Entrenamiento de Perceptrones Multicapa

Aprendizaje Automático I

El objetivo de esta práctica es comenzar a aprender a entrenar perceptrones multicapa en Julia. Para ello, se hará uso de la librería Flux, cuya documentación se puede consultar en https://fluxml.ai/Flux.jl/

La librería Flux contiene funciones para crear redes de neuronas con un número arbitrario de capas de distintos tipos. Esta es una librería pensada para desarrollar proyectos de *Deep Learning*, cuyas RR.NN.AA. suelen tener un número elevado de capas de distinto tipo: *convolucionales*, *maxpooling*, etc. En estas prácticas solamente se desarrollarán perceptrones multicapa, con capas totalmente conectadas (*dense*), teniendo hasta un máximo de dos capas ocultas.

Para crear una RNA en Julia, se puede utilizar la función *Chain*, que recibe como parámetros las capas que va a tener la red (excluyendo la capa de entrada, que no realiza ningún tipo de procesamiento), que pueden ser de naturaleza distinta. Es, por lo tanto, una función con un número variable de parámetros. Para crear cada capa existen distintas funciones, según el tipo de capa que se quiera. Algunos ejemplos son la función *Conv* (para crear capas convolucionales) o *MaxPool* (para crear capas *MaxPooling*), estas capas se utilizan en modelos más avanzados que se verán en asignaturas posteriores. En esta asignatura, al desarrollar perceptrones multicapa, se crearán capas totalmente conectadas con la función *Dense*. Esta función acepta como parámetros el número de entradas, salidas, y la función de transferencia de las neuronas de esta capa. En este aspecto, se distinguen dos casos distintos a la hora de crear RR.NN.AA., según la naturaleza del problema a resolver:

- Problemas de regresión. En este tipo de problemas, la capa de salida suele tener una función de transferencia lineal. Las capas ocultas tienen una función de transferencia no lineal. En los siguientes ejemplos, se utiliza como función de transferencia una sigmoidal en las capas ocultas; podéis encontrar más funciones de transferencia en la documentación de la librería. El primer ejemplo construye una RNA con 10 entradas, una capa oculta con 5 neuronas, y una capa de salida de 1 neurona. El segundo ejemplo construye una RNA con 15 entradas, dos capas ocultas con 12 y 5 neuronas, y una capa de salida con 2 neuronas. Fijaos en que tienen que coincidir los números de neuronas entre distintas capas.

```
ann = Chain(
   Dense(10, 5, σ),
   Dense(5, 1, identity) );

ann = Chain(
   Dense(15, 12, σ),
   Dense(12, 5, σ),
   Dense(5, 2, identity) );
```

- ¿Qué ocurriría si todas las capas de la RNA tienen una función de transferencia lineal?

- Problemas de clasificación. En este caso, como se explicó en clase de teoría, se distinguen dos casos distintos, según si hay dos clases o más de dos clases, recordando que la no pertenencia a ninguna clase se trata como una clase más:

    - Si sólo hay dos clases, se suele hablar de positivos y negativos. En este caso, las salidas deseadas serán 1 o 0, y se tiene por tanto una única neurona oculta. Se desea, por tanto, que esa neurona devuelva valores entre 0 y 1, que se interpretarán como el grado de seguridad que tiene el sistema en que la salida es positiva. Para garantizar que la neurona devuelve un valor acotado entre 0 y 1, se aplica una función sigmoidal en la capa de salida. En el siguiente ejemplo se puede ver una RNA con una capa oculta y una neurona de salida; en este ejemplo, se ha utilizado también la función sigmoidal en la capa oculta, pero esto podría cambiarse.

```
ann = Chain(
   Dense(8, 4, σ),
   Dense(4, 1, σ) );
```

    - Si se tienen más de dos clases, se tiene una neurona de salida por clase. La salida deseada de un patrón es 1 para la neurona correspondiente a la clase a la que pertenece, y 0 para el resto. A esta forma de codificación se le conoce como *one-hot-encoding*. De esta forma, la salida de una neurona para un patrón se puede interpretar como el grado de seguridad de pertenencia de ese patrón a la clase correspondiente a esa neurona. A diferencia del caso

anterior, a las salidas cada neurona de la capa de salida no se aplica una función de transferencia sigmoidal para acotar la salida entre 0 y 1, sino que no se aplica ninguna función (función identidad o *identity*). En su lugar, a las salidas de todas las neuronas se aplica una función *softmax*, que toma valores numéricos sin acotar y devuelve valores numéricos entre 0 y 1 de tal forma que su suma es 1. Por lo tanto, la salida final se puede interpretar como el grado de confianza o seguridad de pertenencia a cada clase. Aunque esta función no constituye una capa de neuronas, en ocasiones, esta última función *softmax* se considera una capa adicional, y, de hecho, desde el punto de vista de la programación con la librería Flux, se realiza efectivamente como si fuera una última capa, como se puede ver en este ejemplo:

```
ann = Chain(
   Dense(9, 5, σ),
   Dense(5, 3, identity),
   softmax );
```

Otra forma de utilizar la función *Chain* es mediante sucesivas llamadas, añadiendo capas a una red ya creada. Para realizar esto, se utiliza el operador de puntos suspensivos al especificar argumentos a una función. A continuación se muestra un ejemplo para dar lugar a una RNA equivalente a la anterior, en el que primero se crea una RNA vacía y se le van añadiendo capas sucesivamente:

```
ann = Chain();
ann = Chain(ann...,  Dense(9, 5, σ) );
ann = Chain(ann...,  Dense(5, 3, identity) );
ann = Chain(ann...,  softmax );
```

Esta variable creada, de cualquiera de ambas formas, se puede usar como función. Por ejemplo, se puede coger la matriz *inputs* creada en la práctica anterior y pasársela a la RNA dando lugar a las salidas de la red simplemente escribiendo lo siguiente:

```
outputs = ann(inputs);
```

- Importante: debido a la formulación usada en el mundo de las RR.NN.AA., las matrices de entradas y salidas de la RNA tienen cada patrón en cada columna, no en cada fila. En la matriz de entradas, por lo tanto, cada fila tendrá cada atributo de

entrada, y en la de salidas deseadas cada fila tendrá una salida de la RNA. En consecuencia, es necesario transponer las matrices creadas en la práctica anterior.

De esta manera, aunque la RNA no esté entrenada, puede ser interesante realizar una llamada similar a esta para verificar que la RNA ha sido creada correctamente, siguiendo el paradigma de la programación defensiva que se os comentaba en la práctica anterior. Una vez se ha verificado que la RNA se ha creado de forma correcta, es momento de entrenarla.

Para entrenar una RNA, como se describe en clase de teoría, se presentan los patrones a la red, se compara la salida que emite con la deseada, y se calcula un valor de *loss* o pérdida, que es el utilizado para modificar los pesos de las conexiones y bias. Por lo tanto, un paso importante es definir esta función de *loss*, que será distinta para problemas de regresión y clasificación. La librería Flux incluye el módulo Losses con una gran cantidad de funciones de *loss* usadas para entrenar RR.NN.AA., en esta asignatura usaremos las más comunes. Por lo tanto, el primer paso es cargar este módulo con

```
using Flux.Losses
```

La forma de utilizar las funciones de *loss* es la misma en todos los casos, indicando como primer argumento las salidas del modelo, como segundo las salidas deseadas (en ambos casos con un patrón en cada columna), y como tercer argumento opcional el *keyword* “agg”, que indica cómo se van a agregar los valores de *loss* para cada patrón. Si no se especifica un valor para este *keyword*, por defecto se realizará una media de todos los valores de *loss* para todos los patrones.

- Para un problema de regresión, la función de *loss* más utilizada es el error cuadrático medio (ECM o MSE, *Mean Square Error*) entre las salidas y las salidas deseadas. Esta función *mse* ya viene definida en el módulo Losses de Flux, aunque es muy sencilla su definición. Se puede utilizar de la siguiente manera, siendo x las entradas e y las salidas deseadas:

```
loss(x, y) = Losses.mse(ann(x), y)
```

  Otras funciones de *loss* que pueden ser interesantes para problemas de regresión son *Flux.Loss.mae* o *Flux.Loss.msle*.

- Para un problema de clasificación la función de error es distinta. Como se muestra en clase de teoría, se utiliza la función *cross-entropy*, para el caso de haber más de 2

clases (una neurona de salida por clase), o la función *cross-entropy* binaria, para el caso de haber 2 clases (sólo una neurona de salida). De esta forma, la función de *loss* para cada caso podría ser una de las siguientes:

```
loss(x, y) = Losses.crossentropy(ann(x), y)
```

```
loss(x, y) = Losses.binarycrossentropy(ann(x), y)
```

Recordad que es importante que tanto en x (entradas) como en y (salidas deseadas) cada patrón debe estar en una columna, al contrario de lo que es habitual. Por este motivo, como se mostrará más adelante, se traspondrán las matrices de entradas y salidas deseadas.

Estas funciones usan la variable *ann*, que se usa como función como ha sido descrito previamente. Por lo tanto, es necesario que esté definida dentro del entorno en el que se define la función.

Una vez definida la función de *loss*, es necesario indicar el optimizador que se usará durante el entrenamiento. El optimizador no es más que la implementación concreta de una variante del algoritmo *backpropagation* vista en clase de teoría. Flux tiene una gran cantidad de implementaciones, desde la clásica basada en descenso de gradiente (*Descent*) o añadiendo también el momento (*Momentum*) hasta otras más avanzadas: *ADAM*, *RADAM*, *AdaMax*, *ADAGrad*, *ADADelta*, *AMSGrad*, *RMSProp*, etc. Posiblemente el optimizador más utilizado es *ADAM*, al que hay que indicar la tasa de aprendizaje, que suele ser un valor pequeño. Como siempre, tenéis más información de estos optimizadores en la documentación de la librería.

De esta forma, entrenar un ciclo la RNA anterior se puede hacer con la función *train!* De la siguiente manera:

```
Flux.train!(loss, params(ann), [(inputs', targets')], ADAM(0.01));
```

- En Julia, cuando una función se define añadiendo "!" (*bang*) como último símbolo, se entiende que modifica el contenido de alguno de sus argumentos, que, por tanto, se ha pasado por referencia.

Como se puede ver, la función *train!* recibe cuatro parámetros:

- Función de *loss* previamente definida.
- Pesos y bias de la RNA. Esto se puede conseguir, como se ve en el ejemplo, con la

función *params*.

- Conjunto de patrones, entradas (*inputs*) y salidas deseadas (*targets*). Como se puede ver, en el ejemplo se le está pasando un array con un solo elemento. Este elemento es una tupla con dos elementos: las matrices de entradas y salidas deseadas. Esta forma de pasar los patrones, que puede parecer rebuscada, tiene su motivación, puesto que cuando el conjunto de patrones es muy grande el calcular las modificaciones a los pesos con todos los patrones puede ser muy costoso. Por ese motivo, los patrones se suelen dividir en subconjuntos (*batches*) para que en cada actualización se realiza con solamente uno de esos subconjuntos. De realizar esto, el array pasado como parámetro en lugar de tener una tupla tendría varias, una por *batch*. Sin embargo, en las prácticas a realizar en esta asignatura no se realizará esto, y se pasan todos los patrones de forma conjunta.

    - **Importante**: Como se indicó anteriormente, estas matrices de entradas y salidas deseadas tienen que tener cada patrón en cada columna, al contrario de lo que es habitual. Por ese motivo, las matrices de entradas y salidas deseadas que se pasan como parámetros están traspuestas, es decir, en lugar de pasar *inputs* y *targets* se pasa *inputs'* y *targets'*. Si las matrices *inputs* y/o *targets* ya tuviesen un patrón en cada columna, no haría falta trasponer la matriz correspondiente.

    - **Importante**: Las matrices que se indiquen en este parámetro serán utilizadas para entrenar la RNA. Por tanto, tienen que ser totalmente disjuntas a las que se utilicen para realizar tests.

- Optimizador. En este ejemplo, es un ADAM con una tasa de aprendizaje de 0.01.

De esta forma se entrena solamente un ciclo. Por lo tanto, para entrenar una RNA es necesario crear un bucle que ejecute esta función mientras que no se cumpla algún criterio de parada. Algunos criterios más comunes pueden ser:

- El error o *loss* de entrenamiento es lo suficientemente bueno.

    - ¿Podría hacerse un criterio de parada similar pero con el error de test? ¿Por qué?

- El número de ciclos de entrenamiento ha alcanzado un máximo prefijado.

- La modificación del error de entrenamiento es inferior a un valor prefijado.
- etc.

De esta forma, es posible entrenar una RNA de tal forma que se minimice el error o *loss* en el conjunto de entrenamiento. Sin embargo, el entrenamiento de RR.NN.AA. no es determinístico, sino que tiene una componente basada en el azar, que es la inicialización aleatoria de los pesos. Cuando esto sucede, para minimizar la componente aleatoria, la RNA se crea y entrena varias veces y se promedian los resultados. Si se quiere entrenar varias RR.NN.AA., será necesario anidar dos bucles, donde el exterior iterará por las distintas redes, y el interior ejecutará los distintos ciclos de entrenamiento de cada red.

- **Importante**: ¿En qué parte del código (fuera de ambos bucles, dentro del primer bucle o dentro del segundo bucle) será necesario poner la llamada a la función *Chain* para crear cada red? ¿Por qué no se debe poner en otra parte?

Mediante este proceso, los pesos y bias, a partir de unos valores inicialmente aleatorios, irán tomando distintos valores hasta que se cumpla uno de los criterios de parada. En este aspecto, es necesario tener en cuenta que los pesos correspondientes a las conexiones que conectan entradas de valor absoluto muy alto tomarán valores de valor absoluto bajo; por el contrario, los pesos de aquellas conexiones que conecten entradas de absoluto bajo tendrán un valor absoluto alto. De esta forma, la RNA es capaz de combinar entradas que estén en intervalos muy distintos, pasándolos a valores de escala similar. Es decir, la RNA es capaz de "aprender" la relación entre las distintas escalas en las que se mueven las entradas.

Sin embargo, a pesar de que la RNA puede realizar este proceso, el entrenamiento de la RNA será mucho más rápido si las entradas proporcionadas están en la misma escala, es decir, si se le ahorra a la RNA el tener que aprender la relación entre las escalas en las que se mueve cada atributo. A este proceso de convertir las entradas para que todas estén en el mismo intervalo se denomina **normalización**, y constituye uno de los tipos de preprocesado más común e importante. Mediante este tipo de preprocesado se consigue que una RNA de arquitectura más sencilla pueda resolver problemas más complejos que sin él, puesto que no necesita emplear parte de la misma para aprender la relación entre las escalas de los atributos de entrada.

- ¿Sería necesario realizar este preprocesado cuando las entradas son los valores de intensidad de cada pixel en una imagen en blanco y negro? ¿Por qué?

Existen muchos otros tipos de preprocesado, como puede ser limpieza de ruido, análisis PCA, etc. Algunos de ellos se verán en una asignatura posterior. Con respecto a la normalización, en clases de teoría se explican más, pero en la práctica se suele usar uno de estos dos tipos:

- Normalización entre máximo y mínimo. Para cada atributo, se toma el menor (*min*) y el mayor (*max*) valor que se tiene, y se cambian todos los valores *v* para pasarlos al nuevo intervalo *[nuevomin, nuevomax]* de la siguiente manera:

$$v' = \frac{v - \min}{\max - \min}(\text{nuevomax} - \text{nuevomin}) + \text{nuevomin}$$

  Generalmente se suele pasar a un intervalo entre [0, 1], por lo que la ecuación queda más sencilla:

$$v' = \frac{v - \min}{\max - \min}$$

  Este tipo de normalización es adecuado cuando se tiene la certeza de que los datos están acotados (tanto superior como inferiormente), es decir, están dentro de un intervalo. Lo que está haciendo, en la práctica, es cambiar el intervalo de los datos por el intervalo [0, 1] y hacer una correspondencia para cada dato a su nuevo valor dentro del nuevo intervalo. Sin embargo, si se sospecha que uno de estos datos podría salirse del intervalo y tomar un valor excesivamente alto o bajo, esta transformación puede ser muy nociva. En este caso, a este valor *atípico* se le asignaría como nuevo valor el 1 si es excesivamente alto, y el resto de valores oscilarían cercanos al 0 con poca diferencia entre ellos, o el 0 si es excesivamente bajo, y el resto de valores oscilarían cercanos al 1 con poca diferencia entre ellos. Si se sospecha que puede haber casos como este, se suele optar por la siguiente forma de normalización.

  - En este tipo de normalización, si *min*=*max*, se puede realizar otro preprocesado distinto en este atributo, ¿en qué consistiría?

- Normalización de media 0. Como se ha dicho, esta forma de normalización es más robusta ante valores atípicos. Para cada atributo, se toma la media y desviación típica de todos los valores que toma, y se hace una transformación sencilla:

$$v' = \frac{v - \mu}{\sigma}$$

De esta manera, los valores de cada atributo pasarán a tener una media 0 y una desviación típica de 1. Algunos valores saldrán de este intervalo, pero esto no supone ningún problema para la RNA, que sencillamente acepta como entradas valores reales.

Es importante tener en cuenta que, se aplique la forma de normalizar que se aplique, esta se realice de forma independiente para cada atributo. Es decir, si se tiene una base de datos con N patrones y L atributos, habría que realizar L normalizaciones distintas.

- Sabiendo que los modelos de Aprendizaje Automático en general suponen que los patrones se distribuyen en filas, pero en el mundo de las RR.NN.AA. se distribuyen en columnas, ¿en qué casos habría que normalizar cada fila por separado y en qué casos habría que normalizar cada columna por separado?

En general, el tener el conocimiento sobre la naturaleza de cada atributo suele ser de gran utilidad para conseguir modelos que ofrezcan mejores resultados. Cuanta más información sobre los datos se "introduzca" dentro del modelo, mejor funcionará este. Un buen ejemplo de introducir información sobre los datos es la normalización de los mismos. Esto podría llevarse al extremo y escoger una forma de normalización distinta, la que se crea que es más adecuada, para cada uno de los atributos. Esto puede llevar a tomar la decisión, por ejemplo, de normalizar el atributo "temperatura" entre máximo y mínimo, y el atributo "distancia" mediante media 0. De todas formas, en la mayoría de los casos se suele optar por una de las dos formas de normalización y aplicarla a todos los atributos de entrada de la RNA.

También es importante tener en cuenta que este proceso ocurre en las salidas de la RNA. Es decir, si las salidas están en intervalos distintos, la RNA tiene que aprender esto también, con lo que se puede "ayudar" a la RNA mediante la normalización de los datos de salida. Este es un proceso que se realiza en problemas de regresión, pero no en problemas de clasificación.

- ¿Por qué no se realiza en problemas de clasificación?

Importante: De esta forma, las entradas (y salidas deseadas) que se aplican a la RNA ya no son los datos originales, sino que han sido transformados. Por lo tanto, una vez entrenada, la RNA ya no está preparada para que se le pasen los datos originales, sino que si se desea

aplicar los datos, estos tendrán que ser transformados de la misma manera. Por este motivo, se aplique la forma de normalizar que se aplique, es necesario guardar los parámetros usados en la normalización **para cada atributo** (máximo y mínimo o media y desviación típica). De igual manera, si se ha normalizado la salida deseada (problemas de regresión), la RNA habrá aprendido a emitir una salida normalizada, por lo que habrá que desnormalizarla, lo cual conlleva nuevamente que habrá que guardar los parámetros de normalización de las salidas deseadas. En resumen, para aplicarle nuevos datos a la RNA, el proceso será el siguiente:

1. Normalizar los datos según los parámetros de normalización que ya se tienen y que se utilizaron en el conjunto de entrenamiento.

2. Aplicar los datos normalizados a la RNA.

3. En caso de problemas de regresión, desnormalizar las salidas de la RNA.

Al estar trabajando con problemas de clasificación, el valor de *loss* no suele resultar de fácil interpretación. Para problemas de clasificación existen distintas métricas que serán el objetivo de prácticas posteriores. Por ahora, para valorar la bondad de salida de la RNA, únicamente utilizaremos la precisión en la clasificación, definida como el ratio de patrones bien clasificados (número de patrones bien clasificados dividido entre el número de patrones en total). Se han distinguido dos casos distintos:

- Cuando se tienen dos clases, la RNA tiene una única neurona de salida. Como se ha descrito, se suele usar como función de transferencia una función sigmoidal, que devuelve un valor entre 0 y 1. Simplemente pasando un umbral (normalmente en 0.5), se puede clasificar el patrón en “positivo” o “negativo” según si la salida es mayor o menor que el umbral respectivamente.

- Cuando se tienen más de dos clases, como resultado de aplicar la función *softmax* se tendrán distintos valores de seguridad, certidumbre o probabilidad de pertenencia a cada clase. Por lo tanto, un patrón se clasifica como la clase con probabilidad más alta.

Importante: Este proceso de normalización, a pesar de que aquí se ve aplicado a RR.NN.AA., es algo común en el resto de técnicas de Aprendizaje Automático. Por tanto, el código a desarrollar relativo a normalización será utilizado también en el resto de modelos.

Para esta práctica, se pide:

- Desarrollar una función para crear RR.NN.AA. para resolver problemas de clasificación. Esta función debe recibir la topología (número de capas y neuronas y funciones de activación en cada capa), conjunto de entrenamiento, y condiciones de parada.

    - Tened en cuenta que la función de transferencia de la capa de salida no viene dada por el usuario sino por el propio problema (regresión/clasificación). De igual manera, el número de neuronas de las capas de entrada y salida vienen dadas por el problema a resolver.

  Una forma sencilla de crear esta RNA es que reciba la topología como un parámetro llamado *topology* de tipo *Array{Int64,1}*, que contiene el número de neuronas de cada capa oculta (vacío para redes sin capas ocultas), y cree la RNA de la siguiente manera:

    - Crea una RNA vacía con la siguiente línea*:*

```
ann = Chain();
```

    - Crear una variable *numInputsLayer*, inicialmente igual al número de entradas de la RNA.

    - Si hay capas ocultas, es decir, si el vector *topology* no está vacío, iterar por este vector (el valor del bucle será igual al número de neuronas en cada capa) y en cada iteración crear una capa oculta donde el número de entradas será igual al valor de la variable *numInputsLayer* y el número de salidas igual al valor actual del bucle. Usar la misma función de transferencia σ en todas las capas ocultas. Después de esto, actualizar el valor de *numInputsLayer* al valor usado en esa iteración. Esto se puede hacer con las líneas:

```
for numOutputsLayer = topology
      ann = Chain(ann..., Dense(numInputsLayer, numOutputsLayer, σ) );
      numInputsLayer = numOutputsLayer;
end;
```

      Si estas líneas se escriben en el script sin estar dentro de un bucle o función, al compilar el código dará varios *warning* y el código no funcionará

correctamente. Esto se corrige automáticamente al usar este código dentro de una función. Para usarlo en el cuerpo principal, habría que escribir la línea *global ann, numInputsLayer;* al principio del bucle.

- Finalmente, añadir la capa final, con el número de neuronas y función de transferencia adecuada al número de clases como se ha descrito anteriormente, añadiendo la función *softmax* si hay más de dos clases de salida.

Una vez creada la RNA, esta función debería implementar un bucle en el que se entrene la RNA con el conjunto de entrenamiento pasado como parámetro hasta que se cumple uno de los criterios de parada pasados como parámetros.

- La función debería devolver la RNA entrenada.

- Desarrollar una función que dado una matriz de salidas deseadas (*targets*) y otra de salidas emitidas por la RNA (*outputs*), calcule la precisión en un problema de clasificación.
  - Desarrollar esta función de tal forma que funcione para los casos de tener 2 clases (una neurona de salida) como más de dos clases (una neurona de salida por clase).

- Integrar estas funciones con el código resultante de la práctica anterior para poder entrenar RR.NN.AA. con el problema seleccionado. Ejecutar varias veces para entrenar distintas redes con distintas arquitecturas. ¿Cuál es la que dará una mayor precisión en el conjunto de entrenamiento? Probar también distintos valores de tasa de aprendizaje.
  - Una vez entrenada una red (como resultado de la llamada a la función anterior), se puede pasar el conjunto de entrenamiento para calcular el valor de precisión.

- Desarrollar dos funciones que reciban los datos de la matriz de entradas y a partir de ellos calculen los parámetros de normalización, una función para cada una de las formas de normalizar que se han descrito aquí.
  - Para calcular valores máximos y mínimos se pueden usar las funciones *maximum* y *minimum*. Al pasarle una matriz, calculan los valores máximos y

mínimos de la matriz respectivamente. Sin embargo, se puede usar un parámetro extra utilizando el *keyword* “dims” para indicar la dimensión en la cual se aplicará la función correspondiente.

- ¿Cuál será el resultado de ejecutar *maximum([1 2; -2 -1], dims=1)* y *maximum([1 2; -2 -1], dims=2)*?

- Para calcular medias y desviaciones típicas, se puede usar las funciones *mean* y *std*. Al igual que las funciones anteriores, como parámetro una matriz, y un parámetro extra opcional con el *keyword* “dims” para indicar la dimensión en la cual se aplicará la función correspondiente.

  - Para poder usar estas dos últimas funciones, cargar el módulo *Statistics* (mediante *using Statistics*).

Para poder utilizar estas funciones con otros modelos de aprendizaje automático en prácticas posteriores, estas deberían recibir un parámetro adicional, de tipo booleano, que indique si los patrones están dispuestos en filas o en columnas.

Este ejercicio está pensado para desarrollar habilidad en programación vectorial. A continuación se detallan los pasos que se podrían dar para escribir el código para calcular la precisión con más de 2 clases, y los patrones dispuestos en filas (si están dispuestos en columnas, habrá que cambiar el valor del *keyword* “dims”):

- En primer lugar, es necesario hallar el valor máximo de salida para cada patrón. Esto se puede hacer con la siguiente línea:

```
vmax = maximum(outputs, dims=2)
```

- Una vez se tiene, se puede crear una matriz booleana de la misma dimensionalidad que la matriz de salidas deseadas y que la matriz de salidas de la RNA, donde cada valor indique la pertenencia a la clase correspondiente de ese patrón. Esto se puede hacer con:

```
outputs = (outputs .== vmax)
```

- Una vez la variable outputs tiene la forma deseada (matriz de valores booleanos), se compara con la matriz targets de la siguiente manera:

```
classComparison = targets .== outputs
```

- En esta matriz, para cada patrón, cuando la clase coincide, todos los elementos de esa fila serán *true*. En cambio, cuando la clase no coincida, más de un elemento de esa fila serán *false*. Por lo tanto, una forma de saber para un patrón si está bien clasificado es mirar si en su fila todos los elementos son *true*. Esto se puede comprobar con la función *all*, que recibe un array y devuelve *true* si todos los elementos son *true*, pero también acepta el *keyword* "dims", con el que aplica esta misma función a través de la dimensión indicada. Para hacerlo en las filas, habría que hacer así:

```
correctClassifications = all(classComparison, dims=2)
```

- Finalmente, sólo resta hacer la media de esta matriz. Recordad que se puede operar con valores booleanos igual que con valores reales, en este caso será tratados como 0 o 1.

```
accuracy = mean(correctClassifications)
```

- Estos últimos pasos podrían haberse dado mirando, en lugar de las coincidencias entre las dos matrices, los patrones en los que no hay coincidencias. Para eso se puede usar la función *any*, que recibe una matriz de valores booleanos y devuelve *true* si hay algún valor igual a *true*, y acepta también los el *keyword* "dims". Se calcularía de esta forma la tasa de error en lugar de la precisión, pero esta se puede calcular a partir de la primera simplemente restándosela a la unidad. El código sería de esta manera:

```
classComparison = targets .!= outputs

incorrectClassifications = any(classComparison, dims=2)

accuracy = 1 - mean(incorrectClassifications)
```

- Desarrollar dos funciones que reciban una matriz de datos y los valores de normalización y realicen la normalización de los datos de la matriz. Dejamos a vuestra elección si queréis optar por una de estas posibilidades:
    - Modifique la propia matriz pasada como parámetro (en este caso el nombre de la función debería terminar en "!").
    - No modifique la matriz, y en su lugar cree una matriz nueva con los datos

normalizados. Para realizar esto, puede ser útil utilizar la función *copy*.

- Si se realiza la primera opción, es inmediato hacer la segunda, ¿cómo se haría?
- En el caso de normalizar por máximo y mínimo, ¿qué se podría hacer en caso de que para un atributo el mínimo fuese igual al máximo?

Al igual que antes, estas funciones deberían estar preparadas para normalizar datos dispuestos en filas o en columnas.

- Para el caso de trabajar con RR.NN.AA., ¿habría que normalizar las filas o las columnas?
- ¿Por qué no normalizamos la matriz con las salidas deseadas?

- A no ser que se indique lo contrario (por ejemplo, en el caso de entrenar una RNA), desarrollar estas funciones sin que haya ningún bucle en el interior.
- Integrar estas funciones en el código que se está desarrollando en las prácticas.
- Repetir los experimentos realizados anteriormente, con los datos sin normalizar, y comparar los resultados con los datos normalizados.
  - ¿Existen diferencias importantes al utilizar datos normalizados?
  - ¿Cuál es la mejor topología?

Aprende Julia:

Como cualquier otro lenguaje, Julia permite la creación de funciones. Existen varias formas de crear funciones, las más comunes son estas dos:

- Si la operación a realizar es sencilla, la función se puede declarar en una línea sin necesidad de la palabra reservada *function*. Este es el caso de operaciones que se pueden realizar en una línea de código o en unas pocas. En este último caso, se pueden usar los paréntesis para encerrar las acciones a realizar, y ";" para separarlas. El valor a devolver por la función será lo último que se evalúe, aunque también se puede utilizar la palabra reservada *return*. A continuación, un par de

ejemplos de esta forma de declarar funciones:

```
suma(x::Float64, y::Float64) = x+x;
mse(outputs::Array{Float64,1}, targets:: Array{Float64,1}) = mean((targets.-outputs).^2);
mediaMayorQue0(valores::Array{Float64,1}) = mean(valores[valores.>0]);
mediaMayorQue0(valores::Array{Float64,1}) = ( positivos=valores.>0; mean(valores[positivos]); )
```

- En muchos otros casos, una función puede realizar muchas operaciones complejas que no sea práctico escribir en una sola línea. En este caso, se puede declarar la función con la palabra reservada *function*, y devolver el resultado con la palabra reservada *return*. Como es habitual en los lenguajes de programación, al evaluar *return*, se sale inmediatamente de la función. Si no se usa *return*, el resultado que devuelva función será el último evaluado. A continuación se muestra el ejemplo anterior:

```
function mediaMayorQue0(valores::Array{Float64,1})
   positivos=valores.>0;
   return mean(valores[positivos]);
end;
```

En el paso de parámetros no es obligatorio (aunque sí recomendable) indicar el tipo de los parámetros. Si no se hace, se supone que son de tipo *Any*. Sin embargo, una práctica habitual en Julia consiste en sobrecargar las funciones, es decir, definir funciones con el mismo nombre pero distintos parámetros o de distinto tipo. Al llamar a una función, se ejecutará la función correcta, si alguna de las que están definidas encaja con los parámetros pasados. Esto permite definir distintos comportamientos.

Una característica interesante de las funciones de Julia es que permiten devolver más de un valor. Esto lo realizan gracias al tipo *Tuple{...}*, que designa tuplas donde cada elemento tiene un tipo determinado, por ejemplo *Tuple{Float64,Float64}*, *Tuple{Array{Float64,2}, Int64}* o *Tuple{Float64, Tuple{Int64, Int4}}*. En general, si se desea devolver más de un elemento, se devuelve una tupla con los elementos que se quiere devolver, por ejemplo:

```
function mediaMayorQue0(valores::Array{Float64,1})
   positivos=valores.>0;
   return (positivos, mean(valores[positivos]))
end;
```

```
(positivos, media) = mediaMayorQue0([1.2 , -1.3 , 5.5 , -3.8 , -2.1])
```

Cuando se crean tuplas en las que todos los elementos tienen el mismo tipo, una forma sencilla de crear el tipo es mediante *NTuple*, indicando el número de elementos y el tipo. Por ejemplo, la siguiente línea se evalúa a *true*:

```
Tuple{Float64,Float64}==NTuple{2,Float64}
```

Como se ha descrito anteriormente, los nombres de aquellas funciones que modifican valores de alguno de sus parámetros se suelen terminar en “!”.