Sobreentrenamiento

Aprendizaje Automático

Una de las cuestiones más importantes en el mundo del *Machine Learning* es saber distinguir correctamente entre errores de entrenamiento y test. El proceso de entrenamiento permitirá modificar los parámetros de la RNA (pesos y bias) de tal forma que se minimice el *loss* **en el conjunto de patrones usados para el entrenamiento**. Si este conjunto de patrones es lo suficientemente representativo, entonces la RNA funcionará bien en su entorno de trabajo, es decir, con patrones nuevos que no están en el conjunto de entrenamiento. En cambio, si este conjunto de patrones no ha sido bien escogido, la RNA podría no funcionar bien con patrones nuevos. La única forma de saberlo es precisamente evaluar la RNA con patrones nuevos y calcular el error con estos patrones de **test**. Dado que el conjunto de test se tiene desde el principio, esta evaluación con el conjunto de test se realiza a la vez que con el conjunto de entrenamiento (pero sólo se entrena con el conjunto de entrenamiento).

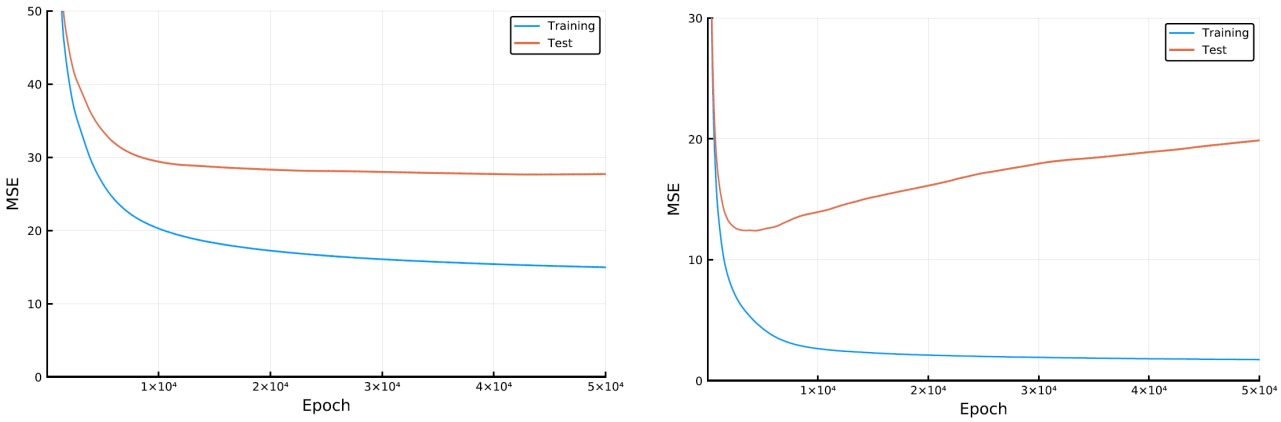
Por lo tanto, para saber cómo de bien entrenada está una RNA, esto se puede saber evaluando el conjunto de test, no el de entrenamiento. Habitualmente se tiene un solo conjunto de datos, con lo que se suele dividir en 2 subconjuntos: entrenamiento y test. Para realizar esta división existen varias alternativas; en estas prácticas se realizarán dos distintas:

- *Hold Out*. Consiste en realizar un experimento con una partición simple del conjunto de datos, dejando un porcentaje determinado para test, que suele oscilar alrededor de un 20% y un 30%.

- Validación cruzada (*crossvalidation*). Consiste en dividir el conjunto de datos en k subconjuntos disjuntos y realizar k experimentos. En el k-ésimo experimento, se separa el k-ésimo conjunto para hacer test, y se entrena con los k-1 restantes. Un valor habitual de k es 10, con lo que se tiene un *10-fold crossvalidation*.

Concretamente, en esta práctica nos centraremos en la primera, dejando la segunda para una práctica posterior. La técnica *hold out* permite partir el conjunto de datos en dos subconjuntos: entrenamiento y test. El subconjunto de entrenamiento permite ajustar los parámetros del modelo (pesos de conexiones y bias), mientras que el subconjunto de test permite probar la RNA (o, en general, el modelo que se esté utilizando, como un árbol de decisión) entrenada en patrones nuevos, no presentes en el conjunto de entrenamiento.

Si el conjunto de entrenamiento está bien escogido (y, por lo tanto, es lo suficientemente representativo), el proceso de entrenamiento podría ser similar a la figura de la izquierda, en el que ambos errores, entrenamiento y test, van disminuyendo en cada ciclo. Sin embargo, en general no se puede asegurar que el conjunto de entrenamiento esté libre de ruido y todo lo bien escogido que sería deseable, por lo que al entrenar suele ocurrir el fenómeno de **sobreentrenamiento** o sobreajuste. Esto se puede ver claramente en la figura de la derecha, en la que ambos errores van disminuyendo hasta que llega un punto en el que el error de test empieza a aumentar. A partir de ese punto el sistema empieza a sobreajustarse.



Para evitar el sobreentrenamiento, en el mundo de las RR.NN.AA. existen una serie de técnicas denominadas de **regularización**, que son cada una de naturaleza muy distinta, pero que todas tienen como objetivo evitar el sobreentrenamiento. Estas técnicas se estudian con más profundidad en asignaturas posteriores; en estas prácticas vamos a estudiar una de las más conocidas, vista en clase de teoría, llamada “Parada temprana” o *Early Stopping*.

Esta técnica se basa en dividir el conjunto de patrones no en 2 conjuntos, sino en 3: entrenamiento, validación y test. Los conjuntos de entrenamiento y test realizan las funciones ya descritas. Por su parte, el objetivo del conjunto de validación es el de intentar evitar el sobreentrenamiento controlando el proceso de entrenamiento, pero no se usa para modificar pesos ni bias. Existen varias formas de controlar el proceso de entrenamiento, aquí usaremos dos de ellas de forma conjunta:

- Nuevo criterio de parada del proceso de entrenamiento. Se tiene un nuevo parámetro: número máximo de ciclos consecutivos sin mejorar el *loss* en el conjunto de validación. Si transcurren estos ciclos sin mejorar el mejor *loss* de validación alcanzado hasta el momento, se para el entrenamiento. Como el conjunto de validación no se usa para modificar pesos ni bias, da una estimación de cuál podría ser el error de test. Por tanto, cuando aumenta el error de validación, está aumentando el error en patrones que no ha visto, por lo que se estima que el error de test aumentaría, y la red se estaría sobreentrenando.

- Una vez parado el proceso de entrenamiento, el conjunto de validación dictamina cuál es la red que se devuelve. Durante todos los ciclos de entrenamiento se va calculando el *loss* en el conjunto de entrenamiento, necesario para modificar pesos y bias, pero también en el conjunto de validación. Una vez que se para el entrenamiento, se devuelve la RNA correspondiente no al último ciclo, sino correspondiente al ciclo con un menor error en el conjunto de validación. Esto se realiza porque se estima que esta RNA es la que dará un menor error en el conjunto de test, es decir, la que se comportará mejor con patrones nuevos, es decir, la que generalizará mejor.

Una vez parado el proceso de entrenamiento y devuelta una RNA, se puede evaluar el conjunto de test para valorar cómo ha sido entrenada la RNA.

- **Importante**: Los errores (o precisiones, o la métrica que se use) en los conjuntos de entrenamiento y validación son interesantes, pero el que realmente importa es el de test.

Sin embargo, en la mayoría de las ocasiones se tienen los 3 conjuntos (entrenamiento, validación y test) desde el principio, por lo que a medida que se evalúan los conjuntos de entrenamiento y validación, también se evalúa el de test, lo cual es útil para sacar gráficas. Sin embargo, estos valores en el conjunto de test no se pueden utilizar para tomar decisiones.

- Si lo que queremos conseguir es una RNA que de un error en test lo menor posible, ¿por qué no se escoge la RNA correspondiente al ciclo con un menor error en test?

Por lo tanto, en el bucle de entrenamiento es necesario mantener una referencia a cuál es el mejor error de validación encontrado hasta el momento, así como una copia de la RNA que ha dado ese error de validación, e ir actualizando ambos en cada ciclo si el error de validación mejora. También es necesario llevar un contador con cuántos ciclos consecutivos se lleva sin mejorar el error de validación.

Como se puede ver, el conjunto de test no tiene ningún tipo de participación en el proceso de entrenamiento. Esto es lo correcto, puesto que se está evaluando una RNA con patrones totalmente nuevos. Sin embargo, desde un punto de vista muy estricto, sí hay una parte en todo este proceso en la que el conjunto de test está teniendo una pequeña influencia: en la normalización de los datos.

Habitualmente se parte de un conjunto de datos que primero se normaliza, para luego ser dividido en entrenamiento/test. Si se hace de esta manera, los patrones usados para test tienen cierta influencia en el cálculo de los valores de los parámetros de normalización, lo cual no sería correcto. Sin embargo, el impacto que suelen tener en estos parámetros y en general en el funcionamiento de

la RNA es tan bajo, que no se suele tener en cuenta esto y se normalizan todos los patrones juntos. De todas formas, lo realmente correcto sería calcular los valores de los parámetros de normalización solamente con el conjunto de entrenamiento, o con los conjuntos de entrenamiento y validación, si se va a hacer.

Para este ejercicio, se pide realizar dos conjuntos de experimentos, sin usar validación y posteriormente usando conjunto de validación. En primer lugar, lo que se pide es lo siguiente:

- Desarrollar una función llamada *holdOut* que dados dos parámetros, N (igual al número de patrones) y P (valor entre 0 y 1, que indica el porcentaje de patrones que se separarán para el conjunto de test), devuelva una tupla con dos vectores con los índices de los patrones que serán utilizados para entrenamiento y test (el primero para entrenamiento y el segundo para test). La suma de longitudes de ambos vectores tiene que ser igual a N, y estos dos vectores tienen que ser disjuntos.

    - Esta función se puede hacer en de una forma muy sencilla usando la función *randperm* (para usarla, cargar el módulo con *using Random*)

    A partir de esta función, partir una base de datos en dos subconjuntos se realiza de una forma inmediata.

    Para realizar esta función no se permite el uso de bucles.

- Desarrollar otra función llamada *holdOut*, igual que en la anterior y que, basándose en ella, tome 3 parámetros: N (número de patrones), Pval (tasa de patrones en el conjunto de validación) y Ptest (tasa de patrones en el conjunto de test), y devuelva una tupla con 3 vectores, con los índices de los elementos de los conjuntos de entrenamiento, validación y test. La suma de las longitudes de estos 3 vectores tiene que ser igual a N.

    - Para realizar esto, simplemente hay que hacer dos llamadas a la función *holdOut* desarrollada previamente. La primera llamada separa el conjunto de test, y la segunda el conjunto de validación. La primera llamada se puede hacer usando los propios parámetros N y Ptest. Sin embargo, la segunda llamada utilizará la longitud del vector de elementos resultante de eliminar los elementos de test de la primera llamada. En este punto, tened en cuenta que Pval hace referencia la tasa de patrones en el conjunto de test con respecto al conjunto original, no con respecto a este subconjunto, que es más pequeño. Por ese motivo, en la segunda llamada esta tasa de validación debe de ser transformada para que sea más grande (para tomar el

mismo número de elementos de un conjunto más pequeño, se necesita una tasa más grande). Se deja como ejercicio sencillo el cálculo de la tasa de elementos para el conjunto de validación para esta segunda llamada.

Al igual que en la función anterior, para realizar esta no se permite el uso de bucles.

- Modificar la función *trainClassANN* de entrenamiento de la RNA realizada en la práctica anterior para que acepte los siguientes parámetros **opcionales** a mayores de los definidos en la práctica anterior:

    - Conjunto de validación, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}*, es decir, una tupla de 2 matrices: entradas y salidas deseadas. Su valor por defecto son matrices vacías. La matriz de entradas deberá ser convertida a *Float32* en el interior de la función.

    - Conjunto de test, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}*, es decir, una tupla de 2 matrices: entradas y salidas deseadas. Su valor por defecto son matrices vacías. La matriz de entradas deberá ser convertida a *Float32* en el interior de la función.

    - *maxEpochsVal*, de tipo *Int*, que define el número de ciclos sin mejorar el mejor *loss* de validación encontrado hasta el momento que han de transcurrir para parar el entrenamiento. Es decir, define un nuevo criterio de parada. Su valor por defecto es 20.

    Con estos parámetros, esta función debería implementar la estrategia de **parada temprana**, para lo cual es necesario tener cuidado con que la RNA que se devuelve al final de la función debe ser:

        - Si se ha pasado un conjunto de validación como parámetro (es decir, si este no es vacío), la RNA a devolver no tiene que ser la que se está entrenando, sino la que haya obtenido un mejor error de validación.

        - Si no se ha pasado un conjunto de validación como parámetro (es decir, si este es vacío), la RNA a devolver deberá ser la correspondiente al último ciclo de entrenamiento. Por lo tanto, esta nueva función de entrenamiento debería funcionar igual que la anterior en caso de que no se pase como parámetro un conjunto de validación.

Además, esta función debería devolver una tupla con 4 elementos. El primero debe de ser la RNA entrenada, y los 3 siguientes deberían de ser vectores con los valores de *loss* obtenidos en los conjuntos de entrenamiento, validación y test en cada ciclo. Al igual que en la función del ejercicio anterior, si se entrenan n ciclos, los vectores con los valores de *loss* a devolver deberían tener n+1 elementos, siendo el primero de ellos el valor de *loss* en el conjunto de entrenamiento/validación/test (validación y test si estos conjuntos no están vacíos) correspondientes al "ciclo 0", es decir, antes de entrar en el bucle de entrenamiento ni realizar ninguna llamada a la función *train!* Además, los valores de *loss* deberán estar en *Float32*, puesto que es en este tipo en el que se realizarán todas las operaciones.

- Para añadir elementos al final de un vector, consultar la función *push!*
- Tened en cuenta que los valores de *loss* obtenidos con la RNA con pesos aleatorios, previos a realizar el entrenamiento, se suelen considerar como el ciclo 0, y deben estar recogidos en los vectores de valores de *loss* (entrenamiento, validación y/o test)
- En los casos en que no haya conjuntos de validación y/o test, los vectores de *loss* correspondientes deberán estar vacíos.

Como se pone en la descripción anterior, será necesario almacenar la mejor RNA alcanzada hasta el momento y actualizarla en algunas iteraciones del bucle. Para realizar esto, no se puede hacer sencillamente una asignación a una nueva variable, puesto que sólo se asignaría el puntero, que apuntaría a la misma dirección de memoria con la RNA que se modificaría en la siguiente iteración. Para hacer una copia de un objeto de tal forma que todos los objetos (y datos, como en este caso pesos y bias) que contiene se copien también, de forma recursiva, se puede usar la función *deepcopy*.

Esta función sustituye a la del ejercicio anterior, por lo que no se basa en ella. Esto quiere decir que el entrenamiento del caso más básico (sin conjunto de validación) no se realiza mediante una llamada a la función del ejercicio anterior, sino que hay que escribir de nuevo el bucle. De hecho, si se intenta llamar a la función del ejercicio anterior, esto provocaría una llamada recursiva a esta misma función, lo que daría lugar a un error por desbordamiento de la pila (*Stack Overflow*).

Al igual que en la práctica anterior, para realizar esta función se puede utilizar un único bucle, para iterar por los ciclos de entrenamiento o *epochs*. Por lo tanto, en la realización de esta función no hay que hacer dos bucles distintos, para los casos de tener y no tener conjunto de

validación (es decir, que este conjunto esté vacío), sino un solo bucle, en cuyo interior se haga la comprobación de si el conjunto de validación está vacío o no, para calcular los valores de *loss* correspondientes y hacer la parada temprana si es el caso.

- Dado que había dos versiones de la función de entrenamiento de RR.NN.AA. (la segunda aceptaba como salidas deseadas un vector), modificar de la misma manera esta función para que las salidas deseadas sean vectores en lugar de matrices para el caso de clasificación en dos clases. Tened en cuenta que en este caso los tipos de los conjuntos de entrenamiento, validación y test pasados como parámetros serán *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}}* para los tres y por tanto será necesario llamar a la función *reshape* en los tres.

  Esta función debe devolver lo mismo que la anterior. Además, al igual que en la práctica anterior, esta función no debe utilizar ningún bucle.

- Integrar el código con el resultante de las prácticas anteriores, de tal manera que la secuencia de pasos sea la siguiente:

  1. Cargar la base de datos, teniendo los patrones en filas y atributos y salidas deseadas en columnas.

  2. Utilizar la función *holdOut* para dividir el conjunto de datos en entrenamiento, validación y test con los porcentajes que se desee. Estos porcentajes pueden ser igual a 0. Se tendrá, por lo tanto, 6 matrices: entradas y salidas deseadas en entrenamiento, validación y test (algunas podrían estar vacías si el porcentaje correspondiente es 0).

  3. Calcular los valores de los parámetros correspondientes al tipo de normalización que se va a usar con vuestros datos (máximo/mínimo o media/desviación típica para cada atributo), únicamente del conjunto de entrenamiento.

     - En esta parte no se pide normalizar el conjunto de entrenamiento, sino calcular los valores de normalización a partir del conjunto de entrenamiento.

  4. Con estos valores calculados en el paso anterior, normalizar conjuntos de entrenamiento, validación y test.

  5. Entrenar distintas arquitecturas, y, para cada una de ellas, sacar gráficas de cómo ha sido la evolución de los valores de *loss* de entrenamiento, validación y test en la misma gráfica, incluyendo el ciclo 0.

- ¿El error de test ha disminuido siempre o ha llegado un punto en el que ha empezado a aumentar?
- ¿Cómo es la evolución de los 3 valores de precisión?
- ¿Por qué se suele parar el entrenamiento?
- ¿A qué ciclo se corresponde la RNA que devuelve la función?

Al final de este documento se incluyen las firmas de las funciones a realizar en este ejercicio.

Aprende Julia:

Julia dispone de una librería que permite mostrar todo tipo de gráficas de una forma muy sencilla. Para cargarla, simplemente hay que poner *using Plots*, la documentación se encuentra en http://docs.juliaplots.org/

En realidad, Plots.jl no es un paquete para mostrar gráficas, sino un interfaz para un conjunto de librerías que permiten representar gráficas. Plots.jl provee de un conjunto de funciones que permiten, de una forma unificada, mostrar gráficas usando una u otra librería, con las mismas llamadas. Es decir, lo que hace Plots.jl es interpretar los comandos y generar las gráficas usando otra librería de gráficos, a la que se refiere como un *backend*. Por lo tanto, es posible cambiar de librería gráfica (*backend*) sin necesidad de realizar modificaciones en el código, puesto que las llamadas a las funciones correspondientes son las mismas para todos los *backends*.

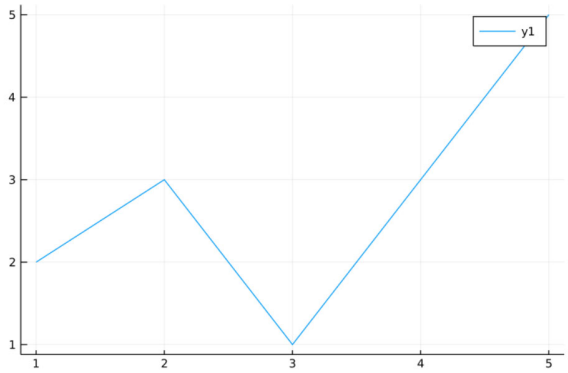
Por lo tanto, en primer lugar, después de cargar Plots, es seleccionar el *backend* con el que se quiere trabajar. Julia provee de una gran cantidad muy usados, los cuales hay que instalar de la manera habitual si se quieren utilizar. Si no se indica cuál se quiere usar, Julia por defecto escoge uno dependiendo de cuáles están instalados. Para ver qué *backend* está siendo utilizado, se puede escribir simplemente *backend()*.

Algunos de los backends más comunes son Plotly, PyPlot, PlotlyJS y GR. En general, para utilizar un *backend*, simplemente hay que hacer una llamada a una función con nombre el propio backend, pero en minúsculas. Para los 4 puestos como ejemplos a principios de este párrafo, sería con llamadas a *plotly()*, *pyplot()*, *plotlyjs()* y *gr()* respectivamente. Como se ha dicho antes, en caso de que no esté instalado habrá que instalarlo de la forma habitual, escribiendo *Pkg.add("Plotly")*, *Pkg.add("PyPlot")*, *Pkg.add("PlotlyJS")* o *Pkg.add("GR")* respectivamente. En https://docs.juliaplots.org/latest/backends/

podéis encontrar más información sobre los distintos *backends*.

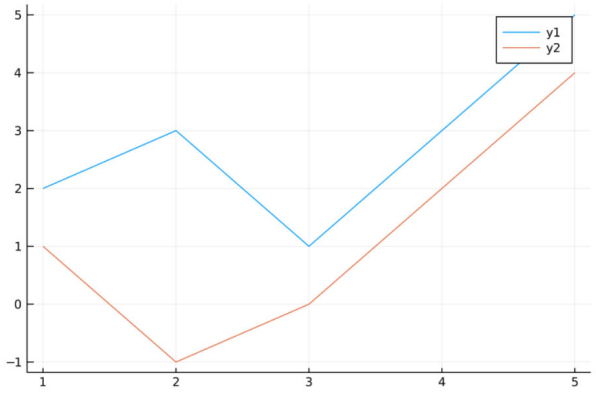
Una vez cargado el *backend* deseado, o con el *backend* por defecto, ya se puede comenzar a hacer gráficas. La forma más sencilla es utilizar la función *plot*, que recibe dos parámetros: la serie que se pone en el eje x, y la serie que se pone en el eje y, ambos como vectores, por ejemplo:

```
plot(1:5, [2, 3, 1, 3, 5])
```



En general, si se quiere poner más de una serie en la gráfica, es suficiente con especificar una columna por serie en la matriz que se le pasa para los datos del eje y, por ejemplo:

```
plot(1:5, [2 1; 3 -1; 1 0; 3 2; 5 4])
```



Otra forma de hacerlo es añadiendo más series al objeto *plot* que se ha creado. Es decir, la llamada a *plot* devuelve un objeto que puede ser modificado mediante posteriores llamadas a *plot!* y pasándolo como primer parámetro (recordad que cuando el nombre de una función termina en "!" se modifica el argumento pasado). El ejemplo anterior se podría realizar de la siguiente manera:

```
g = plot(1:5, [2, 3, 1, 3, 5])
plot!(g, 1:5, [1, -1, 0, 2, 4])
```

Otra posibilidad sería la siguiente:

```
g = plot()
plot!(g, 1:5, [2, 3, 1, 3, 5])
plot!(g, 1:5, [1, -1, 0, 2, 4])
```

En cualquiera de estas tres posibilidades, si esto se realiza en el intérprete de comandos, aparecerá la gráfica. De hecho, en la segunda y tercera posibilidad aparecerá más de una gráfica, a medida que las vayamos creando. Sin embargo, si esto se realiza en un script, la gráfica no aparece de forma automática. Si queremos que aparezca, hay que hacer una llamada a *display* de esta manera:

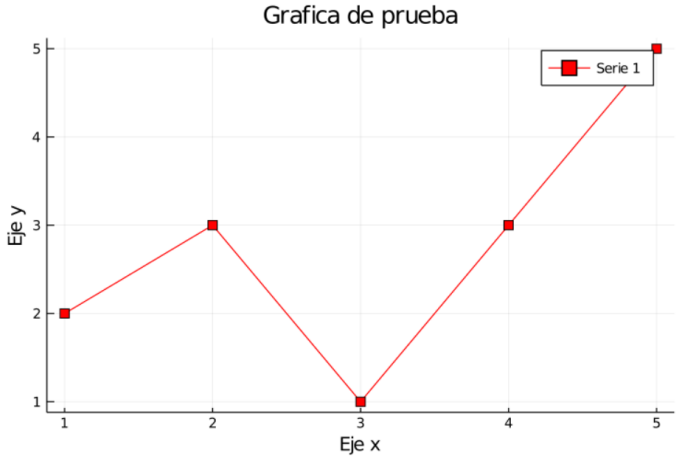
```
display(g)
```

Aparte de la función *plot*, existen muchas otras funciones que permiten representar otros tipos de gráficas con los nombres habituales, como *heatmap*, *plot3d*, *scatter*, *histogram*, *boxplot*, *violin*, etc., todos ellos con su correspondiente función terminada en "!". Como siempre, podéis consultar la ayuda de una función escribiendo, por ejemplo:

```
? boxplot
```

Generalmente se desea añadir más información en las gráficas que simplemente los datos utilizados en la misma. Esta información suele incluir título, etiquetas en los ejes, leyenda, marcadores en cada punto, tipo de línea, colores, etc. Esto se suele realizar mediante atributos pasados como *keywords* en las llamadas a las funciones, como pueden ser *axis*, label, *line*, *fill*, *marker*, *ticks*, *title*, *xlabel*, *ylabel*, por ejemplo:

```
plot(1:5, [2, 3, 1, 3, 5], xaxis = "Eje x", yaxis = "Eje y", title = "Grafica de prueba", marker = :square, color = :red, label = "Serie 1")
```



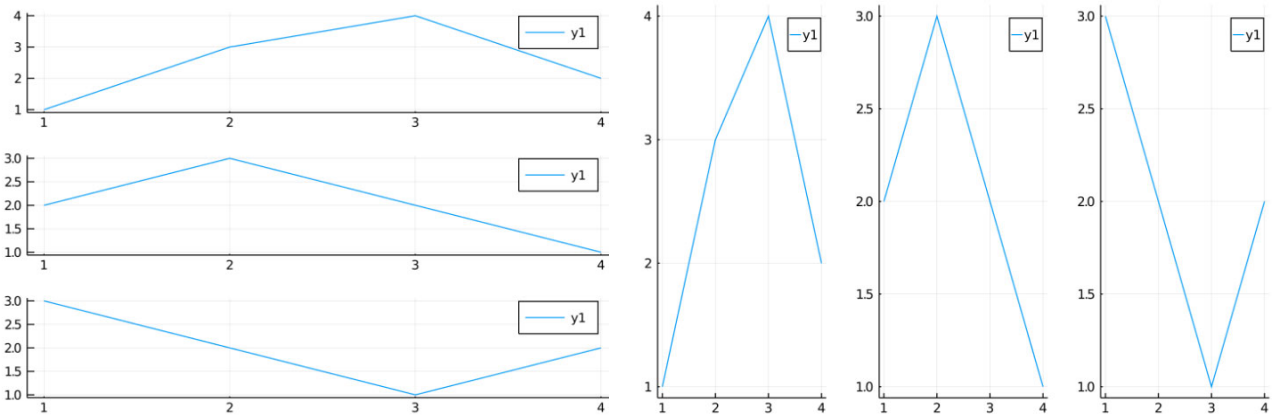
En https://docs.juliaplots.org/latest/attributes/ tenéis una descripción rápida de los *keywords* más comunes. Por otra parte, en https://docs.juliaplots.org/latest/generated/supported/ tenéis una lista

completa de los distintos tipos de series, keywords, marcadores, estilos de línea y escalas soportados por cada *backend*. Como podéis ver, la lista de gráficas y argumentos es bastante extensa, lo que da una idea de las enormes posibilidades que ofrece Julia a la hora de representar gráficas.

Otra cuestión común a la hora de mostrar gráficas es combinar varias dentro de una única ventana. Para realizar esto existen varios métodos, siendo el más sencillo y utilizado mediante el uso del *keyword layout* en la llamada a la función correspondiente para generar la gráfica que contenga todas. En este sentido, lo más sencillo es generar las gráficas de forma independiente, y combinarlas más tarde en una nueva llamada usando el *keyword layout*, que recibe como parámetro una tupla con el número de filas y columnas de la matriz de gráficas, por ejemplo:

```
p1 = plot(1:4, [1, 3, 4, 2]);
p2 = plot(1:4, [2, 3, 2, 1]);
p3 = plot(1:4, [3, 2, 1, 2]);
plot(p1, p2, p3, layout = (3,1));
plot(p1, p2, p3, layout = (1,3));
```

En el primer caso, se genera la siguiente imagen de la izquierda, mientras que en el segundo caso se genera la imagen de la derecha:



En https://docs.juliaplots.org/latest/layouts/ tenéis más información sobre las formas de combinar gráficas.

Finalmente, otra acción muy común es el guardar las gráficas que se generan. Esto se puede realizar mediante la función *savefig*, que recibe como parámetros la gráfica a guardar y el nombre del archivo. A partir de la extensión indicada en el nombre del archivo, Julia ya guarda la gráfica en el formato indicado. Algunos de los formatos más típicos son pdf, png o ps. A pesar de ser una forma muy sencilla de guardar las gráficas, no todos los tipos de archivos están soportados por todos los

backends. En https://docs.juliaplots.org/latest/output/ podéis consultar los formatos de archivo soportados por cada *backend*.

Firmas de las funciones:

A continuación se muestran las firmas de las funciones a realizar en los ejercicios propuestos. Tened en cuenta que, dependiendo de cómo se defina la función, esta puede contener o no la palabra reservada *function* al principio:

```
function holdOut(N::Int, P::Real)
function holdOut(N::Int, Pval::Real, Ptest::Real)

function trainClassANN(topology::AbstractArray{<:Int,1},
    trainingDataset::  Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}};
    validationDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}=
       (Array{eltype(trainingDataset[1]),2}(undef,0,size(trainingDataset[1],2)),
             falses(0,size(trainingDataset[2],2))),
    testDataset:: Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}=
       (Array{eltype(trainingDataset[1]),2}(undef,0,size(trainingDataset[1],2)),
             falses(0,size(trainingDataset[2],2))),
    transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)),
    maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.01,
    maxEpochsVal::Int=20)

function trainClassANN(topology::AbstractArray{<:Int,1},
    trainingDataset::  Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}};
    validationDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}}=
       (Array{eltype(trainingDataset[1]),2}(undef,0,size(trainingDataset[1],2)),
             falses(0)),
    testDataset:: Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}}=
       (Array{eltype(trainingDataset[1]),2}(undef,0,size(trainingDataset[1],2)),
             falses(0)),
    transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)),
    maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.01,
    maxEpochsVal::Int=20)
```