Redes de Hopfield

Modelos Avanzados de Aprendizaje Automático I

Las redes de Hopfield son un tipo de red neuronal recurrente que ha sido fundamental en el desarrollo de la Inteligencia Artificial y el Aprendizaje Automático. Introducidas por John Hopfield en 1982, estas redes están diseñadas para realizar tareas de almacenamiento y recuperación de patrones de manera eficiente. La principal característica de las redes de Hopfield es su capacidad de operar como una memoria asociativa, lo que significa que pueden almacenar un conjunto de patrones y, a partir de un patrón incompleto o ruidoso, recuperar el patrón completo más cercano al original almacenado.

Estructuralmente, una red de Hopfield consta de un conjunto de neuronas binarias, donde cada neurona está interconectada con todas las demás. Estas conexiones tienen pesos sinápticos que se ajustan durante el proceso de entrenamiento para que la red pueda almacenar múltiples patrones de manera estable. Una vez entrenada, la red puede actualizar sus estados neuronales de manera iterativa hasta que alcanza un estado de equilibrio, conocido como un estado de energía mínima, que corresponde al patrón almacenado más parecido al de entrada. Las aplicaciones de las redes de Hopfield son diversas e incluyen la reconstrucción de imágenes, la resolución de problemas de optimización y la simulación de procesos biológicos. En este ejercicio, se utilizarán redes de Hopfield con dos fines distintos: almacenamiento y reconstrucción de imágenes, y clasificación de imágenes, concretamente el *dataset* MNIST, descrito en el ejercicio 1.

Para entrenar una red de Hopfield, dada una matriz S de dimensiones Nxm, con N el número de instancias y m el número de atributos, este entrenamiento se puede realizar con:

$$W = \frac{1}{N} S^T S$$

En el terreno de las redes de neuronas, se entiende que cada instancia está en cada columna; sin embargo, como se puede ver en esta formulación, en este ejercicio se entenderá que cada instancia está en cada fila. Dada la simplicidad de la formulación, cambiar entre una y otra es tan sencillo como cambiar la formulación de S^TxS a SxS^T. En cualquier caso, la matriz W tendrá una dimensión mxm. Además, las redes de Hopfield tienen los pesos de la diagonal igual a 0, para que ninguna neurona tenga conexión consigo misma, y las neuronas no suelen tener *bias*. En el modelo original, este conjunto de entrenamiento S estaba formado por valores binarios que podían ser -1 o 1. Este es el modelo que se va a usar en este ejercicio. Sin embargo, se realizarán las funciones para que los valores binarios que acepte sean también 0 y 1.

Una vez entrenado, ante una nueva instancia S, vector de longitud m (que se corresponde con una matriz con una columna, es decir, de dimensiones mx1), la salida de la RNA se puede calcular mediante WxS, que devuelve otra instancia S', también vector. Ese nuevo estado se puede volver a utilizar como entrada, y este proceso se puede repetir de forma indefinida hasta que se alcance un estado estable, es decir, que la salida sea igual que la entrada. Otra posibilidad es alcanzar un estado periódico, es decir que vaya oscilando entre los mismos valores.

Como en el resto de modelos de redes de neuronas, es posible utilizar más de una instancia como entrada, es decir, la matriz S tendrá dimensiones mxN, dando una salida correcta S'=WxS de dimensiones mxN, o a partir de una matriz S de tamaño Nxm, realizando S'=SxW, de tamaño Nxm. Sin embargo, para cada instancia el número de veces que es necesario utilizarla como entrada para que dé un estado estable es distinto, con lo que es conveniente utilizar cada instancia por separado.

Para utilizar una matriz para entrenar una red de Hopfield, o como entrada de la misma, es necesario realizar dos pasos:

- Umbralizar la matriz para convertirla en valores binarios.
- Cambiar las dimensiones de la matriz para convertirla en un vector, o matriz con una columna. Es decir, tomar los valores de la matriz y almacenarlos de forma consecutiva.

Una vez que se tiene la salida de una red de Hopfield, para convertirla en matriz, es necesario hacer el proceso inverso: tomar el vector de salida S y transformarlo en una matriz de dos dimensiones.

En este ejercicio, se crea un nuevo tipo, llamado *HopfieldNet*, para que sea más fácil definir las funciones y operaciones a realizar. Como una red de Hopfield únicamente es una matriz de pesos, este tipo se define como:

```
HopfieldNet = Array{Float32,2}
```

Además, en este ejercicio se dan las siguientes funciones hechas:

- *runHopfield*. Recibe una red de Hopfield y permite simular su comportamiento ante una instancia o un conjunto de instancias. Esta función está sobrecargada y tiene 3 métodos, según lo que reciba como segundo argumento:
    - Si recibe una instancia como un vector (de tipo *AbstractArray{<:Real,1}*), ejecuta la red de Hopfield hasta que encuentra un estado estable. Cada iteración de la red de Hopfield se realizará mediante una función llamada *stepHopfield* a desarrollar.
    - Si recibe un conjunto de instancias como una matriz con una instancia por fila (de tipo

AbstractArray{<:Real,2}), hace llamadas al método anterior de forma separada para crear la matriz de estados.

- Si recibe un conjunto de imágenes en formato NCHW (de tipo *AbstractArray{<:Real,4}*), hace lo siguiente:
  - Convierte la matriz de imágenes en una matriz bidimensional donde cada imagen será una fila (usando la función *reshape*).
  - Hace una llamada al método anterior con esta matriz, que devolverá los estados finales.
  - Convierte esta matriz de estados finales de nuevo en un conjunto de imágenes en formato NCHW (usando la función *reshape*).

- *showImage*. Esta función, con 3 métodos, permite representar imágenes de una forma sencilla. Esta función ha sido descrita en el ejercicio 1, y en este ejercicio puede ser de gran utilidad para comprobar que las salidas de las funciones a desarrollar son correctas.

En este ejercicio, se pide realizar las siguientes funciones:

- *trainHopfield*. Esta función permite entrenar una red de Hopfield. Recibe un único argumento, y es el *dataset* a utilizar para entrenar la red de Hopfield. Es necesario desarrollar 3 versiones o métodos, que recibirán este argumento con tipo distinto. Estos métodos harán lo siguiente, según el tipo de este argumento:

  - *AbstractArray{<:Real,2}*. En este caso recibe una matriz donde en cada fila se tiene una instancia, y, por lo tanto, el número de columnas es el número de atributos. Los valores de esta matriz deben ser binarios, con valores -1 o 1. Esta función entrena una red de Hopfield siguiendo la ecuación anterior (S^TxS/N), y posteriormente pone los elementos de la diagonal a 0. El tipo de elementos de la matriz resultante ha de ser *Float32*.

  - *AbstractArray{<:Bool,2}*. En este caso recibe una matriz de valores booleanos (0 o 1). Lo que se realiza aquí es una conversión a valores reales, -1 o 1, y se devuelve el resultado de llamar al método anterior. Para ello, si el argumento se llama *trainingSet*, se puede hacer mediante

```
(2. .*trainingSet) .- 1
```

  - *AbstractArray{<:Bool,4}*. Recibe un *dataset* de imágenes umbralizadas (valores booleanos) en formato NCHW, es decir, con 4 dimensiones. Lo que tiene que hacer es

convertir esta matriz en una matriz de dos dimensiones mediante la función *reshape*, de igual manera que lo hace el método correspondiente de la función *runHopfield*, y devolver el resultado de una llamada a la función anterior.

Para el desarrollo de la primera versión de esta función (la que recibe un argumento de tipo *AbstractArray{<:Real,2}*) se permite el uso de un bucle, para poner los elementos de la diagonal a 0. Para las otras dos versiones no se permite el uso de bucles.

- *stepHopfield*. Esta función permite, a partir de un estado, calcular el siguiente aplicando la red de Hopfield. Esta red de Hopfield será el primer argumento que reciba (de tipo *HopfieldNet*). Esta función estará sobrecargada, y habrá que desarrollar dos métodos distintos, según el tipo del segundo argumento:
    - *AbstractArray{<:Real,1}*. Recibe un vector de valores reales binarios (-1 o 1), es decir, un estado de la red, y lo aplica a la red para calcular el siguiente estado. Para realizar esto, se darán los siguientes pasos:
        - En primer lugar, se convierte el vector de entrada en valores de tipo *Float32*.
        - Multiplica la matriz de pesos por este vector.
        - Se aplica un umbral en 0 para transformar el resultado del paso anterior en valores binarios -1 o 1. Para hacer esto, utilizar la función *sign*.
        - Como resultado, esta función deberá devolver un vector de elementos de tipo *Float32*.
    - *AbstractArray{<:Bool,1}*. Recibe un vector de valores booleanos, que será usado como entrada a la red de Hopfield, y se devolverá un vector de valores booleanos con la salida de la red. Para hacer esto, se harán los siguientes pasos:
        - Transformar este vector en valores binarios -1 o 1 de igual manera que en la función *trainHopfield*.
        - Con este nuevo vector, llamar al método anterior. Como resultado, se tendrá un vector de valores reales binarios -1 o 1.
        - Convertir este vector de valores reales en un vector de valores booleanos. Esto se hará fácilmente haciendo un *broadcast* de la operación >= comparando la salida con 0. De esta manera, esta función devolverá un vector

de valores booleanos.

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

- *addNoise*. Función que, dado un conjunto de imágenes en formato NCHW con valores booleanos, añade ruido a las imágenes en un determinado ratio. Los parámetros que recibe esta función son los siguientes:

    - *datasetNCHW*, de tipo *AbstractArray{<:Bool,4}*, con el conjunto de imágenes en las que añadir ruido. Esta matriz no debe ser modificada, por lo que lo primero que hay que hacer en esta función es hacer una copia (mediante la función *copy*) de esta variable.

    - *ratioNoise*, de tipo *Real*, con el ratio de píxeles a cambiar, como un valor entre 0 y 1. Un valor de 0 indica que las imágenes a devolver son iguales a las originales, mientras que un valor de 1 indica que son las imágenes "opuestas".

    Para hacer esta función, puede ser interesante tener en cuenta dos cuestiones:
    - La función *length*, aplicada a una matriz de cualquier dimensionalidad, devuelve el número total de elementos de esa matriz.
    - Cualquier matriz, tenga la dimensionalidad que tenga, puede ser referenciada con un único índice, con un valor desde 1 hasta el número total de elementos.

    De esta manera, si noiseSet tiene el conjunto de imágenes a modificar, los índices de los píxeles a modificar se calcularán mediante

```
indices = shuffle(1:length(noiseSet))[1:Int(round(length(noiseSet)*ratioNoise))];
```

    También es posible realizar el cálculo de los índices mediante la función *randperm*:

```
indices = randperm(length(noiseSet))[1:Int(round(length(noiseSet)*ratioNoise))];
```

    Únicamente será necesario cambiar los valores de los píxeles indicados en el vector *indices*.
    Para el desarrollo de esta función no se permite el uso de bucles.

- *cropImages*. Función que, dado un conjunto de imágenes en formato NCHW con valores booleanos, elimina la información de la parte derecha de las imágenes. Los parámetros que recibe esta función son los siguientes:

    - *datasetNCHW*, de tipo *AbstractArray{<:Bool,4}*, con el conjunto de imágenes en las que eliminar la parte derecha. Esta matriz no debe ser modificada, por lo que lo

primero que hay que hacer en esta función es hacer una copia (mediante la función *copy*) de esta variable.

- *ratioCrop*, de tipo *Real*, con el ratio a cambiar en el sentido horizontal, como un valor entre 0 y 1. Un valor de 0 indica que las imágenes a devolver son iguales a las originales, con un valor de 0.5, las imágenes tendrán toda la mitad derecha en color negro, mientras que un valor de 1 provocará que las imágenes sean totalmente negras.

Esta función no modificará la matriz, sino que tomará una copia de esta y pondrá los píxeles correspondientes del lado derecho a 0.

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

Estas dos funciones permiten modificar el *dataset* de imágenes cargadas en el ejercicio 1 en formato NCHW. Es interesante tomar las imágenes, umbralizarlas, entrenar una red de Hopfield, modificarlas mediante una de estas funciones, y aplicar la red para ver qué imágenes es capaz de reconstruir. Para esto, es útil ir mostrando las imágenes en pantalla para poder ver los resultados.

  - ¿Qué valores de umbralización son los más adecuados?
  - ¿Qué ratio de ruido o de borrado permite la reconstrucción?

- *randomImages*. Esta función crea una matriz de imágenes aleatorias en formato NCHW, con valores booleanos. Esta función recibe como parámetros:

  - *numImages*, de tipo *Int*, con el número de imágenes a generar.

  - *resolution*, de tipo *Int*, con el número de píxeles de alto y ancho de las imágenes (todas serán cuadradas).

  Por lo tanto, esta función devolverá una matriz de tamaño *numImages* x 1 x *resolution* x *resolution*, de valores booleanos. La densidad de valores a 1 y a 0 debería de ser similar. Para ello, se utilizará la función *randn*, indicando las dimensiones de la matriz a devolver, y se comparará la matriz resultado con 0 mediante una operación de *broadcast*.
  Para el desarrollo de esta función no se permite el uso de bucles.

  Al igual que en las dos funciones anteriores, es interesante cargar las imágenes de prueba con la función del ejercicio 1, entrenar una red de Hopfield, aplicar estas imágenes aleatorias

y visualizar los resultados.

Con estas funciones, es interesante explorar la capacidad de las Redes de Hopfield para memorizar patrones (en este caso, imágenes), y reconstruirlas. Una posible experimentación podría ser cargar todo el *dataset* de imágenes (mediante la función *loadImagesNCHW*), umbralizarlas y entrenar una red de Hopfield mediante la función *trainHopfield*. Posteriormente, corromper las imágenes mediante *addNoise* o *cropImages*, e intentar reconstruirlas mediante *runHopfield*. Todo este proceso puede ser visualizado mediante las funciones *showImage*.

- ¿Hasta qué ratio de cambio de imágenes (ruido o eliminación de la parte derecha) la red de Hopfield es capaz de reconstruir la imagen original?

Otra posibilidad es crear imágenes nuevas aleatorias mediante la función *randomImages*, y posteriormente llamar a la función *runHopfield* para obtener alguna de las imágenes memorizadas, que puede ser visualizada mediante *showImage*.

El resto de funciones a desarrollar en este ejercicio se refieren a la clasificación de imágenes, y son las siguientes:

- *averageMNISTImages*. Esta función recibe un conjunto de imágenes en formato NCHW, con las imágenes de MNIST, y devuelve un nuevo conjunto de imágenes, con una sola imagen por dígito. Esta función recibe dos parámetros:
    - *imageArray*, de tipo *AbstractArray{<:Real,4}*, con el conjunto de imágenes.
    - *labelArray*, de tipo *AbstractArray{Int,1}*, con el mismo número de elementos que imágenes en el parámetros anterior. Contiene, para cada imagen, el dígito correspondiente.

  Esta función deberá devolver una tupla con dos elementos, por este orden:
    - Un conjunto de imágenes en formato NCHW con una imagen por dígito, que será el resultado de promediar todas las imágenes de ese dígito. El tipo de elemento de esta matriz deberá ser el mismo que el de la matriz de entradas.
    - Vector *labels* con las etiquetas de las imágenes de la matriz devuelta, con un solo valor por dígito.

  Para hacer esto, se darán los siguientes pasos:
    - Tomar los dígitos a considerar haciendo

```
labels = unique(labelArray)
```

- Crear la matriz de salida en formato NCHW, donde N será la longitud de *labels*, no la longitud de *labelArray*.
- Hacer un bucle para cada valor de 1 a N, con N es la longitud de *labels* (igual que antes, no la longitud de *labelArray*). Para cada valor, tomar el dígito correspondiente de *labels*, promediar las imágenes que corresponden a ese dígito, y asignarla a la matriz. Este promedio se puede hacer mediante

```
dropdims(mean(imageArray[labelArray.==labels[indexLabel], 1, :, :], dims=1), dims=1)
```

- Finalmente, devolver una tupla donde el primer elemento es la matriz creada. Los valores de esta matriz deberán ser del mismo tipo que los valores de la matriz de entrada *imageArray*. El segundo valor de la tupla será el vector *labels* con los dígitos utilizados.

Por ejemplo, si *labelArray* tiene como valores [2,4,4,5,2,4,2,5], el vector *labels* tendrá [2,4,5], y se creará una matriz NCHW donde N=3. Los elementos situados en [1,1,:,:] serán un promedio de todas las imágenes de dígito 2, los elementos situados en [2,1,:,:] serán un promedio de todas las imágenes de dígito 4, y los elementos situados en [3,1,:,:] serán un promedio de todas las imágenes de dígito 5.

Para el desarrollo de esta función se permite un único bucle.

- *classifyMNISTImages*. Función que permite clasificar imágenes de MNIST. La forma de clasificarlas es mediante la comparación con una plantilla de imágenes. Si una imagen es igual a alguna que está en la plantilla, se devolverá el dígito de la plantilla. Para ello, ambos grupos de imágenes deberán de haber sido umbralizados, es decir, sus valores serán booleanos. Esta función recibe como parámetros:

  - *imageArray*, de tipo *AbstractArray{<:Bool,4}*, con las imágenes a clasificar umbralizadas en formato NCHW.

  - *templateInputs*, de tipo *AbstractArray{<:Bool,4}*, con las imágenes usadas como plantilla (y, por lo tanto, umbralizadas también) en formato NCHW.

  - *templateLabels*, de tipo *AbstractArray{Int,1}*, un vector con las etiquetas de las imágenes usadas como plantilla. Este vector no deberá contener etiquetas repetidas. Este parámetro y el anterior son el resultado de una llamada a la función anterior.

  Para hacer esta función, dar los siguientes pasos:

  - Crear un vector *outputs* de longitud igual al número de imágenes a clasificar.

Inicializar los valores de este vector a -1.

- Hacer un bucle que itere por las imágenes de la plantilla (no de las imágenes a clasificar). En el interior de este bucle, hacer en cada ciclo:
  - Se toma la etiqueta y la imagen plantilla. Para tomar la imagen plantilla y poder compararla después con las imágenes a clasificar, esta deberá estar en formato NCHW, es decir, que tenga 4 dimensiones, y las dos primeras solamente contendrán una entrada. Esto se puede hacer de la siguiente manera, suponiendo que *indexLabel* contiene el índice de la etiqueta en el vector *templateLabels*, no la etiqueta en sí:

```
template = templateInputs[[indexLabel], :, :, :];
```

  - Se compara esta imagen plantilla con todas las imágenes del conjunto a clasificar *imageArray*, generando un vector con las coincidencias, es decir, qué imágenes son iguales a la plantilla de esta iteración. Esto se hará con

```
indicesCoincidence = vec(all(imageArray .== template, dims=[3,4]));
```

  - Actualizar el vector *outputs* en estas posiciones de coincidencia poniendo como valor de clasificación la etiqueta de esta imagen de plantilla.
- Finalmente, esta función devolverá el vector *outputs*, es decir, devolverá un vector de valores enteros.

Para el desarrollo de esta función se permite un único bucle.

- *calculateMNISTAccuracies*. Esta función realiza todo el proceso previo para calcular las precisiones al clasificar dígitos del *dataset* MNIST utilizando redes de Hopfield. Esta función recibe como parámetros:
  - *datasetFolder*, de tipo *String*, con el *path* a la carpeta donde está el *dataset* MNIST, que se cargará con la función desarrollada en el ejercicio 1.
  - *labels*, de tipo *AbstractArray{Int,1}*, vector con las etiquetas a considerar en este problema de clasificación, pudiendo alguna de ellas tener el valor de -1. Este vector no debe contener valores repetidos, y será pasado como argumento a la función de cargar el *dataset* MNIST. Este vector dirá qué dígitos se toman para hacer la clasificación. Por ejemplo, si se quiere resolver un problema de 2 clases, separando entre unos y sietes, este vector será igual a [1,7]. Este sería un ejemplo de una estrategia “uno contra uno”; si se quiere hacer un “uno contra todos”, otro ejemplo

podría ser [1,-1], donde se intenta clasificar el dígito 1 separándolo del resto de dígitos. Se podrían hacer múltiples combinaciones, como [2,5,8] o [3,6,8,-1].

- *threshold*, de tipo *Real*, con el valor de umbral a utilizar a la hora de pasar las imágenes a valores booleanos.

Para hacer esta función, dar los siguientes pasos:

- Cargar el *dataset* MNIST con la función desarrollada en el ejercicio 1, pasando como argumento el vector con las etiquetas a considerar, y el tipo de dato igual a *Float32*. Esta llamada devolverá 4 valores: matrices de imágenes de entrenamiento y test, y etiquetas de entrenamiento y test (no por este orden).
- Con las imágenes de entrenamiento y las etiquetas de entrenamiento, hacer una llamada a la función *averageMNISTImages*, que devolverá las imágenes a usar como plantillas de entrenamiento, y las etiquetas de estas plantillas.
- Umbralizar las tres matrices de imágenes: entrenamiento, test y plantillas, haciendo un broadcast de la operación >= comparando con el umbral pasado como parámetro.
- Entrenar una red de Hopfield con la matriz de plantillas (umbralizada).
- Calcular la precisión en el conjunto de entrenamiento haciendo:
  - Ejecutar esta red de Hopfield con la matriz de imágenes de entrenamiento (umbralizada). Esto devolverá una nueva matriz de imágenes.
  - Llamar a la función *classifyMNISTImages* pasando esta matriz como argumento, así como la matriz de imágenes plantilla (umbralizada) y sus etiquetas. Esta llamada devolverá un vector con las predicciones de etiquetas de cada imagen del conjunto de entrenamiento.
  - Finalmente, se calculará la precisión en el conjunto de entrenamiento comparando este vector con el vector de etiquetas del conjunto de entrenamiento.
- Realizar operaciones similares al paso anterior para el conjunto de test.

Por tanto, esta función deberá devolver una tupla con dos valores, por este orden: precisión en el conjunto de entrenamiento y de test.

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

En esta última función es interesante experimentar con distintos valores de umbral, así como distintos dígitos a clasificar. Por ejemplo, para separar entre unos y sietes,

- ¿Qué valores de umbral ofrecen mejores precisiones?
- ¿Qué estrategia ofrece mejores resultados, “uno contra uno” o “uno contra todos”?

- ¿Qué dígitos son más fáciles de separar entre ellos, y del resto?

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 trainHopfield(trainingSet::AbstractArray{<:Real,2})
function trainHopfield(trainingSet::AbstractArray{<:Bool,2})
function trainHopfield(trainingSetNCHW::AbstractArray{<:Bool,4})
function stepHopfield(ann::HopfieldNet, S::AbstractArray{<:Real,1})
function stepHopfield(ann::HopfieldNet, S::AbstractArray{<:Bool,1})
function addNoise(datasetNCHW::AbstractArray{<:Bool,4}, ratioNoise::Real)
function cropImages(datasetNCHW::AbstractArray{<:Bool,4}, ratioCrop::Real)
function randomImages(numImages::Int, resolution::Int)
function averageMNISTImages(imageArray::AbstractArray{<:Real,4}, labelArray::AbstractArray{Int,1})
function classifyMNISTImages(imageArray::AbstractArray{<:Bool,4},
       templateInputs::AbstractArray{<:Bool,4}, templateLabels::AbstractArray{Int,1})
function calculateMNISTAccuracies(datasetFolder::String, labels::AbstractArray{Int,1},
       threshold::Real)
```