Entrenamiento de Perceptrones Multicapa

Aprendizaje Automático

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) );
```

2 capa oculta

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

2 capas ocultas

- ¿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 el modelo (es decir, la RNA a evaluar), como segundo las entradas, y como tercer argumento las salidas deseadas correspondientes a estas entradas (en ambos casos con un patrón en cada columna).

- 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(model, x, y) = Losses.mse(model(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(model, x, y) = Losses.crossentropy(model(x), y)
```

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

  Cada una de estas funciones se podría declarar en las distintas ramas de una sentencia if. Sin embargo, Julia presenta problemas al realizar este tipo de declaraciones de funciones. Por

este motivo, estas dos declaraciones se pueden unir en una sola, con la siguiente línea de código:

```
loss(model, x,y) = (size(y,1) == 1) ? Losses.binarycrossentropy(model(x),y) : Losses.crossentropy(model(x),y);
```

- Como se puede ver, en primer lugar se comprueba el número de filas de la matriz de salidas deseadas (y), ¿por qué se comprueba el número de filas y no el de columnas?

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.

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. Por ejemplo, para crear un optimizador, esto se haría con una línea similar a la siguiente, donde *learningRate* es una variable definida previamente:

```
opt_state = Flux.setup(Adam(learningRate), ann)
```

En este ejemplo, el optimizador es un ADAM con una tasa de aprendizaje igual a *learningRate*, que suele tomar valores entre 0.001 y 0.1. Un valor de 0.01 es habitual, aunque habría que probar distintos valores hasta encontrar uno que ofrezca buenos resultados en el problema en cuestión.

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

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

- 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. Compara la salida emitida con la deseada
- Modelo a entrenar.
- 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, creado anteriormente.

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.

Una cuestión a tener en cuenta es que, si bien la llamada a la función *train!* debería de estar en el interior de un bucle, la creación del optimizador debería de estar fuera de este, antes del mismo. Dependiendo del optimizador que se utilice, algunos tienen parámetros que son modificados cada vez que se llama a la función *train!*, con lo que, si se hace en el interior del bucle, esta modificación de parámetros se pierde al estar instanciando una nueva versión del optimizador de cada vez.

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.

Una cuestión importante a la hora de entrenar consiste en el tipo de datos que usan las redes de neuronas. Estos pueden ser *Float32* o *Float64*, e incluso se pueden mezclar. Por defecto, los pesos de las capas utilizan el tipo *Float32*, y, si se usan entradas de tipo *Float64*, el entrenamiento y la ejecución serán más lentas, puesto que para operar entre estos dos tipos será necesario ir haciendo conversión de tipos. El tipo más utilizado en redes de neuronas es *Float32*, puesto que provee de la precisión suficiente para la inmensa mayoría de operaciones a realizar, y ocupa la mitad que *Float64*, por lo que es mucho más rápido reservar memoria, además de la cantidad de memoria que se ahorra. Otro factor a tener en cuenta es que el tipo de datos utilizado en las tarjetas gráficas. En esta práctica, se podrán utilizar ambos tipos, pero se recomienda usar *Float32* al trabajar con redes de neuronas artificiales.

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 valor 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, para que el proceso de entrenamiento resulte más eficaz y eficiente, lo mejor es proveer todas las entradas en la misma escala, mediante un proceso de normalización previo. De esta manera, se evita que la red de neuronas tenga que aprender también la relación entre las escalas de cada atributo.

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 bondad es la precision en la clasificacion de un problema de Aprendizaje Automatico
el ratio de patrones bien clasificados

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 llamada *oneHotEncoding*, que contenga el código desarrollado en la práctica anterior relativo a la codificación de una entrada o salida categórica. Es decir, que reciba un vector de valores y lo codifique como se explicó en esa práctica. Esta función recibirá dos parámetros, llamados *feature* y *classes*, ambos de tipo *AbstractArray{<:Any,1}*, es decir, son vectores que contienen cualquier tipo de valor. El primero tiene los valores de ese atributo o salida deseada para cada patrón, y el segundo tiene los valores de las categorías. Esta función debería realizar las siguientes tareas:
  - Si el número de valores en el vector *classes* es igual a 2, se compara el vector del atributo con una de las dos clases realizando un *broadcast* del operador == para generar, de esta manera, un vector de valores booleanos. Posteriormente, se transforma ese vector en una matriz bidimensional de una columna y se devuelve. Para hacer esto último, consultad la función *reshape*.
  - Si el número de valores en el vector *classes* es mayor que 2, en primer lugar se crea una matriz de valores booleanos (del tipo *Array{Bool,2}* o *BitArray{2}*) con tantas filas como patrones y tantas columnas como categorías (una columna por categoría). Posteriormente, se itera sobre cada columna/categoría, y se asignan los valores de

esa columna como el resultado de comparar el vector *feature* con la categoría correspondiente realizando un *broadcast* igual que en el punto anterior.

En ambos casos, la matriz resultante debería tener tantas filas como patrones. Es decir, que estos se sitúen en las filas, no en las columnas.

Al igual que en la práctica anterior, a pesar de que no es necesario, para esta función se permite realizar un único bucle, que recorra las clases, únicamente para el caso de tener más de dos clases.

- Sobrecargar esta función llamada *oneHotEncoding* de estas dos maneras:
    - Una que reciba un único parámetro llamado *feature*, de tipo *AbstractArray{<:Any,1}*, y que tome las categorías y realice una llamada a la función anterior. Para extraer las categorías se puede usar la función *unique*. Desarrollar esta función sin declararla explícitamente mediante la palabra *function*.
    - Otra función que reciba un único parámetro llamado *feature*, de tipo *AbstractArray{Bool,1}*, y por tanto será un vector que contiene para cada patrón solamente dos posibilidades. Al ser dos categorías, la matriz de salida deberá tener una única columna, que tenga los mismos elementos. Una forma sencilla de hacer esta función es utilizando la función *reshape*.

  Para el desarrollo de estas funciones no se permite realizar ningún bucle.

- Desarrollar una serie de funciones que permitan normalizar los datos utilizando el código desarrollado en la práctica anterior. Para esto, desarrollar las siguientes funciones:
    - Dos funciones llamadas *calculateMinMaxNormalizationParameters* y *calculateZeroMeanNormalizationParameters* que reciban un parámetro de tipo *AbstractArray{<:Real,2}* y devuelvan una tupla con dos valores, cada uno de ellos será una matriz con una fila, con los mínimos y máximos de cada columna (primera función) o medias y desviaciones típicas para cada columna (segunda función). Para el desarrollo de estas funciones no se permite utilizar ningún bucle.
    - Cuatro funciones para normalizar entre máximo y mínimo. La primera de ellas, llamada *normalizeMinMax!* recibe dos parámetros, una matriz de valores a normalizar (de tipo *AbstractArray{<:Real,2}*) y los parámetros de normalización (de tipo *NTuple{2, AbstractArray{<:Real,2}}*), ejecute el código de la práctica anterior

referido a normalizar entre máximo y mínimo y devuelva la misma matriz con los datos normalizados. Posteriormente, realizar otra función del mismo nombre (sobrecargada) pero con un único parámetro que sea la matriz de datos, y lo que hará será calcular los parámetros de normalización con la función desarrollada en el punto anterior y llamar a la función *normalizeMinMax!*. Estas dos funciones terminan en "!" porque se modifica la matriz de valores pasada como parámetro. Realizar otras dos funciones de nombres *normalizeMinMax* que realicen las mismas, pero que no modifiquen la matriz de datos, es decir, que creen una nueva, la modifiquen y se devuelva esta. Para ello, consultar la función *copy*. Estas dos funciones deberían realizar llamadas a las funciones anteriores. Ninguna de estas cuatro funciones debe utilizar bucles en su interior.

    - Realizar otras cuatro funciones análogas para el caso de realizar una normalización de media 0, cuyos nombres serán *normalizeZeroMean!* y *normalizeZeroMean*. Al igual que en las anteriores, ninguna de estas cuatro funciones debe utilizar bucles en su interior.

- Desarrollar una función llamada *classifyOutputs*, que reciba un parámetro llamado outputs de tipo AbstractArray{<:Real,1}, es decir, un vector, con las salidas de un modelo (no necesariamente una RNA) que emita únicamente una salida, para un problema de clasificación binaria. Este vector deberá contener un valor de salida real para cada patrón, y realizará la clasificación devolviendo el vector de valores binarios correspondientes. Esta función recibirá un parámetro opcional, llamado *threshold*, de tipo *Real*, con un valor por defecto de 0.5. Esta función se puede implementar realizando un *broadcast* del operador >= para generar de esta manera el vector de valores booleanos. Para la realización de esta función no se permite el uso de bucles.

- Desarrollar una función llamada *classifyOutputs* (es decir, sobrecargar la función anterior) que reciba en este caso una matriz en lugar de un vector, como un parámetro llamado outputs de tipo AbstractArray{<:Real,2} con las salidas de un modelo (no necesariamente una RNA) **con un patrón en cada fila** y lo convierta a una matriz de valores booleanos que cada fila sólo tenga un valor a true, que indica la clase a la que se clasifica ese patrón. Para realizar esto, mirar en primer lugar el número de columnas que tiene la matriz *outputs*, y en función de ello hacer lo siguiente:

    - Si tiene una columna, convertir esta matriz de una columna en un vector y hacer una llamada a la función anterior. Para realizar esto, es necesario que la función reciba

como parámetro opcional el valor del umbral (*threshold*), con un valor por defecto de 0.5. Para convertir la matriz en un vector, esto se puede hacer de una forma sencilla con *outputs[:]*. Como resultado de esta llamada, se obtiene un vector, que será necesario convertir de nuevo en una matriz con una columna mediante la función *reshape*.

- Si tiene más de una columna, se deberá crear una matriz de valores booleanos del mismo tamaño, y, para cada fila, poner a *true* la columna con un valor mayor. Esto se puede realizar sin necesidad de escribir ningún bucle.

Este ejercicio junto con el siguiente está pensado para desarrollar habilidad en programación vectorial. Por este motivo, al igual que en las funciones anteriores, no debe contener bucles en su interior. A continuación se detallan los pasos que se podrían dar para escribir el código para este segundo escenario, cuando se tiene más de una columna (suponiendo que cada patrón está en cada fila):

- En primer lugar, es necesario averiguar, para cada fila, en qué columna está el valor máximo de salida para cada patrón. Esto se puede hacer con la función findmax, que devuelve una tupla con dos valores: el máximo en cada fila o columna, y las coordenadas en la matriz en la que se encontró ese máximo, que es lo que realmente nos interesa. Con el *keyword dims* se puede indicar si se quiere buscar los máximos en las filas o en las columnas. La línea de código sería la siguiente:

```
(_,indicesMaxEachInstance) = findmax(outputs, dims=2);
```

- Una vez se tiene, se puede crear una matriz booleana de la misma dimensionalidad que la matriz de salidas, donde cada valor indique la pertenencia a la clase correspondiente de ese patrón. Esta matriz se inicializa con todos los valores a *false*, por lo que se puede crear fácilmente con la función *falses*. Esto se puede hacer con:

```
outputs = falses(size(outputs));
```

- Finalmente, se asignan a *true* en esta matriz los valores de los índices que contienen los mayores valores de cada fila, que están recogidos en la variable *indicesMaxEachInstance* creada anteriormente. Esto se puede realizar se compara con la matriz targets de la siguiente manera:

```
outputs[indicesMaxEachInstance] .= true;
```

- Desarrollar una función llamada *accuracy* dado una matriz de salidas deseadas (*targets*) y otra de salidas emitidas por un modelo (no necesariamente una RNA) (*outputs*), calcule la precisión en un problema de clasificación. Ambas matrices deberán tener un número de filas igual al número de patrones, es decir, cada patrón estará colocado en cada fila. 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).

  Para hacer esta función, desarrollar cuatro funciones con el mismo nombre:

  - Una primera en la que targets y outputs sean de tipo *AbstractArray{Bool,1}*, es decir, vectores de valores booleanos. La precisión será simplemente el valor promedio de la comparación de ambos vectores.

  - Otra función en la que *targets* y *outputs* sean de tipo *AbstractArray{Bool,2}*, es decir, matrices bidimensionales de valores booleanos. En este caso, será necesario examinar el número de columnas. Si sólo tienen una columna, se realizará una llamada a la función anterior tomando como vectores la primera columna de *targets* y *outputs*. Si el número de columnas es mayor que 2, será necesario comparar ambas matrices mirando en qué filas no coinciden los valores.

    - ¿Qué ocurre si el número de columnas es igual a 2?

  - Otra función en la targets sea de tipo *AbstractArray{Bool,1}* (un vector de valores booleanos) mientras que outputs sea de tipo *AbstractArray{<:Real,1}*, es decir, sean salidas reales que no se hayan interpretado todavía como valores de pertenencia a clase “positivo”/”negativo”. En este caso, la función podría aceptar un parámetro opcional *threshold* con un valor por defecto de 0.5, y lo que haría sería pasarle el umbral al vector *outputs*, y llamar a la función anterior *accuracy*.

  - Finalmente, otra función en la targets sea de tipo *AbstractArray{Bool,2}* (una matriz de valores booleanos) mientras que outputs sea de tipo *AbstractArray{<:Real,2}*, es decir, sean salidas reales que no se hayan interpretado todavía como valores de pertenencia a N clases (N: número de columnas). En este caso, es necesario nuevamente distinguir si se tienen 1 o más de 2 columnas. En el primero caso se hará una llamada a *accuracy* con los vectores correspondientes a la primera columna de *outputs* y *targets*. En el segundo caso, se hará una llamada a la función *classifyOutputs* para convertir *outputs* en una variable de tipo *AbstractArray{Bool,2}*,

y posteriormente hacer una llamada a *accuracy*.

Este ejercicio junto con el anterior está pensado para desarrollar habilidad en programación vectorial, por lo que no se permite el uso de bucles en ninguna de estas funciones. A continuación se detallan los pasos que se podrían dar para escribir el código para calcular la precisión con más de 2 clases, los patrones dispuestos en filas, y tanto *outputs* como *targets* ya son de tipo *AbstractArray{Bool,2}*:

- 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 nueva 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 una función llamada *buildClassANN* para crear RR.NN.AA. para resolver problemas de clasificación. Esta función debe recibir la topología (número de capas ocultas y neuronas en cada una, y opcionalmente funciones de activación en cada capa oculta), el número de neuronas de entrada y el número de neuronas de salida.

    - 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 *AbstractArray{<:Int,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. Si no se ha indicado la función de transferencia de cada capa oculta, 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.

  Como se indica, para el desarrollo de esta función se puede emplear un bucle.

- Crear una función llamada *trainClassANN* que cree una RNA para realizar clasificación (mediante una llamada a la función anterior) y la entrene. Para ello, 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. Esta función debería aceptar los siguientes parámetros:

  - *topology*, de tipo *AbstractArray{<:Int,1}* con la topología (capas ocultas) de la RNA.

  - *dataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}* con la matriz de entradas y salidas deseadas. A partir de estas matrices se puede obtener el número de neuronas de entrada y salida, necesarias para realizar la llamada a la función anterior, mediante la función *size*.

    Como se puede ver, la matriz de entradas puede contener datos de tipo *Float64* o *Float32*. Como se ha explicado anteriormente, por defecto las redes de neuronas implementan sus parámetros (pesos y bias) en *Float32*, por lo que es mejor utilizar este tipo, además de por los otros motivos ya explicados (eficiencia, uso de GPU, menor uso de memoria). Para esto, se podría utilizar como tipo del dataset *Tuple{AbstractArray{Float32,2}, AbstractArray{Bool,2}}*. Sin embargo, se deja al usuario la posibilidad de usar tanto *Float32* como *Float64*, y será su responsabilidad usar un tipo de datos coherente en todo el uso de las redes de neuronas, que, como se ha dicho, habitualmente es *Float32*.

  - Parámetros opcionales que controlen otros aspectos del algoritmo como el criterio de parada del algoritmo, con valores por defecto, por ejemplo: *maxEpochs* (de tipo *Int*, con un valor por defecto 1000), *minLoss* (de tipo *Real*, con valor por defecto 0), o

learningRate (de tipo *Real*, con valor por defecto 0.01).

Es importante destacar que el conjunto de patrones aquí utilizado tendrá cada patrón en una fila, mientras que la librería Flux (en general, la mayoría de las librerías de AA.GG. lo hacen) esperan matrices en las que cada patrón esté en una columna. Esto no supone un problema, únicamente es necesario trasponer las matrices en ciertas ocasiones, como cuando se proveen a la función para entrenar un ciclo la RNA, al calcular el valor de *loss*, o al tomar las matrices de salidas que se obtienen al ejecutar la RNA.

Recordad que una llamada a la función *train!* entrena la RNA un solo ciclo. Por tanto, es necesario, en primer lugar, crear la RNA mediante una llamada a la función anterior, y posteriormente ejecutar un bucle en el que en cada iteración se llame una única vez a la función *train!* Es decir, si se quiere entrenar durante n ciclos, este bucle tendrá que realizar n iteraciones.

Esta función debería devolver una tupla con dos elementos, donde el primero debe ser la RNA entrenada y el segundo un vector con los valores de *loss* en cada ciclo de entrenamiento.

Para la realización de esta función se permite realizar un único bucle, que itere sobre cada ciclo de entrenamiento.

Esta función será modificada en el siguiente ejercicio.

- En ocasiones, cuando el problema de clasificación sea de dos clases, en lugar de tener las salidas deseadas como una matriz de una sola columna, se tendrá como un vector. Para contemplar estos casos, se pide crear una función del mismo nombre que la anterior y que acepte los mismos argumentos con la excepción de que el segundo argumento, *dataset*, sea de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}}*. Esta función debería únicamente convertir el vector de salidas deseadas en una matriz con una columna y llamar a la función anterior. Para hacer esta conversión, consultar la función *reshape*. Por este motivo, no se permite la realización de bucles en el desarrollo de esta función. Además, igual que la anterior, será modificada en la siguiente práctica.

- Integrar estas funciones con el código resultante de la práctica anterior para poder entrenar RR.NN.AA.

Una vez que la función de entrenamiento devuelve una RNA entrenada, se puede simular

pasándole un conjunto de entradas (con los patrones en columnas, es decir, traspuesto) y devolverá las salidas para ese conjunto (igualmente, los patrones estarán en columnas, por lo que habrá que trasponer las salidas de la RNA). Estas salidas, junto con las salidas deseadas para ese conjunto de entradas, se pueden aplicar a la función *accuracy* para calcular la precisión en ese conjunto.

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.

- 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?

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

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::Float32, y::Float32) = x+x;
mse(outputs::Array{Float32,1}, targets:: Array{Float32,1}) = mean((targets.-outputs).^2);
mediaMayorQue0(valores::Array{Float32,1}) = mean(valores[valores.>0]);
mediaMayorQue0(valores::Array{Float32,1}) = ( 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{Float32,1})
   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. Cuando se realizar una llamada a la función, Julia comprueba que exista en memoria alguna definición con ese nombre en la cual los tipos de los argumentos pasados encajen con los definidos y, si existe, la ejecutará con esos parámetros.

En los ejemplos puestos más arriba, se han definido los parámetros como *Float32* o *Array{Float32,2}*, lo cual hace que las funciones estén definidas para esos tipos concretos, y por lo tanto, si se realizan llamadas con parámetros de tipo *Float64* o *Array{Float64,2}* no se encontrará una definición de función adecuada en memoria, y devolverá un error. Para solucionar esto, es conveniente utilizar las propiedades de subtipado como se ha visto en la práctica anterior para definir los tipos más genéricos con los que pueda trabajar la función. Por ejemplo, en lugar de usar Float32, se pueden usar los tipos *AbstractFloat*, *Real* o *Number* (este último tipo incluye los números complejos). En lugar de usar *Array{Float32,2}*, se podría usar *Array{<:AbstractFloat}*, *Array{<:Real}* o *Array{<:Number}*. Además, es necesario tener en cuenta que existen otros tipos que se comportan como arrays pero que no son arrays. Un ejemplo importante y que se va a usar en esta asignatura es al trasponer matrices, lo cual crea un objeto que no es de tipo *Array*, sino de tipo *LinearAlgebra.Adjoint* pero que se puede usar como si fuera un array porque tiene definidas las operaciones como tal. Ambos tipos, *Array* y *LinearAlgebra.Adjoint* son subtipos de *AbstractArray*, como se puede ver en el siguiente ejemplo:

```
julia> m = [1. 2.; 3. 4.];
```

```
julia> isa(m, Array)
true
julia> isa(m, AbstractArray)
true
julia> typeof(m')
LinearAlgebra.Adjoint{Float64,Array{Float64,2}}
julia> isa(m', Array)
false
julia> isa(m', AbstractArray)
true
```

Dado que ambos tipos, *Array* y *LinearAlgebra.Adjoint* son subtipos de *AbstractArray*, para permitir que un argumento sea indistintamente de un tipo o de otro, este será el tipo que se usará en los parámetros de las funciones. Por lo tanto, las funciones anteriores quedarán de la siguiente manera:

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

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

Para el caso de que un argumento sea un vector o matriz de valores booleanos, como es el caso de las salidas deseadas en un problema de clasificación, el tipo a utilizar será *AbstractArray{Bool,N}*, donde N es la dimensionalidad de la matriz. Como ya se ha mencionado en la práctica anterior, este tipo es un supertipo tanto de *Array{Bool,N}* como de *BitArray{N}*.

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{Float32,Float32}*, *Tuple{Array{Float32,2}, Int64}* o *Tuple{Float32, 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::AbstractArray{<:Real,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 "!".

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 oneHotEncoding(feature::AbstractArray{<:Any,1},
    classes::AbstractArray{<:Any,1})
function oneHotEncoding(feature::AbstractArray{<:Any,1})
function oneHotEncoding(feature::AbstractArray{Bool,1})

function calculateMinMaxNormalizationParameters(dataset::AbstractArray{<:Real,2})
function calculateZeroMeanNormalizationParameters(dataset::AbstractArray{<:Real,2})

function normalizeMinMax!(dataset::AbstractArray{<:Real,2},
    normalizationParameters::NTuple{2, AbstractArray{<:Real,2}})
function normalizeMinMax!(dataset::AbstractArray{<:Real,2})
function normalizeMinMax( dataset::AbstractArray{<:Real,2},
    normalizationParameters::NTuple{2, AbstractArray{<:Real,2}})
function normalizeMinMax( dataset::AbstractArray{<:Real,2})

function normalizeZeroMean!(dataset::AbstractArray{<:Real,2},
    normalizationParameters::NTuple{2, AbstractArray{<:Real,2}})
function normalizeZeroMean!(dataset::AbstractArray{<:Real,2})
function normalizeZeroMean( dataset::AbstractArray{<:Real,2},
    normalizationParameters::NTuple{2, AbstractArray{<:Real,2}})
function normalizeZeroMean( dataset::AbstractArray{<:Real,2})

function classifyOutputs(outputs::AbstractArray{<:Real,1}; threshold::Real=0.5)
function classifyOutputs(outputs::AbstractArray{<:Real,2}; threshold::Real=0.5)

function accuracy(outputs::AbstractArray{Bool,1}, targets::AbstractArray{Bool,1})
function accuracy(outputs::AbstractArray{Bool,2}, targets::AbstractArray{Bool,2})
```

```
function accuracy(outputs::AbstractArray{<:Real,1}, targets::AbstractArray{Bool,1};
    threshold::Real=0.5)
function accuracy(outputs::AbstractArray{<:Real,2}, targets::AbstractArray{Bool,2};
    threshold::Real=0.5)

function buildClassANN(numInputs::Int, topology::AbstractArray{<:Int,1}, numOutputs::Int;
    transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)))

function trainClassANN(topology::AbstractArray{<:Int,1},
    dataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}};
    transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)),
    maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.01)
function trainClassANN(topology::AbstractArray{<:Int,1},
    (inputs, targets)::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}};
    transferFunctions::AbstractArray{<:Function,1}=fill(σ, length(topology)),
    maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.01)
```