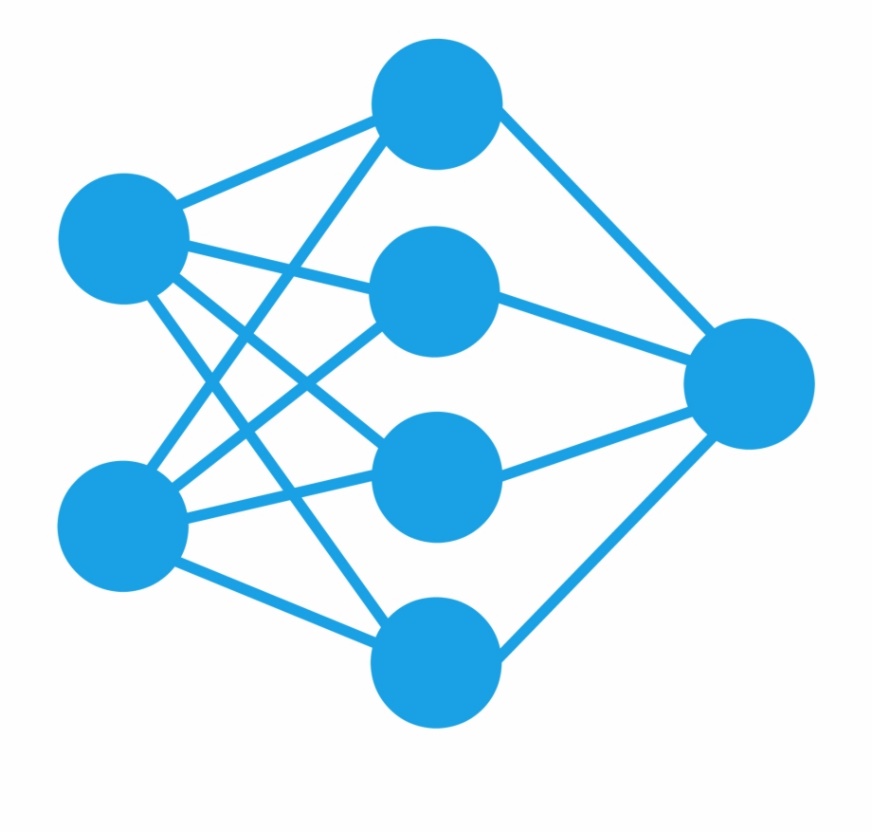
Práctica 1: Identificación y control neuronal



**SISTEMAS DE CONTROL INTELIGENTE**

**05/10/2019**

**Autores:** Álvaro de las Heras Fernández

Álvaro Maestre Santa

# Índice

[**1.Introducción** 3](#_Toc22999191)

[**2.Desarrollo de la práctica** 3](#_Toc22999192)

[**Parte I** 3](#_Toc22999193)

[Ejercicio 1 3](#_Toc22999194)

[Ejercicio 2 4](#_Toc22999195)

[Ejercicio 3 7](#_Toc22999196)

[Ejercicio 4 10](#_Toc22999197)

[**Parte II** 12](#_Toc22999198)

[Ejercicio 1 12](#_Toc22999199)

[Ejercicio 2 13](#_Toc22999200)

[**Parte III** 16](#_Toc22999201)

[Ejercicio 1 16](#_Toc22999202)

[Ejercicio 2 18](#_Toc22999203)

[**3.Conclusión** 22](#_Toc22999204)

[**4.Bibliografía** 22](#_Toc22999205)

## 

# 1.Introducción

El objetivo de realizar esta práctica consiste en realizar una serie de **simulaciones** de robots a los que se aplicarán **control clásico** y **neurocontroladores**, realizando **comparaciones** entre ambos y realizando variaciones a los controladores neuronales, para observar como afecta a la **trayectoria**. Toda esto se realizará en tres partes, siendo la primera una como introducción a las redes neuronales, la segunda una comparación entre controles y la tercera para seguir trayectorias preconfiguradas.

# 2.Desarrollo de la práctica

## Parte I

### Ejercicio 1

En este ejercicio se ha realizado un clasificador neuronal mediante un perceptrón. Para ello se dispondrán de puntos con 4 tipos de clases que son los siguientes:

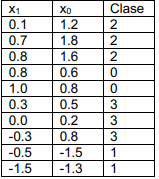


Fig. 1 Datos que se clasificarán.

Una vez introducimos los datos en Matlab, se ha de crear y entrenar el perceptrón que simplemente se ha de llamar por su nombre, y después entrenarle con **train()** al que introduciremos los puntos y clases, estas últimas como objetivos.

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

**La red con los datos que recibe por entrada es capaz de separar los datos en sus 4 clases mediante las 2 rectas delimitando con ellas 4 áreas. Todo este proceso lo hace el perceptrón iterando 8 veces (probando distintas posiciones con las rectas), hasta conseguir tener los puntos clasificados.**

**La capa de salida tiene 2 neuronas como indica la Fig. 2, esto se debe a la cantidad de clases que hay que clasificar. En este caso con al tener cuatro clases como mínimo necesitábamos 2 salidas para que se puedan asignar a las clases cuando se clasifique y entrene la red [00=Clase 0 01=Clase 1 10=Clase 2 11=Clase 3], por lo que eran necesarias 2 neuronas a la salida. Si únicamente hubiéramos tenido que clasificar en dos clases con una única neurona en la capa de salida hubiera sido suficiente.**

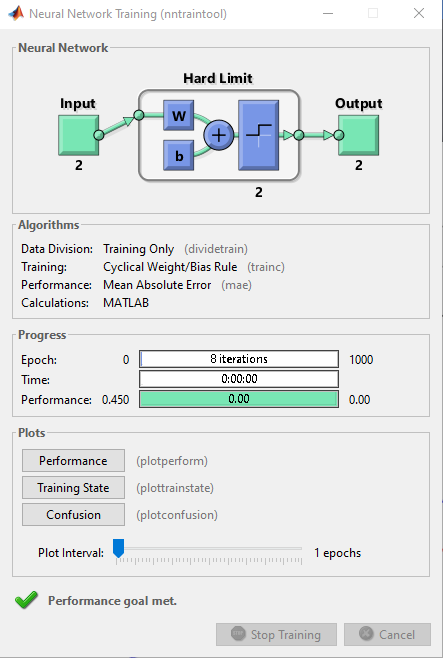
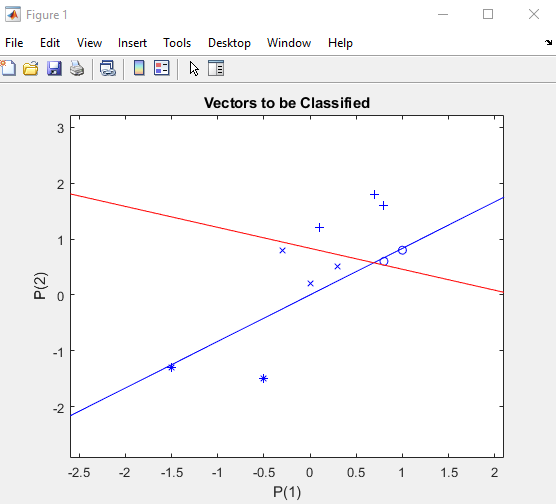
 

Fig. 2 Salida del entrenamiento y representación del resultado, respectivamente.

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

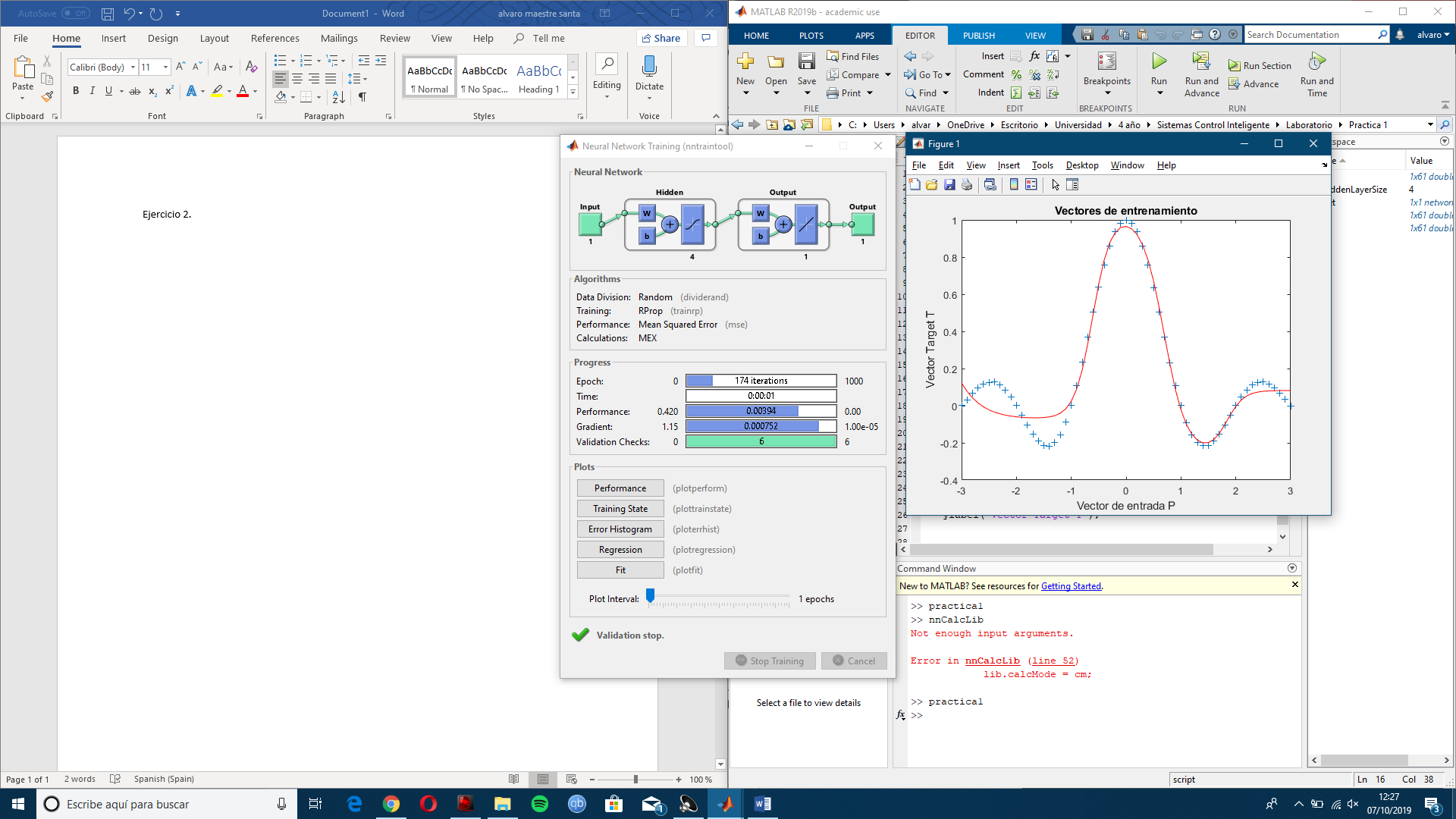
Al incorporar ese nuevo punto ya **no se puede clasificar**, haciendo que el perceptrón itere indefinidamente, llegando al límite máximo de 1000 iteraciones. La razón de que ahora no pueda clasificarlo se debe a uno de los principales inconvenientes del perceptrón es que cuando se tiene un problema que no es **linealmente separable**, no puede **converger** porque **no tiene solución** haciendo que busqué una solución **indefinidamente**.

### Ejercicio 2

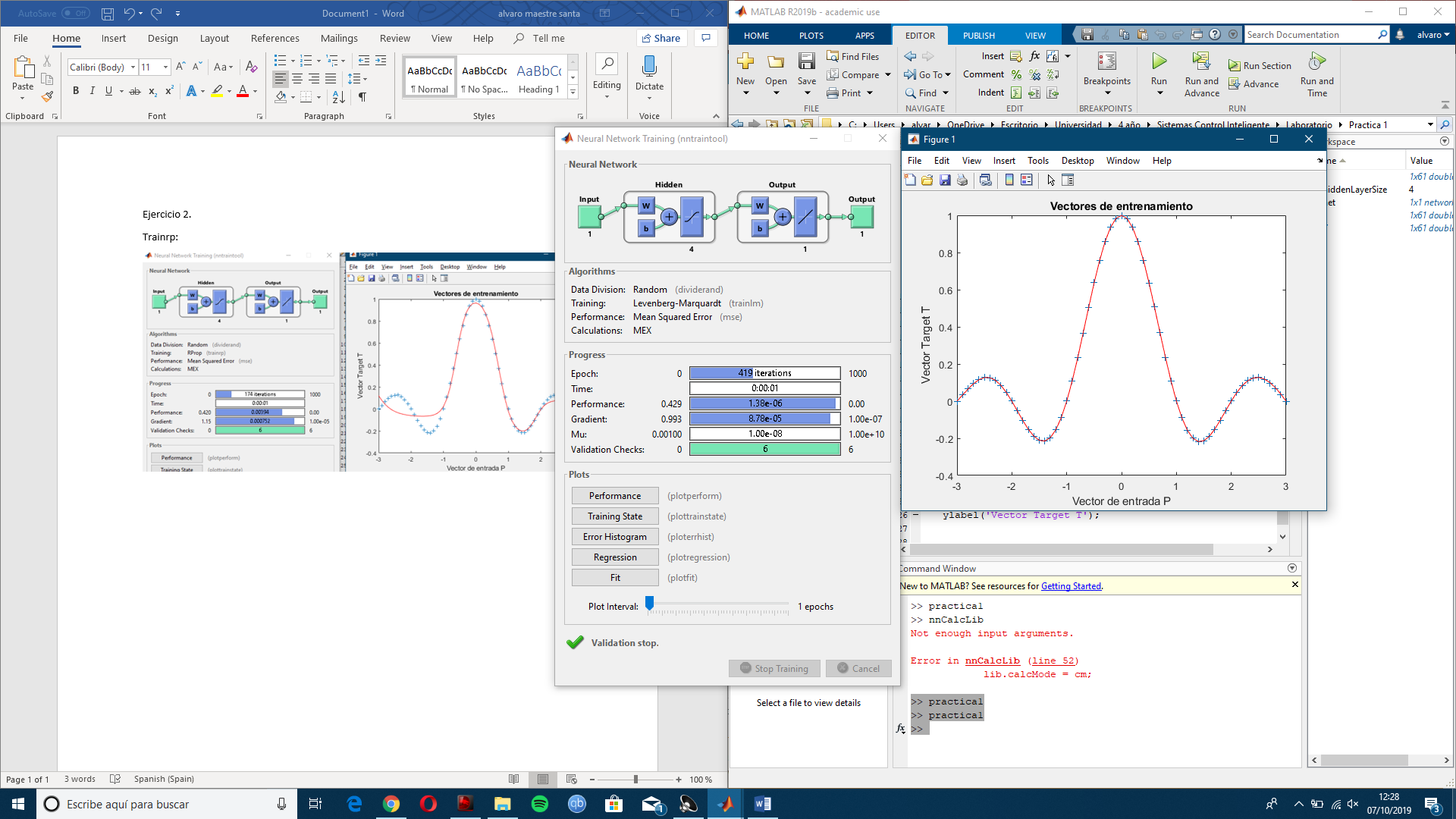
(Explicar mejor)

En este ejercicio te pide que cambies el tipo de entrenamiento de las redes y los compares, para ello hemos utilizado 3 tipos de entrenamientos, que son: Trainrp, Trainbr, Traingd. A parte hemos modificado también las neuronas para ver cómo se comporta con más y menos neuronas. Los resultados son los siguientes:

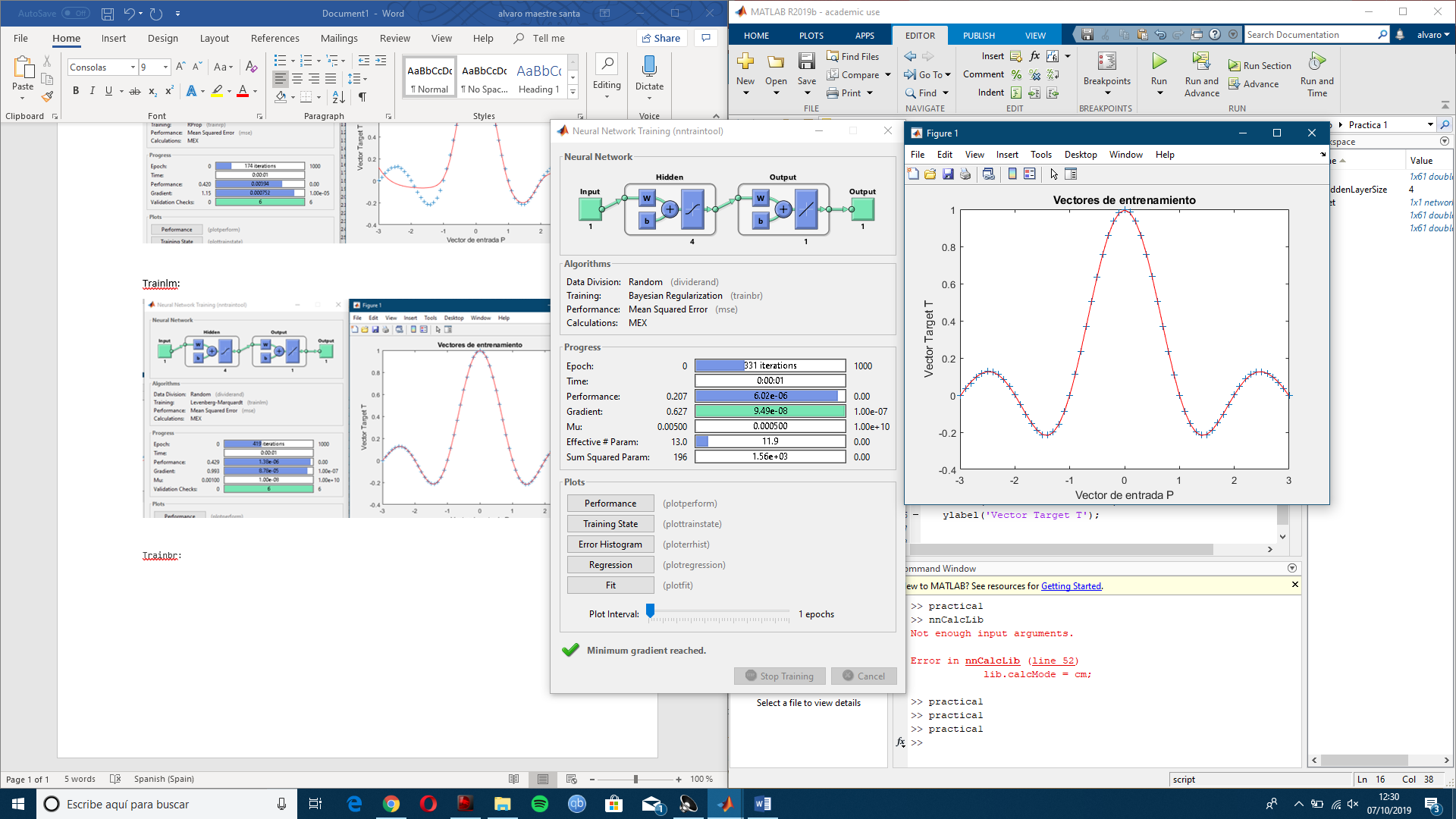
* **Trainrp(4 neuronas):** Resilient Backpropagation.



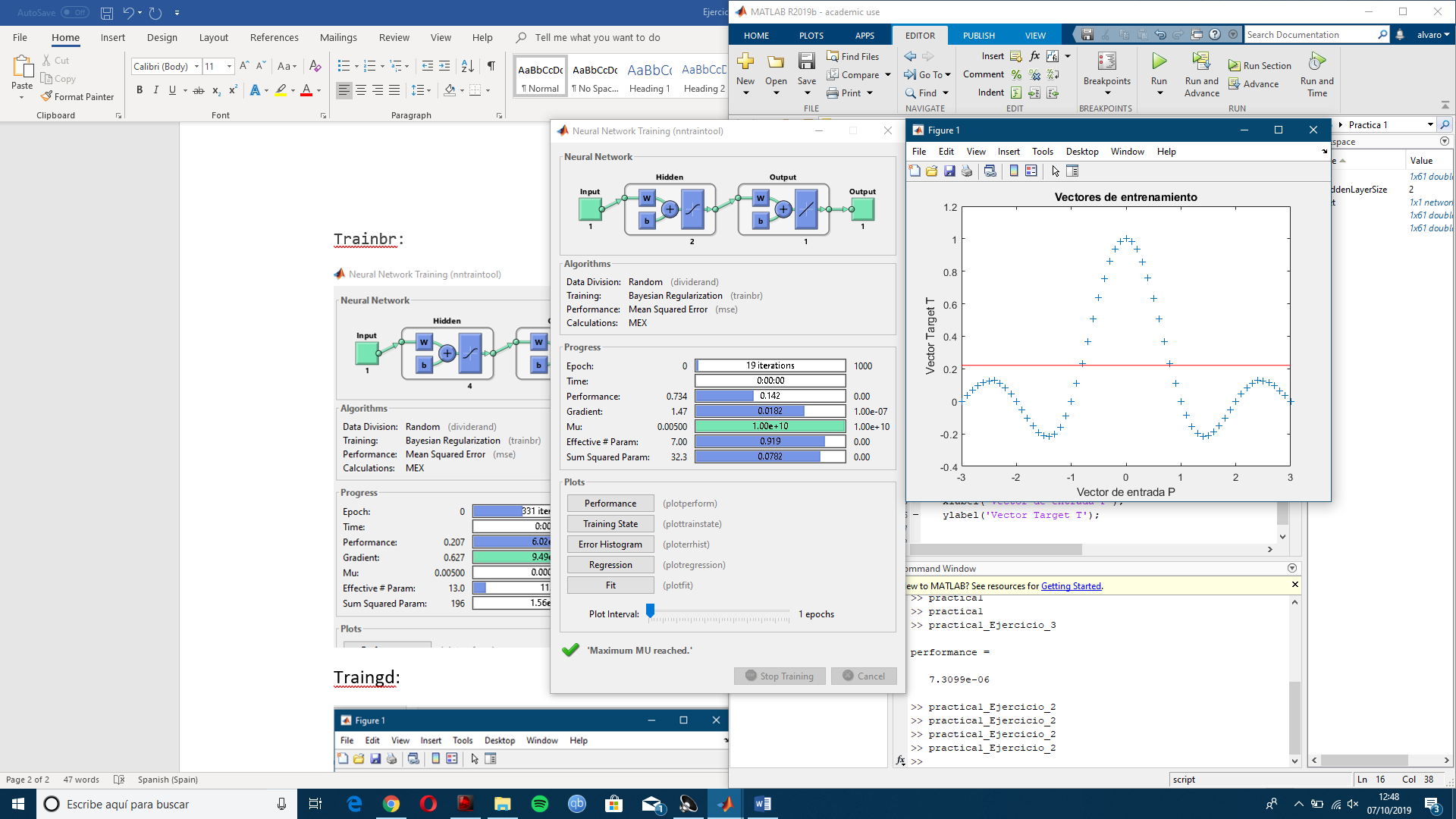
* **Trainlm(4 neuronas):** Levenberg-Marquardt.



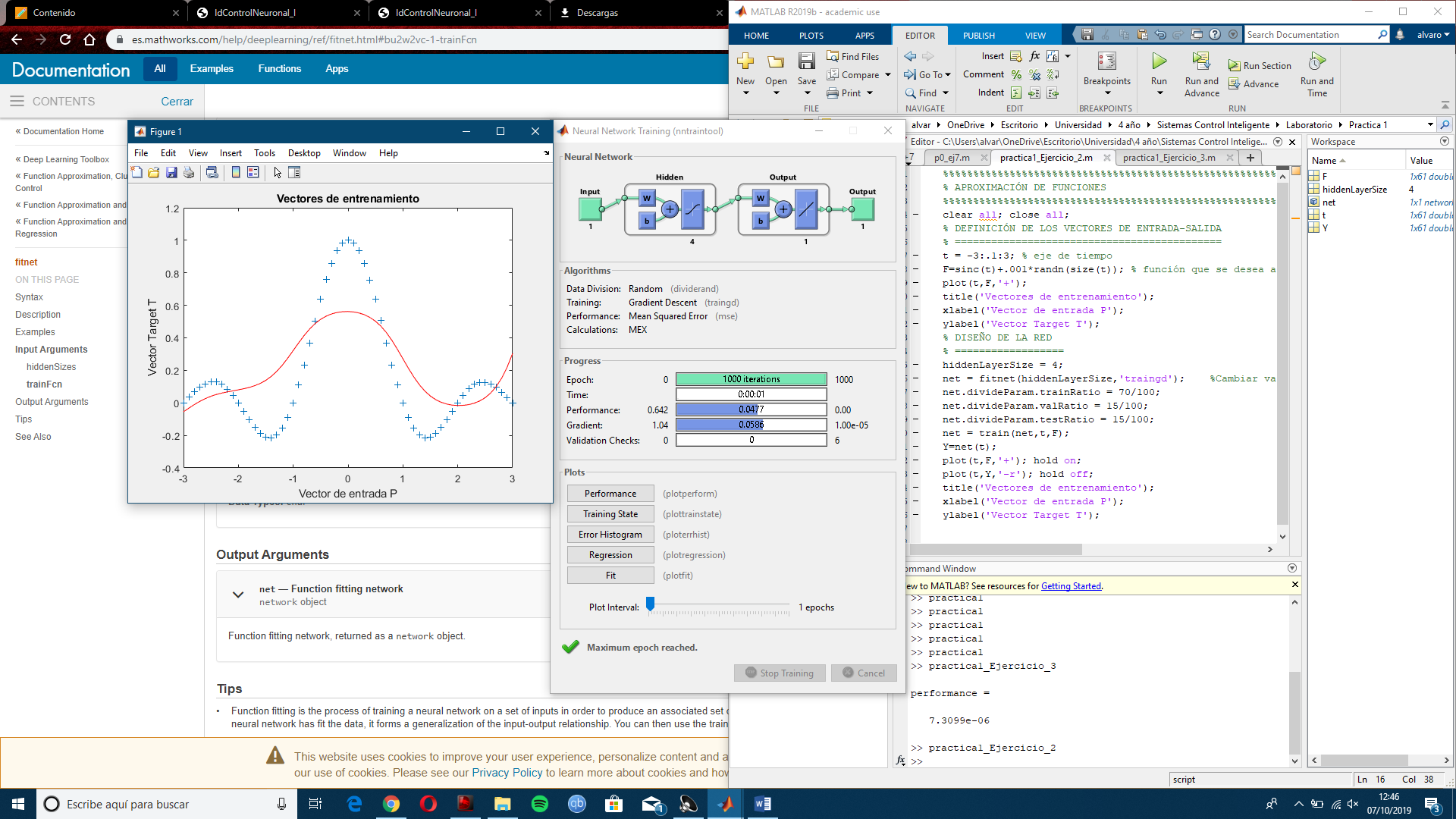
* **Trainbr (4 neuronas):** Bayesian Regularization.



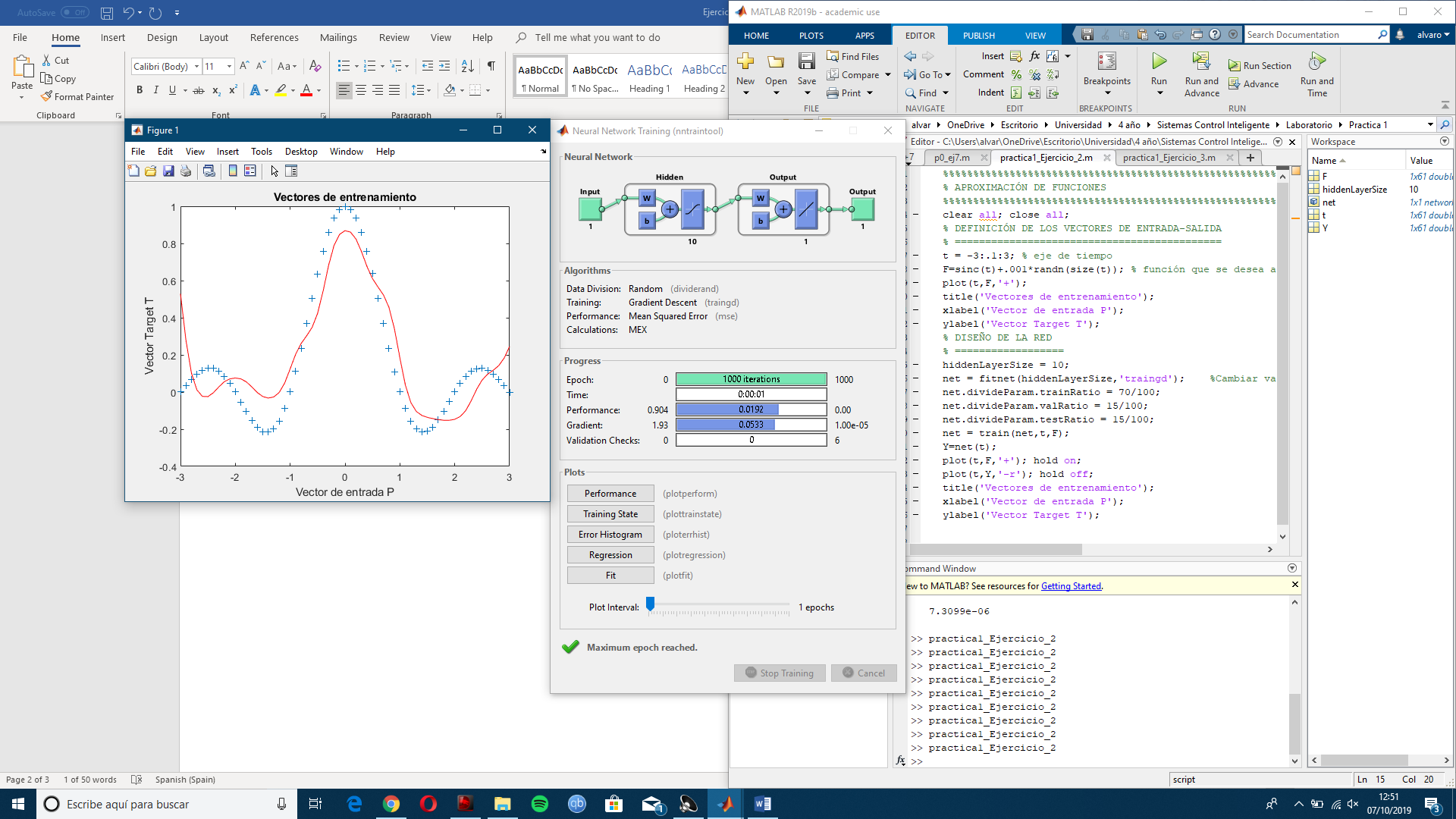
* **Trainbr (2 neuronas):** Bayesian Regularization.



* **Traingd (4 neuronas):** Gradient Descent.



* **Traingd (10 neuronas):** Gradient Descent.



**Conclusión**

Según los resultados obtenidos, podemos decir lo siguiente.

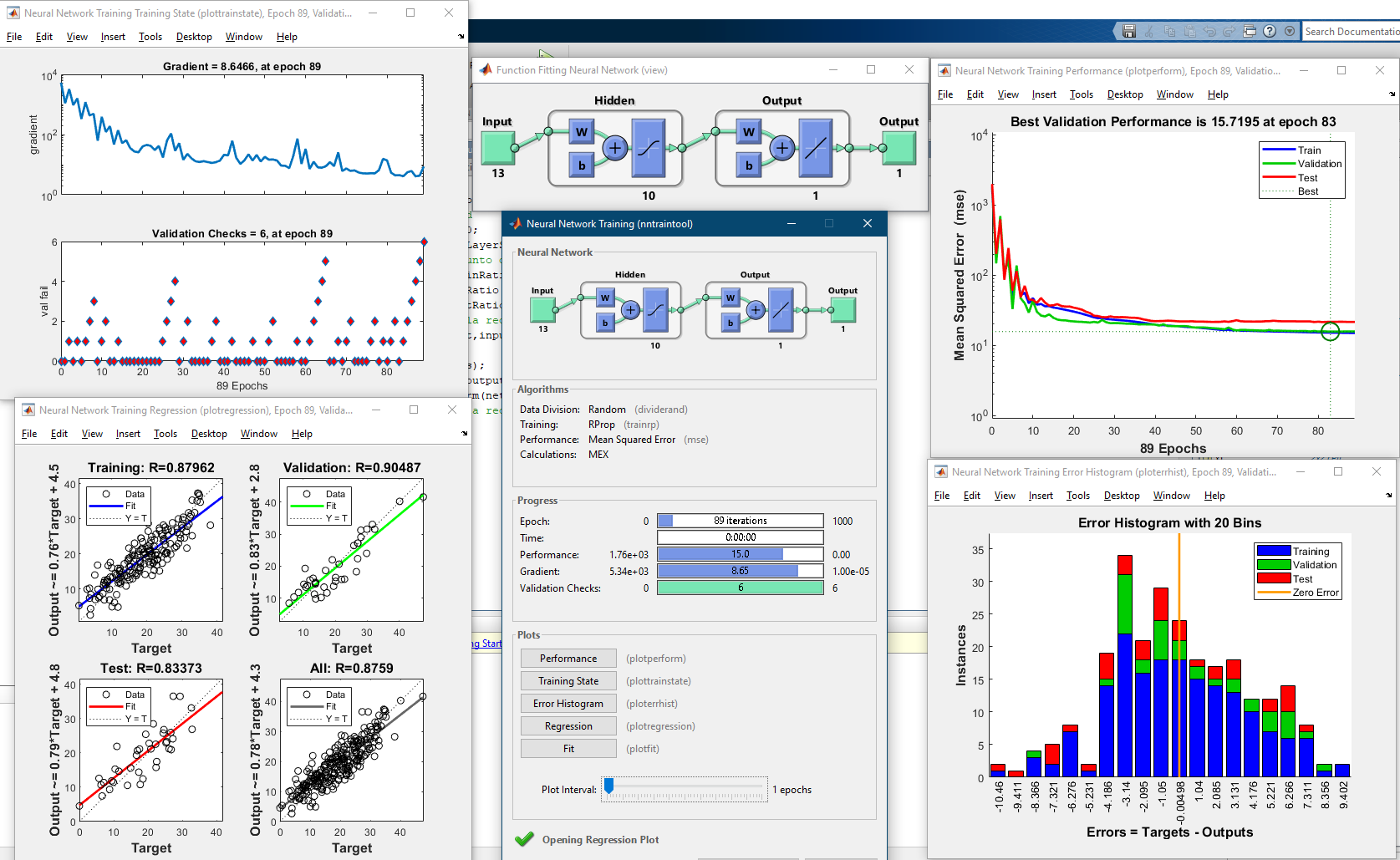
* Los métodos de entrenamiento que más se acercan al resultado perfecto son: **Trainbr** *y* **Trainlm**. Tenemos que destacar que **Trainlm**, obtiene una mejor performance en el entrenamiento. ¿Por qué? compara los valores y habla de las graficas
* El método que menos se aproxima al resultado es: ***Traingd****.*
* En el caso de poner 2 neuronas, vemos como no es capaz de acercarse al resultado, sin embargo, en el caso de 10 neuronas, vemos como mejora considerablemente el resultado, en este caso, no hemos notado sobre entrenamiento, ya que mejora en todos los aspectos.

### Ejercicio 3

(Explicar las gráficas que se verán si los valores que dan son buenos o no y hablar datos)

En este ejercicio, vamos a comparar diversas graficas de diferentes métodos de entrenamiento, estas gráficas son: performance, estado del entrenamiento, histograma de los errores y la regresión. El entrenamiento con mejores resultados, le modificaremos la forma en la que se divide los datos, para así ver lo que sucede con las gráficas.

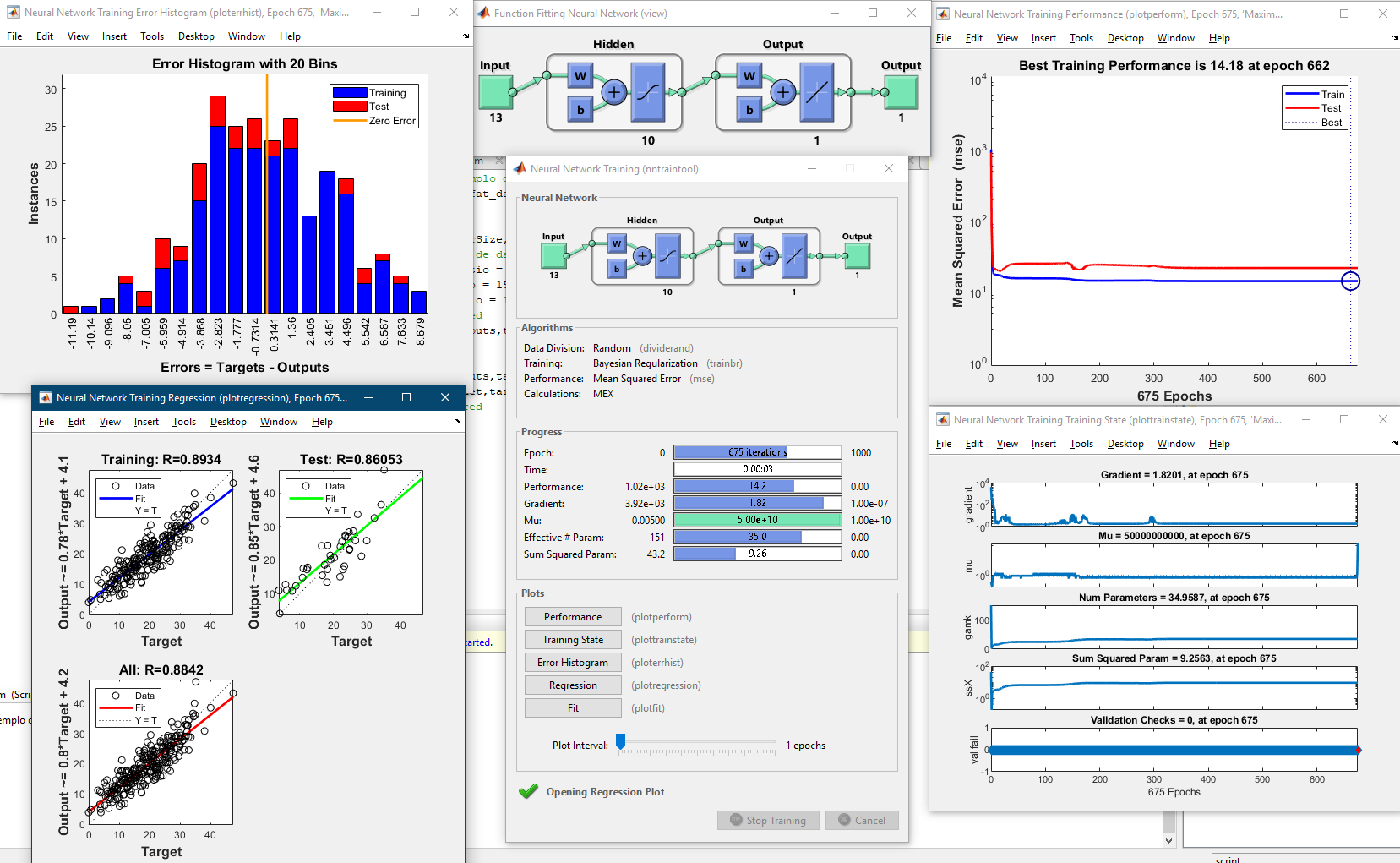
* **Trainrp**: Resilient backpropagation.



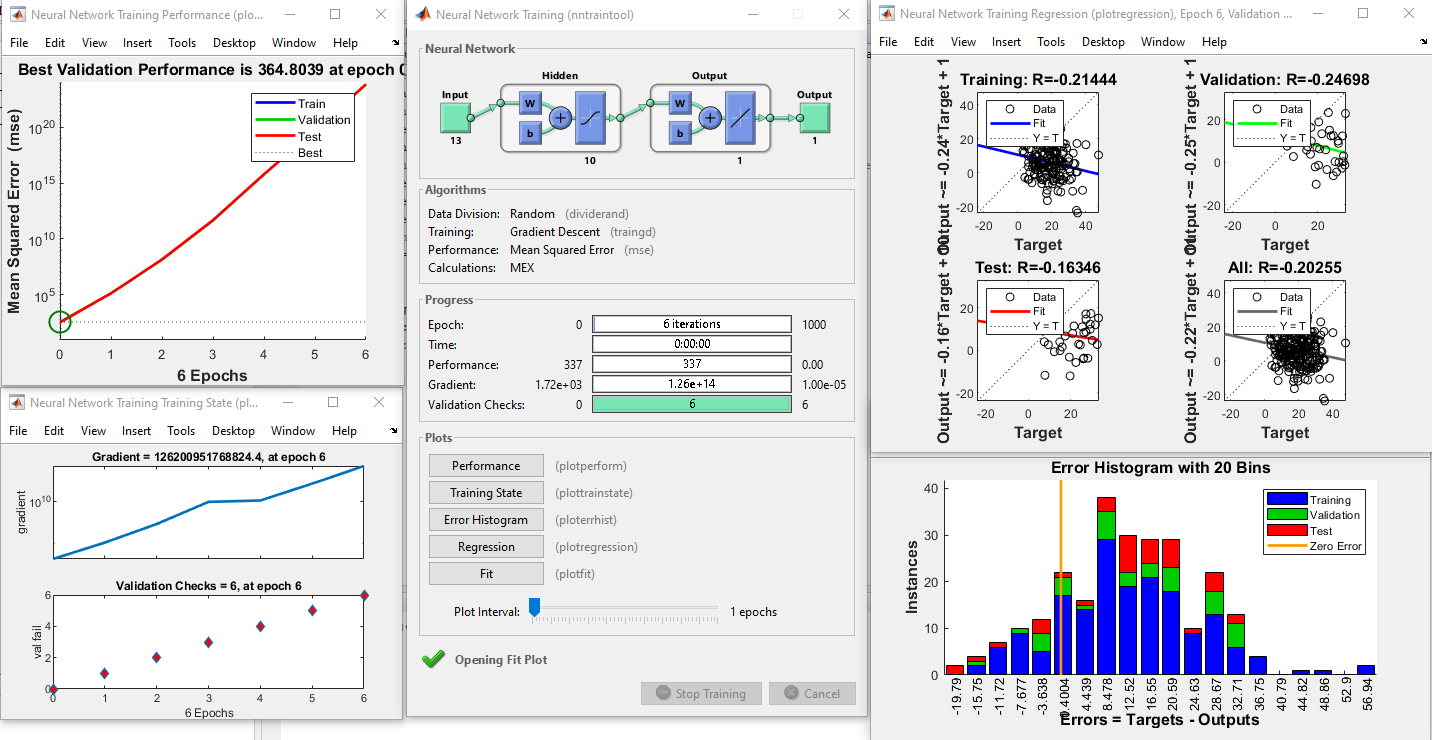
* **Trainlm**: Levenberg-Marquardt.



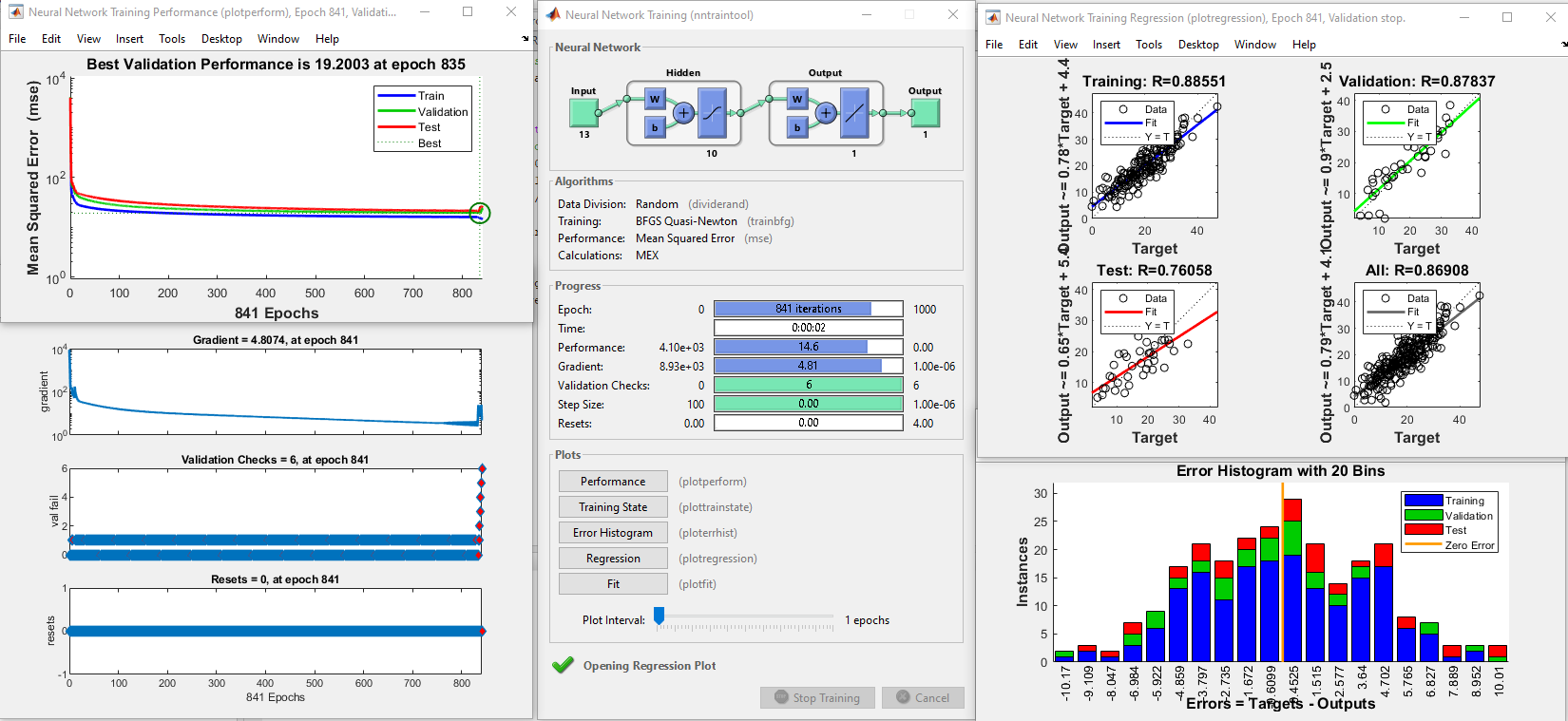
* **Trainbr**: Bayesian Regularization.



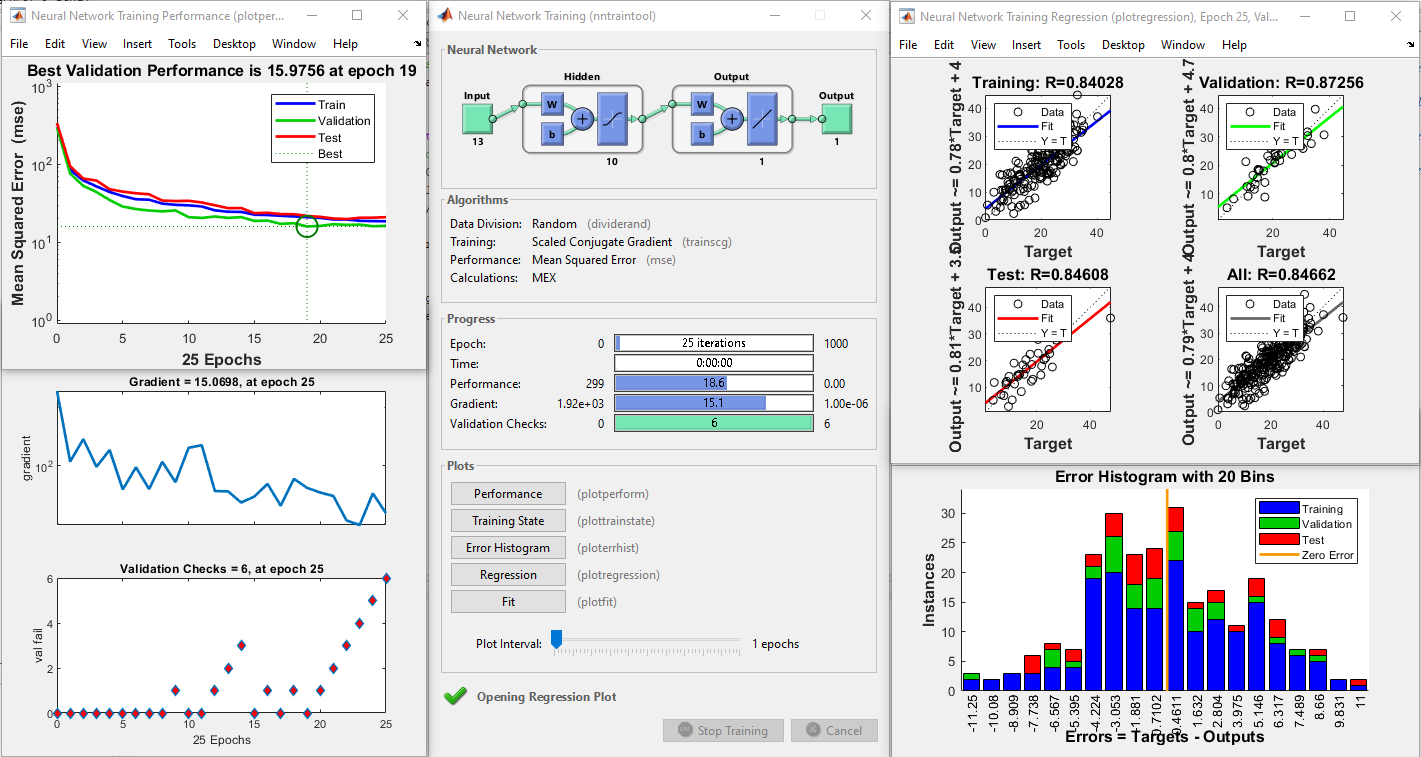
* **Traingd:** Gradient descent backpropagation.

****

* **Trainbfg**: BFGS quasi-Newton backpropagation.

****

* **Trainscg**: Scaled conjugate gradient backpropagation.

****

En el caso de la gráfica fit, no se ha podido mostrar, ya que tiene varios elementos en los datos de entrada.

CONCLUSION

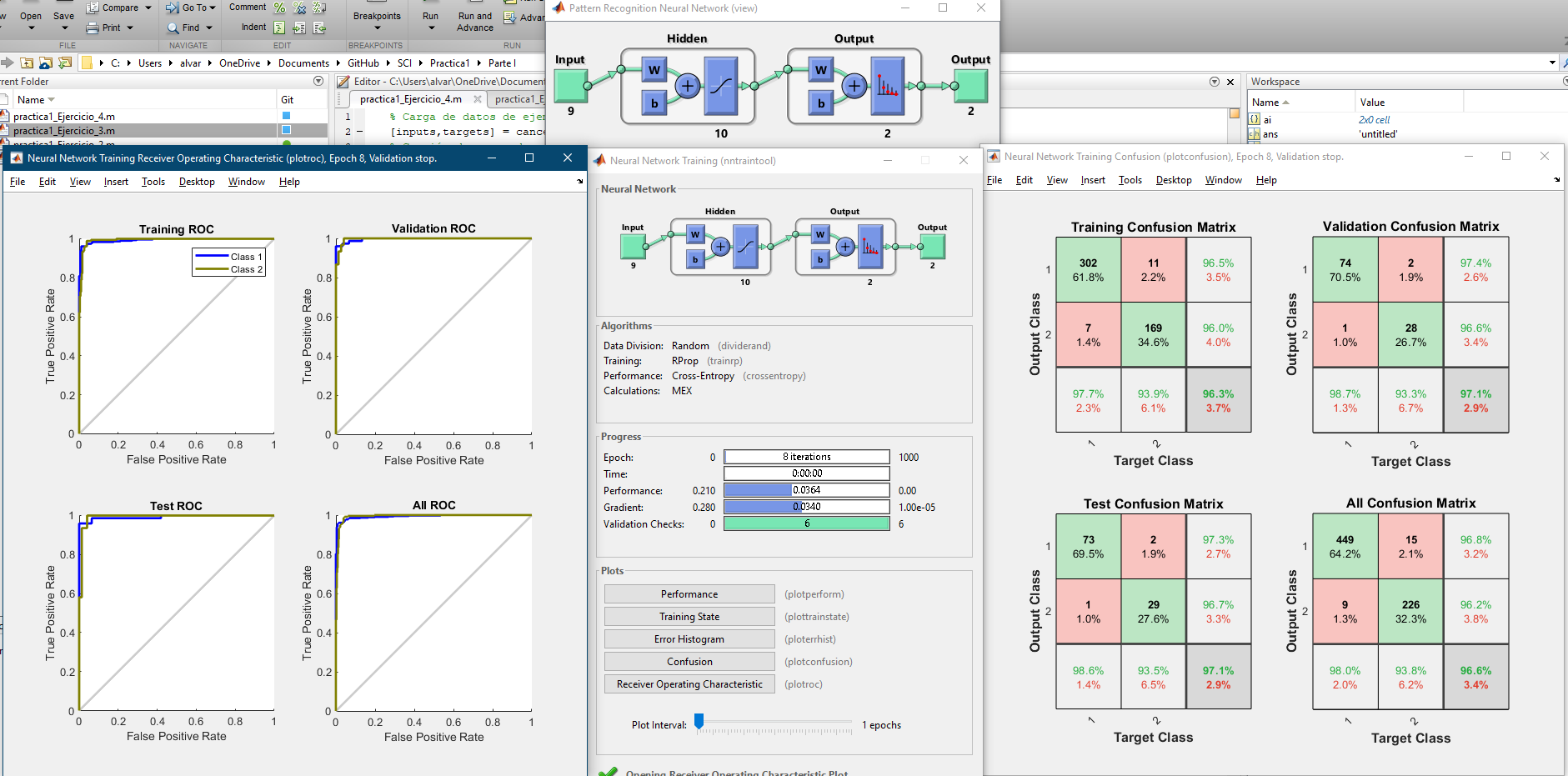
### Ejercicio 4

(Explicar las gráficas que se verán si los valores que dan son buenos o no y pies de foto)

(Mencionar que datos se van a usar)

Este ejercicio es muy similar al anterior, solo que vamos a analizar otro tipo de gráficas sobre los mismos entrenamientos estudiados previamente, estas gráficas son: las matrices de confusión y curvas roc (características operativas del receptor). También, como en el anterior ejercicio, el entrenamiento con mejor resultado, será el que le modificaremos la división de datos, para ver como repercuten en estas nuevas gráficas estudiadas.

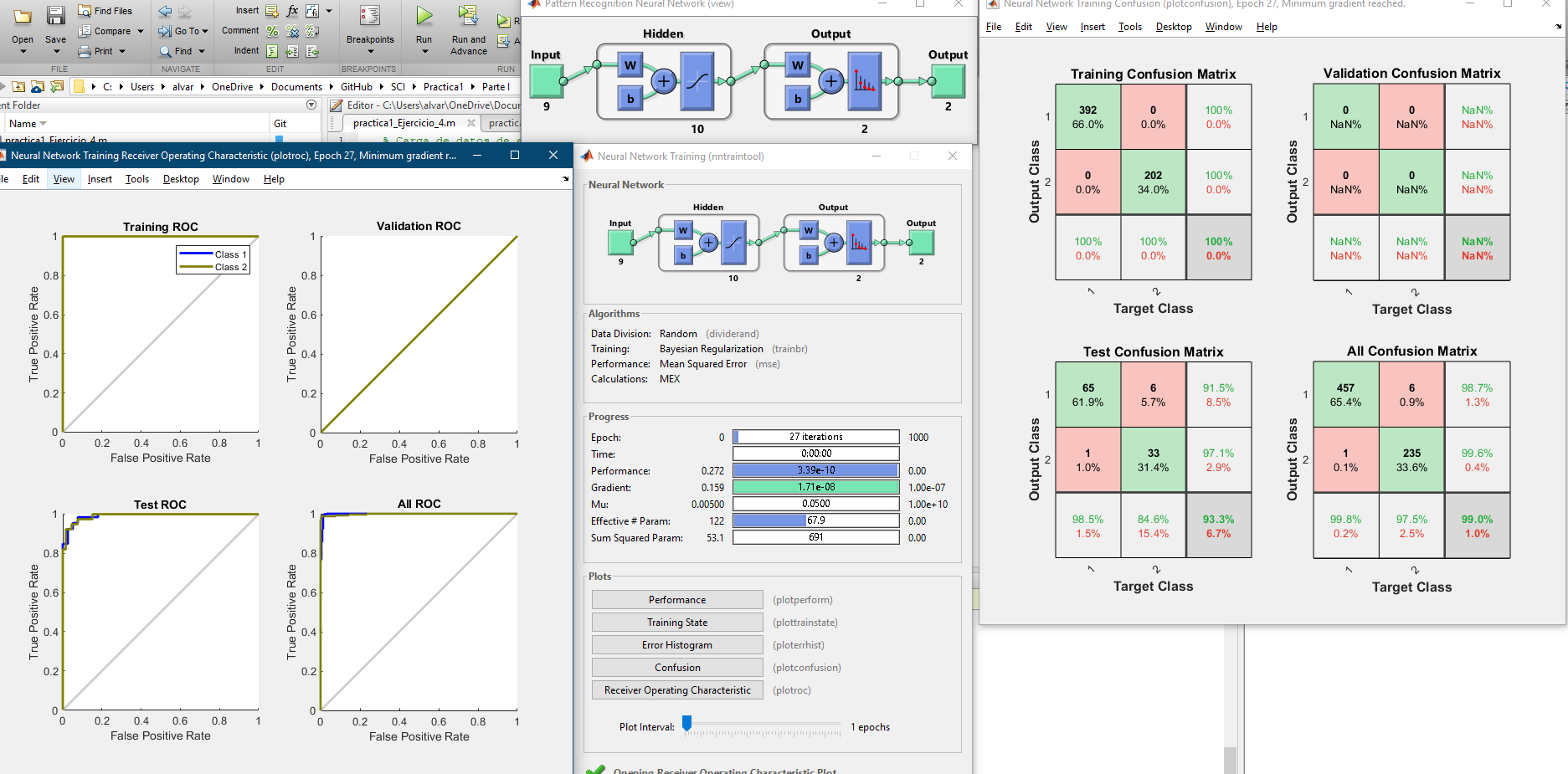
* **Trainrp**: Levenberg-Marquardt.



* **Trainlm**: Levenberg-Marquardt.



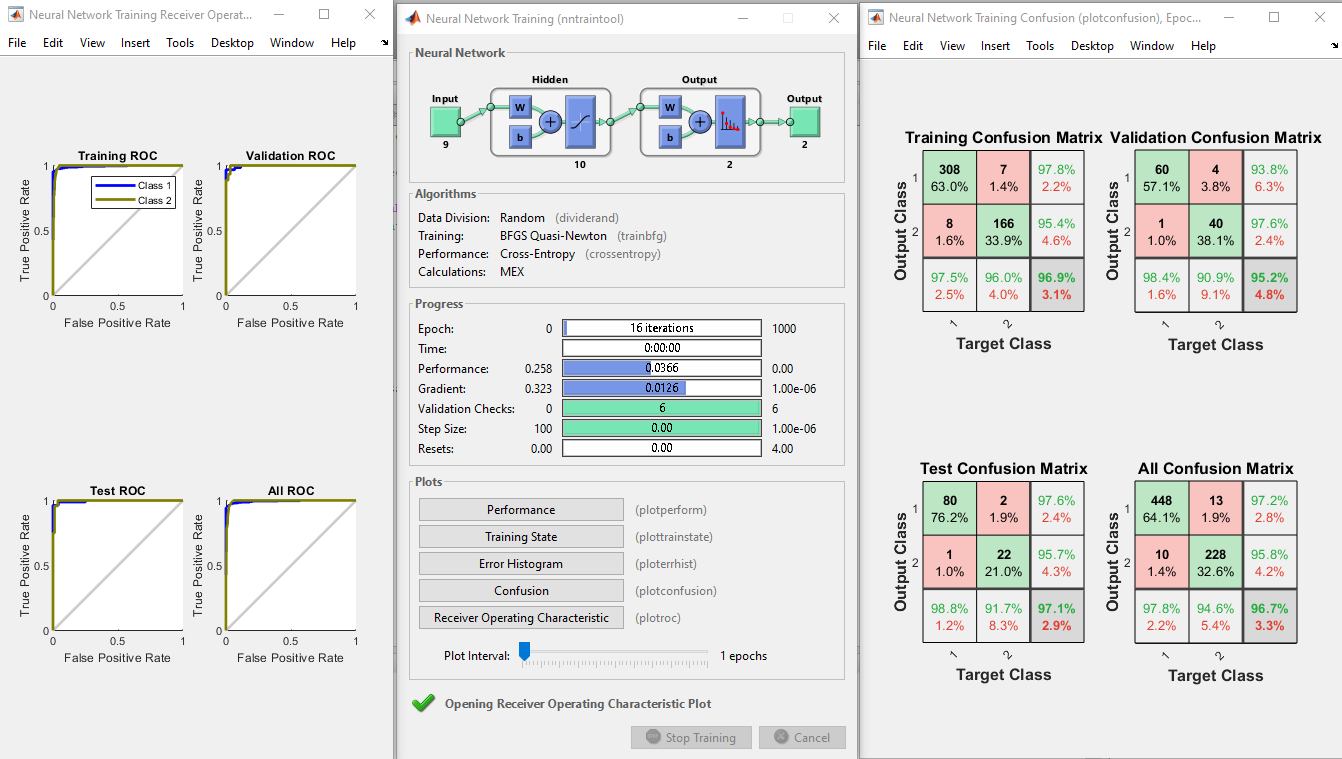
* **Trainbr**: Bayesian Regularization.



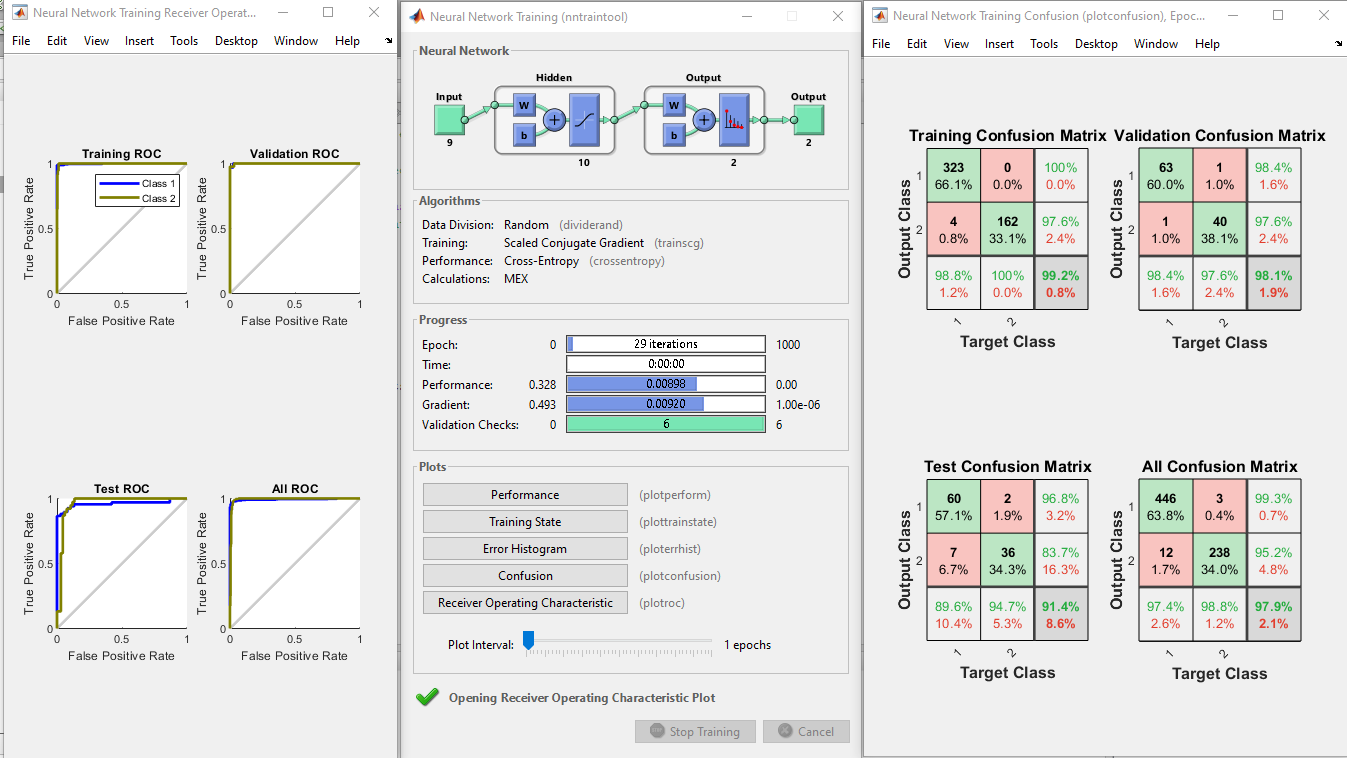
* **Traingd**: Gradient descent backpropagation.



* **Trainbfg**: BFGS quasi-Newton backpropagation.



* **Trainscg**: Scaled conjugate gradient backpropagation.



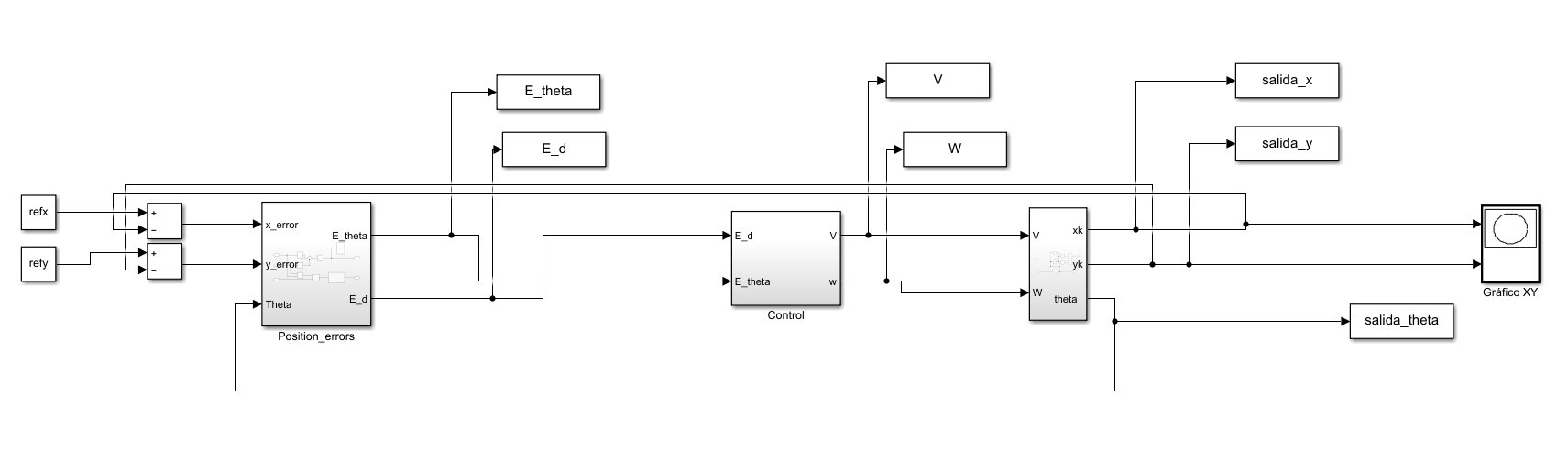
CONCLUSION

## Parte II

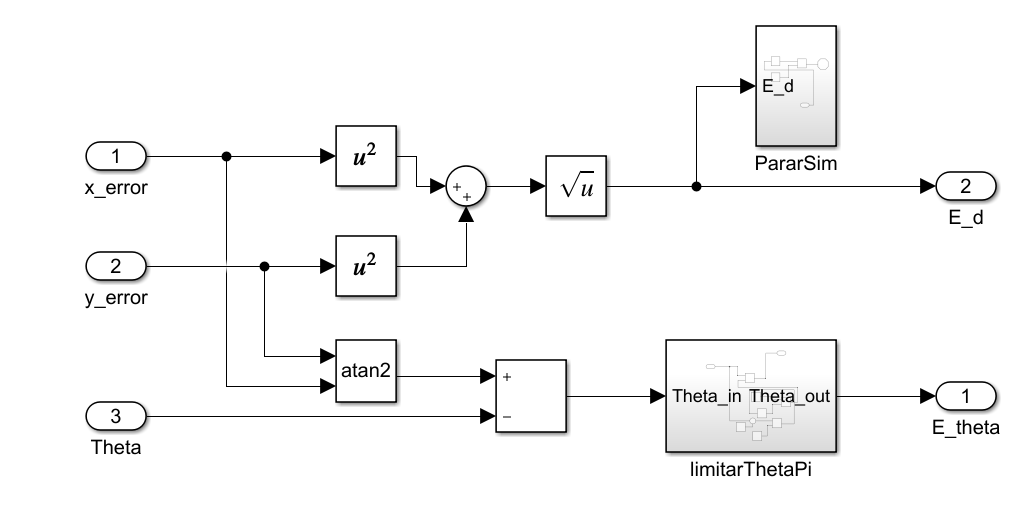
### Ejercicio 1

En esta parte el objetivo es, diseñar un sistema que simule el **desplazamiento** del robot hasta unas **coordenadas** mediante un **controlador**, que trataremos como una **caja negra**.

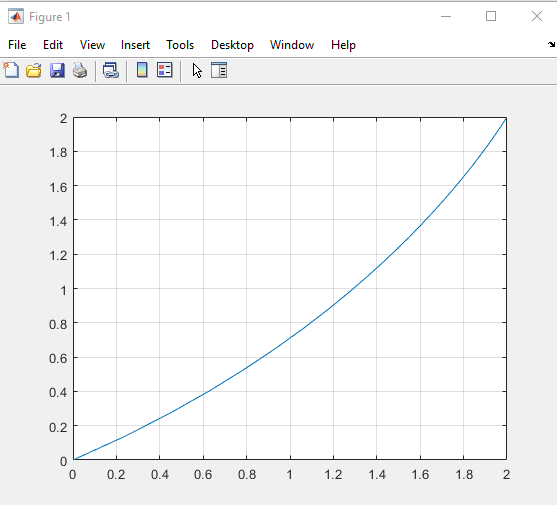
Lo primero ha sido crear un diagrama de bloques, que es el siguiente:



Luego hemos programado **Position\_errors** en base a las condiciones y fórmulas que nos decía el enunciado, quedando como se muestra en la siguiente imagen, añadiendo un módulo de **parada** en función al error y otro para limitar ángulos a **[-pi,pi]:**



Una vez configurado y programado todo, el robot es capaz de moverse y hacer una trayectoria, como se puede apreciar en la siguiente imagen:

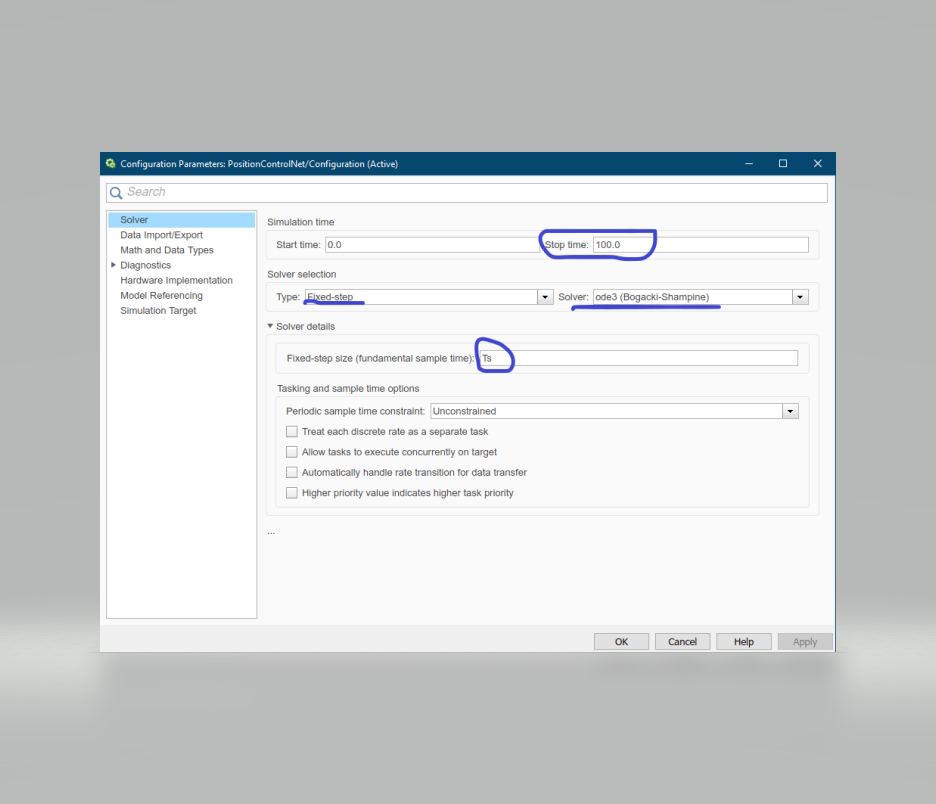


En este caso, hemos programado al robot para que vaya desde la posición x = 0 e y = 0 a la posición x = 2 e y = 2. Por lo tanto, podemos decir, de que el robot está bien programado, ya que llega al punto, el cual, le hemos dicho que vaya.

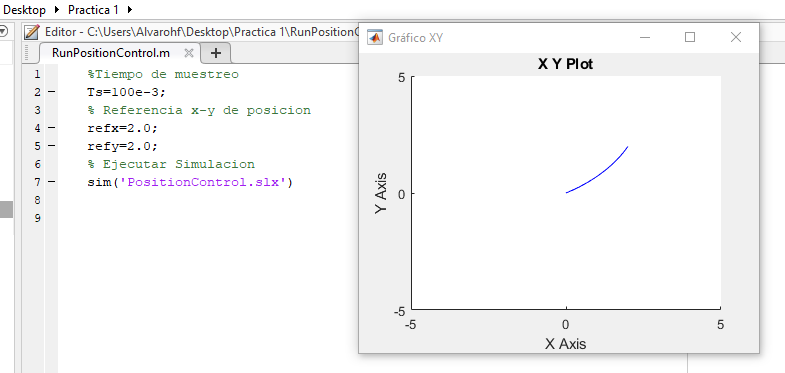
### Ejercicio 2

En este ejercicio, tenemos que crear una red neuronal que **emule** la **caja negra** del ejercicio anterior, actuando como un **neurocontrolador**.

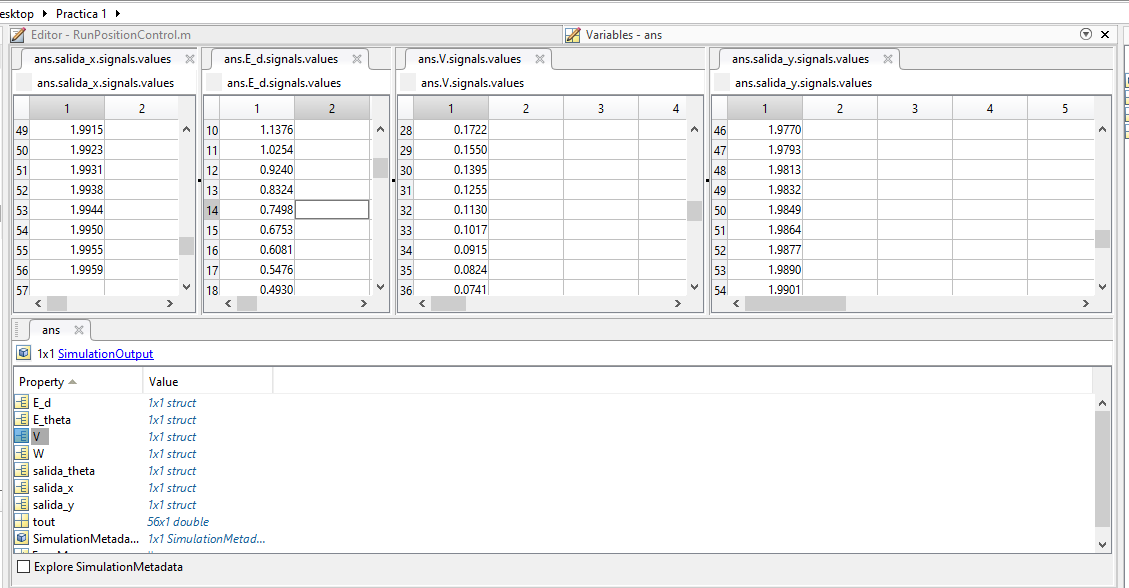
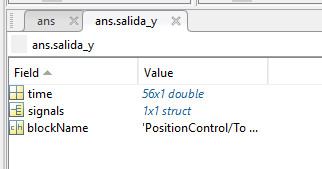
1. En el primer apartado, tenemos que configurar el diagrama, limitando la duración a **100 segundos** de la simulación, añadiendo la variable **Ts** como periodo de **muestreo** y poniendo el **solver** en **ode3** y Fixed-step.



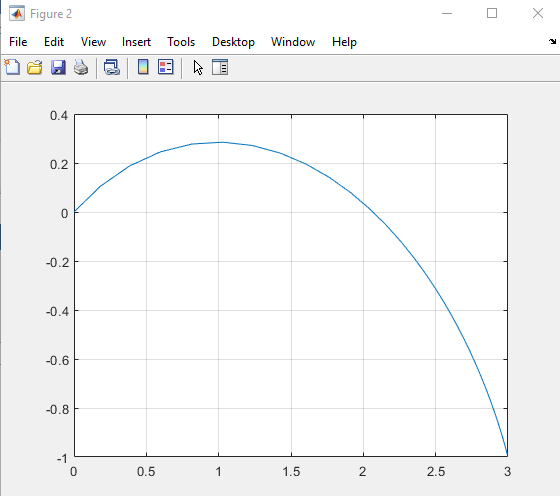
1. Seguidamente, copiamos el código proporcionado por el profesor, que simulará el diagrama desde el entorno de comandos de Matlab.



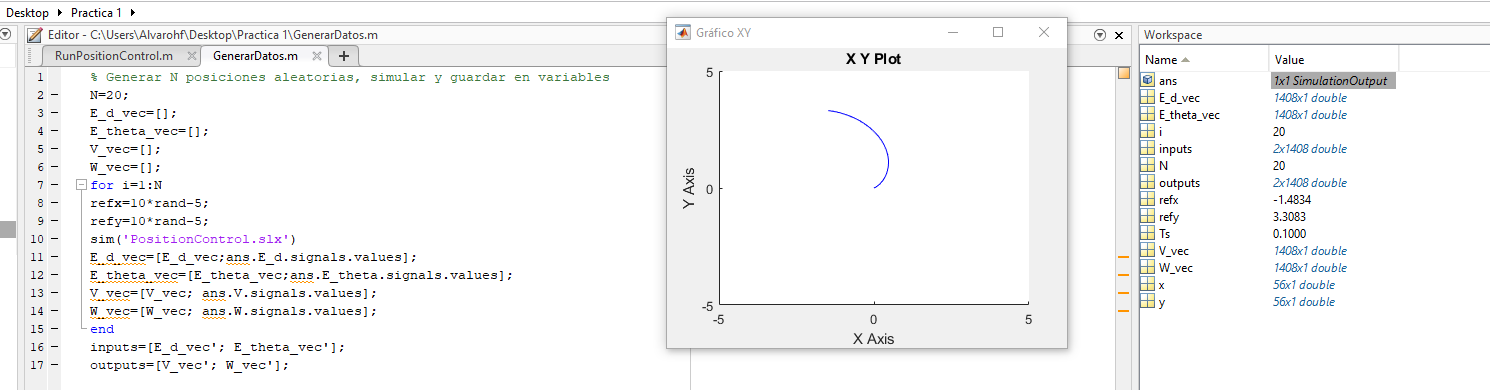
1. Ahora vamos a comprobar que se generan correctamente las **variables** que contienen las **salidas** y **entradas** del controlador. Viendo que se han creado correctamente y contienen datos y tiempos.

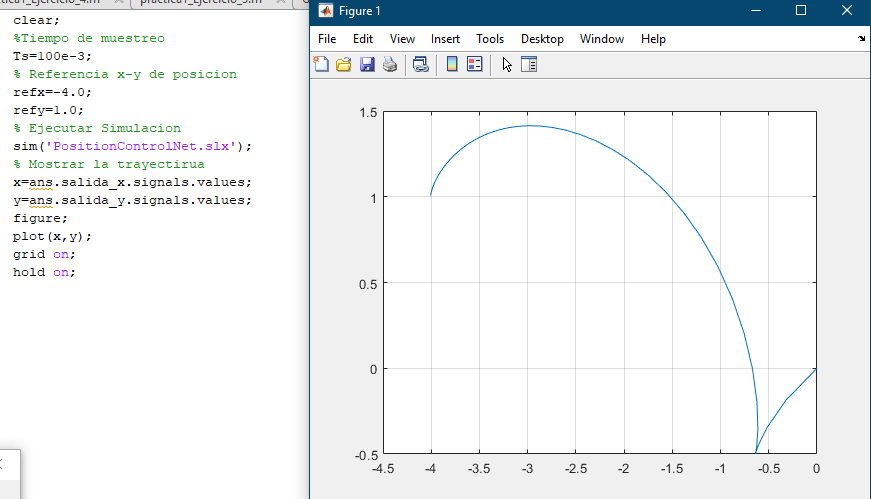
1. Ahora se ejecuta y vemos la **trayectoria** que hace el robot, con una gráfica creada a partir de los puntos generados en la **simulación**.



1. Ahora repetiremos los procesos anteriores pero con **diferentes datos** y muchas **más veces** para así generar un conjunto de datos que servirá para **entrenar** a la red.



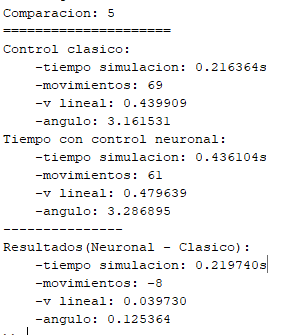
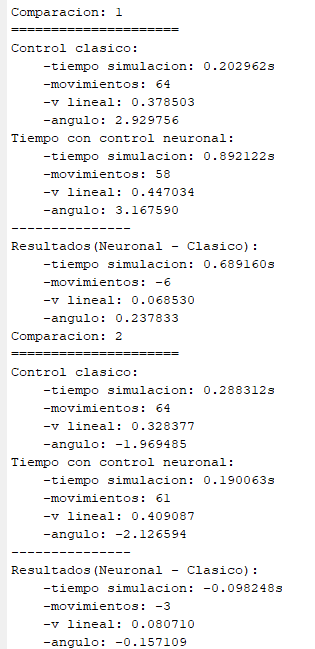
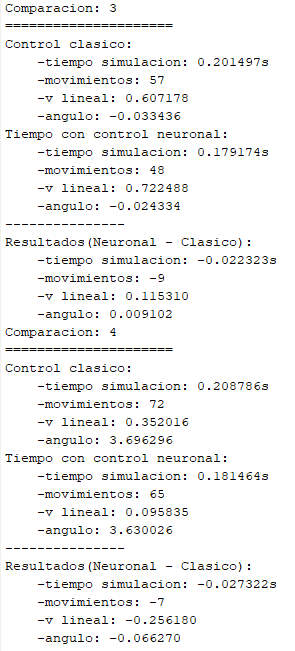
1. Después se nos pide que entrenemos una **red** para **sustituir** el módulo de **control** del sistema del robot, para ello usando los métodos vistos anteriormente, obteniendo una red que es capaz de llegar al punto de destino a partir del origen.



1. Como último apartado de este ejercicio, tenemos que **comparar** el resultado de la red neuronal, con la caja negra para diferentes **puntos**. Para ello, hemos creado un **script**, el cual realiza 5 comparaciones (se pueden poner las que se quieran):



Podemos apreciar, que el módulo de control neuronal funciona **correctamente** llegando siempre al punto de **destino**, aunque en muchos casos **difiere** de la **ruta** seguida por el módulo de control original. Para poder realizar más comparaciones hemos medido también **tiempos**, **velocidades** lineales, **movimientos** y **ángulos**.



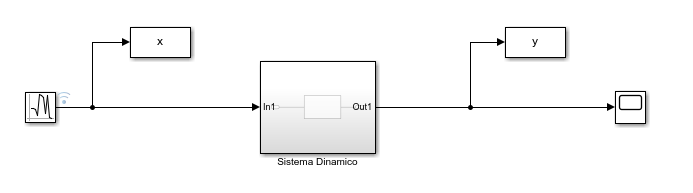
En estos resultados vemos que en el **tiempo**, es mucho **menor** en el caso del **control neuronal** al igual que el número de **movimientos**, esto se debe a que tiende a hacer **menos giros** y más avances en línea recta, como las **medias de ángulos** corroboran. Teniendo en cuenta esto vemos que la **velocidad lineal** es **mayor** en el caso de la **red neuronal**, motivo que explica porque la simulación la realiza en un menor tiempo que la del control normal. Estos **datos** tienen que verse con **cuidado** porque se tratan de **medias**, que pueden **no representar** correctamente al **conjunto** de datos.

## Parte III

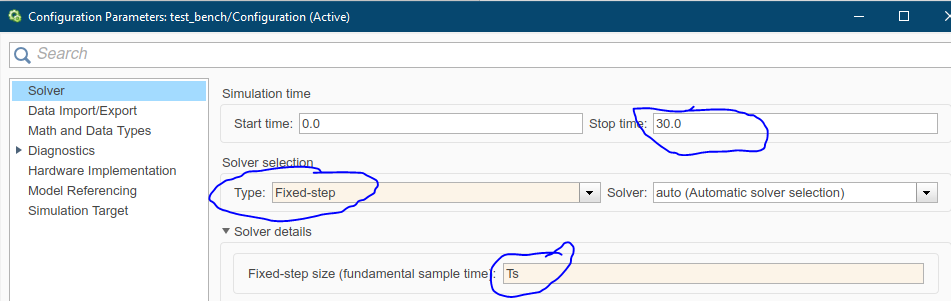
### Ejercicio 1

(Poner pie de foto mira el ejercicio 2 como guia)

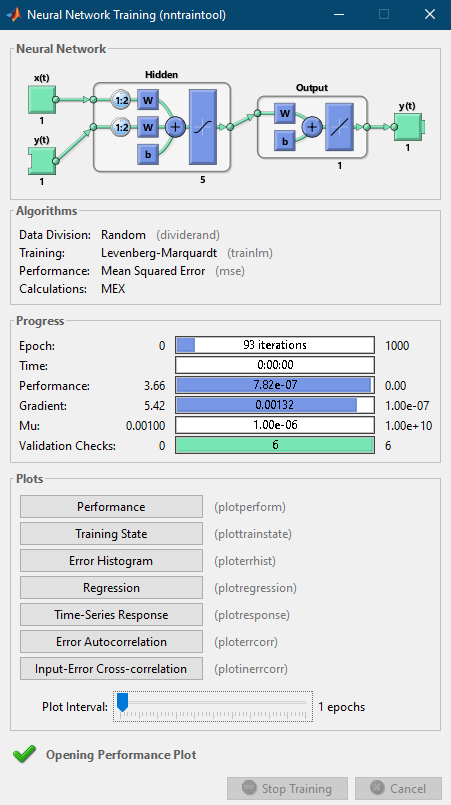
En este ejercicio nos piden que identifiquemos un sistema utilizando una red recursiva de tipo **NARX**. Para ello la primera parte consiste en crear un sistema que emule la respuesta del sistema, se compone de un generador de datos, una caja negra y un **Scope** para visualizar en forma de ondas los datos. Lo podemos ver en la siguiente imagen.



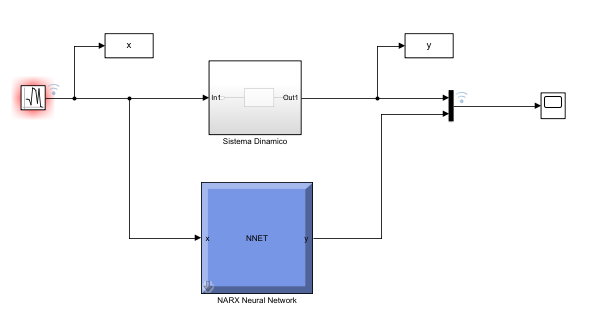
Con el sistema creado, lo configuramos poniendo un tiempo de simulación de **30s** la frecuencia como la variable **Ts** y el solver con **Fixed-step** (Para poder poner Ts), además de tiempo discreto.



Una vez configurado, copiamos el código que nos dan en la práctica para así poder entrenar una red NARX. La red entrenada obtuvo los siguientes datos, que son buenos resultados al tener una buena **perfomance** muy próxima al 0:



Una vez obtenida la red NARX entrenada, modificamos el sistema anterior mostrado, añadiendo la NARX entre el **Scope** y la entrada **x** para así **comparar** ambos sistemas.



Dado este sistema, podemos ver como la red generada y el modelo de sistema, dan casi los mismos resultados en la gráfica.



Como se puede apreciar, el **error** es muy **pequeño** de la red NARX frente al modelo original, cabe destacar que al **principio** es donde más error hay, ya que es donde menos coinciden las líneas del modelo original y la red NARX entrenada, en el resto de los casos coincide a la perfección, viendo así que la red NARX está bien entrenada y funciona correctamente en este caso.

### Ejercicio 2

1. Para la creación del esquema de simulink hemos utilizado el modelo anterior de la parte control, sustituyendo el módulo de control por el nuevo, y el sistema de referencia con el TrajectoryGeneration.slx. Además, en Matlab declaramos en un script los **valores iniciales** para x\_0, y\_0 y theta\_0.
2. En esta parte hemos creado el script RunTrajectoryControl.m en el que aparte de indicar los valores iniciales, hemos sacado las **posiciones actuales** y las de **referencia** (exportándolas del simulink al workspace), para compararlas en un **gráfico** usando **plot** y **hold on** para mostrar ambas. Al hacer esta comparación podremos ver si se asemeja al objetivo que en este caso serían los puntos de la trayectoria generada, que como podemos ver es bastante precisa, excepto en algunas zonas donde difiere un poco al comienzo.

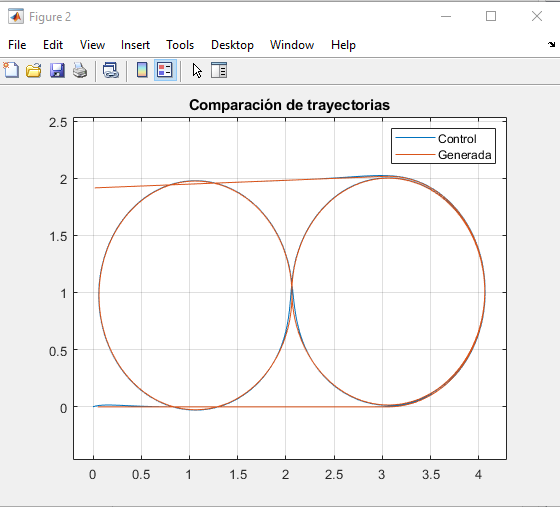


Fig. 1 Comparación de las trayectorias del sistema.

1. Siguiendo como base el ejemplo del ejercicio anterior hemos entrenado varios modelos neuronales, que han requerido de bastantes pruebas de **acierto y error**, con distintas neuronas. Al tener que sustituir solo el controlador los datos que nos interesaban eran los **errores** como **entradas** y las **velocidades** como **salidas**. Otro cambio que hemos tenido que hacer ha sido el tamaño de los arrays de tipo cell, haciendo que tuvieran tamaño 2 para permitir que introducir los dos errores y las dos velocidades dejándolo así **mat2cell (**inputs, 2, **ones(**nT,1**)).**  Realizando bastantes entrenamientos obtuvimos bastantes redes, muchas de ellas no funcionaban correctamente, y muy pocas que lo hacían relativamente bien hasta que se cambiaban las posiciones. Por lo que optamos por generar **más datos** con distintas **referencias** para solventar el problema, obteniendo así un modelo con un buen rendimiento y cuyo funcionamiento en la práctica era bueno y permitía variar sus posiciones iniciales, aunque solo funciono bien en la de 10 neuronas. Los criterios de selección de las redes por **experimentación** se basaban en una primera etapa en el **performance** de la red, que era el **error cuadrático medio**, escogiendo aquellos que tuvieran uno muy cercano al **cero**, pero también era necesario comprobar que la **validación** se realizará correctamente y no nos produjera una red **sobreentrenada,** que memorizase los datos de entrenamiento. Esto último lo hicimos viendo las gráficas que **plotperform** ofrecía y viendo cuando se paraba y como estaban los datos en comparación al resto. Con estas redes entrenadas generamos un bloque de **simulink** que pusimos en la red para probar finalmente, escogiendo así los mejores teóricamente. Las neuronas que escogimos para los distintos modelos fueron de 5, 10, 15 y 20 neuronas, trabajando con un único dato generado y después con un conjunto de 50 datos. Como se observa la mayoría con un único dato ofrecen una performance muy buena al ser cercana al 0 destacando la de **15 neuronas** con un error cuadrático de 2.71e-06.

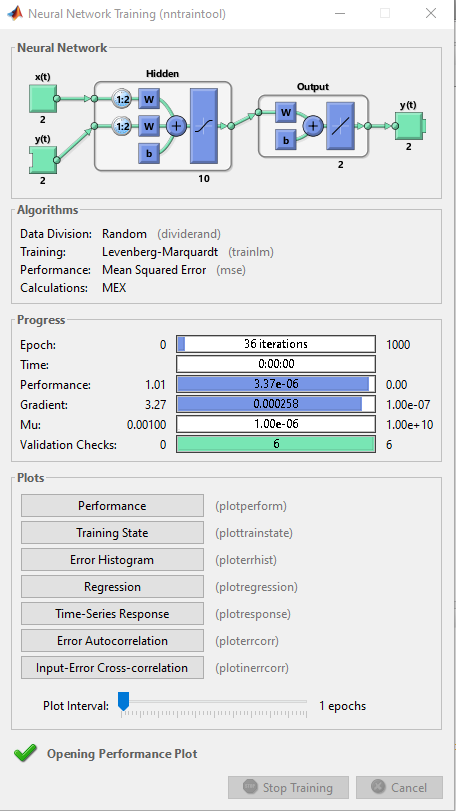
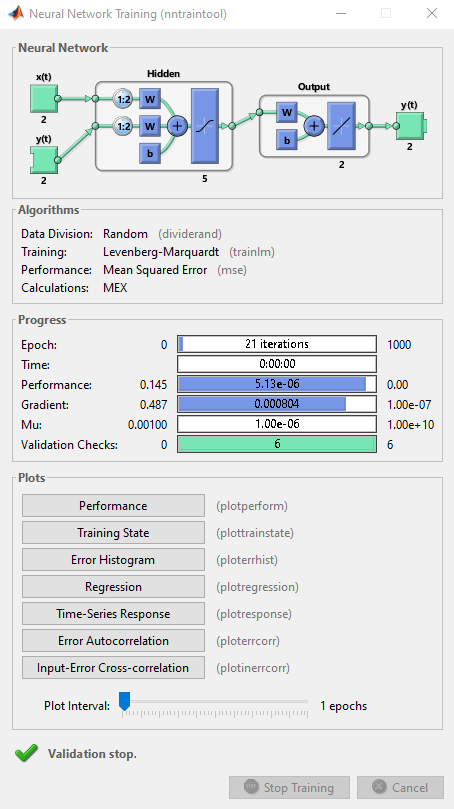
 

Fig. 1 Entrenamiento redes de 5 y 10 neuronas de un solo dato.

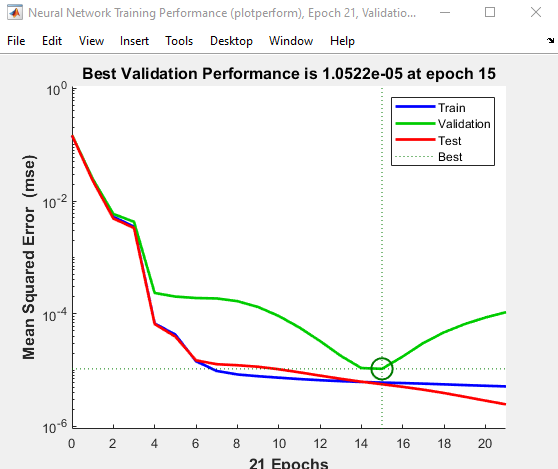


Fig. 1 Validación de la red de 5 neuronas de un solo dato.

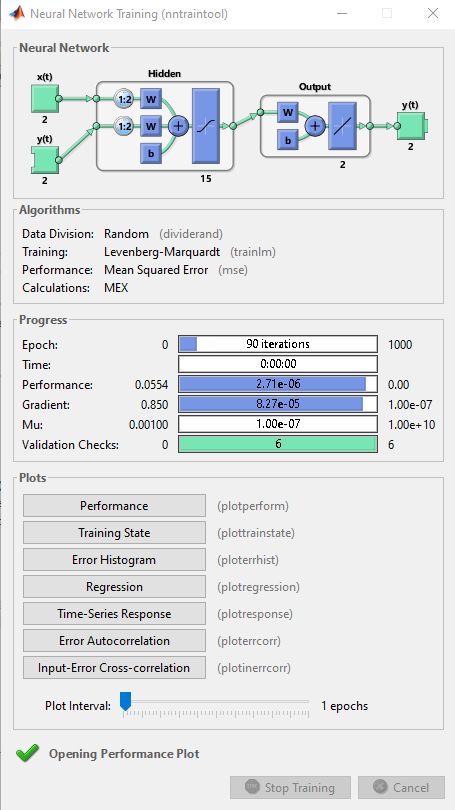
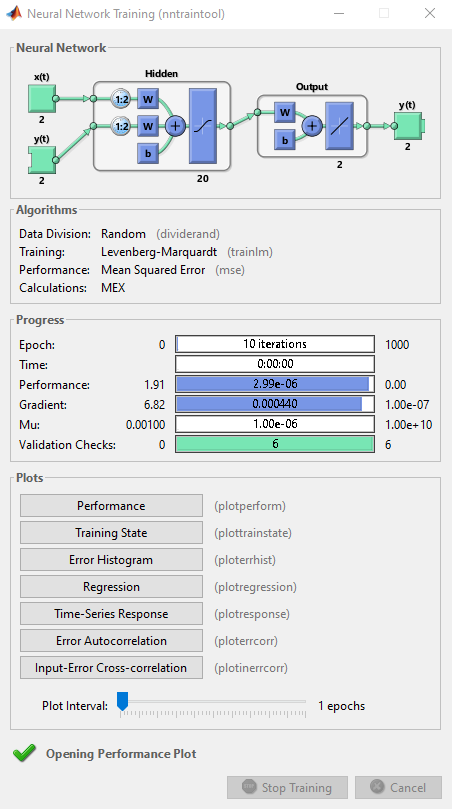
 

Fig. 1 Entrenamiento redes de 15 y 20 neuronas de un solo dato.

Con las de múltiples datos este error ya sube bastante porque aumenta la dificultad del entrenamiento de la red. El **mejor** resultado de rendimiento se obtiene con la de **20 neuronas** con un 0.008 y el **peor** con la de **5 neuronas** con un 0.0123, siendo la diferencia muy alta con respecto a las demás.

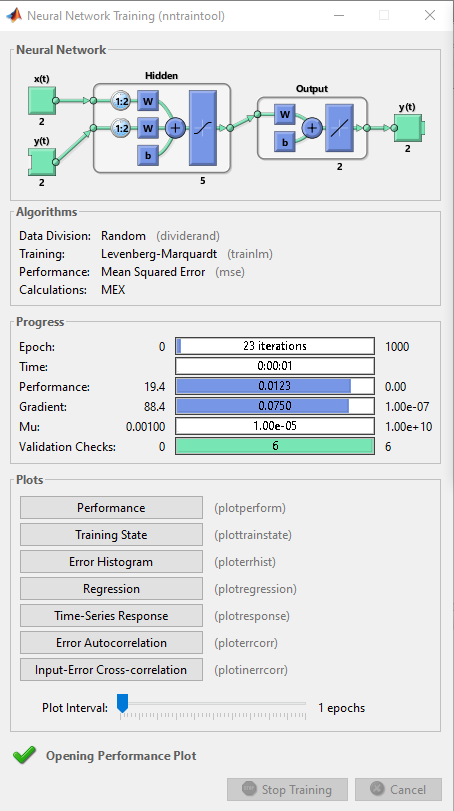
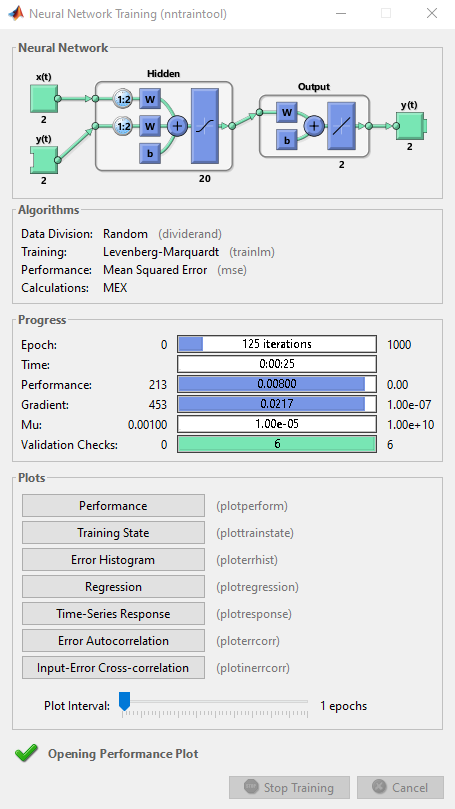
 

Fig. 1 Entrenamiento redes de 5 y 20 neuronas de múltiples datos.

1. En esta parte lo que hemos hecho ha sido un pequeño **script** para **comparar** superponiendo las **trayectorias** del control original sobre el control de la red neuronal, para así observar si las redes en la **práctica** son igual de efectivas. Para ello hemos cogido los modelos anteriores para hacer prueba y error y ver los resultados. Sin embargo, la mayoría de los casos a pesar de tener un rendimiento **teórico** alto con una buena validación **distan** de aproximarse a los resultados **reales** salvo el caso de la red de 10 neuronas de múltiples datos, el resto o realizan trayectorias **aleatorias** o imitan la que deben recorrer **erróneamente**. En definitiva se trata de realizar muchas pruebas hasta tener suerte y dar con una buena red, aunque con ciertas modificaciones se puede dar con resultados mejores (más neuronas, más datos, etc.).

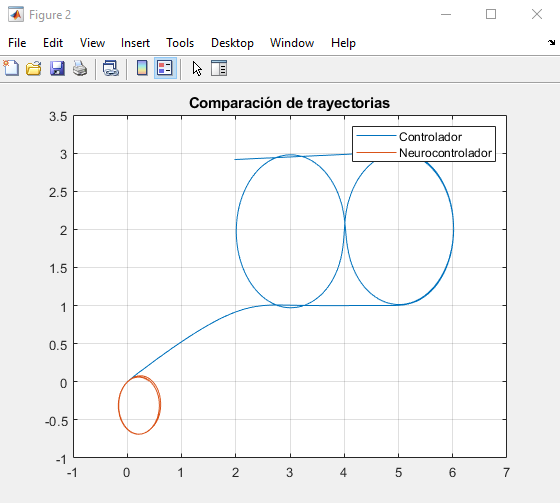


Fig. 1 Trayectoria aleatoria generada por la red de 5 neuronas y un solo dato.

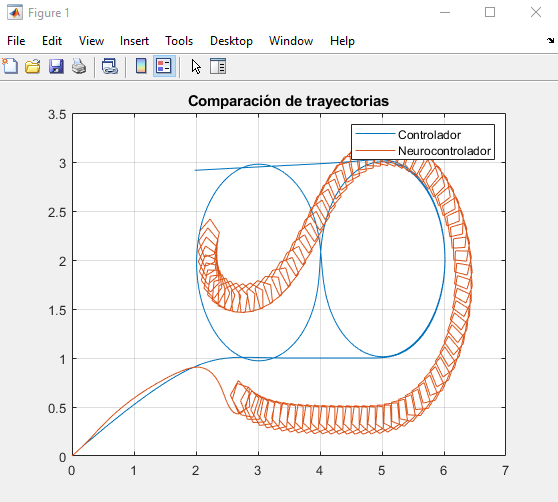
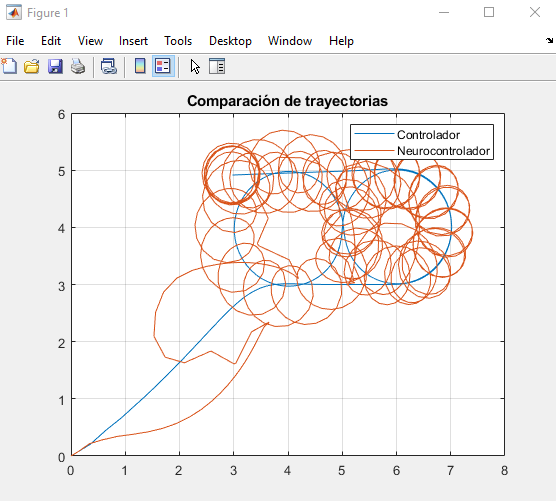
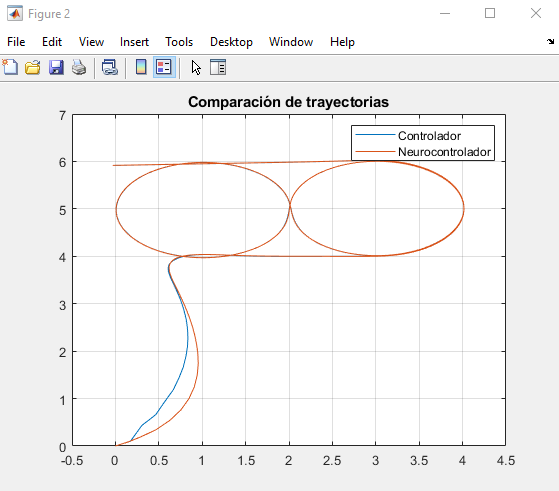
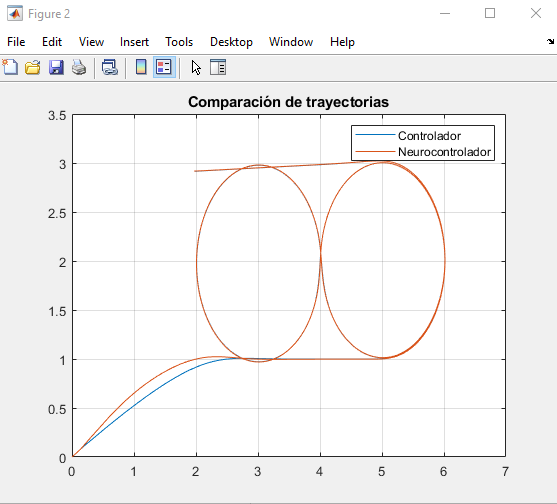
 

Fig. 1 Trayectorias erróneas generadas por las redes de 5 y 15 neuronas de múltiples datos.

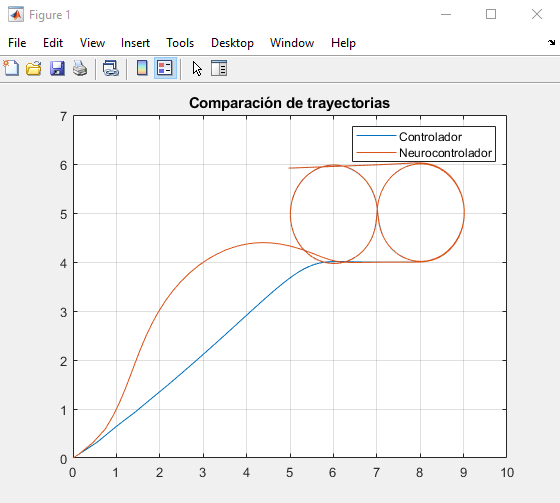


Fig. 1 Trayectorias aproximadas generadas por la red de 10 neuronas y múltiples datos.

## 3.Conclusión

Se puede concluir viendo todas las resoluciones de los ejercicios que Terminarlo

## 4.Bibliografía

* <https://es.mathworks.com/help/map/ref/wraptopi.html>
* <https://es.mathworks.com/help/deeplearning/ref/trainrp.html>
* <https://es.mathworks.com/help/deeplearning/ref/traingd.html>
* <https://es.mathworks.com/help/deeplearning/ref/trainscg.html>
* <https://es.mathworks.com/help/deeplearning/ref/trainbfg.html>