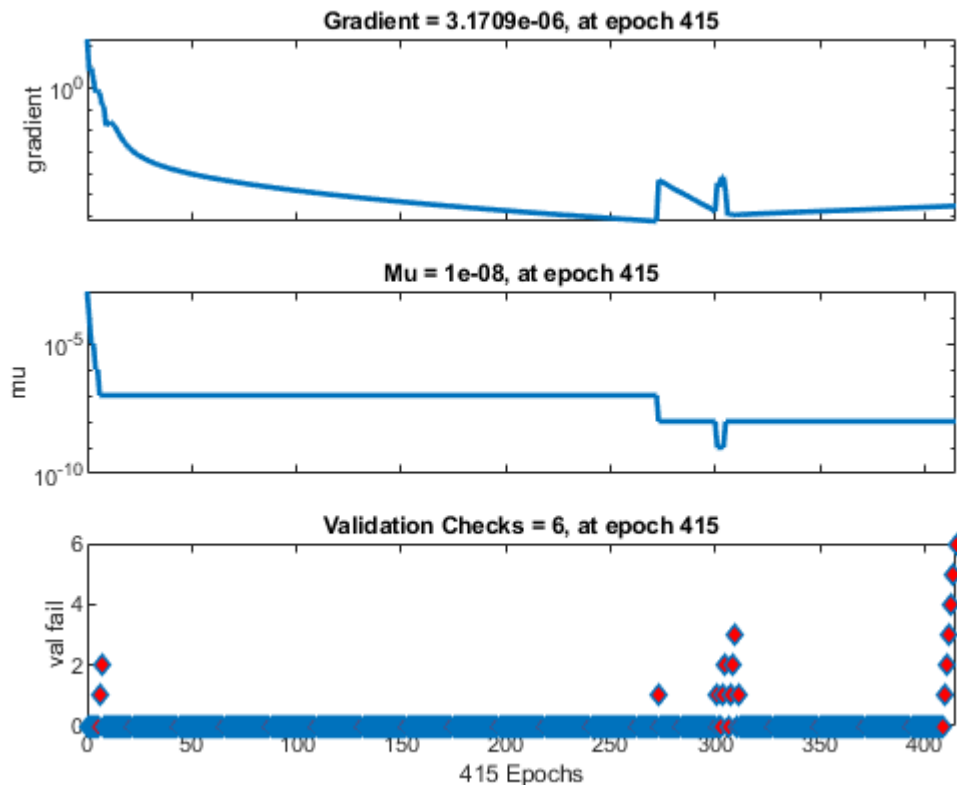
La gráfica *plotperform* (TR) traza el error frente a la época para el entrenamiento, la validación y los rendimientos de prueba del registro de entrenamiento (TR) devuelto por la función de entrenamiento.



En este tipo de gráfica se observa con facilidad como el error va disminuyendo a medida que van avanzando las épocas. El círculo que se encuentra a la derecha de la imagen marca el valor mínimo del error de los datos de validación. Después de que ocurrieran 6 incrementos consecutivos del error de validación, se detiene la ejecución de la función de entrenamiento. Finalmente, se elige el menor de éstos como resultado, siendo el mejor rendimiento de la validación.

plottrainstate

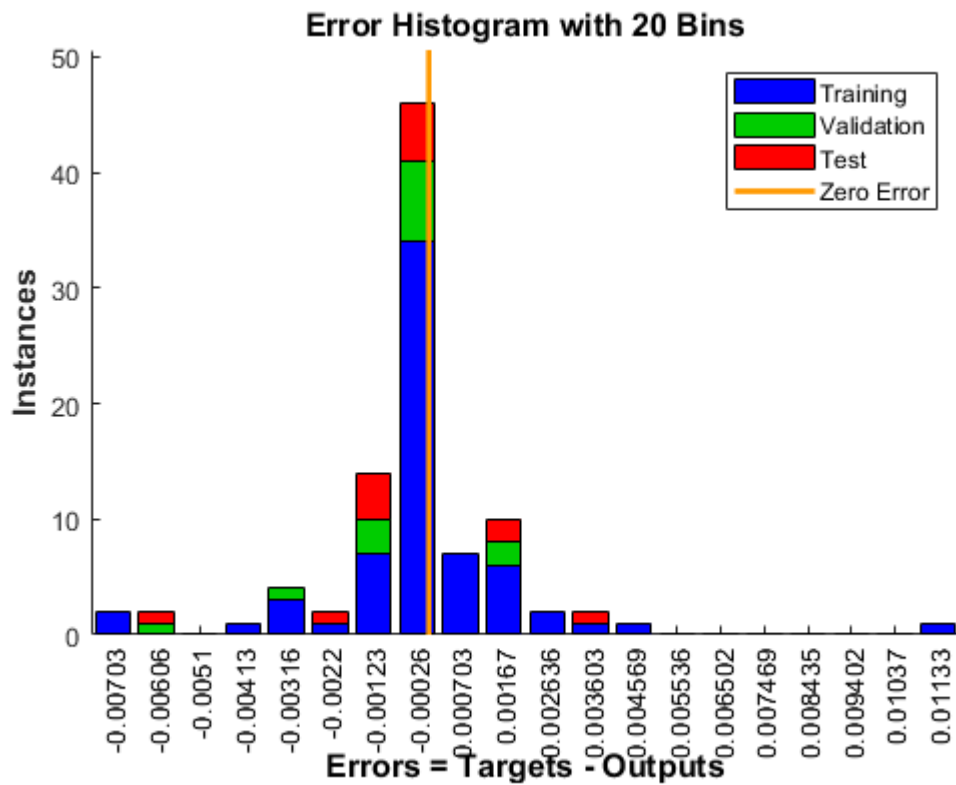
La gráfica *plottrainstate* (tr) traza el estado de un registro de entrenamiento (tr) devuelto por la función observando así su evolución.



La ventana obtenida muestra tres gráficas. La primera muestra la evolución del gradiente. El gradiente representa la velocidad de los cambios en el proceso de *backpropagation* en la próxima iteración. Se observa que al inicio del entrenamiento es un valor alto y desciende a medida que se acerca a los mínimos. La segunda gráfica muestra la ganancia (μ) que se obtiene con este tipo de entrenamiento. Este valor cambia dependiendo del método de entrenamiento usado llegando a ser inexistente en algunos casos. La última gráfica muestra los incrementos de errores consecutivos. Como hemos explicado en la gráfica anterior, en esta se ve cómo aparecen los 6 incrementos consecutivos que provoca que se llegue al final del aprendizaje.

ploterrhist

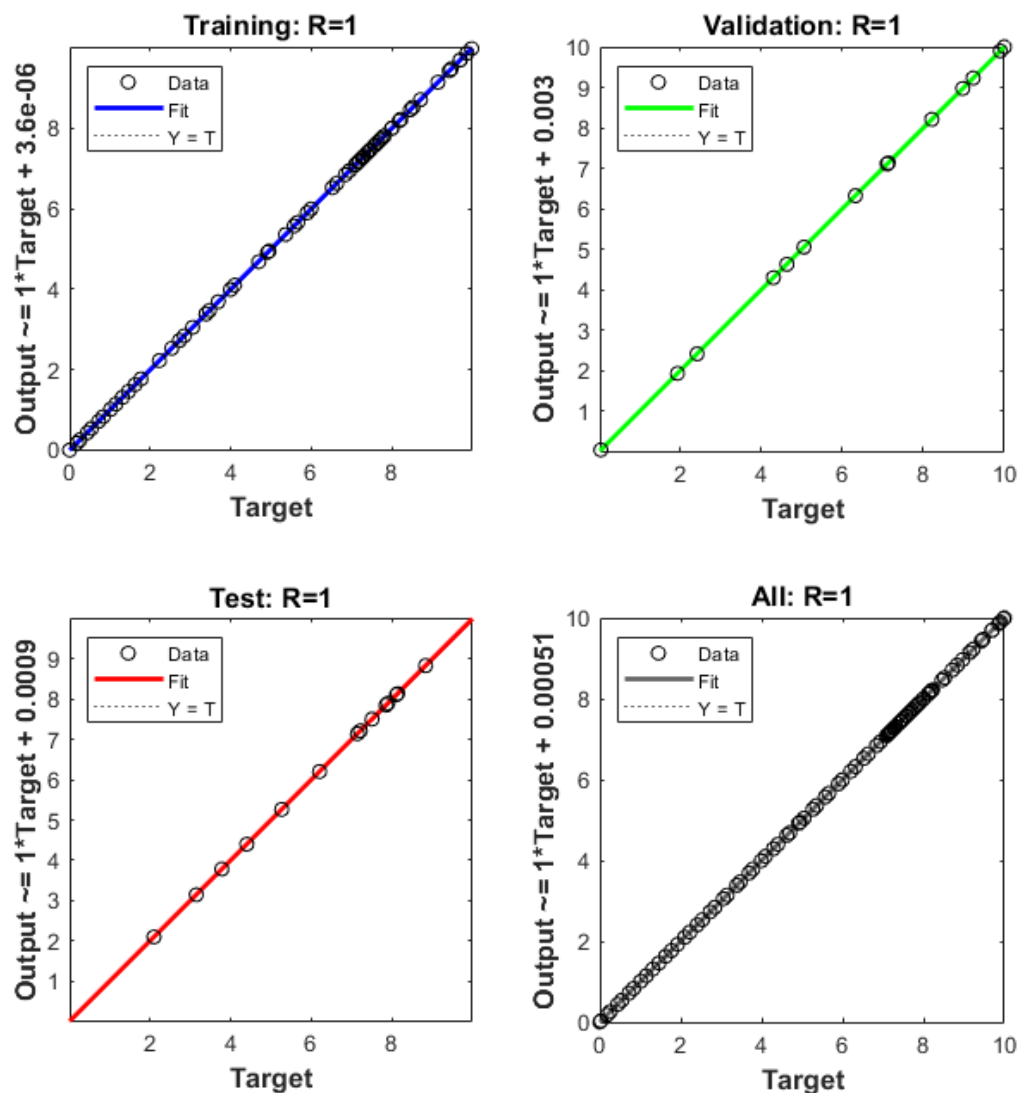
Esta gráfica muestra el histograma de error de los datos de entrada.



Lo que destaca de la observación de la gráfica es que la mayoría de los datos han tenido una tasa de error de -0.00026 resaltando los datos de entrenamiento. También se ve como casi todos los datos de entrenamiento se van agrupando de forma que el eje central es el valor de error cero.

plotregression

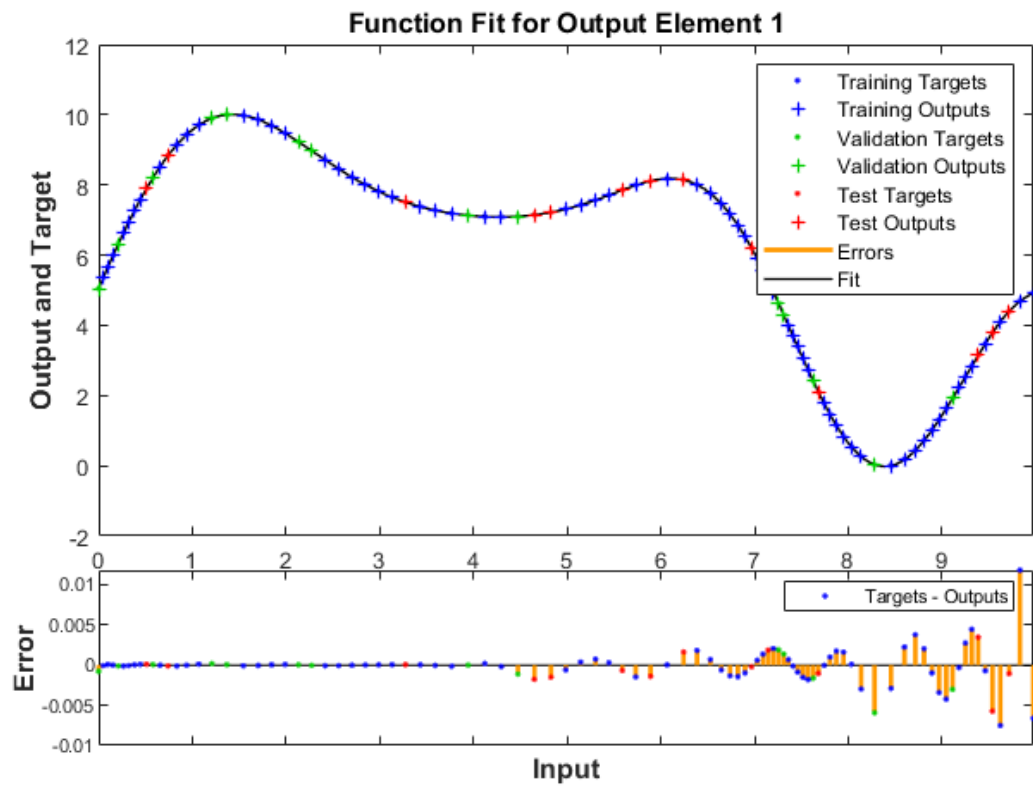
La gráfica dibuja las funciones lineales del estado final de la red neuronal.



En estas gráficas se muestra la relación que hay entre las salidas obtenidas (*output*) y el resultado esperado (*target*) sobre el conjunto de datos que estamos usando. Con estos se calcula una recta de regresión para una máxima aproximación a los valores de los datos, cuya inclinación y posición se obtiene calculando el valor de la media y la desviación media de los datos. En un entrenamiento perfecto el valor de *output* sería igual al valor de *target* y en uno más realista se usa la función lineal de regresión para ajustar lo máximo posible los valores a una recta.

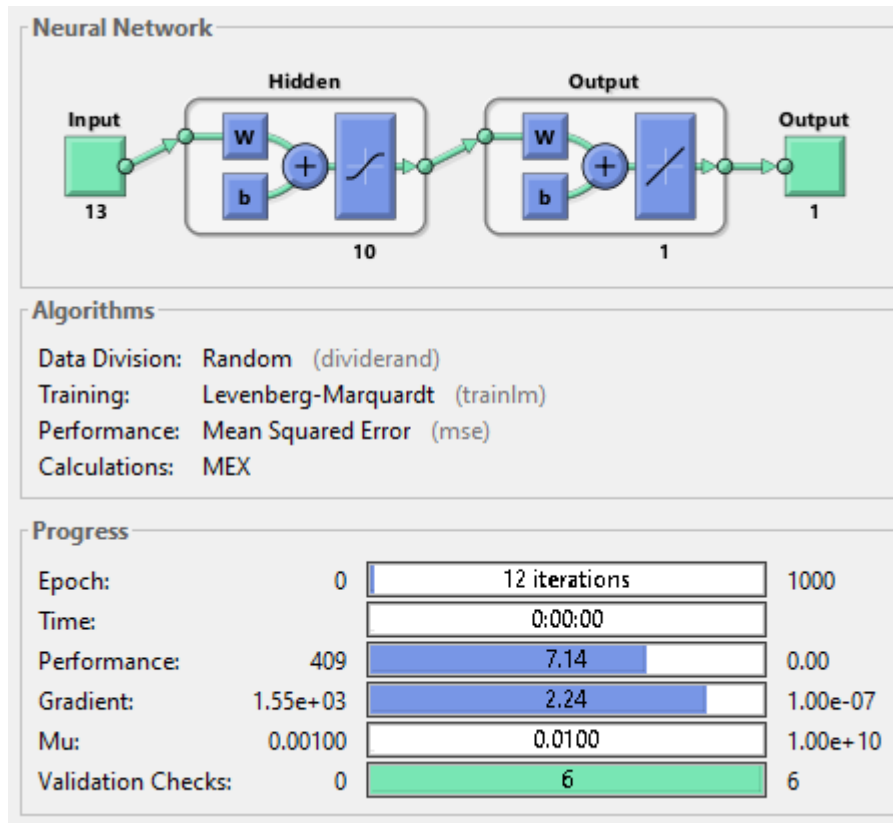
plotfit

La gráfica traza la salida de la red en función de las entradas y también traza los datos de salida esperados (*target*) y obtenidos (*output*) con los valores en las entradas. También se muestra el error en cada punto de la función.

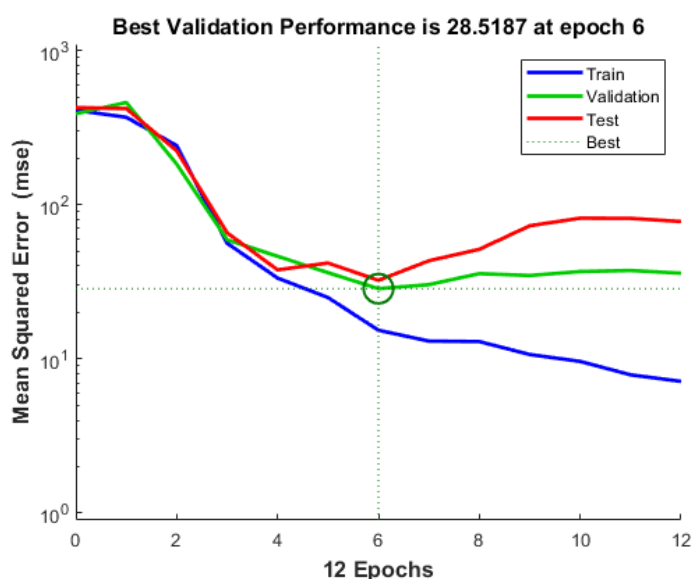


bodyfat_dataset

Ahora, usamos el dataset que se llama *bodyfat_dataset*. El método de entrenamiento utilizado es el de *trainlm* con 10 neuronas en la capa oculta. La red neuronal está compuesta por 13 entradas y una salida. También podemos observar que la época se completa en 12 iteraciones. Estos datos se pueden ver en la siguiente imagen. A continuación obtenemos las diferentes gráficas disponibles.

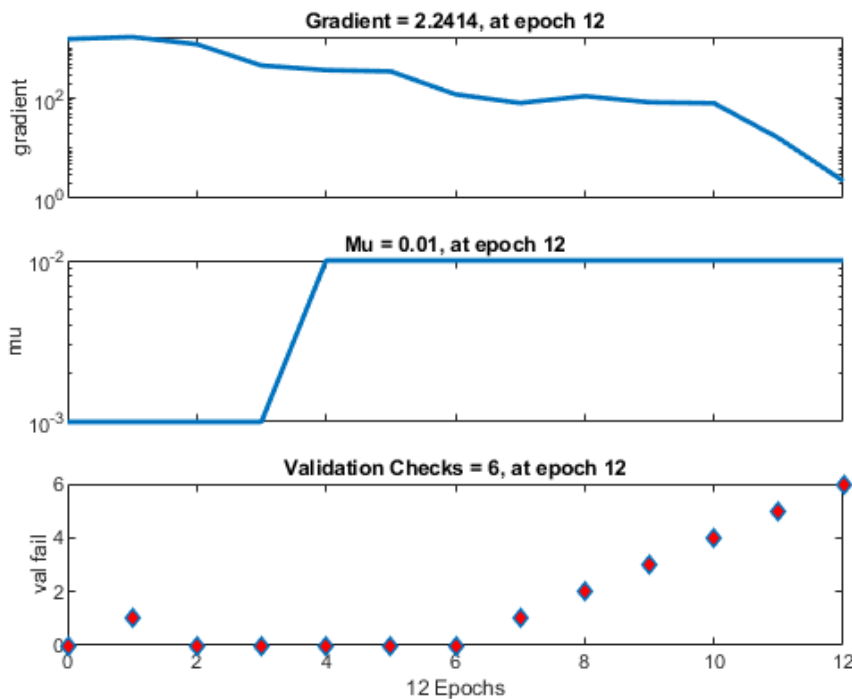


plotperform



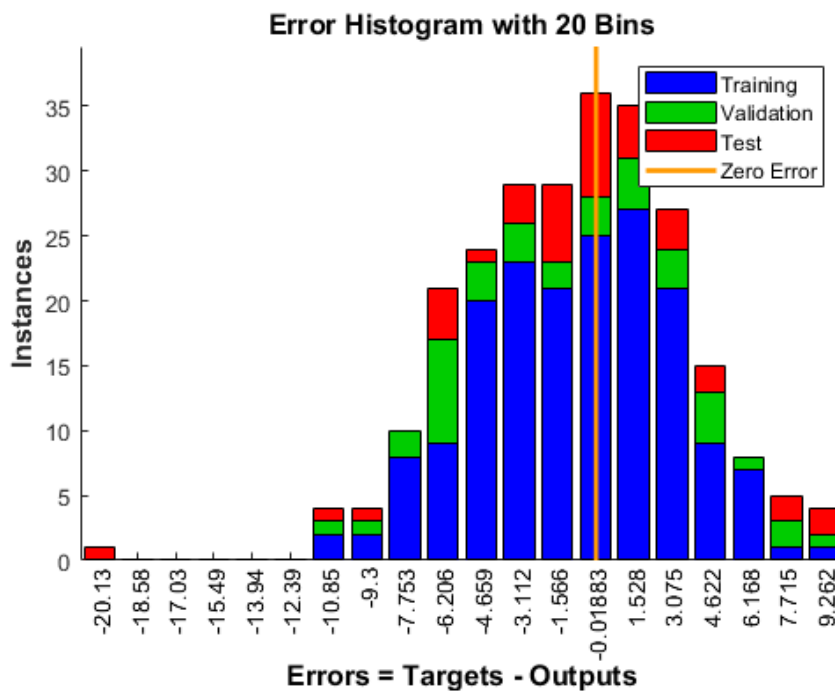
En esta gráfica se observa con facilidad como el error va disminuyendo a medida que van avanzando las épocas. También se ven mucho mejor los incrementos de los que se hablaba en la explicación anterior de esta gráfica. El círculo, que marca el valor mínimo del error de los datos de validación, se encuentra justo antes de estos 6 incrementos.

plottrainstate



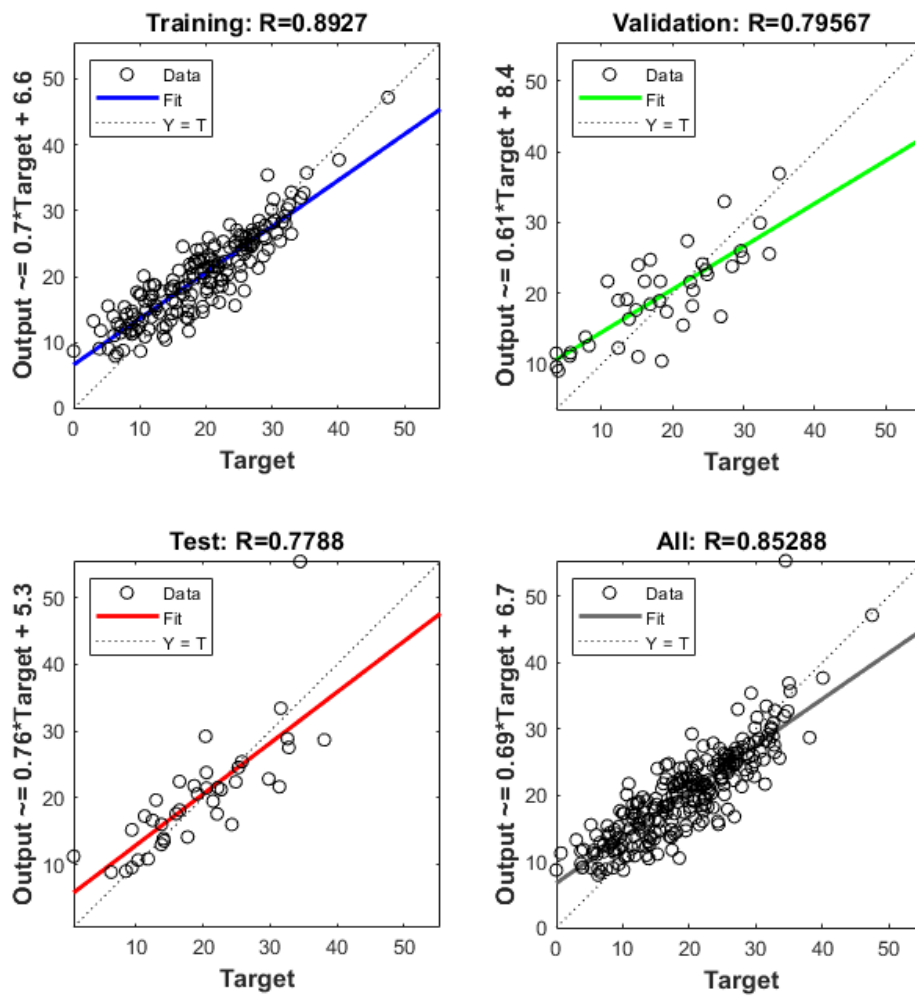
En la primera gráfica, se observa que al inicio del entrenamiento es un valor alto y desciende a medida que se acerca a los mínimos, como es habitual. La segunda gráfica nos muestra como a partir de la cuarta iteración el nivel de aprendizaje de la red neuronal aumenta. La última gráfica muestra de otra forma los incrementos de errores consecutivos que se observan también en la primera gráfica.

ploterrhist



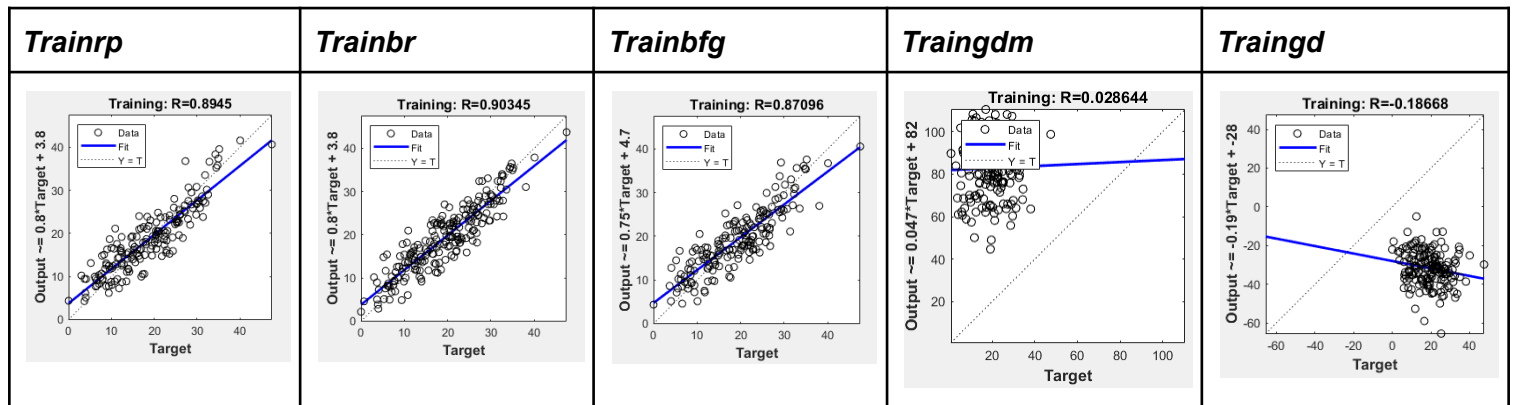
Lo que destaca de la observación de la gráfica es que la mayoría de los datos han tenido una tasa de error de -0.01883 resaltando los datos de entrenamiento.

plotregression



Se puede subrayar, en comparación con estas mismas gráficas del apartado anterior, que las rectas de regresión calculadas tienen una peor aproximación a la salida esperada en un entrenamiento perfecto ya que los datos se encuentran más dispersos.

Cambio en el método de entrenamiento

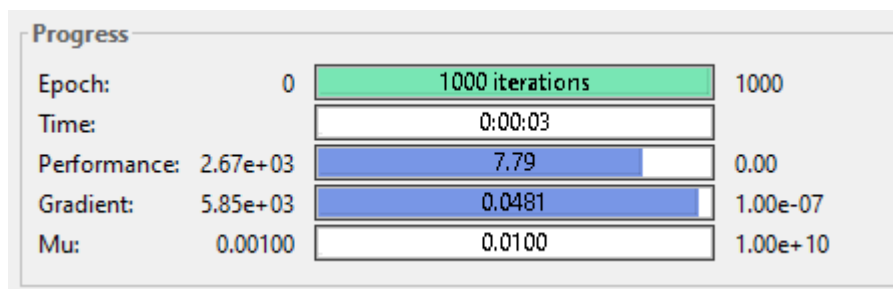


Las mejores funciones de entrenamiento son con las que se obtiene una media más cercana a 0 y una desviación más cercana a 1. Con la observación de las gráficas, se intuye que las mejores aproximaciones con las de *trainrp*, *trainbr* y *trainbfg*. Las dos primeras tienen una desviación de 0.8 y una media de 3.8 y la tercera gráfica tiene una desviación de 0.75 y una media de 4.7, por lo tanto, de ello se concluye que en este caso los mejores métodos de entrenamiento son *trainrp* y *trainbr*. Cabe destacar que las funciones con peores resultados sean las que utilizan el algoritmo de descenso del gradiente de forma más pura.

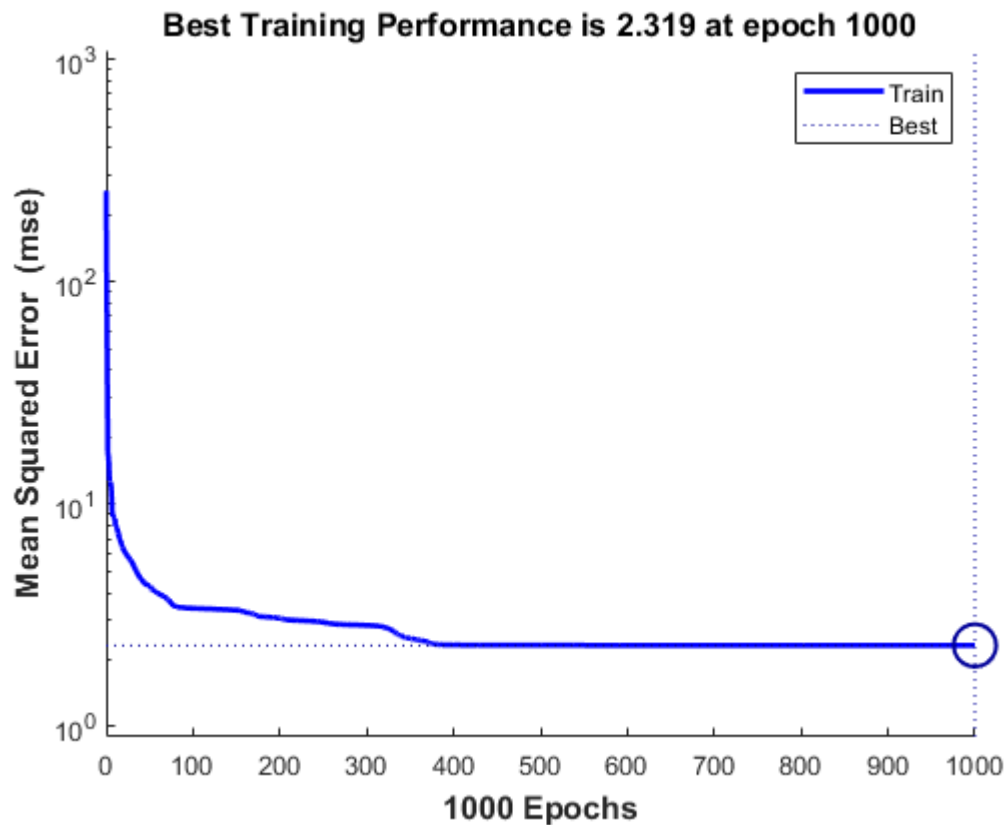
Modificación de la división de entrenamiento, validación y test

Vamos a comparar diferentes cambios en la división de los datos de entrenamiento, validación y test y comparamos los resultados entre ellos.

trainRatio = 100/100, *valRatio* = 0/100, *testRatio* = 0/100

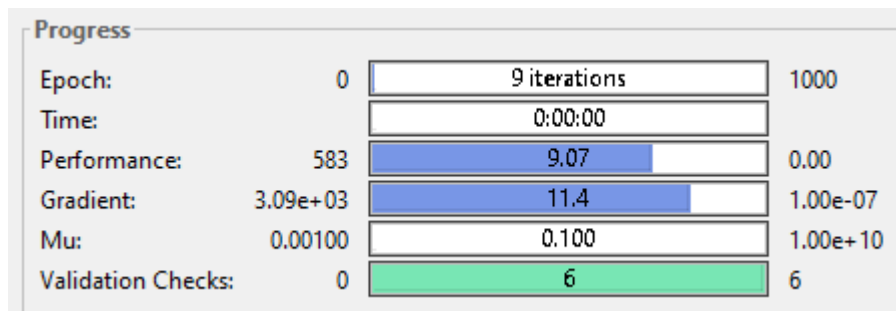


Si todos los datos son de entrenamiento, la red entrenará hasta agotar sus iteraciones. Esto ocurre porque no tiene datos de test para saber cuándo detener el entrenamiento.

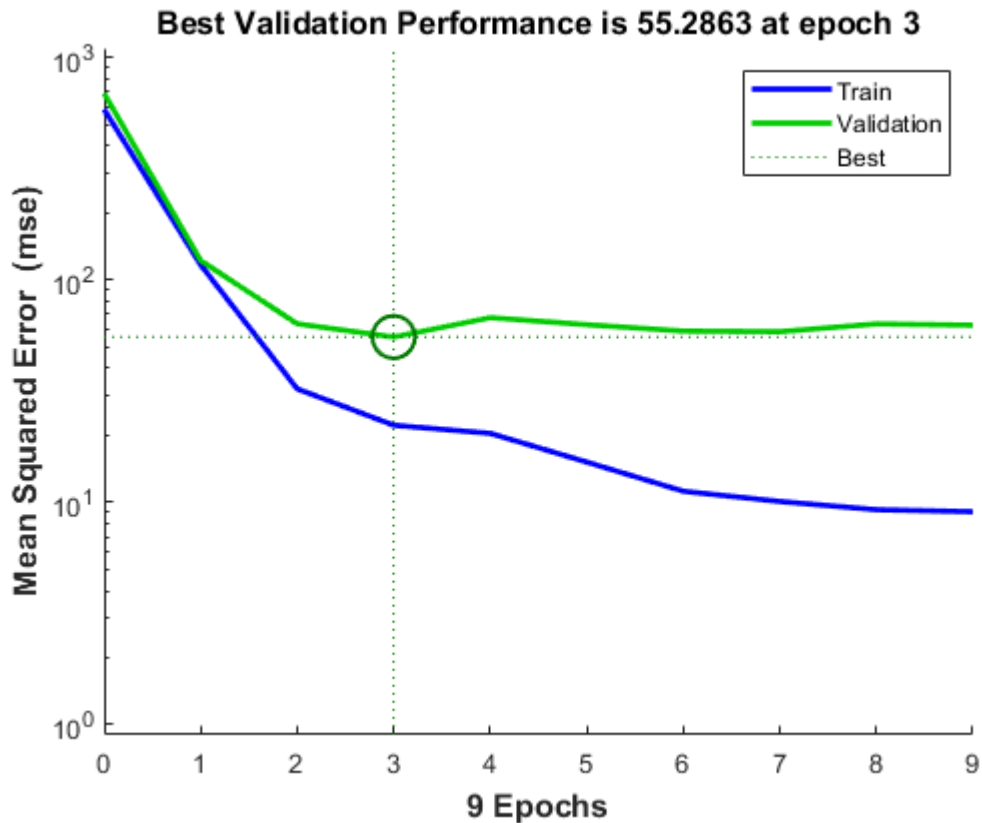


En la gráfica se observa que tiene muy buen rendimiento, ya que se realizan todas las iteraciones posibles.

trainRatio = 80/100, valRatio = 20/100, testRatio = 0/100



Cuando los datos de validación son demasiado bajos la red terminará muy pronto superando los 6 aumentos consecutivos.

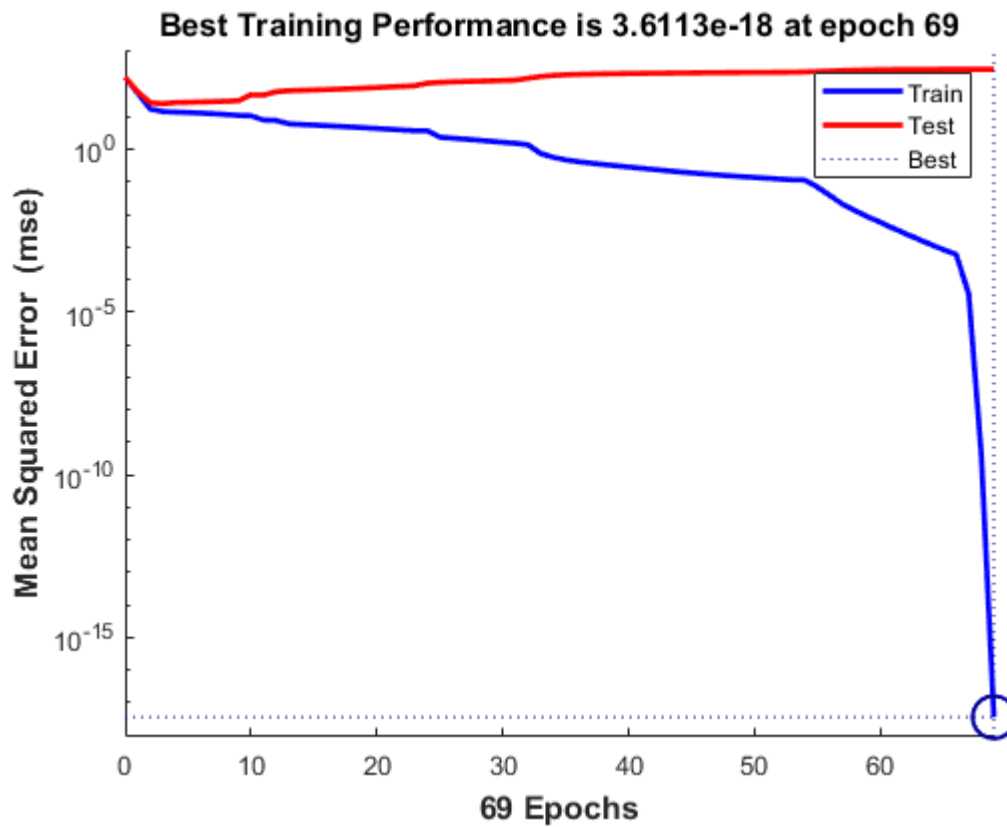


La gráfica nos muestra los malos resultados que se obtienen con esta división. El error cuadrático es muy alto. Tanto en el apartado anterior como este, al no tener datos de test, no se consigue disminuir en gran medida el error.

trainRatio = 50/100, valRatio = 0/100, testRatio = 50/100

Progress				
Epoch:	0	<div><div></div></div>	69 iterations	1000
Time:			0:00:00	
Performance:	160	<div><div></div></div>	3.61e-18	0.00
Gradient:	354	<div><div></div></div>	7.07e-08	1.00e-07
Mu:	0.00100	<div><div></div></div>	1.00e-07	1.00e+10

Parece que se obtiene un progreso correcto aunque lo habitual es que éste se detenga debido a los datos de validación, que en este supuesto no existen.

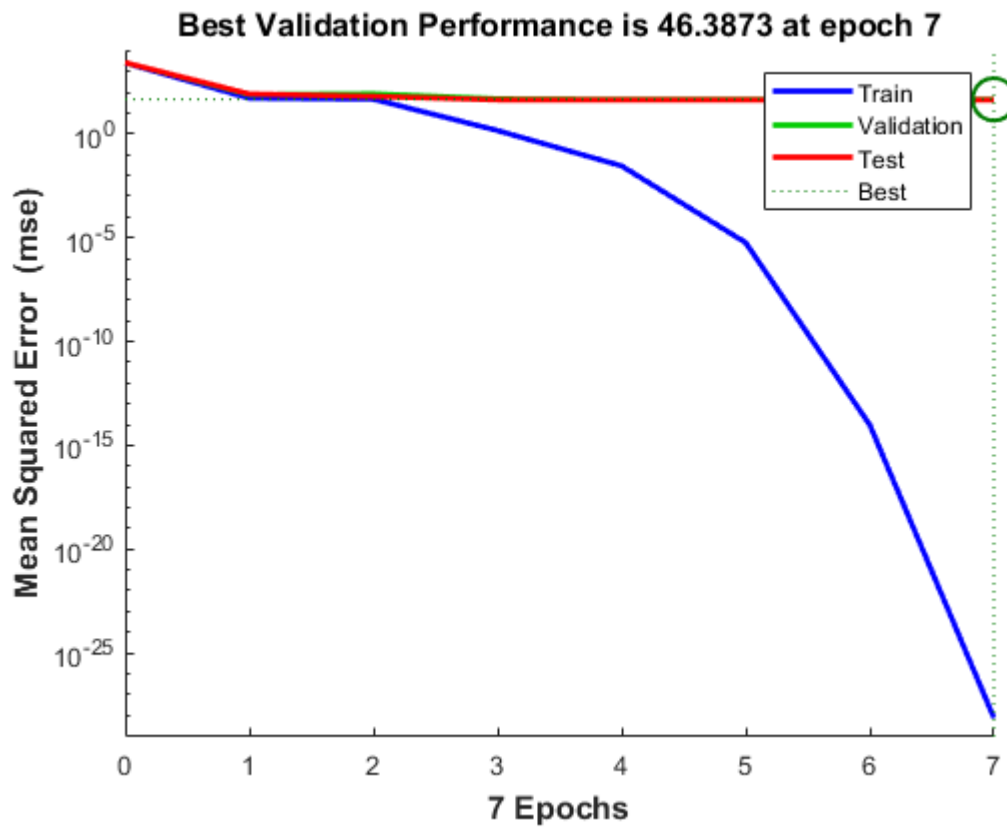


Los datos de test consiguen disminuir más fácilmente el error de los datos de entrenamiento pero no tiene datos para validar los resultados.

trainRatio = 10/100, valRatio = 50/100, testRatio = 40/100

Progress				
Epoch:	0	7 iterations		1000
Time:		0:00:00		
Performance:	2.64e+03	7.48e-29		0.00
Gradient:	7.93e+03	4.80e-13		1.00e-07
Mu:	0.00100	1.00e-10		1.00e+10
Validation Checks:	0	0		6

En este caso, destaca las pocas iteraciones que se han realizado en el entrenamiento.



En esta gráfica se concluye que el entrenamiento tiene que ser mayor porque no se obtienen buenos resultados con pocos datos de entrenamiento. Los datos de validación y test sí hacen un buen trabajo cuando éstos están compensados. Entonces la división que se ofrece por defecto al inicio del ejercicio, es una distribución bastante buena.

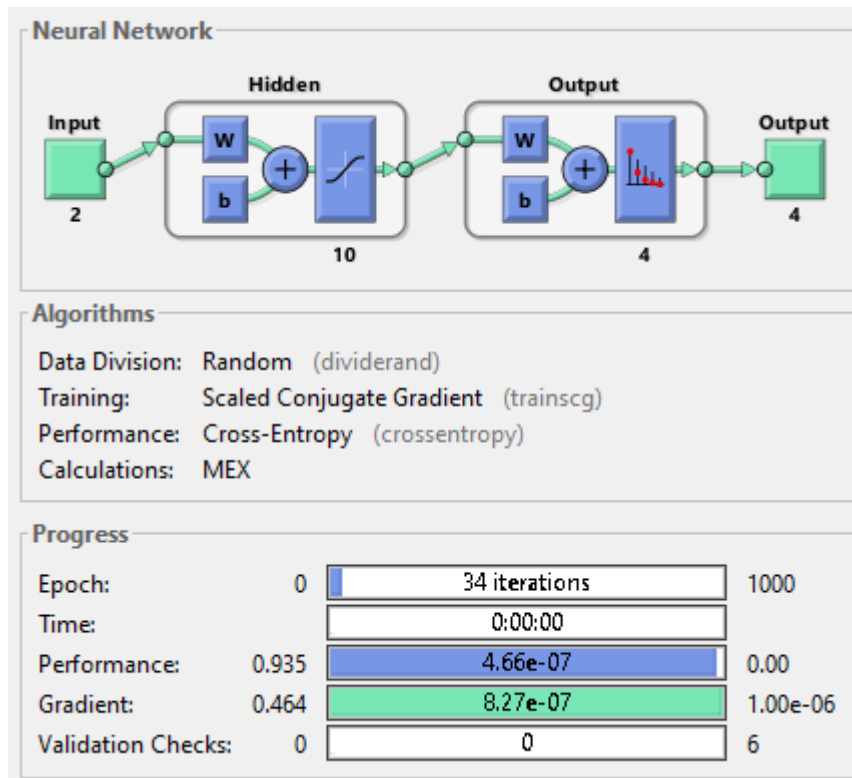
Ejercicio 4. Clasificación.

Código

```
1 % Ejercicio 4
2
3 clear all;
4 close all;
5 nnet.guis.closeAllViews();
6
7 % Carga de datos de ejemplo disponibles en la toolbox
8 [inputs,targets] = simpleclass_dataset; % conjunto de datos usados
9
10 % Creación de una red neuronal para el reconocimiento de patrones
11 hiddenLayerSize = 10;
12 net = patternnet(hiddenLayerSize); % método de entrenamiento usado
13
14 % División del conjunto de datos para entrenamiento, validación y test
15 net.divideParam.trainRatio = 70/100;
16 net.divideParam.valRatio = 15/100;
17 net.divideParam.testRatio = 15/100;
18
19 % Entrenamiento de la red
20 [net,tr] = train(net,inputs,targets);
21
22 % Prueba
23 outputs = net(inputs);
24 errors = gsubtract(targets,outputs);
25 performance = perform(net,targets,outputs)
26
27 % Visualización
28 view(net)
29
```

simpleclass_dataset

En este caso usaremos una red optimizada para la clasificación llamada *patternnet*. El conjunto de datos que vamos a usar es *simpleclass_dataset*. En lugar de las gráficas específicamente relacionadas con la aproximación de una función, en el caso de una tarea de clasificación, se ofrecen las gráficas que vamos a mostrar a continuación.



plotconfusion

Esta función de MATLAB traza una matriz de confusión para los objetivos de etiquetas verdaderas y las salidas de etiquetas predichas.

En la gráfica de matriz de confusión, las filas corresponden a la clase predicha (Output Class) y las columnas corresponden a la clase verdadera (Target Class). Las celdas diagonales corresponden a observaciones que están correctamente clasificadas. Las celdas fuera de la diagonal corresponden a observaciones clasificadas incorrectamente. Tanto el número de observaciones como el porcentaje del número total de observaciones se muestran en cada celda.

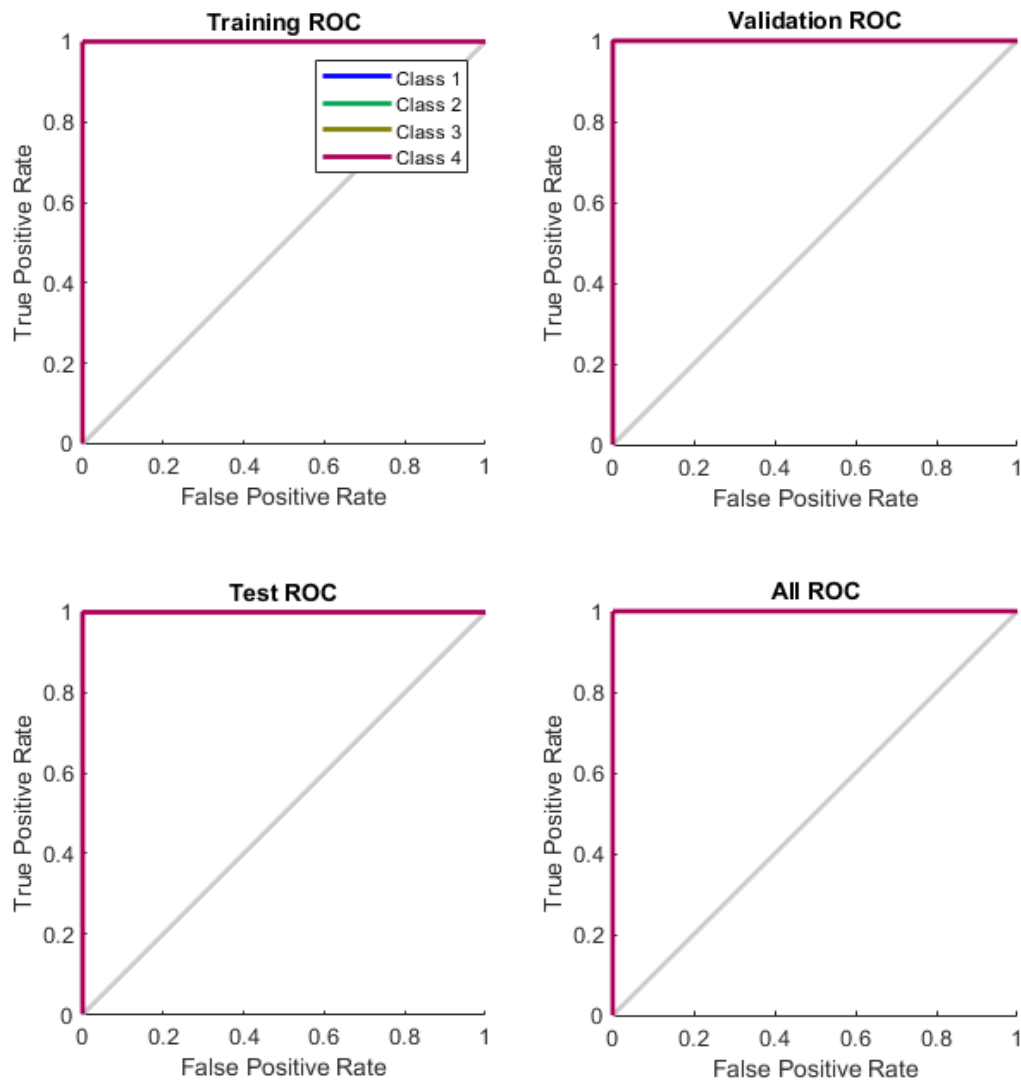
La columna en el extremo derecho del gráfico muestra los porcentajes de todos los ejemplos que se prevé que pertenezcan a cada clase que están clasificados correcta e incorrectamente. La fila en la parte inferior del gráfico muestra los porcentajes de todos los ejemplos pertenecientes a cada clase que están clasificados correcta e incorrectamente. La celda en la parte inferior derecha del gráfico muestra la precisión general.



Con las matrices obtenidas se puede ver que no hay ningún error a la hora de predecir las clases dándole los inputs.

plotroc

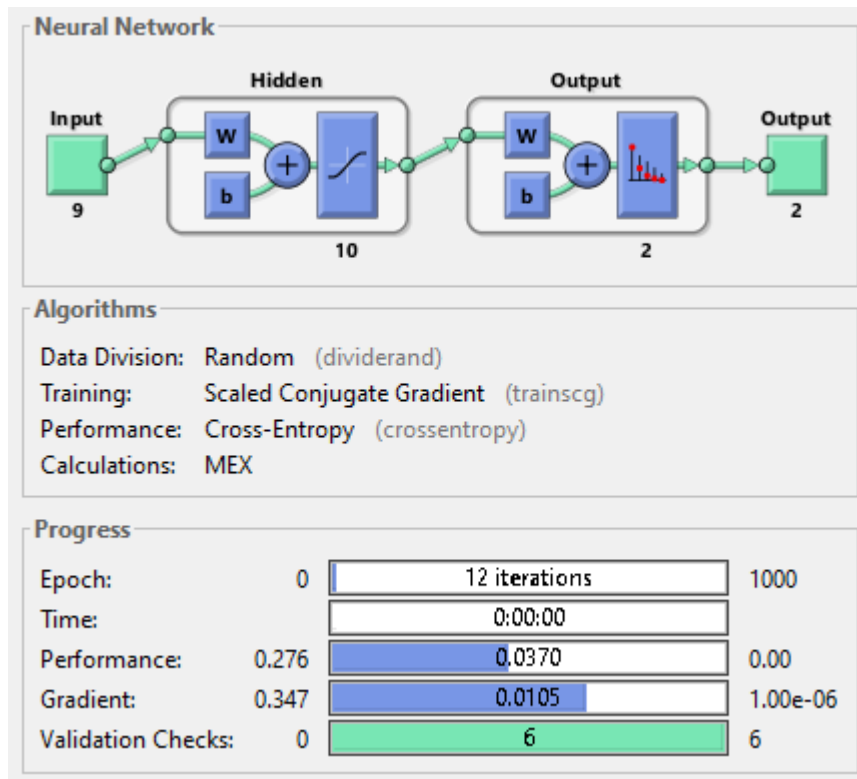
Esta gráfica traza la característica de funcionamiento del receptor para cada clase de salida. Cuanto más abrace cada curva a los bordes izquierdo y superior de la trama, mejor será la clasificación.



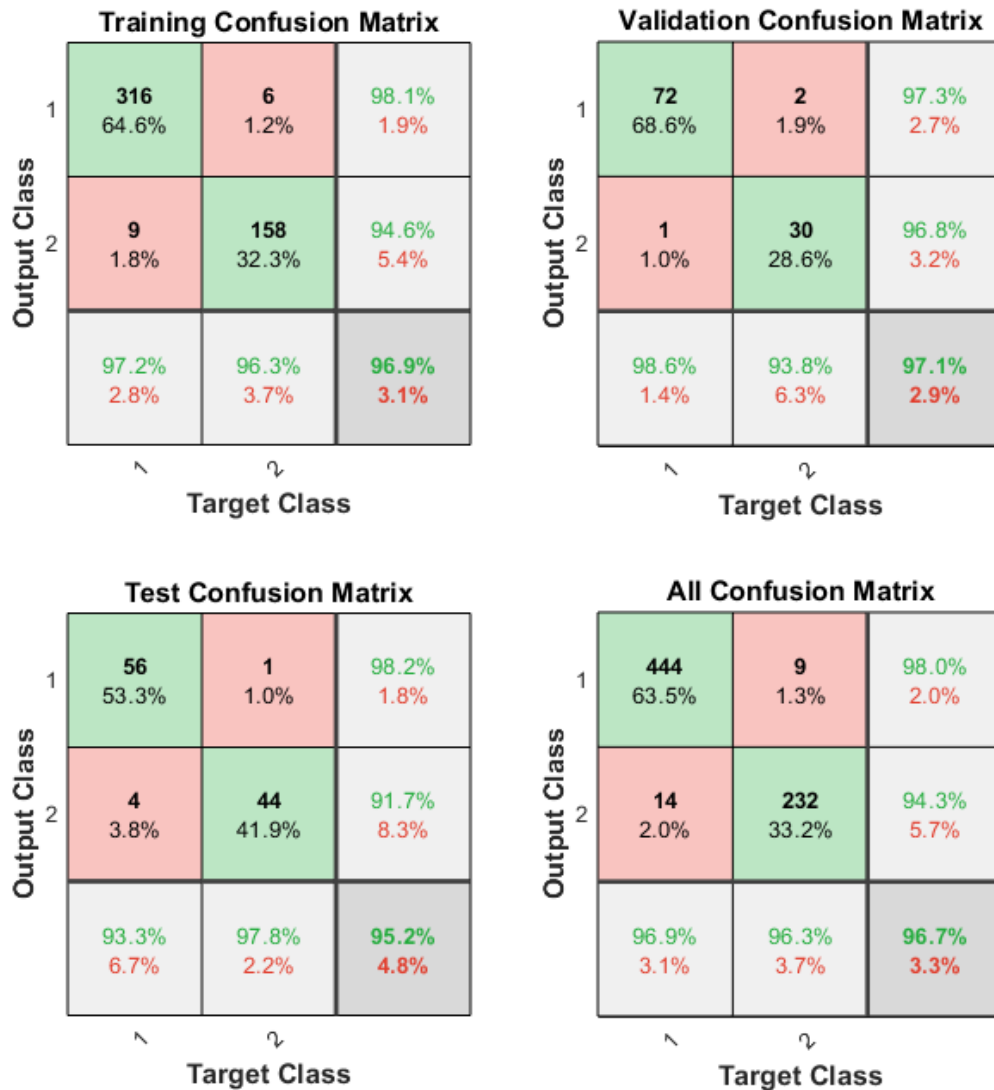
Por lo tanto, en estas gráficas se puede concluir que se realiza una clasificación muy buena.

cancer_dataset

El conjunto de datos que vamos a usar ahora es *cancer_dataset* y comparamos los resultados anteriores.

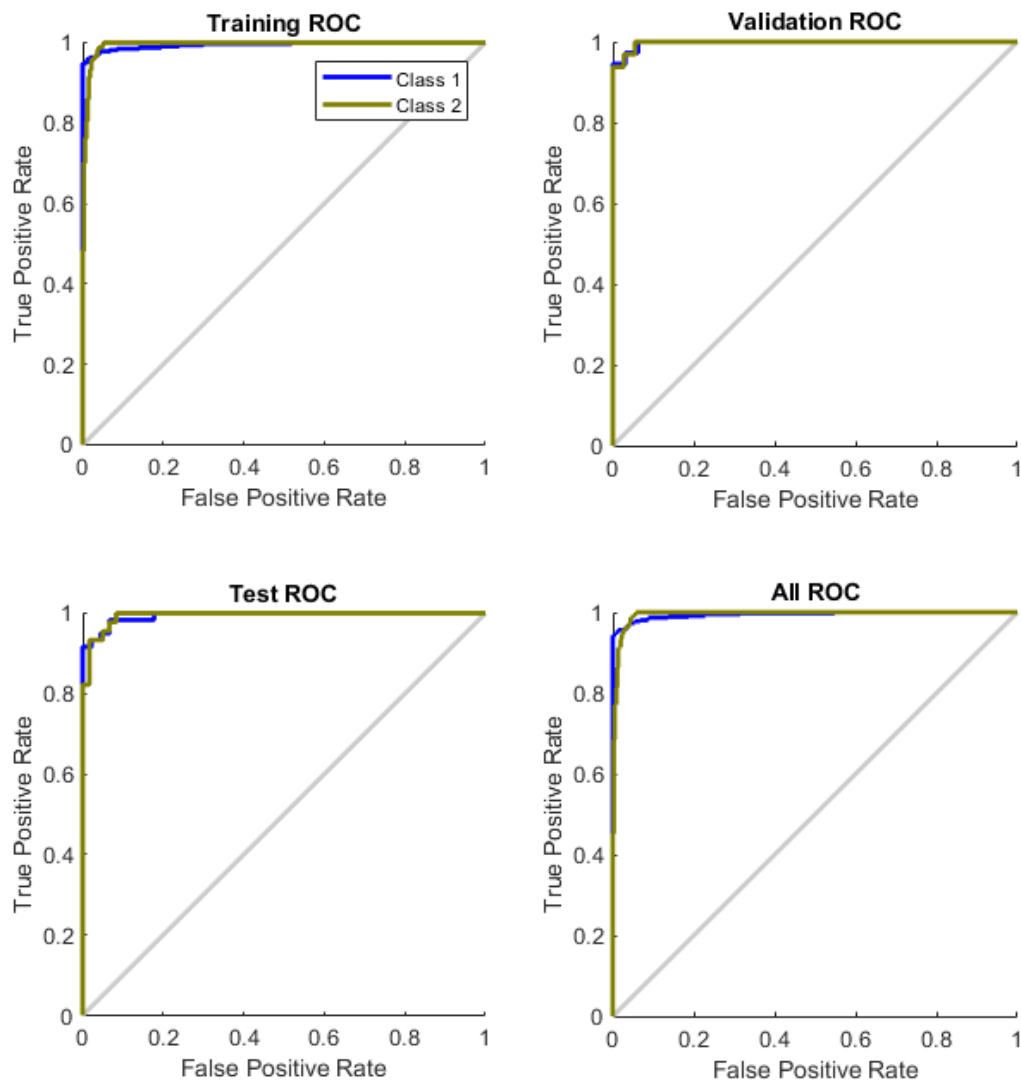


plotconfusion



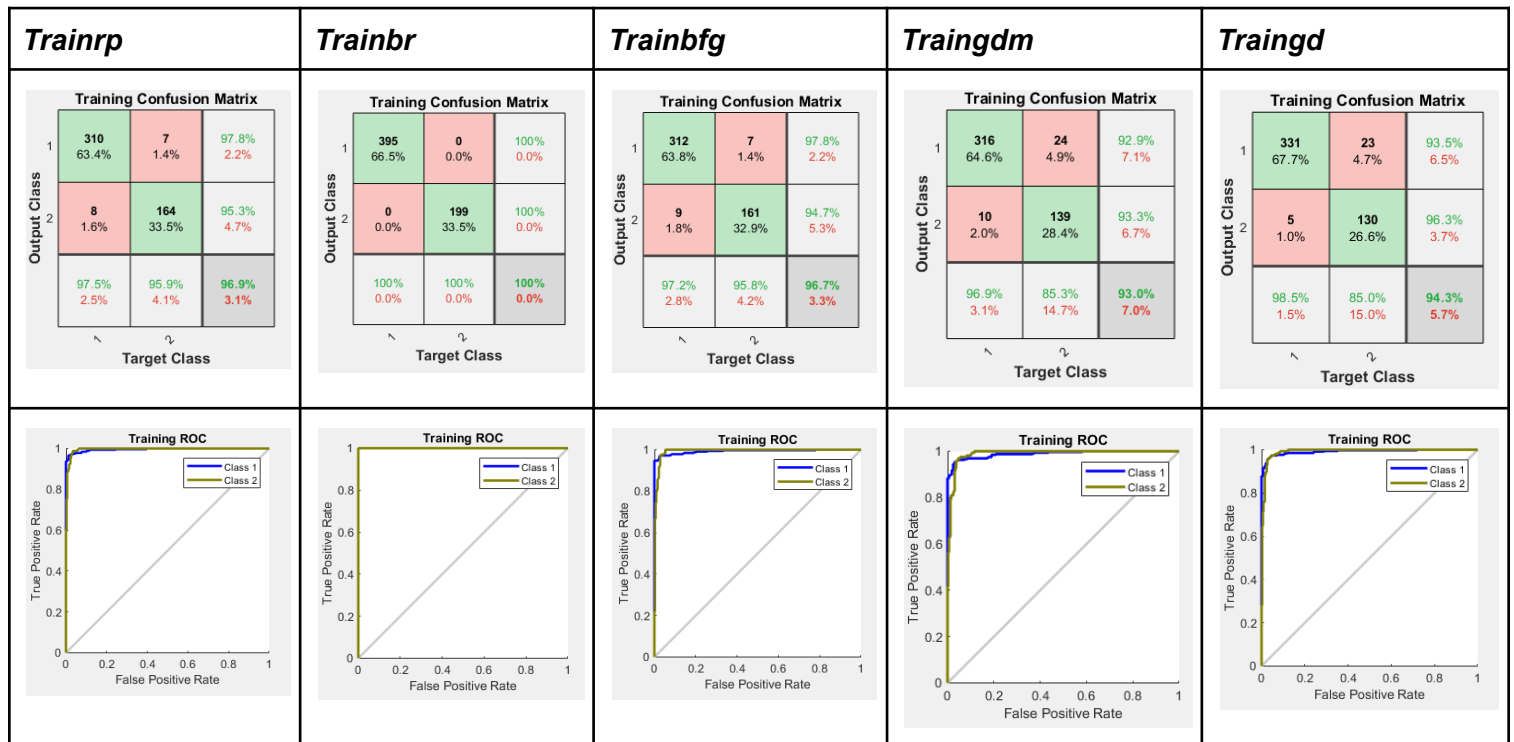
En estas matrices se puede observar como hay un pequeño porcentaje de datos que no se encuentran en las celdas diagonales, es decir, no corresponden a observaciones que están correctamente clasificadas.

plotroc



El pequeño porcentaje que se mencionaba anteriormente se refleja en estas gráficas en las que aparecen pequeñas perturbaciones en la curva de la esquina superior izquierda.

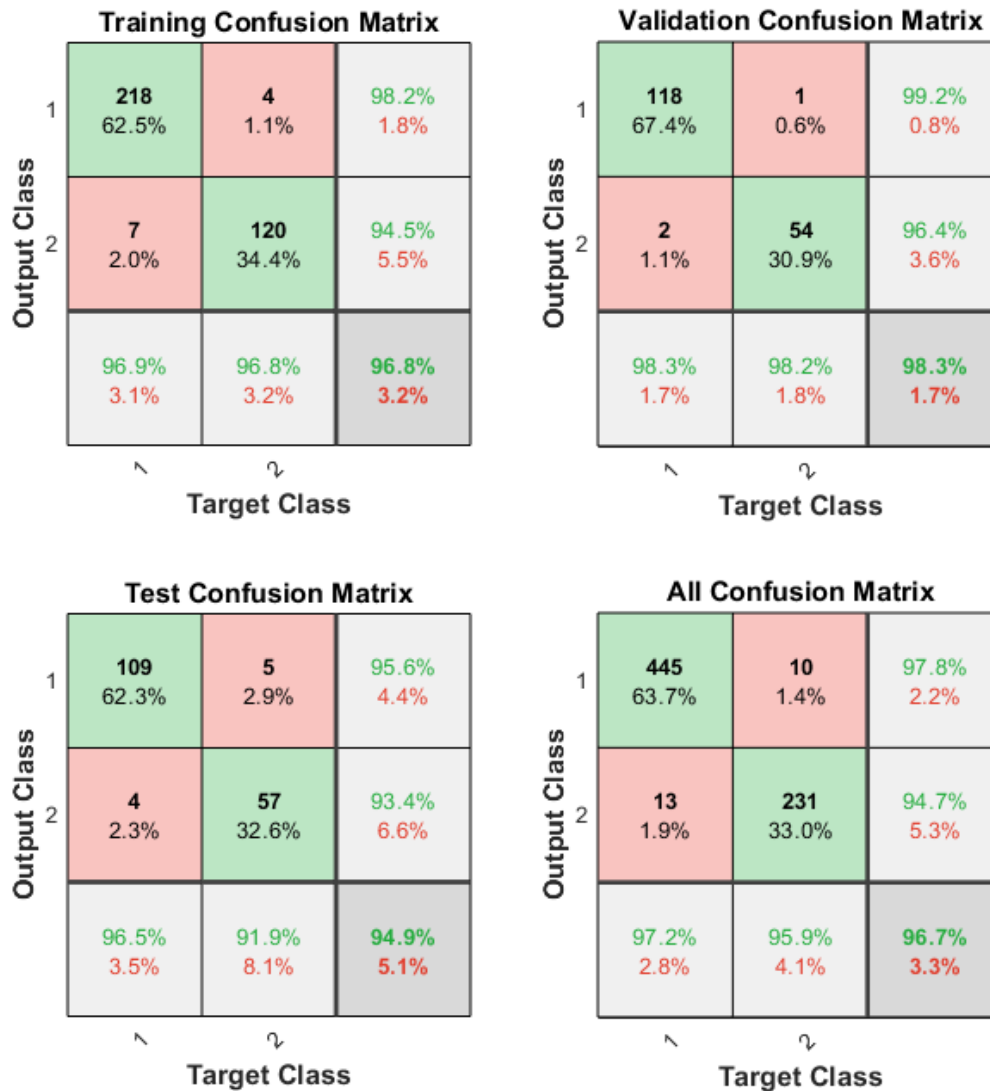
Cambio en el método de entrenamiento



Usando diferentes métodos de entrenamiento con el conjunto de datos anterior observamos cómo con el *trainbr* se consigue una clasificación perfecta de los elementos mientras que con el resto de entrenamientos aparecen las perturbaciones en las gráficas y algunos porcentajes de datos mal clasificados.

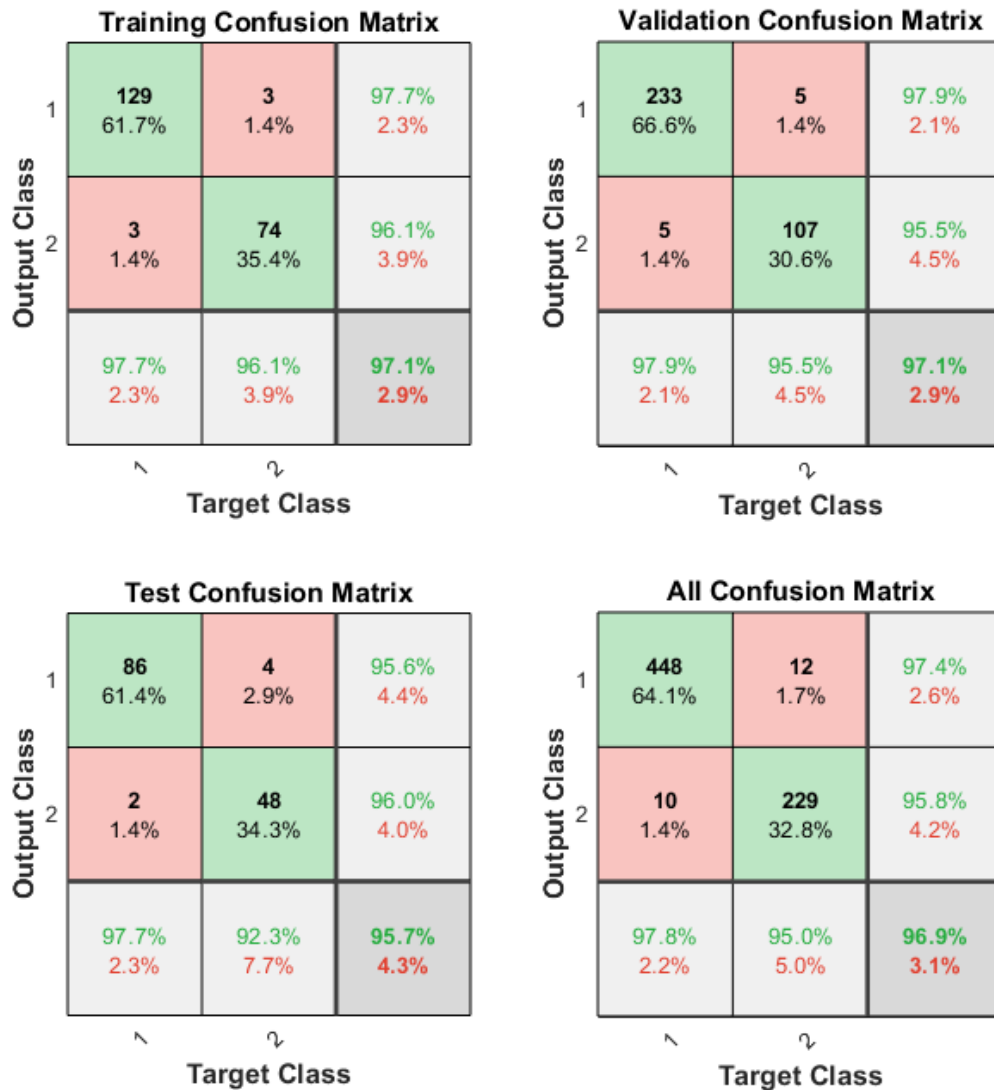
Modificación de la división de entrenamiento, validación y test

$trainRatio = 50/100$, $valRatio = 25/100$, $testRatio = 25/100$



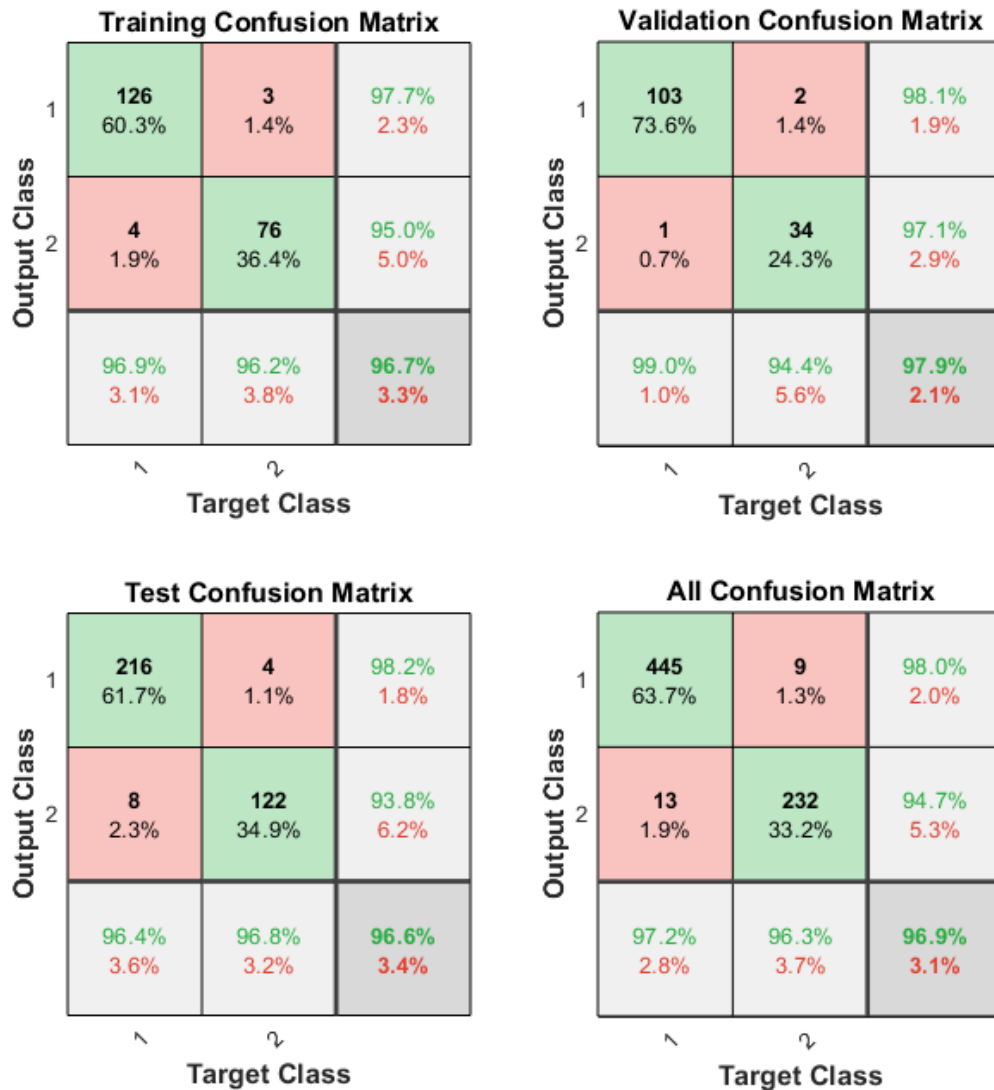
En las matrices de confusión observamos que el error aumenta ligeramente ya que los datos que tiene para entrenar son pocos en comparación con los datos que se usan en la comprobación de la predicción.

$trainRatio = 30/100$, $valRatio = 50/100$, $testRatio = 20/100$



Los errores son muy parecidos a la división anterior. Destacan los errores de los datos de test que disminuyen. Como anteriormente, sigue habiendo pocos datos para entrenar a la red por lo tanto no es un entrenamiento óptimo.

$trainRatio = 30/100$, $valRatio = 20/100$, $testRatio = 50/100$



En este caso aumentan los errores para los datos de test en comparación con los errores de los datos de entrenamiento y los datos de validación. Los datos de entrenamiento siguen siendo muy pocos, por lo tanto, la red no es capaz de encontrar más patrones para poder utilizarlos para clasificar los datos de test y los de validación.

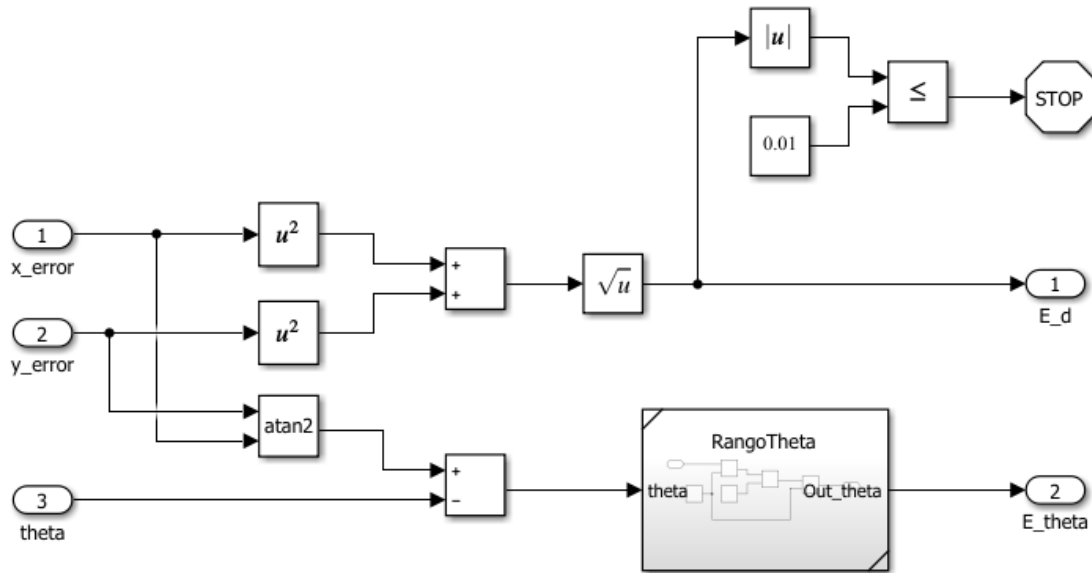
Parte II

Código

```
1 - clear all;
2 - close all;
3
4     % Tiempo de muestreo
5 - Ts = 100e-3;
6
7     % Referencia x-y de posicion
8 - refx = 5.0;
9 - refy = 5.0;
10
11    % Generar N posiciones aleatorias, simular y guardar en variables
12 - N = 20;
13 - E_d_vec = [];
14 - E_theta_vec = [];
15 - V_vec = [];
16 - W_vec = [];
17 - for i = 1:N
18 -     refx = 10 * rand - 5;
19 -     refy = 10 * rand - 5;
20
21     % Ejecutar Simulación
22 -     sim("PositionControl.slx")
23
24     E_d_vec = [E_d_vec ; E_d.signals.values];
25     E_theta_vec = [E_theta_vec ; E_theta.signals.values];
26     V_vec = [V_vec ; V.signals.values];
27     W_vec = [W_vec ; W.signals.values];
28 - end
29 - inputs = [E_d_vec' ; E_theta_vec'];
30 - outputs = [V_vec' ; W_vec'];
31
32    % Entrenar red neuronal de 13 neuronas en la capa oculta
33 - net = feedforwardnet([13]);
34 - net = configure(net, inputs, outputs);
35 - net = train(net, inputs, outputs);
36
37    % Generar bloque de Simulink con el controlador neuronal
38 - gensim(net,Ts)
39
40    % Mostrar
41 - x = salida_x.signals.values;
42 - y = salida_y.signals.values;
43
44 - figure;
45 - plot(x, y);
46 - grid on;
47 - hold on;
```

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

a) Esquema general de un control de posición.



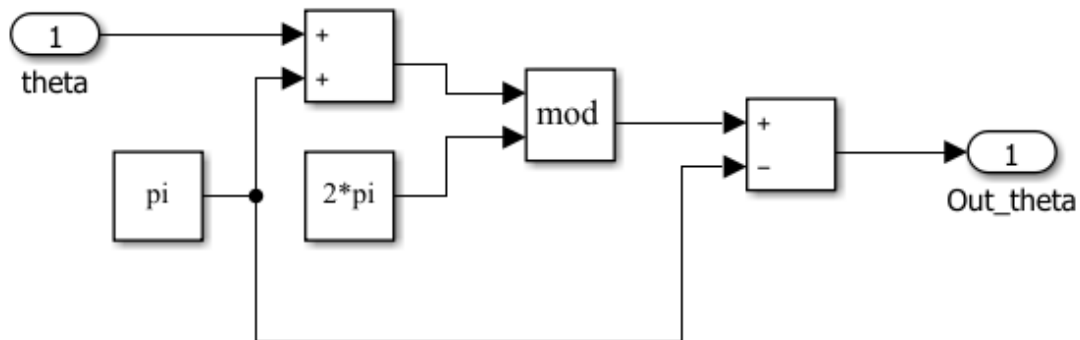
PositionErrors.slx

Este es el esquema de Simulink diseñado para calcular los errores con los que se realiza el control de posición del robot. Estos errores vienen dados por las siguientes fórmulas:

$$E_d = \text{sqrt}((refx - x_k)^2 + (refy - y_k)^2)$$

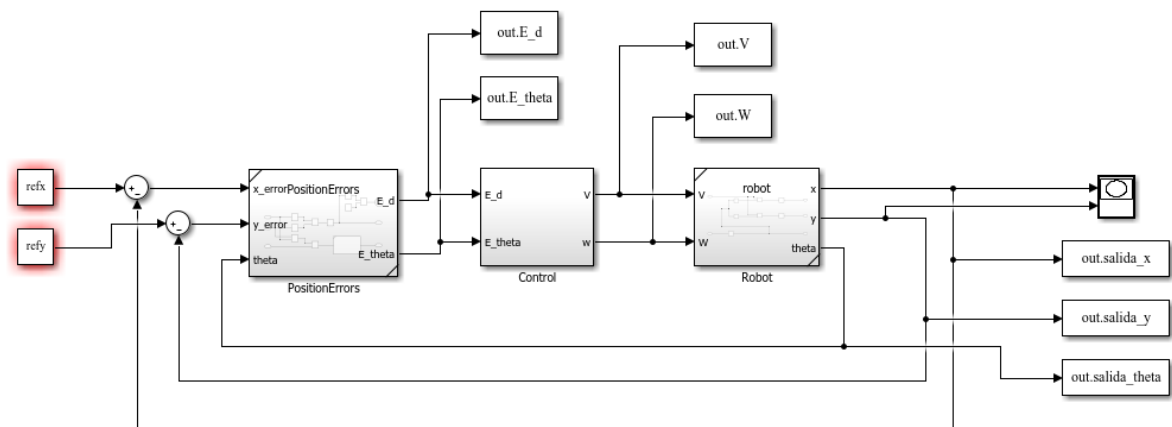
$$E_\theta = \text{atan2}(refy - y_k, refx - x_k) - \theta_k$$

Se añade una comprobación para que E_θ siempre tenga un valor comprendido en el intervalo $[-\pi, \pi]$. Esta comprobación y, en caso necesario, transformación se realiza dentro del subsistema RangoTheta que se define en el siguiente esquema:



RangoTheta.slx

Para transformar θ a un ángulo dentro del rango deseado $[-\pi, \pi]$, primero le sumamos π para que quede en el rango $[0, 2\pi]$. En el caso de que el ángulo no pertenezca a este intervalo, aplicando el módulo de 2π , nos dará como resultado un ángulo comprendido entre esos valores. Luego solo falta volver a restar π para volver al rango deseado $[-\pi, \pi]$.

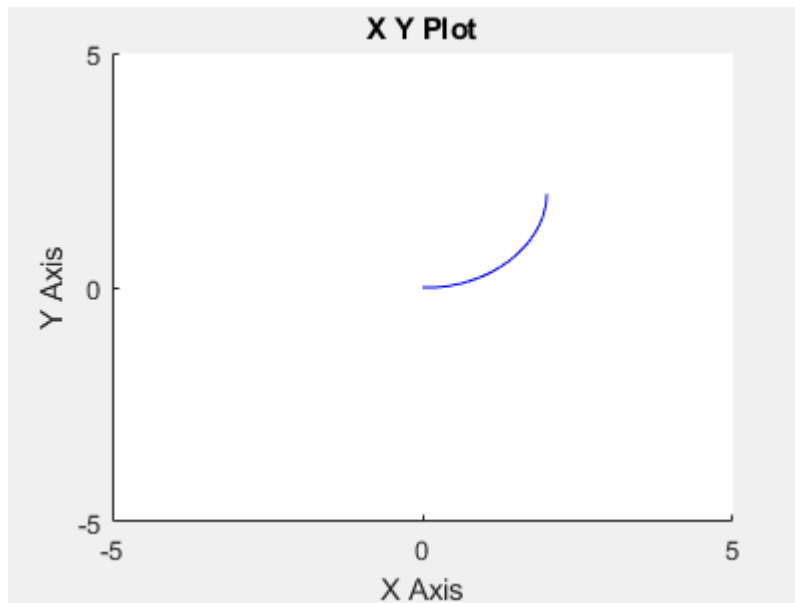


PositionControl.slx

Finalmente obtenemos el esquema general de un control de posición uniendo los bloques de PositionErrors, Control y Robot como se muestra en la imagen.

b) Simular el diagrama PositionControl.

Creamos un script con el nombre "RunPositionControl.m" y añadiendo el código que se nos proporciona. En la salida de la ejecución nos aparece la siguiente gráfica:



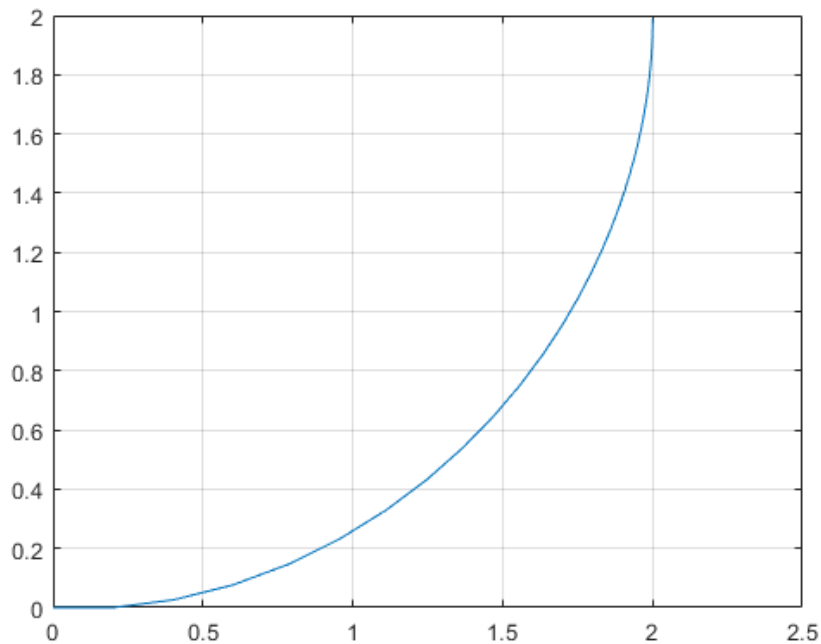
c) Ejecutar el script RunPositionControl.

En la ejecución también comprobamos que se generan las variables que contienen las salidas y entradas del controlador y las salidas del robot durante la simulación. En el workspace aparecen las variables E_d, E_theta, V y W y las variables salida_x, salida_y y salida_theta, respectivamente.

Workspace		
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	58x1 double	
Ts	0.1000	
V	1x1 struct	
W	1x1 struct	
x	58x1 double	
y	58x1 double	

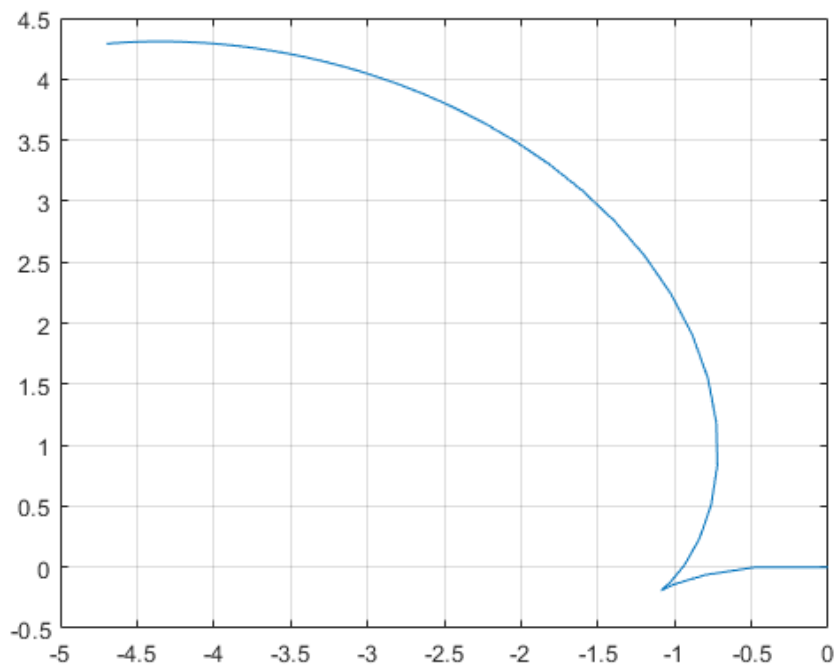
d) Trayectoria del robot.

Añadimos el código que se nos da en el script anterior para mostrar la trayectoria del robot mediante el comando plot. Su ejecución nos produce la siguiente gráfica de salida:



e) Varias simulaciones del controlador.

Realizamos 30 simulaciones y guardamos el valor a lo largo del tiempo de las entradas (E_d y E_{θ}) en una matriz de "inputs" y salidas (V y W) en una matriz de "outputs" del bloque controlador. La salida de la última simulación es la siguiente gráfica:



f) Red neuronal con una capa oculta.

Para encontrar el número de neuronas de la capa oculta hacemos varias pruebas de experimentación. Esta tabla refleja los resultados obtenidos.

Neuronas	Best Validation Performance
1	0.66119 en la época 12
2	0.028973 en la época 383
3	0.014816 en la época 30
4	0.00308 en la época 127
5	0.0011347 en la época 26
6	0.0044599 en la época 41
7	0.0016179 en la época 171
8	0.0020556 en la época 117
9	0.000139 en la época 32
10	0.00075801 en la época 274
11	0.00035525 en la época 74
12	7.8579e-05 en la época 188
13	4.5503e-08 en la época 661
14	2.9537e-05 en la época 162
15	1.7539e-06 en la época 1000
16	1.2441e-06 en la época 1000
17	0.0028912 en la época 5
18	1.0874e-05 en la época 263
19	0.00040982 en la época 150
20	0.0001591 en la época 84

El valor elegido es el de 13 neuronas porque es el que menor valor de validation performance tiene y, por lo tanto, el mejor.

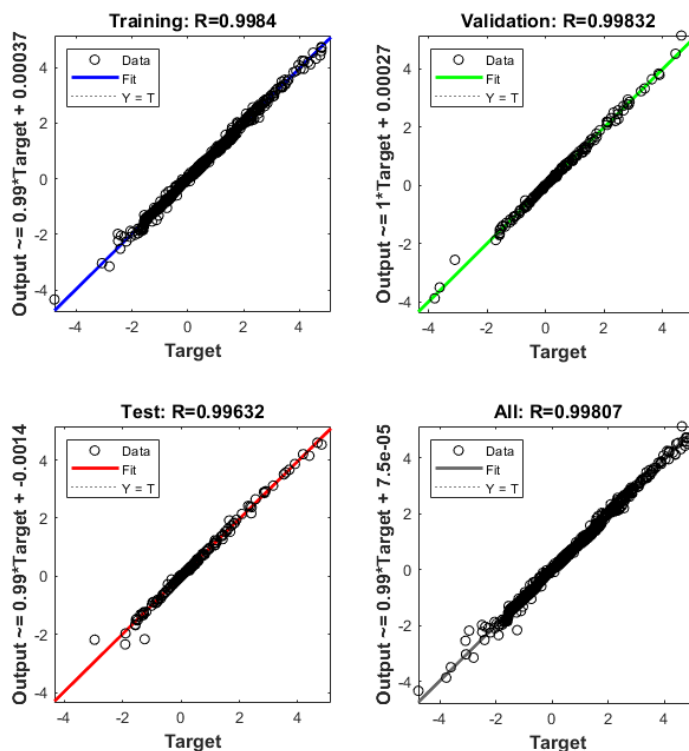
Para obtener estos datos, se usa el siguiente código. Con él conseguimos 20 gráficas y de ellas, el mejor valor de validation performance y la época.

```

32 % Entrenar red neuronal de 1 a 20 neuronas en la capa oculta
33 - for i = 1:20
34 -     net = feedforwardnet([i]);
35 -     net = configure(net, inputs, outputs);
36 -     [net, tr] = train(net, inputs, outputs);
37 -     figure;
38 -     plotperform(tr);
39 - end

```

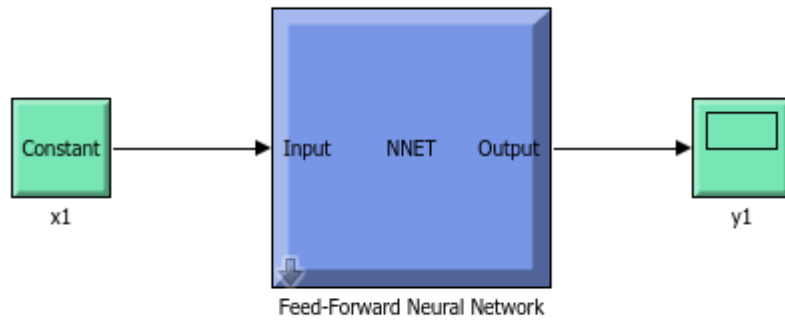
Una vez elegido el mejor número de neuronas, ejecutamos nuevamente y observamos la gráfica de regresión para discutir los resultados.



Se puede ver cómo los datos de entrenamiento, de validación y de test se ajustan bastante bien a la recta de regresión. Se puede decir que los datos apenas se alejan de las rectas calculadas y por lo tanto es un buen entrenamiento.

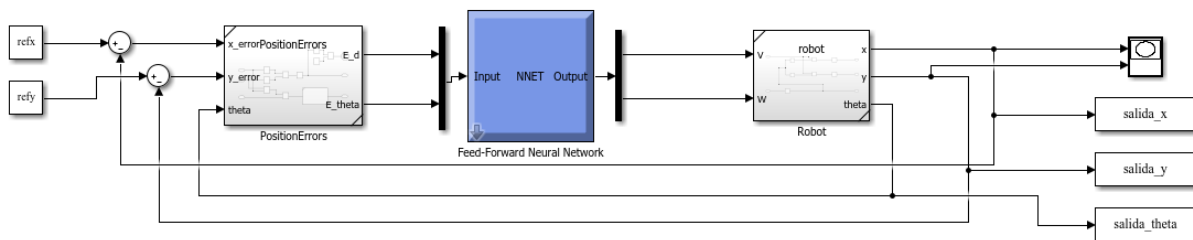
g) Un bloque con la red neuronal.

Generamos, mediante la orden gensim de Matlab, el siguiente bloque con la red neuronal.



h) Utilizar la red neuronal.

Cambiamos el bloque controlador y colocamos en su lugar la red neuronal. Utilizamos bloques multiplexores y demultiplexores para adaptar las señales de entrada y salida..

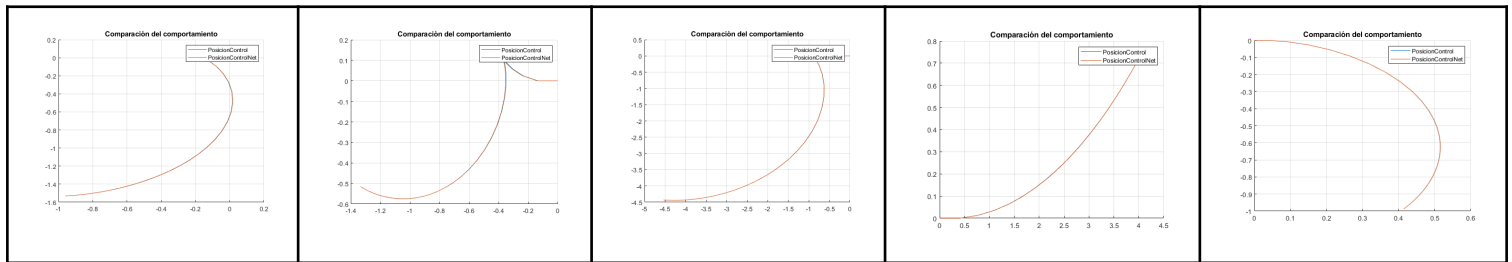


i) Comparar el comportamiento.

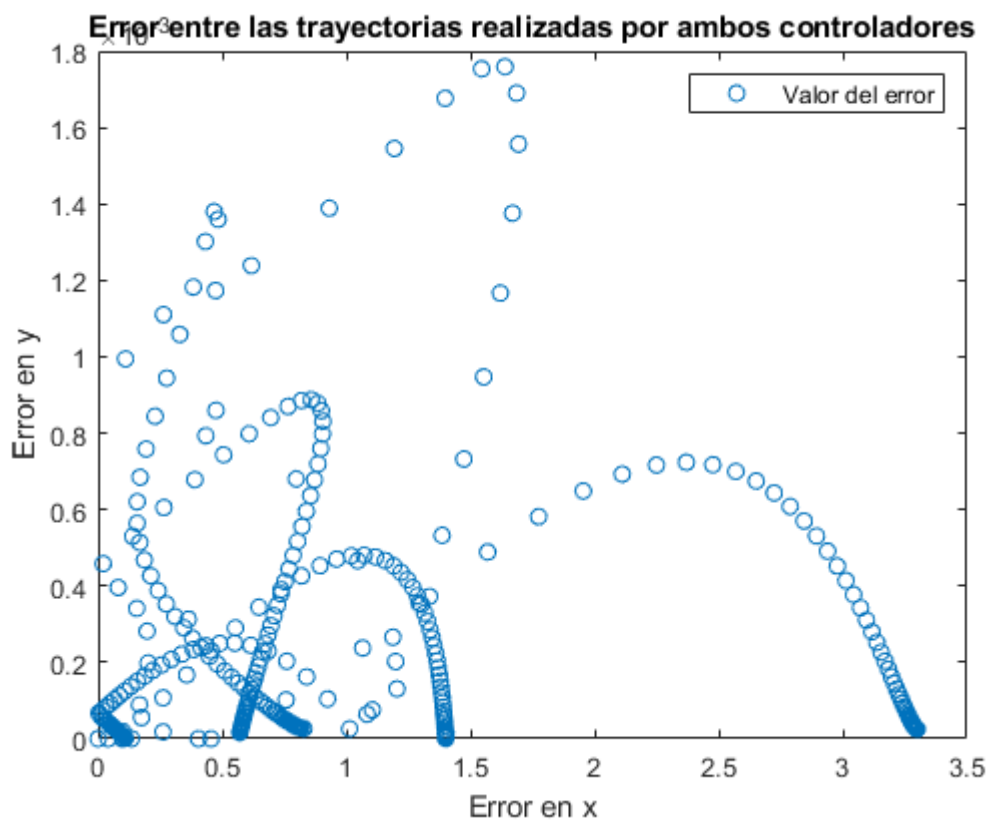
Para realizar la comparación, usamos el siguiente código. También calculamos el error entre las trayectorias realizadas por ambos controladores.

```
1 - clear all;
2 - close all;
3
4 - % Tiempo de muestreo
5 - Ts = 100e-3;
6
7 - % Generar N posiciones aleatorias, simular y guardar en variables
8 - N = 5;
9 - errores_x = [];
10 - errores_y = [];
11
12 - for i = 1:N
13 -     refx = 10 * rand - 5;
14 -     refy = 10 * rand - 5;
15
16 -     % Ejecutar Simulación de PosicionControl
17 -     sim("PositionControl.slx")
18 -     x = salida_x.signals.values;
19 -     y = salida_y.signals.values;
20
21 -     % Ejecutar Simulación de PosicionControlNet
22 -     sim("PositionControlNet.slx")
23 -     x_net = salida_x.signals.values;
24 -     y_net = salida_y.signals.values;
25
26 -     figure(i);
27 -     hold on;
28 -     posicionControl = plot(x, y);
29 -     posicionControlNet = plot(x_net, y_net);
30 -     hold off;
31 -     grid on;
32 -     legend("PosicionControl", "PosicionControlNet");
33 -     title("Comparación del comportamiento");
34
35 -     error_x = [];
36 -     error_y = [];
37 -     for j = 1:min(length(x), length(x_net))
38 -         error_x = [error_x ; abs(x(j) - y_net(j))];
39 -         error_y = [error_y ; abs(y(j) - y_net(j))];
40 -     end
41 -     errores_x = [errores_x ; error_x];
42 -     errores_y = [errores_y ; error_y];
43 - end
44
45 - figure(i + 1);
46 - errores = plot(errores_x, errores_y, 'o');
47 - title('Error entre las trayectorias realizadas por ambos controladores');
48 - legend('Valor del error');
49 - xlabel('Error en x');
50 - ylabel('Error en y');
```

Para una mejor comparación de las gráficas, las colocamos en una tabla.



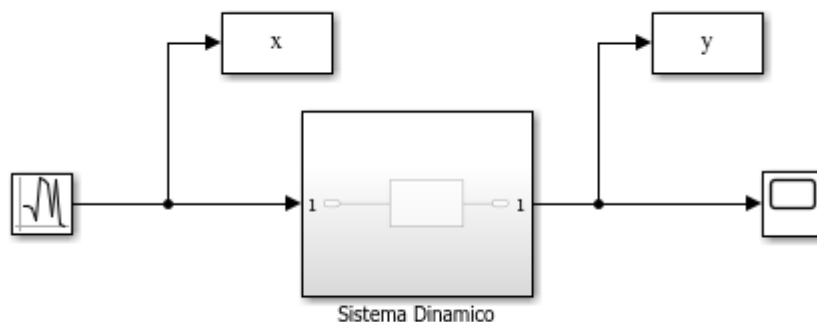
En todos los casos se sigue bastante bien la trayectoria deseada aunque no es perfecta. Esto lo podemos saber por la siguiente gráfica de los errores entre las trayectorias realizadas por ambos controladores.



Parte III

Ejercicio 1. Identificación de un sistema utilizando una red recursiva de tipo NARX

Se diseña el archivo en el que se simula el comportamiento del sistema ante una entrada aleatoria y se almacenan la variable de entrada al sistema “x” y la variable de salida “y”. El sistema se define por el siguiente diagrama:



test_bench.slx

Mediante el siguiente código se ejecuta la simulación y se generan los arrays de inputs y outputs con las entradas y salidas del sistema a lo largo de la simulación.

```
1 - clear all;
2 - close all;
3
4   % Generación de datos de simulación
5 - Ts = 0.1;
6 - sim('test_bench.slx')
7 - inputs = x.signals.values';
8 - outputs = y.signals.values';
```

Los valores guardados podemos consultarlos en el workspace.

Workspace	
Name ▲	Value
inputs	1x301 double
outputs	1x301 double
tout	301x1 double
Ts	0.1000
x	1x1 struct
y	1x1 struct

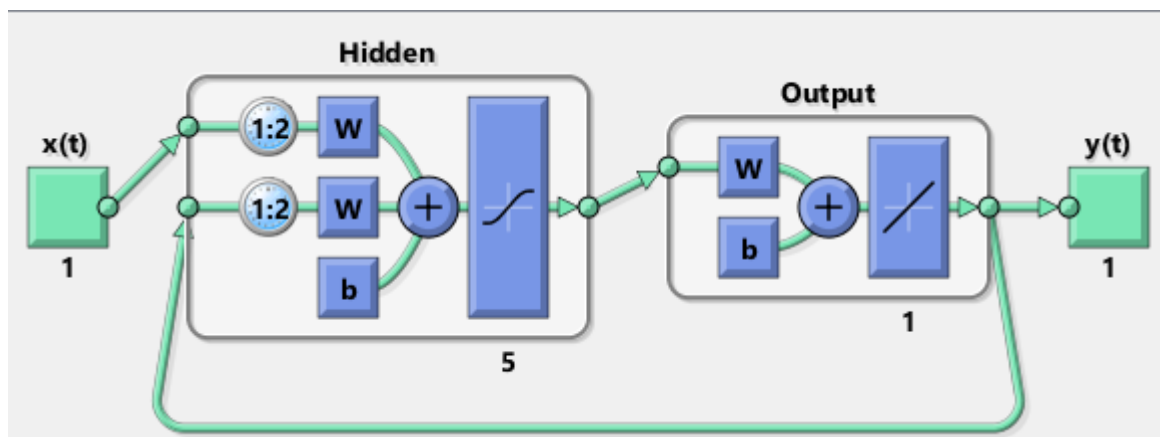
Ahora se crea una red NARX con 5 neuronas en la capa oculta, 2 retardos en la entrada y en la salida realimentada. La red va a ser de tipo feedforward, por lo que hay que transformar las entradas y las salidas. Luego se entrena la red y se convierte en recursiva. Todo ello, se visualiza en el siguiente código.

```

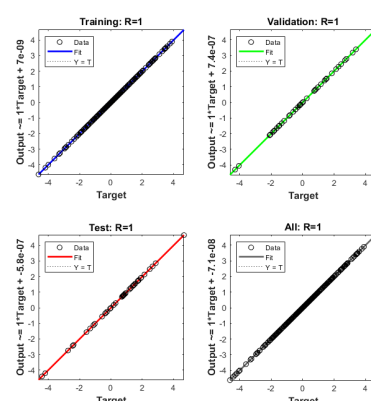
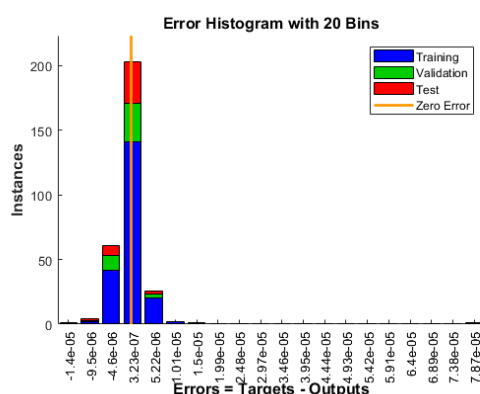
10 % Definición del modelo NARX
11 - N = 5;
12 - net = narxnet(1 : 2, 1 : 2, [N]);
13 - view(net)
14
15 - nT = size(inputs, 2);
16 - inputsc = mat2cell(inputs, 1, ones(nT, 1));
17 - outputsc = mat2cell(outputs, 1, ones(nT, 1));
18
19 - [x, xi, ai, t] = preparets(net, inputsc, {}, outputsc);
20
21 - net = train(net, x, t, xi, ai);
22 - net = closeloop(net);
23 - view(net)

```

La red obtenida es la siguiente:

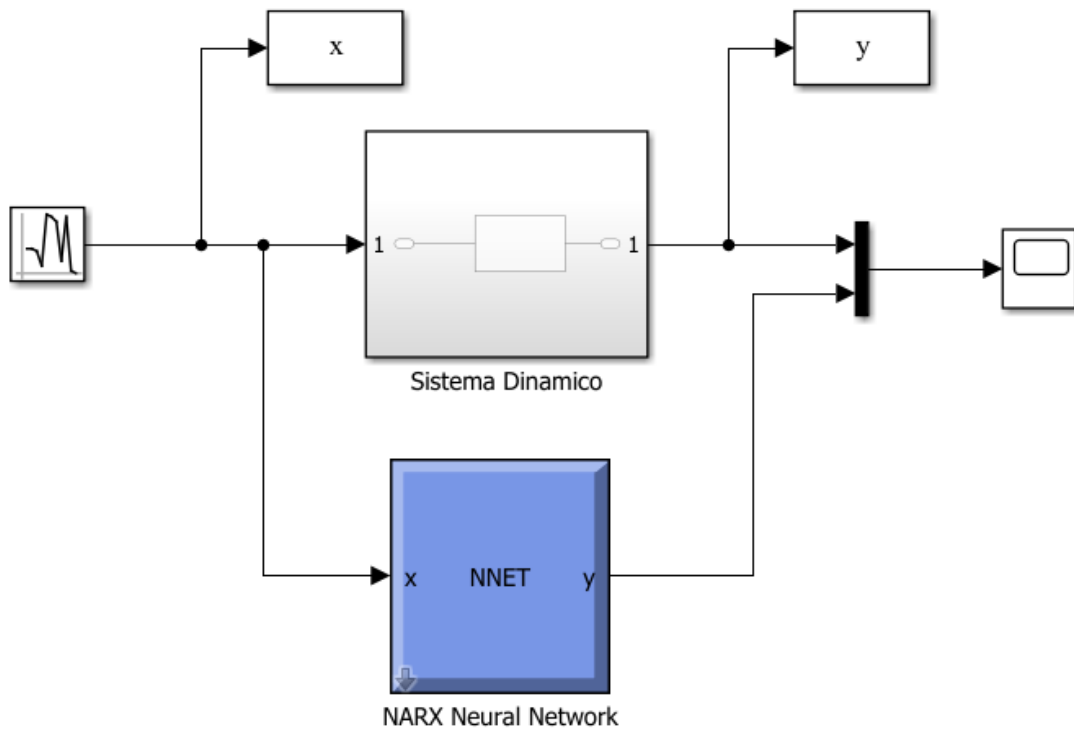


Mirando el error de entrenamiento, de validación y de test, podemos ver que ha sido un buen entrenamiento.

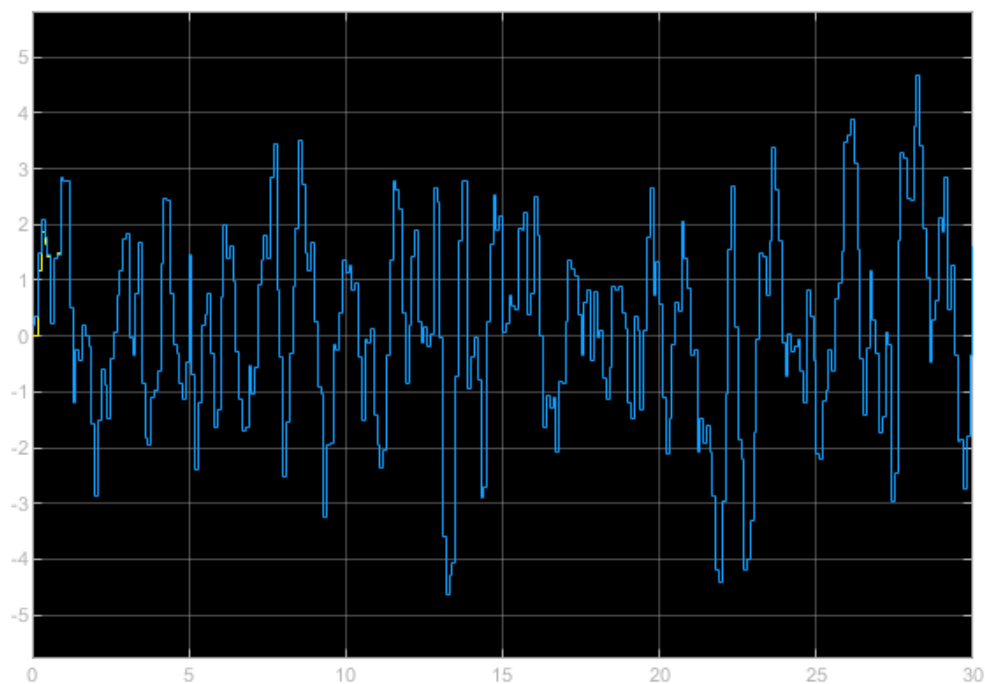


En el histograma de los errores podemos ver que están bastante aproximados al error cero. Esto se puede confirmar cuando en las gráficas de regresión vemos que los datos están muy próximos a las rectas calculadas.

Ahora generamos el modelo de Simulink de la red entrenada mediante el comando gensim.



Visualizamos el error entre la red neuronal y el sistema mediante el “scope” de dos canales y obtenemos la siguiente gráfica.

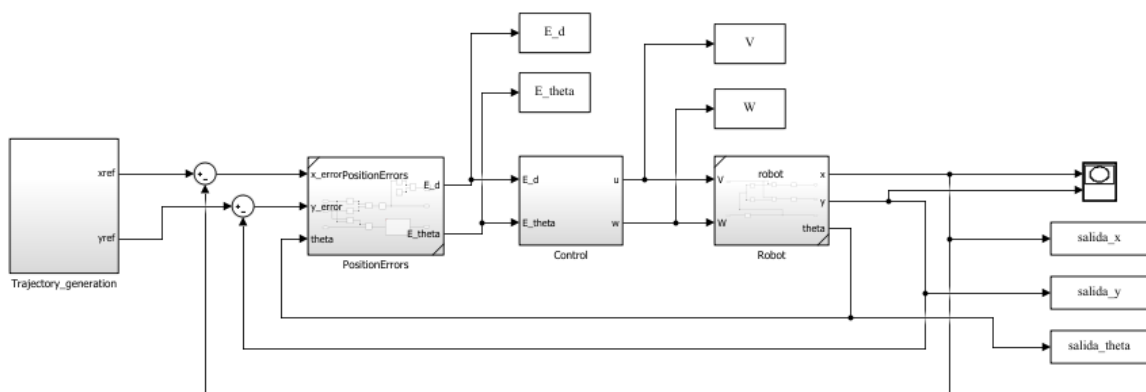


Ejercicio 2. Control para seguimiento de trayectorias mediante redes recurrentes.

Desarrollo del ejercicio.

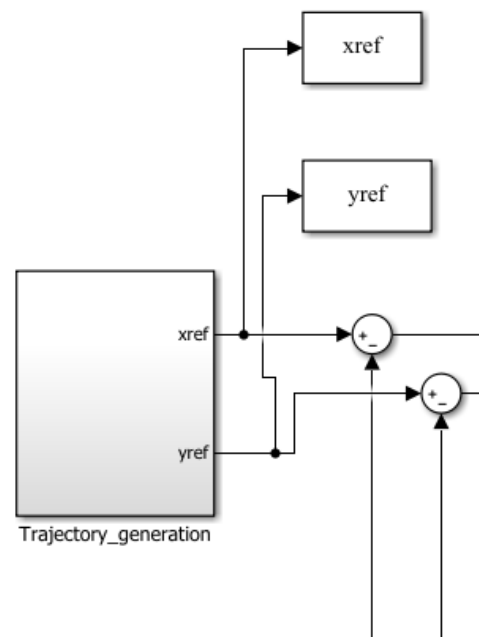
a) Implementar el esquema

Utilizamos como controlador el proporcionado en el archivo “controlblackboxTrajectory.slx”. Usamos los parámetros de la simulación de sesiones anteriores. Guardamos el esquema de Simulink con el nombre TrajectoryControl.slx. Su diseño es el siguiente:



b) Generar el script

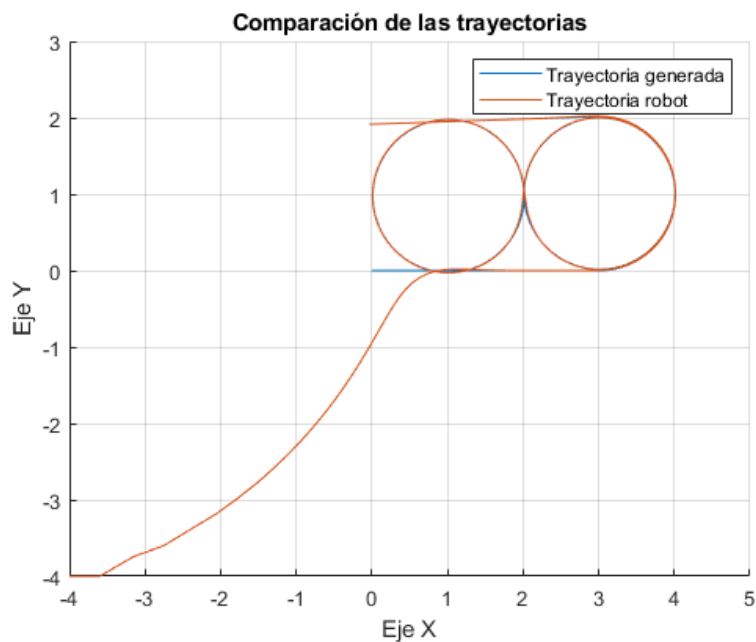
Añadimos bloques al inicio del esquema para poder observar en el workspace de Matlab el valor de las referencias de posición, x_{ref} e y_{ref} .



Generamos el script de Matlab llamado "RunTrajectoryControl.m" donde se muestra el seguimiento de la trayectoria seguida por el robot.

```
1 - clear all;
2 - close all;
3
4 - % Tiempo de muestreo
5 - Ts = 100e-3;
6
7 - % Parámetros iniciales de la trayectoria
8 - x_0 = 0;
9 - y_0 = 0;
10 - th_0 = 0;
11
12 - % Ejecutar Simulación
13 - sim('TrajectoryControl.slx');
14
15 - xref = salida_xref.signals.values';
16 - yref = salida_yref.signals.values';
17 - x = salida_x.signals.values';
18 - y = salida_y.signals.values';
19
20 - figure(1);
21 - hold on;
22 - generada = plot(xref, yref);
23 - robot = plot(x, y);
24 - hold off;
25 - grid on;
26 - legend('Trayectoria generada', 'Trayectoria robot');
27 - title('Comparación de las trayectorias');
28 - xlabel('Eje X');
29 - ylabel('Eje Y');
```

Con el comando plot añadido en el código podemos visualizar la trayectoria del robot y también compararla con la trayectoria generada. En la siguiente gráfica se puede observar ambos recorridos.



c) Diseñar y entrenar una red neuronal recursiva de tipo “narx”

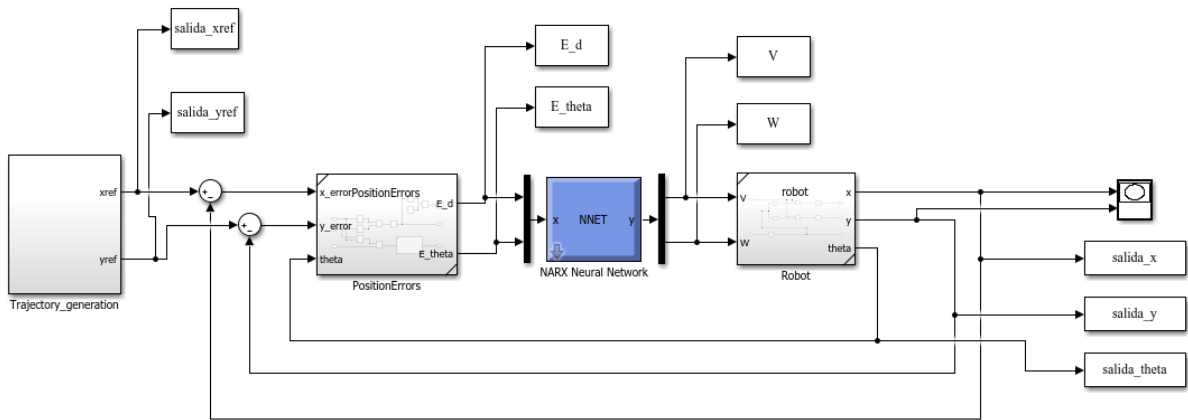
Ahora se crea una red NARX con una capa oculta, dos retardos en la entrada y un retardo en la realimentación de la salida. Se entrena la red y se convierte en recursiva. Todo ello, se visualiza en el siguiente código del script RunTrajectoryControl2.m.

```

1 - clear all;
2 - close all;
3
4 - % Tiempo de muestreo
5 - Ts = 100e-3;
6
7 - % Parámetros iniciales de la trayectoria
8 - th_0 = 30*pi/180;
9 - x_0 = 0;
10 - y_0 = 0;
11 - N = 1;
12
13 - E_d_vec=[];
14 - E_theta_vec=[];
15 - V_vec=[];
16 - W_vec=[];
17
18 - for num = 0:N
19 -     for num2 = 0:N
20 -         x_0 = num;
21 -         y_0 = num2;
22 -         % Ejecutar Simulación
23 -         sim('TrajectoryControl.slx');
24 -         E_d_vec=[E_d_vec;E_d.signals.values];
25 -         E_theta_vec=[E_theta_vec;E_theta.signals.values];
26 -         V_vec=[V_vec; V.signals.values];
27 -         W_vec=[W_vec; W.signals.values];
28 -     end
29 - end
30
31 - inputs=[E_d_vec'; E_theta_vec'];
32 - outputs=[V_vec'; W_vec'];
33
34 - % Definición del modelo NARX
35 - Ne = 10;
36 - net = narxnet(1 : 2, 1, [Ne]);
37 - view(net)
38
39 - nT = size(inputs, 2);
40 - inputsc = mat2cell(inputs, 2, ones(nT, 1));
41 - outputsc = mat2cell(outputs, 2, ones(nT, 1));
42
43 - [x, xi, ai, t] = preparets(net, inputsc, {}, outputsc);
44
45 - net = train(net, x, t, xi, ai);
46 - net = closeloop(net);
47 - view(net)
48
49 - gensim(net,Ts)

```

El diseño de la red neuronal recursiva de tipo "narx" que sirve para emular el comportamiento del controlador proporcionado es el siguiente:



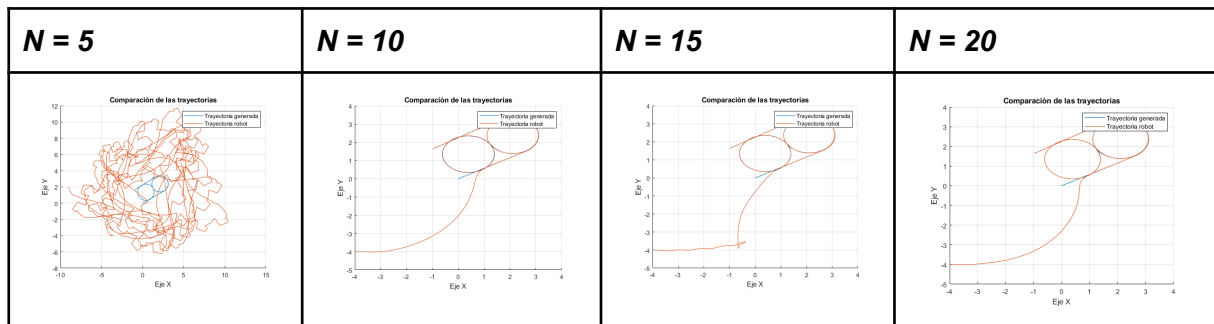
Usamos RunTrajectoryControlNet.m para poder ejecutar el nuevo diseño con la red neuronal sustituyendo al bloque de control. El código es el siguiente:

```

1 - clear all;
2 - close all;
3
4 - % Tiempo de muestreo
5 - Ts = 100e-3;
6
7 - % Parámetros iniciales de la trayectoria
8 - x_0 = 0;
9 - y_0 = 0;
10 - th_0 = 30*pi/180;
11
12 - % Ejecutar Simulación
13 - sim('TrajectoryControlNet.slx');
14
15 - xref = salida_xref.signals.values';
16 - yref = salida_yref.signals.values';
17 - x = salida_x.signals.values';
18 - y = salida_y.signals.values';
19
20 - figure(1);
21 - hold on;
22 - generada = plot(xref, yref);
23 - robot = plot(x, y);
24 - hold off;
25 - grid on;
26 - legend('Trayectoria generada', 'Trayectoria robot');
27 - title('Comparación de las trayectorias');
28 - xlabel('Eje X');
29 - ylabel('Eje Y');

```

Para la elección del número de neuronas de la capa oculta, realizamos varias pruebas mediante experimentación. En el valor de Ne del script RunTrajectoryControl2.m se prueban varios valores, se entrena la red, se añade la red al diseño y se ejecuta con RunTrajectoryControlNet.m. Los resultados obtenidos con los diferentes valores estudiados son los siguientes:



Para $N=5$ se observa que la trayectoria del robot no se acerca nada a la trayectoria generada. Esto se debe a que, con 5 neuronas, la red no ha aprendido lo suficiente para llegar a un resultado óptimo. Es un muy mal ajuste.

Para $N=10$ podemos ver ya un muy buen ajuste y, por tanto, suponemos un buen entrenamiento.

Probamos a aumentar el número de neuronas a $N=15$ y la aproximación de la trayectoria del robot a la trayectoria generada es buena pero, a la hora de encontrar esta trayectoria, realiza movimientos irregulares y extraños.

Si se aumenta más el número de neuronas, observamos que vuelve a hacer un buen ajuste a la trayectoria generada y es difícil elegir si el mejor entrenamiento se realiza con 20 o con 10 neuronas.

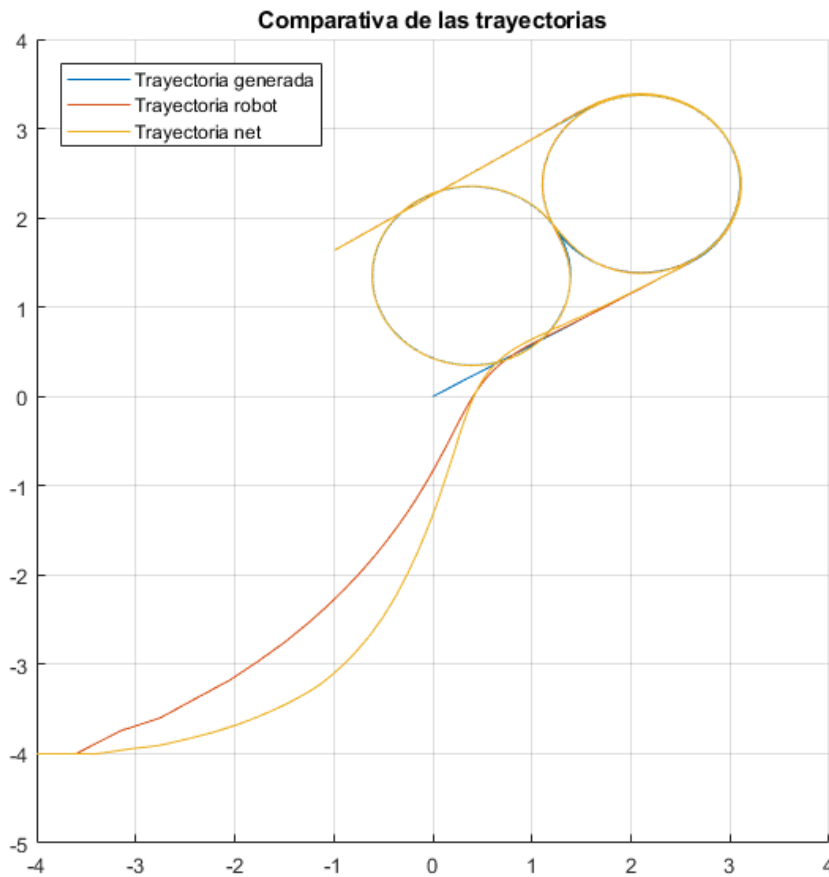
d) Simular el comportamiento y comparar los resultados

Para comparar los resultados entre el comportamiento de los bloques de Simulink correspondientes a las redes neuronales diseñadas anteriormente y el comportamiento del controlador original, se generan diferentes trayectorias variando los valores de x_0 e y_0 y usamos el siguiente código:

```
1 - clear all;
2 - close all;
3
4 % Tiempo de muestreo
5 - Ts = 100e-3;
6
7 % Parámetros iniciales de la trayectoria
8 - x_0 = 0;
9 - y_0 = 0;
10 - th_0 = 30*pi/180;
11
12 % Ejecutar Simulación
13 - sim('TrajectoryControl.slx');
14
15 - xref = salida_xref.signals.values';
16 - yref = salida_yref.signals.values';
17 - x_robot = salida_x.signals.values';
18 - y_robot = salida_y.signals.values';
19
20 % Ejecutar Simulación con la red neuronal
21 - sim('TrajectoryControlNet.slx');
22 - x_net = salida_x.signals.values';
23 - y_net = salida_y.signals.values';
24
25 - figure(1);
26 - hold on;
27 - trayectoria_original = plot(xref, yref);
28 - trayectoria_robot = plot(x_robot, y_robot);
29 - trayectoria_net = plot(x_net, y_net);
30 - hold off;
31 - grid on;
32 - legend('Trayectoria generada', 'Trayectoria robot', 'Trayectoria net');
33 - title('Comparativa de las trayectorias');
```

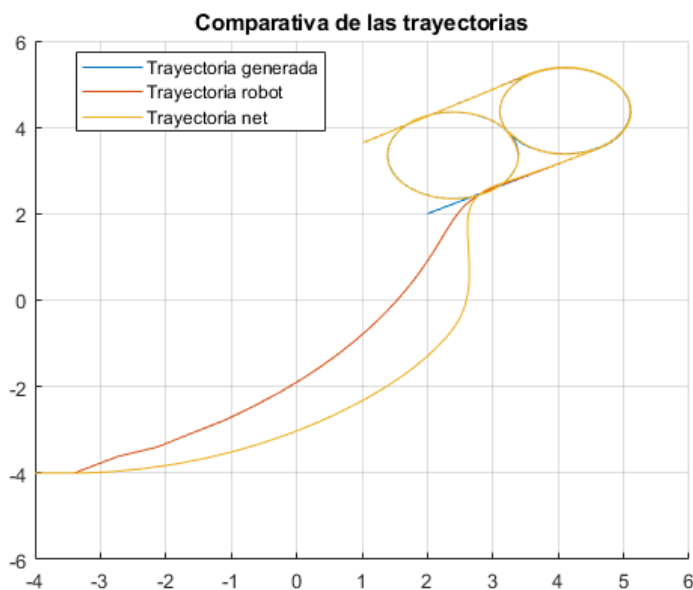
$x_0 = 0, y_0 = 0$

En este caso, la trayectoria generada se inicia en el punto (0,0). Tanto la trayectoria del robot como la de la red se ajustan muy bien al camino deseado.



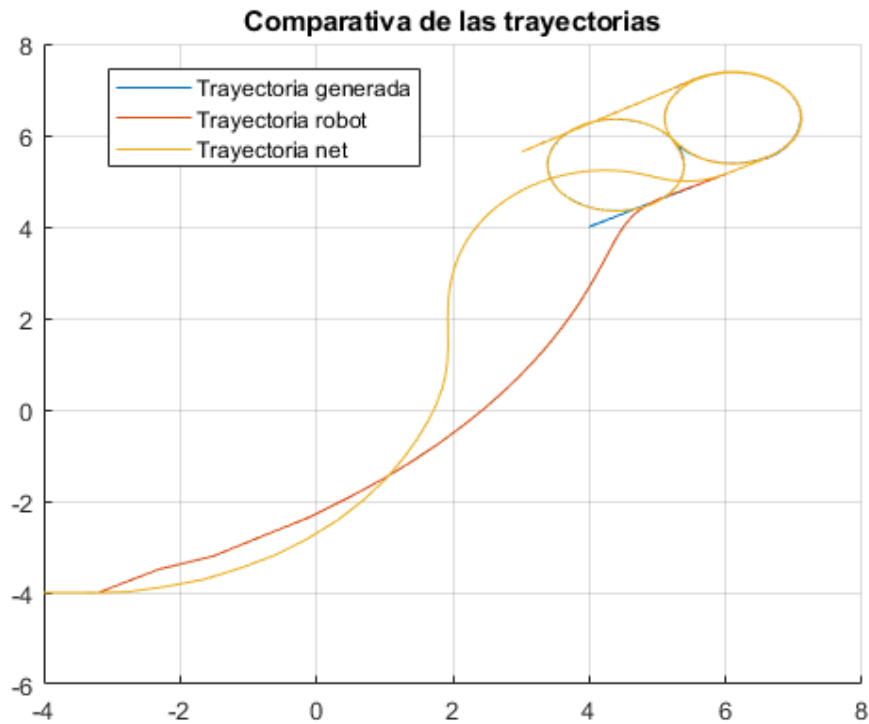
$x_0 = 2, y_0 = 2$

Aumentamos la posición de inicio al punto (2,2). Ahora la trayectoria de la red aumenta también su curvatura hasta encontrar la trayectoria generada, por lo que es un ajuste un poco peor que el anterior aunque aún bastante bueno.



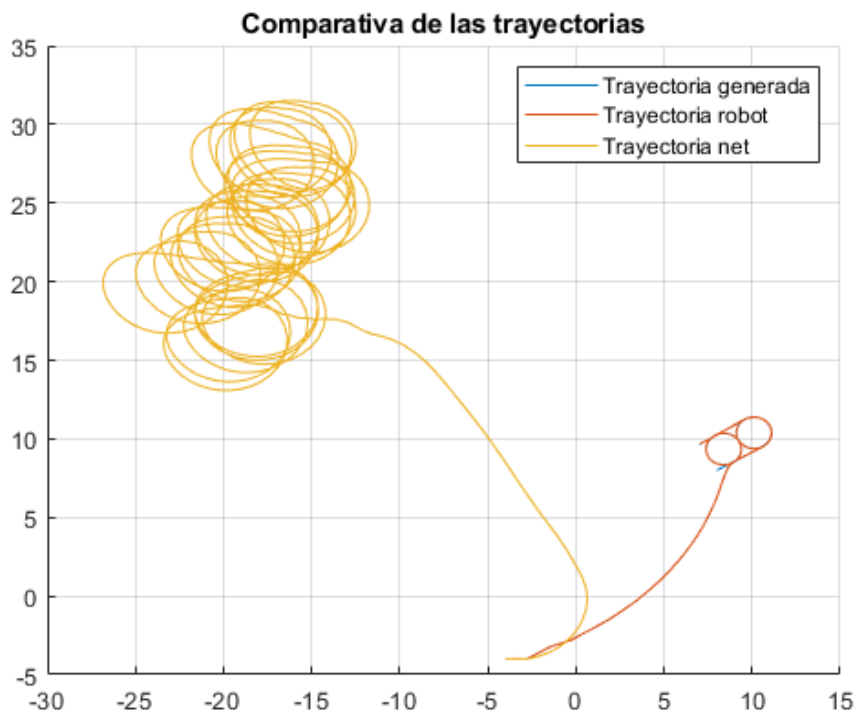
$x_0 = 4, y_0 = 4$

En este punto, ya se empieza a apreciar que la trayectoria de la red se va alejando de la que sigue el robot para completar la trayectoria generada.



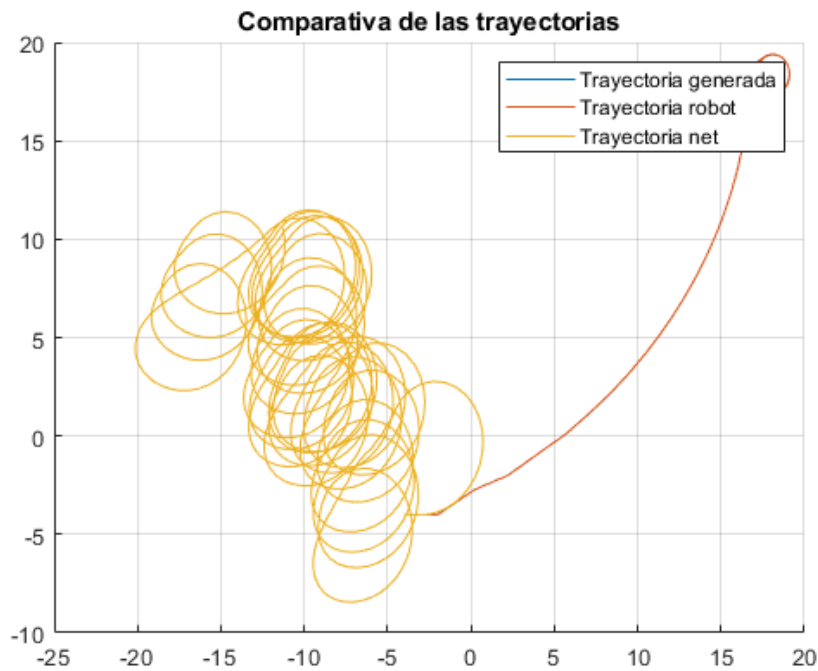
$x_0 = 8, y_0 = 8$

Ahora sí en el punto (8,8) la red no consigue ajustarse a la trayectoria y sigue su propio camino con una muy mala aproximación.



$x_0 = 16, y_0 = 16$

Aumentando aún más el punto inicial no mejoramos la trayectoria, sino que la empeoramos con respecto al mal ajuste anterior.



$x_0 = -2, y_0 = -2$

Como hemos visto, según aumentábamos el punto inicial iban apareciendo peores trayectorias. Entonces probamos a disminuir con respecto a la mejor aproximación hasta ahora que es el punto (0,0). Probamos el punto (-2,-2) y se comprueba que tampoco mejora. También realiza caminos que no se ajustan a la trayectoria generada ni a la del robot, aunque es mejor que algunas de las anteriores,

