Redes en cascada

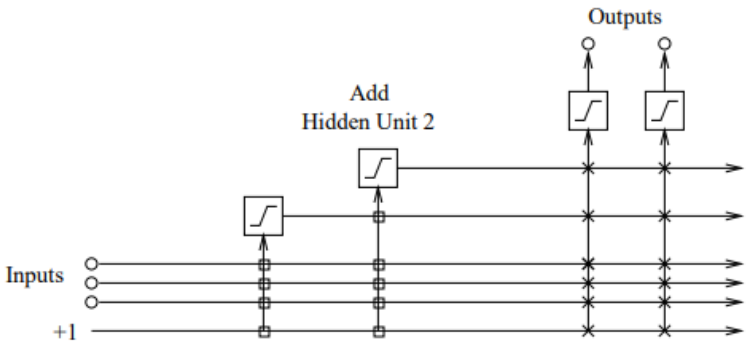
Modelos Avanzados de Aprendizaje Automático I

Las redes de neuronas en cascada son un tipo específico de arquitectura en el campo de las redes neuronales y el aprendizaje profundo. Este enfoque se utiliza para mejorar el rendimiento y la eficiencia de las redes neuronales, especialmente en tareas complejas como el reconocimiento de patrones, la clasificación de imágenes y el procesamiento del lenguaje natural.

Estas redes se basan en tener varias capas, originalmente con una única neurona cada capa. Esta neurona recibe como entrada las salidas de la capa anterior (originalmente un único valor, al ser capas de una única neurona), así como también las entradas que recibe la capa anterior, es decir:

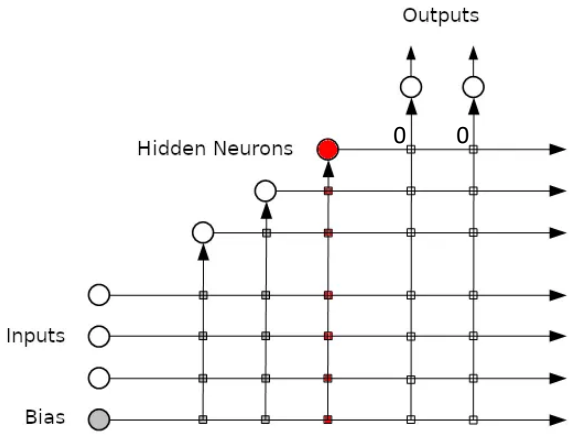
- La neurona de la primera capa oculta recibe como entradas los atributos del problema.
- La neurona de la segunda capa oculta recibe como entradas los atributos del problema y la salida de la neurona de la primera capa oculta.
- La neurona de la tercera capa oculta recibe como entradas los atributos del problema y las salidas de las neuronas de las dos primeras capas ocultas.
- etc.

De esta manera, cada neurona va procesando toda la información anterior y realizando una extracción de características cada vez de orden superior, utilizando las características de orden más bajo ya extraídas. Este enfoque permite a las neuronas trabajar de manera secuencial, mejorando progresivamente la precisión de las predicciones a través de varias capas. Un diagrama habitual para mostrar la estructura de estas redes es el siguiente:



A la hora de entrenar estas redes, este proceso se realiza de forma secuencial, añadiendo una neurona de cada vez y entrenando la red con la neurona añadida. En la clase de teoría se explican estas redes con más detalle, así como el algoritmo de entrenamiento inicialmente propuesto para estas redes (*Quickprop*). Sin embargo, para hacer la práctica más sencilla, en este ejercicio se utilizará el algoritmo *backpropagation* haciendo alguna modificación en su aplicación.

Cuando se añade una neurona, se considera que la RNA "previa" realiza una extracción de características y una posterior clasificación, y se desea partir de ese punto, es decir, de ese valor de *loss*. Por este motivo, en este ejercicio al añadir una nueva neurona a la red, los pesos de las conexiones de esta neurona a las neuronas de salida tendrán un valor de 0. Los pesos de las conexiones entrantes a esta neurona tendrán valores aleatorios; sin embargo, los pesos de las conexiones salientes de 0 hacen que esta neurona no tenga impacto en la red, y por ese motivo el *loss* de la nueva RNA será igual al valor del *loss* de la RNA anterior, sin la neurona. Visualmente, se puede ver en la siguiente figura, en la que se quiere crear una nueva RNA añadiendo una nueva neurona, en color rojo. Los pesos de las conexiones de esa neurona a las neuronas de salida (cuadrados situados en la línea horizontal de esa neurona) tienen un valor de 0, mientras que los pesos de las conexiones entrantes de esa neurona (cuadrados de color rojo situados en la línea vertical de esa neurona) tiene valores aleatorios.



De esta forma, manteniendo el valor de *loss* de la RNA previa, nos aseguramos de que añadir una neurona nunca va a tener un efecto nocivo sobre el funcionamiento de la RNA. Una vez añadida, es necesario entrenar la RNA con la neurona. En este ejercicio, esto se hará en 2 etapas:

1. Mantener la RNA previa sin modificar, y cambiar únicamente los pesos entrantes en la neurona nueva, y los pesos de la capa de salida (esto incluye tanto los pesos de la nueva neurona a las neuronas de salida como los pesos de las neuronas previas a las neuronas de salida). En la figura anterior, esto quiere decir que se modifican las 3 últimas columnas, porque se tienen 2 neuronas de salida. Es decir, se entrenan únicamente las dos últimas capas: la última capa oculta (con la neurona nueva) y la capa de salida. Esto se hace para concentrar el entrenamiento únicamente en la nueva neurona, considerando que las características que extrae la RNA son de interés, e intentando adaptar la nueva neurona a las características ya existentes.

Para hacer esto, se puede entrenar la RNA entera, "congelando" los pesos de las capas que no son las dos últimas. Al congelar estas capas, los pesos de las mismas no resultan modificados mediante la aplicación del algoritmo *backpropagation*.

2. Una vez esta nueva neurona ha sido entrenada y "adaptada" a la red, es momento de ver si es posible obtener mejoras en la misma al haber introducido una nueva neurona que añade características de nivel más alto. Para ello, se entrena la red entera, modificando tanto los pesos relativos a la nueva neurona como los previamente existentes.

Partiendo de una RNA inicial sin capas ocultas, este proceso se repite hasta que se alcanza un valor mínimo de *loss* o hasta que se ha llegado al número máximo de neuronas. Lógicamente, para la primera RNA, sin capas ocultas, solamente se realiza una de las dos etapas de entrenamiento, siendo indiferente cuál de las dos, puesto que en ambos se realizará la misma operación. Además, para el caso de haber añadido una neurona a una RNA sin capas ocultas, la primera etapa no se realiza, al no haber características previas extraídas.

En este ejercicio, se dan las siguientes funciones hechas:

- *indexOutputLayer*. Recibe una RNA (de tipo *Chain*) y devuelve el índice en el que se encuentra la capa de salida, que será igual a la longitud de la RNA si esta es de clasificación con 2 clases, y la longitud – 1, si es de clasificación con más de dos clases, porque en este último caso es necesario aplicar la función *softmax*.

En este ejercicio, se pide realizar las siguientes funciones:

- *newClassCascadeNetwork*. Crea una nueva RNA para resolver problemas de clasificación sin capas ocultas para resolver problemas de clasificación. Recibe como entradas:
    - *numInputs*, de tipo *Int*, con el número de entradas de la RNA.
    - *numOutputs*, de tipo *Int*, con el número de salidas de la RNA.

  Esta función deberá devolver una RNA, de tipo *Chain*, sin capas ocultas. Es, por tanto, una versión simplificada de la función desarrollada en la asignatura de FAA, en la que solamente hay que examinar el número de salidas para crear la RNA:
    - Si hay dos salidas, se creará una RNA de tipo *Chain*, con una sola capa *Dense*, con función de activación σ.
    - Si hay más de dos salidas, se creará una RNA de tipo *Chain*, con una capa *Dense*, con

función de activación *identity*, seguida por la función *softmax*.

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

- *addClassCascadeNeuron*. Esta función recibe una RNA para problemas de clasificación, y crea una RNA nueva añadiéndole una nueva capa oculta con una neurona, pero que ante las mismas entradas emite las mismas salidas. Recibe como parámetros obligatorios:
    - *previousANN*, de tipo *Chain*, con la RNA en la que se va a añadir la neurona.

  Esta función recibe como parámetro opcional:
    - *transferFunction*, de tipo *Function*, con la función de activación o transferencia.

  Para desarrollar esta función, en primer lugar es necesario referenciar la capa de salida de la RNA y las capas previas, usando para ello la función que se provee *indexOutputLayer*. Suponiendo que la variable *previousANN* contiene la RNA a modificar, esto se puede hacer de la siguiente manera:

```
outputLayer    = previousANN[   indexOutputLayer(previousANN)   ];
previousLayers = previousANN[1:(indexOutputLayer(previousANN)-1)];
```

  En la librería Flux, cada capa densa es una estructura de tipo *Dense* que contiene una matriz de pesos de nombre *weight*, un vector de *bias* de nombre *bias*, y la función de activación, de nombre σ. A partir de la matriz de pesos de la capa de salida se puede obtener el número de entradas y salidas de esa capa oculta, de la siguiente manera:

```
numInputsOutputLayer  = size(outputLayer.weight, 2);
numOutputsOutputLayer = size(outputLayer.weight, 1);
```

  Una vez se tienen, será necesario crear una RNA nueva, con las capas previas de la RNA anterior, la capa con la nueva neurona, y la capa de salida. Es necesario tener en cuenta que la capa con la nueva neurona debe recibir como entradas las salidas de la capa anterior, y emitir como salidas la salida de la neurona, y, además, estas mismas entradas. Esta capa se puede hacer mediante la función *SkipConnection* de Flux que recibe una capa y la forma de concatenar las entradas y las salidas. En este ejercicio, se creará la capa mediante:

```
SkipConnection(Dense(numInputsOutputLayer, 1, transferFunction), (mx, x) -> vcat(x, mx))
```

  De esta forma, se crea una capa densa que recibe el número de entradas correspondiente y emite estas entradas con una salida adicional, creando una nueva matriz en la que la salida

adicional ocupa la última fila. De esta forma, la creación de una nueva RNA será similar a la siguiente:

```
ann = Chain(
        previousLayers...,
        SkipConnection(Dense(numInputsOutputLayer, 1, transferFunction), (mx, x) -> vcat(x, mx)),
        <introducir aquí la(s) capa(s) de salida para un problema de clasificación>
);
```

Se deja este ejemplo no completo, a falta de introducir las posibles capas de salida para un problema de clasificación, que son distintas para los casos de tener 2 o más clases.

Una vez que se tiene creada la nueva RNA, es necesario copiar los pesos de la capa de salida de la anterior red a la nueva. Para ello, hay que tener en cuenta que la matriz de pesos tendrá una fila más, puesto que recibe una entrada más. Para que la nueva neurona no tenga impacto en la RNA, la matriz de pesos y el vector de *bias* de la capa de salida se puede asignar de la siguiente manera:

- La última columna de la matriz de pesos se pondrá a 0, puesto que corresponde con los pesos de las conexiones de la salida de la nueva neurona.
- Las restantes columnas de la matriz de pesos tendrán valores iguales a la matriz de pesos de la capa de salida de la RNA anterior.
- El vector de *bias* de la capa de salida será igual al vector de *bias* de la RNA anterior.

Una forma sencilla de comprobar que la nueva RNA emite las mismas salidas es aplicando una matriz de entradas y comprobando que las salidas que emiten ambas redes (la anterior y la nueva) son iguales.

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

- *trainClassANN!* Entrena una RNA, indicando si se quieren entrenar solamente las dos últimas capas (capa de salida y última capa oculta, con la nueva neurona), o toda la RNA. Recibe como parámetros obligatorios:
  - *ann*, de tipo *Chain*, con la RNA a entrenar.
  - *trainingDataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}* con el lote de datos a utilizar para entrenar.
  - *trainOnly2LastLayers*, de tipo *Bool*, con un valor booleano que dice si se desea entrenar solamente las dos últimas capas o no.

Esta función recibe como parámetros opcionales:

- *maxEpochs*, de tipo *Int*, con el número máximo de ciclos que se va a entrenar.
- *minLoss*, de tipo *Real*, con el valor mínimo de *loss* a obtener.
- *learningRate*, de tipo *Real*, con la tasa de aprendizaje.
- *minLossChange*, de tipo *Real*, con el valor mínimo de cambio en el *loss* para seguir entrenando, como un ratio de cambio.
- *lossChangeWindowSize*, de tipo *Int*, con el tamaño de ventana (número de ciclos) en el que se calcula este valor de cambio en *loss* para seguir entrenando.

Como se puede ver, esta función deberá implementar los siguientes criterios de parada:

- Número máximo de ciclos.
- Si el valor de *loss* obtenido es inferior o igual al mínimo indicado, se para el entrenamiento.
- Si han transcurrido más de *lossChangeWindowSize* ciclos y en los últimos *lossChangeWindowSize* ciclos el ratio de cambio en el *loss* es inferior o igual a *minLossChange*, entonces se para también el entrenamiento. El cálculo del ratio del cambio en el *loss* se realizará mediante la diferencia entre el valor máximo y mínimo de *loss*, dividida entre el valor mínimo de *loss*, estando calculados estos valores dentro de la ventana de los últimos *lossChangeWindowSize* ciclos. Si la variable *trainingLosses* contiene los valores de *loss* de los ciclos de entrenamiento, este cálculo se hará con:

```
lossWindow = trainingLosses[end-lossChangeWindowSize+1:end];
minLossValue, maxLossValue = extrema(lossWindow);
((maxLossValue-minLossValue)/minLossValue <= minLossChange)
```

Esta función recibirá las entradas y salidas deseadas como matrices con las dimensiones adecuadas para realizar el entrenamiento, es decir, las instancias estarán dispuestas en columnas y los atributos en filas.

Esta función deberá devolver un vector con los valores de *loss* de tipo *Float32* correspondientes a este entrenamiento. Este vector de valores de *loss* deberá contener el correspondiente al “ciclo 0”, es decir, el valor de *loss* antes de entrenar la RNA. La función de *loss* a utilizar es igual a la definida en la asignatura de FAA. Además, el optimizador se definirá

de la misma manera, con:

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

Con respecto a entrenar únicamente las dos últimas capas, para congelar parte de la red y que esta no se modifique en el entrenamiento, se dispone de la función *Freeze!*, a la que se puede indicar qué capas se quiere congelar. En este caso, dado que se desea congelar todas las capas menos las dos últimas, cuando se indique, esto se hará mediante:

```
Flux.freeze!(opt_state.layers[1:(indexOutputLayer(ann)-2)]);
```

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

- *trainClassCascadeANN*. Crea una red de neuronas en cascada, indicando, además de los parámetros de entrenamiento y el lote de datos de entrenamiento, el número máximo de neuronas a utilizar. Los parámetros obligatorios que recibe son los siguientes:
    - *maxNumNeurons*, de tipo *Int*, con el número máximo de neuronas que tendrá la red.
    - *trainingDataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}}*, con el lote de datos a usar para entrenar la RNA.

  Esta función recibe como parámetros opcionales los indicados a continuación. Excepto el primero, que contiene la función de activación a utilizar por las neuronas de la RNA, el resto describen cada proceso de entrenamiento llevado a cabo por esta función, que, por lo tanto, serán pasados como argumentos a la función *trainClassCascadeANN* cada vez que se llame.
    - *transferFunction*, de tipo *Function*.
    - *maxEpochs*, de tipo *Int*.
    - *minLoss*, de tipo *Real*.
    - *learningRate*, de tipo *Real*.
    - *minLossChange*, de tipo *Real*.
    - *lossChangeWindowSize*, de tipo *Int*.

  Esta función recibirá las entradas y salidas deseadas como matrices donde cada fila tendrá una instancia. Por lo tanto, lo primero que deberá hacer es trasponer estas matrices.

Además, deberá convertir la matriz de entradas a *Float32*.

Lo primero que deberá hacer esta función es crear una RNA sin capas ocultas, mediante una llamada a la función *newClassCascadeNetwork* previamente desarrollada. Posteriormente, se llamará a la función *trainClassANN!* para entrenar esta primera RNA. Esta llamada devolverá el primer vector con los primeros valores de *loss* de todo el proceso.

Posteriormente, se creará un bucle con tantas iteraciones como *maxNumNeurons*. En cada iteración se hará lo siguiente:

- Llama a la función *addClassCascadeNeuron* para crear una nueva RNA con una nueva capa con una neurona.
- Si el número de capas/neuronas de esta RNA es mayor que 1, es momento de entrenar la RNA congelando todas las capas excepto las dos últimas. Para ello, se llama a la función *trainClassANN!*, indicando *trainOnly2LastLayers=true*, que devuelve el vector de valores de *loss* de este entrenamiento. Este vector se concatenará con el vector global de valores de *loss* que ya se tiene, con una salvedad: el primer valor del nuevo vector recibido será igual que el que último que ya se tiene, por lo que no se concatenan ambos vectores, sino que el vector de valores globales de *loss* se concatena con el vector de valores recibidos, comenzando por el segundo valor.
- Posteriormente, se entrena toda la RNA, llamando de nuevo a la función *trainClassANN!*, indicando *trainOnly2LastLayers=false*, que devolverá los valores de *loss* de este entrenamiento. Al igual que antes, se concatena a partir del segundo valor con el vector de valores globales de *loss*.

Finalmente, esta función devolverá una tupla con estos valores, por orden:

- RNA desarrollada.
- Vector de valores de *loss* de tipo *Float32*.

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

- *trainClassCascadeANN*. Función análoga a la anterior, pero en lugar de recibir las salidas deseadas como una matriz, las recibe como un vector. Esta función recibe como parámetros obligatorios:

  - maxNumNeurons, de tipo Int.

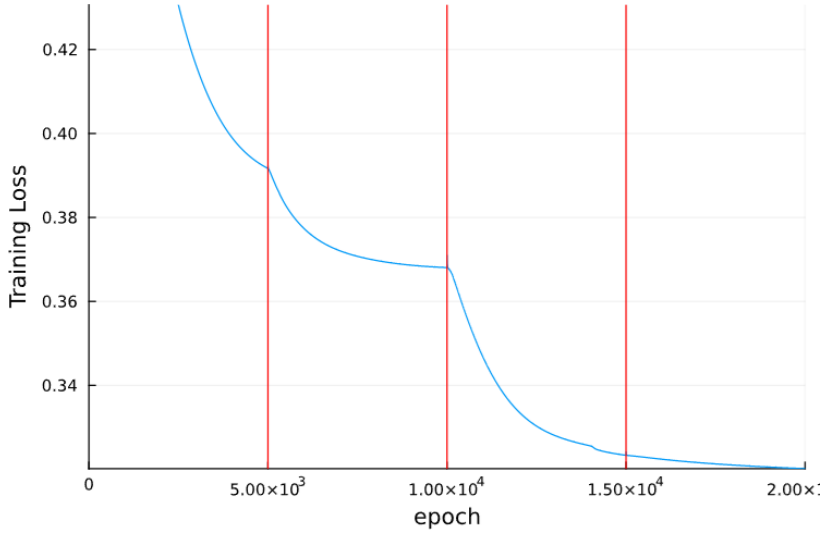
  - *trainingDataset*, de tipo *Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}}*, con el lote de datos a usar para entrenar la RNA.

  Los parámetros opcionales que recibe serán iguales a los de la función anterior.

Esta función lo único que hará será convertir el vector de salidas deseadas en matriz (mediante la función *reshape*) y llamar a la función anterior con los parámetros obligatorios y opcionales. Devolverá, por tanto, lo mismo que la función anterior.

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

Una vez desarrolladas estas funciones, es interesante proceder a entrenar alguna red en cascada y comparar este entrenamiento con el entrenamiento de Redes de Neuronas Artificiales desarrollado en la asignatura de FAA. Un ejemplo de *dataset* en el que probar podría ser el de "colic", con un valor de cambio mínimo en el *loss* de 10^{-6} en una ventana de 5 *epochs*, entrenándose cada etapa durante un máximo de 5000 *epochs*, con una tasa de aprendizaje de 0.001, una función de transferencia sigmoidea y un máximo de 2 neuronas ocultas. Como resultado de este entrenamiento, se puede generar la siguiente gráfica.



En ella, se puede observar claramente cómo la adición de nuevas neuronas permite obtener mejoras importantes en el valor de *loss*.

- ¿A qué se corresponde cada una de estas líneas verticales? ¿Qué partes se entrenan después de cada una de ellas?
- ¿Cuál es el valor de *loss* que se obtiene, comparado con el de una RNA clásica con una capa oculta con el número de neuronas? ¿Qué modelo devuelve un valor menor? ¿Por qué se produce esto? ¿Es más relevante el número de neuronas o el número de conexiones?

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 newClassCascadeNetwork(numInputs::Int, numOutputs::Int)
function addClassCascadeNeuron(previousANN::Chain; transferFunction::Function=σ)
function trainClassANN!(ann::Chain, trainingDataset::Tuple{AbstractArray{<:Real,2},
        AbstractArray{Bool,2}}, trainOnly2LastLayers::Bool;
        maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.001,
        minLossChange::Real=1e-7, lossChangeWindowSize::Int=5)
function trainClassCascadeANN(maxNumNeurons::Int,
        trainingDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,2}};
        transferFunction::Function=σ,
        maxEpochs::Int=1000, minLoss::Real=0.0, learningRate::Real=0.001,
        minLossChange::Real=1e-7, lossChangeWindowSize::Int=5)
function trainClassCascadeANN(maxNumNeurons::Int,
        trainingDataset::Tuple{AbstractArray{<:Real,2}, AbstractArray{Bool,1}};
        transferFunction::Function=σ,
        maxEpochs::Int=100, minLoss::Real=0.0, learningRate::Real=0.01,
        minLossChange::Real=1e-7, lossChangeWindowSize::Int=5)
```