Introducción al Aprendizaje Automático y al lenguaje Julia

Aprendizaje Automático I

Las prácticas de esta asignatura consistirán en el estudio de la aplicabilidad de distintos modelos de Aprendizaje Automático para resolver un problema del mundo real seleccionado por cada equipo de trabajo. Los modelos a utilizar serán los siguientes:

- Redes de Neuronas Artificiales.
- Máquinas de Soporte Vectorial.
- Árboles de Decisión.
- kNN.

Con respecto a la primera técnica, RR.NN.AA., esta ha sido descrita en la asignatura ya cursada Sistemas Inteligentes, por lo que no se realizará en esta asignatura un desarrollo teórico más allá que para complementar lo explicado en la asignatura anterior. Sin embargo, esta técnica sí se utilizará en prácticas, con el cambio importante de que en Sistemas Inteligentes las prácticas fueron desarrolladas en Matlab, y en esta asignatura estas serán en Julia. El objetivo de usar Julia es poder profundizar en todo el proceso de utilización de RR.NN.AA., y, en general, de un sistema de Aprendizaje Automático. Por este motivo, las próximas prácticas estarán orientadas a aprender cómo entrenar correctamente una RNA en Julia.

En general, durante las prácticas se realizarán dos tareas en paralelo:

- En las primeras semanas, mediante los tutoriales que se proporcionarán, realizar tareas de codificación de distintas partes de un sistema de Aprendizaje Automático, tales como una validación cruzada o las métricas a utilizar. Dado que en las primeras semanas el grado de desarrollo de la aplicación práctica va a ser muy preliminar, estas prácticas se realizarán con el objetivo de resolver un problema muy conocido, el problema de clasificación de flores iris. A medida que se vaya desarrollando el código en estas prácticas, este deberá ir siendo integrado en el sistema final que se utilizará. Es decir, en cada práctica se desarrollará una parte del sistema final.
- De forma paralela, cada equipo de trabajo deberá realizar las tareas propias del

problema que ha propuesto resolver, que incluyen adquisición de datos, análisis de datos, carga de la base de datos en Julia, etc. A medida que se avance en los tutoriales, estos avances deberán integrarse y aplicarse para resolver el problema, lo que se verá plasmado en las distintas entregas de memoria.

Por lo tanto, durante las primeras semanas el objetivo será doble: por una parte aprender a implementar las distintas partes de un sistema de aprendizaje automático, y por otra comenzar el trabajo en la aplicación práctica propuesta, integrando el código que se va desarrollando de forma incremental. Una vez se hayan terminado los tutoriales, el trabajo se centrará únicamente en la resolución del problema propuesto, llevándose a cabo mediante distintas aproximaciones.

Como se ha indicado, para desarrollar el código en los tutoriales se utilizará como problema de ejemplo el de flores iris. Esta es posiblemente la base de datos más conocida en el campo de reconocimiento de patrones. Esta base de datos fue publicada por Fisher en el año 1936 como ejemplo de análisis discriminante lineal, y desde entonces ha sido utilizada en multitud de ocasiones como *benchmark* para probar nuevos sistemas o, sencillamente, para el aprendizaje. Esta base de datos contiene 150 instancias pertenecientes a 3 clases: Iris Setosa, Iris Versicolor e Iris Virginica, teniendo 50 instancias cada clase. Es, por lo tanto, un problema de clasificación. Una de estas clases en linealmente separable de las otras dos, mientras que estas no son linealmente separables la una de la otra. Cada instancia consta de 4 atributos, que son longitudes y anchuras de sépalos y pétalos, medidos en centímetros.

La base de datos a utilizar en estas prácticas se puede descargar de la página web del UCI, en la siguiente dirección: https://archive.ics.uci.edu/ml/machine-learning-databases/iris/ Concretamente, el archivo a descargar se llama “iris.names”.

El objetivo de este primer tutorial es instalar y familiarizarse con Julia, desarrollando el código necesario para cargar esta base de datos en Julia y realizar un preprocesado básico. Este preprocesado básico tiene que ver con el uso de entradas y/o salidas categóricas en lugar de ser numéricas. Algunos modelos como las RR.NN.AA. no permiten el uso de valores categóricos, por lo que es necesario convertirlos en valores numéricos. Para realizar esta tarea hay tres opciones:

- Si solamente hay dos categorías, por ejemplo verdadero/falso, verde/azul, madera/metal o caro/barato, ese atributo se transforma en otro único atributo, que

toma el valor *false* o 0 para una categoría y *true* o 1 para la otra.

- Si hay más de dos categorías, por ejemplo rojo/verde/azul, madera/metal/plástico o coche/barco/avión/tren, se transforma en tantos atributos como posibilidades haya, uno para a cada categoría, con valor 1 para aquellas instancias que pertenezcan a ella y 0 para las que no. Por ejemplo, en el caso rojo/verde/azul los patrones con valor “rojo” pasarán a ser (1, 0, 0), los “verde” (0, 1, 0) y los “azul” (0, 0, 1).

- Existe una tercera posibilidad, cuando hay más de dos categorías, que consiste en convertirlas en un único número real. Por ejemplo, A/B/C/D podría convertirse en 0/0.33/0.66/1. Sin embargo, este caso solamente es interesante cuando en el mundo real exista un orden A < B < C < D, y por lo tanto no es aplicable en el caso de flores iris.

En el caso de la base de datos de flores iris, esta situación se produce únicamente en la salida deseada, por lo que es necesario codificarla.

En esta primera semana, se pide:

- Tener un primer contacto con Julia, instalar los paquetes necesarios y aprender los conceptos más básicos de Julia.

- Descargar la base de datos de flores iris de la dirección indicada y cargar la base de datos en Julia.

    - Crear una matriz con las entradas y otra con las salidas deseadas, cada una con el tipo más adecuado.

        - ¿Qué dimensiones tienen las matrices?

- Desarrollar una función que permita codificar los valores categóricos en valores booleanos, distinguiendo los dos casos más comunes (tener únicamente dos clases y tener más de dos clases).

    - El caso a contemplar de estos dos posibles debe ser automáticamente detectado dentro de la función.

    - Esta función debe recibir un vector de entrada (matriz unidimensional) con los datos y devolver un vector (matriz unidimensional) o matriz (bidimensional) de salida, en función de la forma de codificación.

    - Desarrollar este código sin utilizar bucles que recorran todos los patrones (sí se puede hacer un bucle que recorra las clases o los atributos, pero sólo se puede realizar este bucle).
    - Para hacer este código, puede ser útil la función *unique*.
    - Esta función será aplicada a cada una de las entradas/salidas categóricas que tenga el problema escogido.
- Como resultado de esta práctica, después de aplicar esta función a las entradas o salidas categóricas, se debe disponer de dos matrices (entradas y salidas deseadas) para su uso en las siguientes prácticas.

Aprende Julia:

Para la implementación de las soluciones, existen distintas posibilidades en cuanto al lenguaje de programación. Entre la gran cantidad de opciones disponibles, se han tomado en consideración tres posibilidades:

- Matlab. Este es un lenguaje científico con muchos años de trayectoria, y por lo tanto tiene como ventaja que posee una gran cantidad de módulos (llamados *Toolboxes*) para casi cualquier operación que se desee realizar, junto con una excelente documentación. Esto le convierte en un lenguaje muy indicado para iniciarse y aprender. Sin embargo, tiene como principal inconveniente que es necesario adquirir una licencia para utilizarlo. Si bien la Facultad de Informática y la Universidade da Coruña posee licencia para su uso, el hecho de que sea necesario adquirirla para utilizarlo de forma profesional ha hecho que muchas empresas no se decanten por esta opción, por lo que en la práctica a nivel empresarial no es tan utilizado como python.
- Python. Sin duda, es el más utilizado. Es un lenguaje moderno, y sencillo, con una gran cantidad de módulos y abundante documentación, aunque sin llegar al número ni calidad de Matlab. Es gratuito y de código abierto, y de propósito general, lo que ha hecho que sea uno de los lenguajes más utilizados en la actualidad en el mundo empresarial. Además, una de las primeras librerías de *Deep Learning*, llamada *Tensorflow*, fue desarrollada por Google para este lenguaje, lo que ha aumentado

drásticamente la comunidad de desarrolladores de aplicaciones de aprendizaje máquina en este lenguaje. Librerías como Scikit-Learn permiten también el uso de otras técnicas de aprendizaje máquina como árboles de decisión o Máquinas de Soporte Vectorial. El mayor problema que tiene este lenguaje es que no es un lenguaje científico sino de propósito general, y la programación vectorial no está soportada de forma nativa, sino a través de la librería numpy, lo que conlleva una considerable pérdida de rendimiento.

- Julia. Este es un lenguaje emergente, de vida muy corta, y totalmente científico, desarrollado en el Instituto Tecnológico de Massachussets. Su primera versión estable tiene unos pocos años de vida, y actualmente se encuentra bajo un intenso desarrollo. Por este motivo, no posee el mismo número de módulos que Matlab o Python, aunque la cantidad que posee está aumentando muy rápidamente, puesto que el lenguaje es gratuito y de código abierto. Este lenguaje ha sido desarrollado como un punto intermedio entre Matlab, como lenguaje científico, y python, como lenguaje sencillo y de código abierto, con una velocidad de ejecución superior a ambos. Como principales inconvenientes tiene que la comunidad de desarrolladores no es tan grande como la de python, y el número de módulos no es tan grande como el de Matlab, y, dado su corto período de tiempo, todavía no ha hecho presencia en el mundo empresarial.

Entre estos tres lenguajes se ha escogido Julia para realizar las prácticas de esta asignatura, puesto que el tiempo de cómputo de los algoritmos de aprendizaje máquina es un factor clave en las prácticas, y Julia es el que ofrece mejor rendimiento.

Como se ha indicado, a nivel empresarial es un lenguaje que todavía no tiene presencia, sin embargo, esto se ve paliado por dos factores importantes:

- Al haber adquirido destreza en programar en lenguaje python en otras asignaturas, esta asignatura supone la oportunidad de aprender un lenguaje científico que complete los conocimientos de programación.

- A pesar de que a nivel empresarial python sea el lenguaje más utilizado, la librería Scikit-Learn está disponible también en Julia, así que su aprendizaje en esta asignatura implicaría que podrían utilizarla también en su trabajo en una empresa en el lenguaje python sin esfuerzo, al tener conocimientos tanto de dicha librería como de dicho lenguaje.

Como ya se ha indicado, Julia es un lenguaje gratuito y de código abierto. Existen varias distribuciones disponibles en internet; sin embargo, se recomienda instalar la última versión estable de Julia Pro disponible en https://juliacomputing.com/

En esta asignatura se utilizarán distintos paquetes de Julia. Algunos de ellos ya vienen instalados, por ejemplo el paquete Statistics. Para utilizarlo simplemente se puede escribir al inicio de un script "using Statistics". Si solamente se va a usar una o varias funciones y no se desea cargar todo el paquete, se puede indicar esto, escribiendo algo como "using Statistics: mean". Otros paquetes no vienen por defecto en la distribución de Julia, y es necesario instalarlos. Algunos de estos paquetes son los siguientes:

- FileIO: Permite operar con archivos.
- XLSX: Permite cargar archivos de Excel en Julia.
- JLD2: Permite cargar y guardar variables de Julia en archivo. Es necesario tener instalado el paquete FileIO.
- Flux: Permite entrenar Redes de Neuronas Artificiales en Julia.
- ScikitLearn: Librería de python que provee de un interfaz uniforme para entrenar y utilizar modelos de Aprendizaje Automático, así como otras utilidades. En Julia, estas funcionalidades están disponibles en la misma librería.
- Plots: Permite representar gráficas en Julia.
- MAT: Permite cargar variables de Matlab en Julia.
- Images: Permite trabajar con imágenes.

La primera vez que se usan, estos se precompilan. Por ejemplo, si los datos a cargar están en formato de hoja Excel, para cargarlos se puede usar la función *readdata* del paquete XLSX. Para lo cual, este paquete debe estar previamente instalado, lo que se puede hacer escribiendo en la línea de comandos lo siguiente:

```
import Pkg; Pkg.add("XLSX");
```

Y para usar la función en un script, se puede escribir "*using XLSX: readdata*" al principio del script.

Para realizar esta práctica, es necesario realizar dos llamadas para cargar las dos matrices. Si estos datos están en una hoja Excel, estas llamadas serán algo similar a esto:

```
inputs = readdata("iris.xlsx","inputs","A1:D150");
```

```
targets = readdata("iris.xlsx","targets","A1:C150");
```

De esta forma, se cargan dos variables en memoria, una con la matriz de entradas y otra con la matriz de salidas deseadas.

En otras ocasiones, los datos estarán en un archivo de texto con algún tipo de delimitador. Para ello es útil el paquete DelimitedFiles. Este paquete se puede instalar de la forma habitual con

```
import Pkg; Pkg.add("DelimitedFiles");
```

Igualmente, para utilizarlo solamente hay que escribir al principio del *script* lo siguiente:

```
using DelimitedFiles
```

Una vez cargado, para cargar una base de datos, esto se puede hacer con la siguiente línea:

```
dataset = readdlm("iris.data",',');
```

Como se puede ver, el primer parámetro es el nombre del archivo, mientras que el segundo es el delimitador o delimitadores que se van a utilizar. En este caso, se carga toda la base de datos en una única variable llamada *dataset*, que será una matriz bidimensional. En la mayoría de las ocasiones será necesario separar las entradas de las salidas deseadas. Esto se puede hacer con las siguientes líneas:

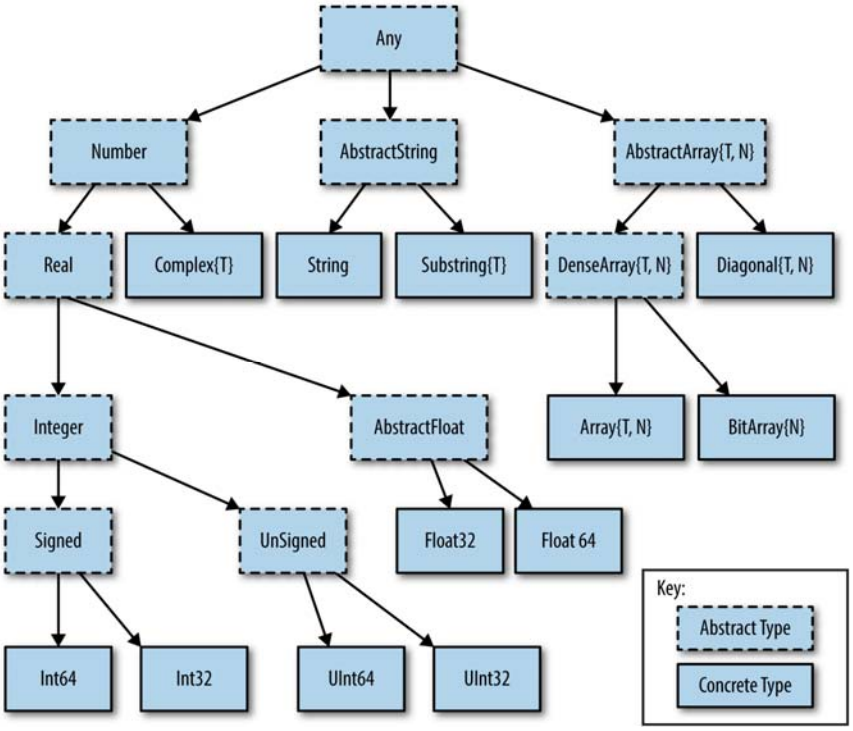
```
inputs = dataset[:,1:4];
```

```
targets = dataset[:,5];
```

El operador de corchetes permite referenciar un dato o una parte de una matriz, el primer elemento se refiere a las filas y el segundo a las columnas. Cuando se pone dos puntos (:), se dice que se quiere referenciar todas las filas o todas las columnas, devolviendo por lo tanto una matriz en lugar de un elemento. Esto será descrito más en profundidad más adelante en este tutorial. En el caso particular de la base de datos de flores iris, las cuatro

primeras columnas son las entradas y la quinta se corresponde con la salida deseada, por ese motivo en estas dos líneas se pueden ver los valores 1:4 y 5. En otra base de datos, las columnas serían distintas.

Es importante tener en cuenta que todas las variables tienen un tipo. Julia provee de una jerarquía de tipos donde el tipo de la raíz se denomina *Any*. Es decir, cualquier variable será de tipo Any. En este caso, al leer de una hoja de cálculo una gran cantidad de datos que pueden ser de naturaleza distinta (números, fechas, cadenas de caracteres, etc.), devuelve una matriz bidimensional donde cada elemento es de tipo *Any*. En Julia, esto se representa como *Array{Any,2}*, donde *Any* indica el tipo de cada elemento de la matriz, y el 2 indica el número de dimensiones. En la siguiente imagen se puede ver la jerarquía de varios tipos comunes en Julia (Fuente: *Learning Julia: Abstract, Concrete, and Parametric Types by Spencer Russell, Leah Hanson*):



Como se puede ver, por ejemplo, una variable de tipo *Int64*, a su vez, es de tipos *Signed*, *Integer*, *Real*, *Number* y *Any*. Para ver de qué tipo es una variable, se puede usar la función *typeof*; para comprobar si una variable es de un tipo determinado, se puede usar la función *isa*.

- ¿Cuál será el resultado de las siguientes llamadas?
  - `typeof(Array{Float64,2})`

- `isa(Array{Float64,2}, Any)`
- `isa(typeof(Array{Float64,2}), Any)`

Generalmente los tipos más utilizados en Aprendizaje Automático para almacenar los datos numéricos serán *Float32* o *Float64*. *Float32* es muy utilizado en el mundo del Aprendizaje Automático porque es el tipo de datos que utiliza la mayoría de las tarjetas gráficas. Sin embargo, el tipo más utilizado, y el que usaremos en esta asignatura, es *Float64*.

Por lo tanto, es necesario convertir los datos que usaremos, de *Array{Any,2}* a *Array{Float64,2}*. Una posibilidad es usar la función *convert*, que intenta hacer una conversión a un tipo especificado, con lo que podría hacerse algo como:

```
inputs = convert(Array{Float64,2},inputs);
```

```
targets = convert(Array{Float64,2},targets);
```

En este último ejemplo se han convertido las salidas deseadas a una matriz bidimensional de valores reales, lo cual es útil en problemas de regresión con varias salidas. Tened en cuenta que esto habría que adaptarlo al problema en cuestión. Por ejemplo, si es un problema de clasificación en dos clases y el archivo tiene valores numéricos de 0 y 1 para cada clase, esto se podría convertir en valores booleanos con algo como:

```
targets = convert(Array{Bool,1},targets);
```

Si el problema es de clasificación y hay más de dos clases, habría que realizar una conversión más compleja, en una matriz donde el número de columnas es igual al número de clases y para cada patrón se tiene un valor de *true* en la columna correspondiente a la clase a la que pertenece y *false* en el resto.

Otra posibilidad es forzar un tipado usando el propio tipo como si fuese una función. Por ejemplo, se puede especificar que un número sea de un tipo determinado escribiendo algo como *Float64(8)*, *Float32(8)*, *Int64(8)*, *UInt32(8)*, etc. Sin embargo, no se puede hacer *Float64(inputs)*, porque *inputs* no es de tipo *Number*, por lo que esa conversión no es posible. Lo que se quiere no es forzar el tipo de la matriz, sino crear una nueva matriz donde cada elemento sea el resultado de forzar el tipo del elemento correspondiente de la matriz inicial, es decir, hacer un ***broadcast*** del forzado de tipo.

Uno de los puntos en los que Julia es más potente es en el manejo de matrices

multidimensionales. Julia permite, entre otras cosas, aplicar funciones a todos los elementos de una matriz y construir la matriz resultado de una forma automatizada. En este caso, se dice que se ha hecho un *broadcast* de esa función en toda la matriz. A continuación, un ejemplo sencillo: se define la función que calcula el cuadrado de un número de la siguiente manera:

```
cuadrado(x::Number) = x*x
```

En este caso, se ha especificado que el tipo del argumento es *Number*, con lo que se obliga a que el argumento sea *Int32*, *Int64*, *Float32*, *Float64*, etc. (si no se especifica el tipo del argumento, por defecto Julia entiende que es de tipo *Any*). Se está, por tanto, definiendo una función **entre números, no entre matrices**. Por ejemplo, la siguiente llamada daría un error:

```
cuadrado([1 2 3])
```

puesto que esta función está definida entre números y se está pasando como argumento una matriz, concretamente de una fila y 3 columnas. Sin embargo, si se desea construir una nueva matriz del mismo tamaño que la matriz original donde cada elemento sea el resultante de aplicar esta función al elemento correspondiente de la matriz original, esto se puede hacer escribiendo un punto '.' después del nombre de la función, de la siguiente manera:

```
cuadrado.([1 2 3])
```

De esta forma, se indica a Julia que se debe aplicar esa función elemento a elemento. Realizar estas operaciones de *broadcast* permite desarrollar código mucho más limpio, puesto que se evita escribir bucles, y eficiente, puesto que Julia es capaz de distribuir estas operaciones paralelas en distintos núcleos. Otra forma alternativa de realizar este proceso sería el siguiente:

```
[cuadrado(x) for x in [1 2 3]]
```

Otro ejemplo es el siguiente, con la operación suma:

```
suma(a::Number, b::Number) = a+b
```

Esta llamada dará error, puesto que no está definida la función 'suma' entre matrices:

```
suma([1 2 3],[2 3 4])
```

Sin embargo, se puede hacer *broadcast* de esta operación entre los elementos de ambas matrices, de la siguiente manera:

```
suma.([1 2 3],[2 3 4])
```

En este caso, ambas matrices tienen que tener el mismo tamaño. Incluso es posible que uno de los argumentos sea un número y el otro una matriz.

- ¿Cuál será el resultado de hacer *suma.(1,[2 3 4])* y *suma.([1 2 3],3)*?

En general, las operaciones matemáticas más comunes tienen implementados operadores de *broadcast*. Por ejemplo, si A y B son dos matrices del mismo tamaño, para operar elemento a elemento se puede hacer A.+B, A.*B, A./B, o A.-B. Otro ejemplo es hacer A.^2, donde se construye una matriz nueva donde cada elemento es el correspondiente, elevado al cuadrado.

Volviendo a los tipos, por lo tanto es equivalente escribir estas líneas:

```
inputs = Float64.(inputs); targets = Float64.(targets);
```

```
inputs = [Float64(x) for x in inputs]; targets = [Float64(x) for x in targets];
```

```
inputs = convert(Array{Float64,2},inputs); targets = convert(Array{Float64,2},targets);
```

De nuevo, esta conversión de las salidas deseadas es útil cuando el problema es de regresión. Si el problema es de clasificación con 2 clases, y nos dan estas salidas deseadas como valores numéricos 0 o 1, serían equivalentes las siguientes líneas:

```
targets = Bool.(targets);
```

```
targets = [Bool(x) for x in targets];
```

```
targets = convert(Array{Bool,1},targets);
```

Para el caso de tener más de dos clases, habría que realizar un procesado un poco más complejo.

De esta manera, se tendrán dos matrices, una con las entradas (llamada *inputs*) y otra con las salidas deseadas (llamada *targets*), ambas de tipo *Array{Float64,2}*. A pesar de que este

es el tipo que más se va a usar en esta asignatura, tened en cuenta que los tipos de datos en Julia son muy flexibles; por ejemplo se podría tener una variable que contenga una matriz tridimensional donde cada elemento sea un vector, el tipo sería *Array{Array{Float64,1},3}*.

- ¿Qué tipo tendrán los objetos [[]], [[8]] y [[8.]]?

De forma genérica, en Aprendizaje Automático se entiende que las matrices contienen cada instancia en cada fila, mientras que en la matriz de entradas las columnas representan los atributos, y en la matriz de salidas deseadas cada columna representa una salida distinta. Por lo tanto, ambas matrices (*inputs* y *targets*) deben tener el mismo número de filas.

Para saber el número de filas y/o columnas de una matriz se puede usar la función *size*. Esta función devuelve una tupla, con el número de elementos igual al número de dimensiones, en la que cada elemento indica el tamaño de esa dimensión. Por ejemplo, la llamada

```
size(inputs)
```

devuelve (150, 4), es decir, 150 filas y 4 columnas. También es posible llamar a esta función indicando de qué dimensión se quiere leer el tamaño. Por ejemplo,

```
size(inputs,1)
```

devolverá un valor de 150.

En este caso, de haber hecho bien la base de datos en el archivo Excel, ambas matrices deberían tener el mismo número de filas. Sin embargo, este tipo de cuestiones en muchas ocasiones deberían ser comprobados con el objetivo de encontrar posibles errores. Para eso, en muchas partes del código suele ser interesante introducir comprobaciones para verificar que todo está correcto. Cuando se ejecute esa comprobación, si no es cierta, el sistema debería dar un error. A esto se llama hacer *programación defensiva*. En el caso de Julia, esto se puede hacer con la macro *@assert*, a la que se indica la comprobación a realizar, y, de forma opcional, el mensaje de error que debería aparecer, por ejemplo:

```
@assert (size(inputs,1)==size(targets,1)) "Las matrices de entradas y salidas deseadas no tienen el mismo número de filas"
```

Llegados a este punto, deberían estar cargadas dos matrices en memoria, ambas con el mismo número de filas. Importante: Al contrario que en el resto de modelos, en el mundo de

M[2,3,1].

¿Cuál será el resultado de las siguientes llamadas? Al represen
fuera una matriz, ¿esta será una matriz fila o una matriz co
llamada devuelve un vector o una matriz bidimensional?

- `typeof([1,2,3])`
- `typeof([1;2;3])`
- `typeof([1 2 3])`

- `typeof([1 2 3; 4 5 6])`

El operador dos puntos (:) es el operador de rango, que vale para crear vectores. En Matlab J:K es equivalente a crear el vector [J, J+1, ..., K] siempre que J < K. Además, J:D:K es equivalente a [J,J+D, J+2* D,..., K]. Por ejemplo, en Matlab son equivalentes las siguientes operaciones:

```
1:3
```

```
[1 2 3]
```

Sin embargo, en Julia existe un pequeño cambio en este aspecto, si bien la operabilidad sigue siendo la misma. El cambio en Julia consiste en que J:D:K no crea un vector sino un elemento de tipo *UnitRange* o *StepRange*, que almacena los índices inicial y final y el incremento, y que se puede utilizar de la misma manera que un vector. De esta forma, se elimina la necesidad de crear vectores en muchas ocasiones en las que no son necesarios, por ejemplo en bucles. Además, esto se realiza de forma transparente al desarrollador, puesto que, como se ha comentado, la operabilidad es la misma que en el caso de crear vectores explícitos.

El operador dos puntos también se puede utilizar para seleccionar filas, columnas o partes de una matriz. Estos operadores se pueden utilizar junto con la palabra *end*, que indica que el rango terminará en el valor último de la fila o columna. Por ejemplo, se puede probar los siguientes ejemplos, que permiten tomar de la matriz M todas las filas y la primera columna, la primera fila y las columnas a partir de la segunda y la segunda fila y todas las columnas respectivamente:

```
M[:, 1]
```

```
M[1, 2:end]
```

```
M[2, :]
```

Al referenciar elementos de una matriz, hay que tener en cuenta que, por cada dimensión en el que no se indique un rango sino un único valor (por ejemplo, una fila o una columna), se perderá esa dimensión.

- ¿Qué dimensiones y tamaños tendrán las matrices resultado de estas tres operaciones? ¿Qué dimensión se “pierde” en cada una?

Como se indicaba, la operabilidad con este objeto es la misma que con vectores. Por ejemplo, estas dos líneas ofrecen el mismo resultado:

```
M[1, end:-1:1]
M(1, [3, 2, 1])
```

Para transponer matrices, se puede usar la función *transpose*, o utilizar el operador ':

```
M'
```

Como se indicó anteriormente, Julia permite aplicar una función a todos los elementos de una matriz y además realizar operaciones entre los elementos de matrices situados en la misma posición. Para esto último, se antepone un punto (.) a operadores como multiplicación, división o potencia:

```
N = [10 20 30;40 50 60];
N*M
N.*M
N./M
N/10
N>30
N.>30
```

En este último ejemplo, se crea una matriz de valores booleanos en la que cada elemento es el resultado de comprar el elemento correspondiente de la primera matriz con el valor 30.

- ¿Por qué da error al intentar ejecutar N*M pero no N.*M? ¿Por qué da error al intentar ejecutar N>30 pero no N.>30?

Al contrario que en Matlab, Julia no permite que las matrices crezcan dinámicamente. Por ejemplo, en Matlab en la matriz anterior es posible dar valor un elemento situado en una posición que no existe. En este caso, no dará error, sino que Matlab aumentará el tamaño de la matriz hasta esa posición, rellenando los valores nuevos con ceros. En Julia en cambio esto sí provocará un error, puesto que las matrices tienen tamaños definidos.

Para declarar una matriz y por tanto reservar memoria para la misma se puede usar el tipo de esa matriz como si fuera una función, poniendo como primer argumento la forma de inicializar los datos, y después un número por cada dimensión. La forma de inicializar los

datos más común es usar *undef*, que indica que no se quieren inicializar los valores de la matriz (tendrán los valores que hubiera en memoria en ese momento). Por ejemplo, la siguiente llamada crea una matriz bidimensional de 10 filas y 6 columnas:

```
M = Array{Float64,2}(undef, 10, 6)
```

- ¿Por qué dará error escribir M = Array{Float64,2}(undef, 10)?
- ¿Cuál será el resultado de las siguientes llamadas?
    - `typeof(Array{Float64,2}(undef, 4, 15))`
    - `typeof(Array{Float64,2})`
    - `isa(Array{Float64,2}, Array{Float64,2})`
    - `isa(Array{Float64,2}(undef, 4, 10), Array{Float64,2})`

Otra posibilidad es usar las funciones *zeros* o *ones*, que crean matrices del tamaño indicado donde todos los elementos son 0 o 1 respectivamente, por ejemplo:

```
zeros(10, 6)
```

```
ones(13)
```

Para concatenar dos vectores, se pueden poner entre corchetes, separando los elementos por ";" Por ejemplo, la siguiente línea permite crear un vector resultado de concatenar los dos vectores indicados:

```
[[1, 2, 3]; [4, 5, 6]]
```

Para unir matrices, esto se realiza de forma similar, utilizando los corchetes, y separando las matrices por espacio, en caso de querer unirlas añadiendo nuevas columnas (poniendo una "al lado" de la otra), o separando las matrices por ";" en caso de querer unirlas añadiendo nuevas filas (poniendo una "debajo" de la otra). Por supuesto, en el primer caso tiene que coincidir el número de filas, y en el segundo tiene que coincidir el número de columnas. Aquí tenéis un ejemplo de ambas situaciones:

```
[[1 2 3; 4 5 6] [7 8 9; 10 11 12]]
[[1 2 3; 4 5 6]; [7 8 9; 10 11 12]]
```

- ¿Cuál será el resultado de cada una de las dos operaciones anteriores?
- Si *[[1, 2, 3]; [4, 5, 6]]* permite concatenar vectores, ¿cuál será el resultado de las siguientes operaciones? ¿Cuál será el tipo del resultado?

  - `[[1, 2, 3]  [4, 5, 6]]`
  - `[[1, 2, 3], [4, 5, 6]]`
  - `[[1  2  3]  [4  5  6]]`
  - `[[1  2  3]; [4  5  6]]`
  - `[[1  2  3], [4  5  6]]`

Julia dispone de una serie de funciones para trabajar con matrices tales como ones, zeros, size, length, max, min, minmax, rand, inv, det, sum, etc. Algunas de las más utilizadas son:

- zeros: recibe como parámetros el tamaño de cada dimensión, y crea una matriz de esa dimensionalidad y tamaño, donde todos los elementos son igual a 0.
- ones: recibe como parámetros el tamaño de cada dimensión, y crea una matriz de esa dimensionalidad y tamaño, donde todos los elementos son igual a 1.
- rand: recibe como parámetros el tamaño de cada dimensión, y crea una matriz de esa dimensionalidad y tamaño, donde todos los elementos son valores aleatorios.
- size: recibe como parámetro una matriz y devuelve una tupla con tantos elementos como dimensionalidad tenga la matriz, donde cada elemento es el tamaño de esa matriz. Puede ser llamada indicando como segundo argumento la dimensión, y devuelve únicamente el tamaño de esa dimensión.
- maximum: recibe como parámetro una matriz y devuelve el valor máximo de la matriz. Opcionalmente, se puede indicar la dimensión mediante la palabra clave “dims=”, y devuelve una matriz que contiene el valor máximo a lo largo de esa dimensión; por lo tanto, la matriz que devuelve tiene una dimensión menos que la original.
- minimum: realiza la operación similar a máximum, pero devolviendo el valor mínimo en lugar del máximo.
- findall: recibe como parámetro una matriz de valores booleanos y devuelve los índices de los valores positivos.
- sum: recibe como parámetro una matriz y devuelve un vector con la suma de los valores de la matriz. Opcionalmente, se puede indicar la dimensión mediante la

palabra clave “dims=”, y devuelve una matriz que contiene la suma a lo largo de esa dimensión; por lo tanto, la matriz que devuelve tiene una dimensión menos que la original.

En general, para ver la ayuda de una función, simplemente escribiendo “?” en el intérprete de comandos se puede escribir el nombre de una función para ver su documentación.

Como ya se ha dicho, Julia permite hacer un *broadcast* de funciones sobre todos los elementos de una matriz, teniendo como resultado una matriz de la misma dimensionalidad, con los resultados de las operaciones. Por ejemplo, llamar a la función cos.(M) devolverá una matriz con los cosenos de cada elemento de la matriz M. La expresión M.>3 devolverá una matriz binaria, del mismo tamaño que M, con unos en aquellos elementos en los que el elemento de M fuese mayor que 3, y ceros en el resto. La expresión M[1,:].=3 asigna el valor 3 a la primera fila de la matriz M.