3 Clusterización con Matlab y Python

OBJETIVOS

- Comprender el funcionamiento de una red competitiva
- Diseñar y configurar una red SOM en Matlab
- Diseñar y configurar una red SOM en Python
- Utilizar el paquete *minisom* de Python
- Entender el funcionamiento de la red LVQ

3.1 Aprendizaje competitivo

Las neuronas en una capa competitiva se distribuyen con objeto de reconocer similitudes entre los patrones de entrada.

Dado un patrón de entrada $p \in Rx1$, con R características, la salida de la red de aprendizaje competitivo $a \in Sx1$, será un vector en el que todas las componentes son

0, salvo la neurona ganadora cuyo valor es 1. Como se observa en la figura 3.1, al introducir un patrón de entrada p, la salida de la primera capa (n) será un vector de S valores (tantos como número de neuronas de dicha capa). Cada uno de los elementos de este vector representará el producto entre el patrón de entrada p y el vector de pesos de cada neurona (junto con las desviaciones en su caso, b). Este vector n es el que se introduce a la capa competitiva con objeto de calcular el de menor distancia. Esta última parte de la capa competitiva tiene por cometido ofrecer un valor 0 para todas las componentes que no tengan la mínima distancia, y un valor de 1 para aquella componente con el valor de mínima distancia.

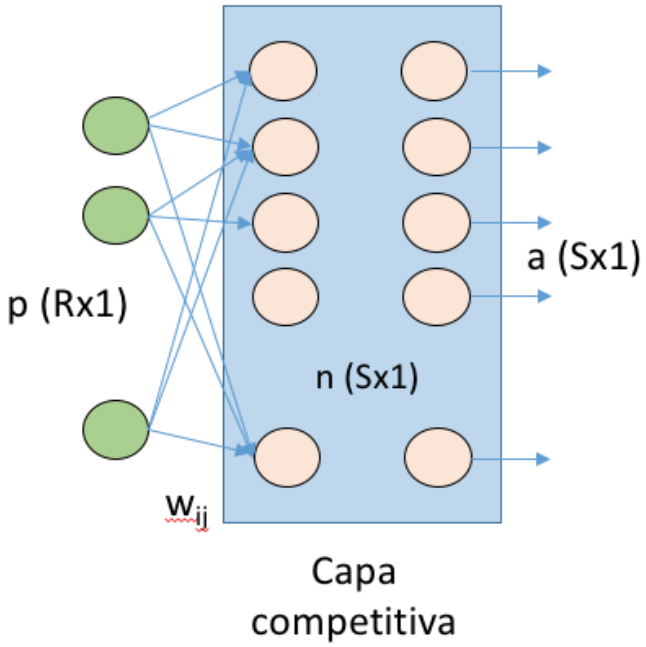

Figura 3.1: Arquitectura de una red con aprendizaje competitivo

3.1.0.1 Creación de una red competitva con Matlab

Supongamos que tenemos un conjunto de patrones (8) con dos características cada uno de ellos:

```
1 p=[.2 .3 .2 .4 .5 .7 .8 .9; .1 .2 .3 .1 .5 .6 .8 .7]
```

En la figura 3.2 se puede observar una distribución en el plano de estos patrones, dado que cada uno de ellos contiene sólo dos características (dos valores).

El objetivo que perseguimos es dividir este conjunto de patrones en dos clases. Pero esta división en dos clases distintas la queremos efectuar de una forma no supervisada

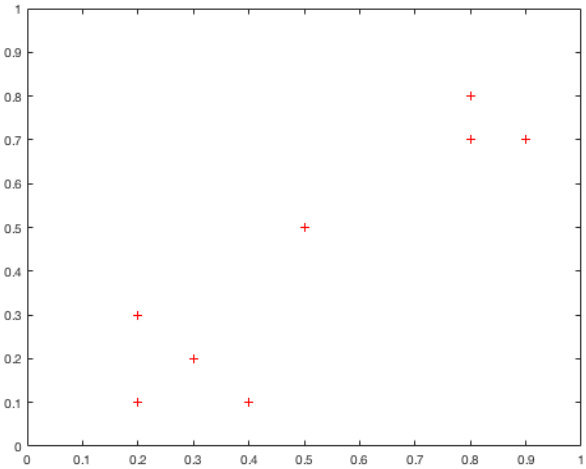

Figura 3.2: Distribución de patrones en el plano

dado que no tenemos ningún conocimiento acerca de a qué clase debe pertenecer cada uno de los patrones de entrada. Lo que esperamos es que la red sea capaz de inferir algún tipo de relación entre los patrones de entrada como para poder clasificar en dos clases distintas. Si conociéramos a qué clase debe pertenecer cada patrón de entrada podríamos desarrollar y entrenar una red como en el capítulo previo realizando un aprendizaje supervisado. Para ello, en primer lugar crearemos una capa de neuronas con dos elementos (cada patrón tiene dos características). Esto lo hacemos mediante el comando *newc*[1]

```
net = newc ([0 1; 0 1], 2)
view(net)
```

Como observamos le estamos indicando al crear la capa de neuronas, que cada patrón de entrada presenta dos características (pasamos como parámetro una matriz con dos filas), y que la primera característica se encuentra dentro del rango $0-1$, al igual que la segunda característica (son los valores que le pasamos en cada una de las columnas de la matriz). El segundo de los argumentos (2) indica a la red que disponga dos neuronas. Creada la arquitectura de la red, es preciso inicializar los pesos y desviaciones de cada una de estas dos neuronas que hemos creado. Esta red, al crearse, ha incializado los pesos y desviaciones de las neuronas. Podemos acceder al valor que presentan estos pesos y desviaciones de la red que hemos creado:

```
net.IW\{1\}
net.b\{1\}
```

[1]Este comando en las últimas versiones de Matlab ha pasado a denominarse como competlayer

Una vez creada la red e inicializados los pesos y desviaciones, es preciso entrenar la red de una forma no supervisada (recordemos que no conocemos a qué clase pertenece cada patrón o dato de entrada). Para ello se puede utilizar la regla de aprendizaje de Kohonen. Esta regla de aprendizaje adapta los pesos de cada una de las neuronas en función de si una neurona resulta ganadora o no (se entiende como neurona ganadora aquella cuyo vector de pesos es más similar al patrón presentado a la entrada). Cuando se tiene una neurona ganadora, su vector de pesos se adapta para hacerlo más próximo al patrón presentado a la entrada. En Matlab se dispone de una función para realizar este aprendizaje, que se denomina *learnk*.

Una de las limitaciones de las redes competitivas es que pueden existir neuronas que no resultan nunca ganadoras. Esto es, si alguna de las neuronas presenta unos vectores de pesos que se encuentran muy alejados de los datos de entrada, quizá no gana nunca y no se adapta a estos patrones de entrada (son *neuronas muerta*s), dejando de ser útiles. Para evitar este efecto, una posibilidad es tener en cuenta el porcentaje en el que una neurona resulta ganadora, utilizando este valor para actualizar las desviaciones (b) en la función de aprendizaje, de forma tal que las desviaciones de aquellas neuronas que se activen muy a menudo (muy ganadoras) disminuyan, mientras que aumenten las desviaciones de las neuronas muy poco ganadoras. Esto es posible realizarlo mediante la función *learncon* en Matlab.

Matlab encapsula estas facilidades que nos ofrece, al crear una red general mediante el comando previo *newc*, y realizar un entrenamiento posterior, en un comando de mayor nivel que es *competlayer*.

Con objeto de comprobar el funcionamiento de estos parámetros, utilizaremos mayor número de datos que los presentados inicialmente en los patrones que hemos puesto como ejemplo hasta el momento (con únicamente 8 patrones). De esta forma haremos uso de un conjunto de datos ya disponible en Matlab como es `iris_dataset` (estos datos se corresponden con valores de flores con un total de 4 atributos o características: longitud del sepalo, anchura del sepalo, longitud del petalo, anchura del petalo). Se dispone de un total de 150 patrones cada uno de ellos con cuatro características (matriz de $4x150$).

```
1 x=iris\_dataset;
```

Podemos ver una representación gráfica de los datos cargados al representarlas gráficamente. Si representamos en un gráfico de 3 dimensiones, tres de sus características, podremos observar la variación espacial de estos patrones como se ve en la figura 3.3. Obviamente en un gráfico sólo podemos representar 3 de las cuatro características de los patrones o datos de entrada.

```
1 plot3(x(1,:),x(2,:), x(3,:), '+r');
```

A continuación, procederemos a crear una red neuronal de tipo competitivo con objeto de realizar una clasificación de estos 150 datos en 2 clases diferentes.

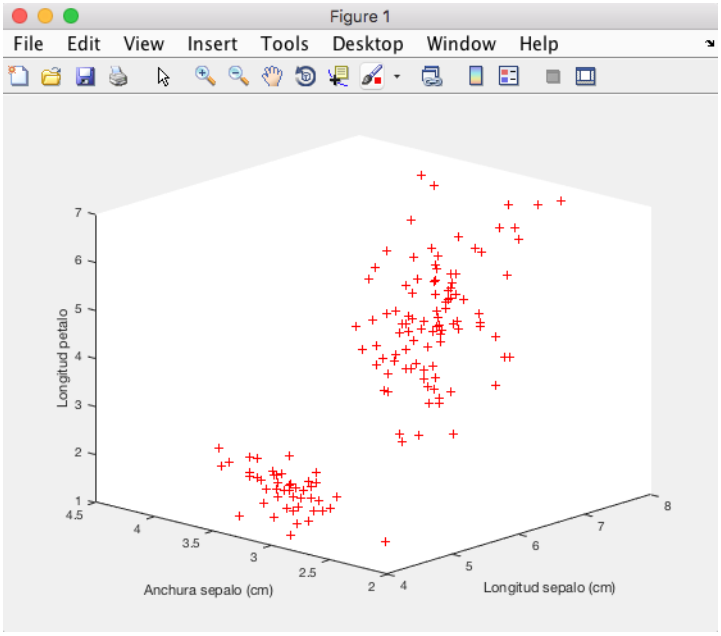

Figura 3.3: Representación gráfica de los 150 patrones de entrada (longitud y anchura del sepalo y longitud del petalo)

```
net = competlayer(2, 0.1)
```

Mediante este comando (*competlayer*) creamos una red competitiva para clasificar en 2 conjuntos (ver figura 3.4). El parámetro 0.1, indica la razón de aprendizaje en la regla de aprendizaje de Kohonen.

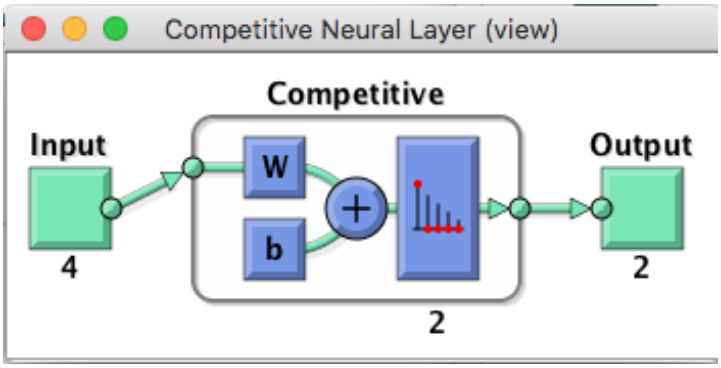

Figura 3.4: Red competitiva para clasificación en 2 conjuntos

```
net = configure(net,x);
w=net.IW\{1\};
```

Con el comando primero (*configure*), estamos configurando la red creada previamente a los datos de entrada. En la variable *w* hemos almacenado los pesos, que representarán los centros de cada una de las clases (en nuestro caso 2) en que deseamos clasificar el conjunto de patrones de entrada *x*. Podemos representar sobre la figura previa en la que habíamos dibujado los 150 patrones de entrada, la posición inicial de dichas clases *w*. Así en la figura 3.5 observamos tanto los 150 patrones de entrada como la posición inicial de dada una de las dos clases antes de realizar ningún entrenamiento sobre la red.

```
hold on;
circles = plot3(w(:,1), w(:,2), w(:,3), 'ob');
```

Como en el caso previo, si bien cada clase vendrá definida por cuatro valores (igual al número de características de cada patrón) sólo representamos los tres primeros valores de cada una de las clases (gráficamente no podemos representar los cuatro valores). Como se observa en la figura 3.5, al crear la red, todas las clases se encuentran en el promedio de los valores de entrada que le hemos pasado mediante el comando *configure*.

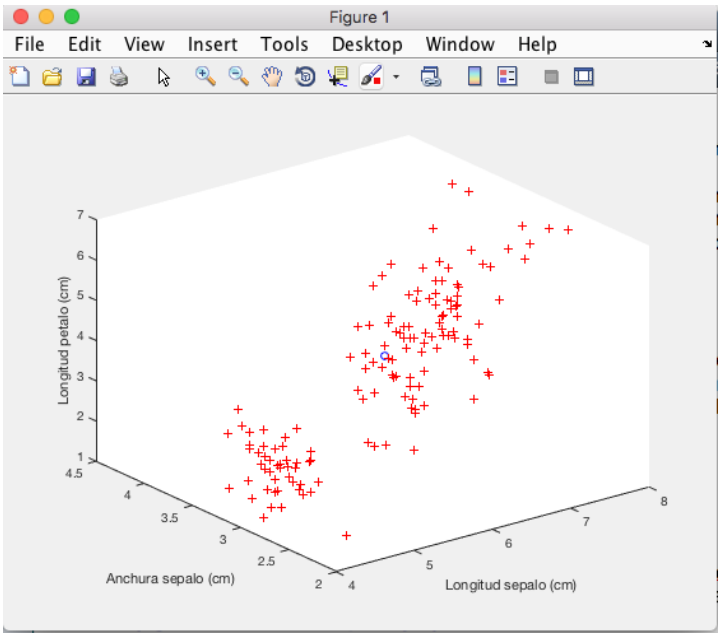

Figura 3.5: Posiciones de las dos clases (en círculo azul) inicialmente al configurar la red.

Una vez inicializada la red, procederemos al entrenamiento de la misma. Para ello seleccionaremos un total de 7 iteraciones.

```
net.trainParam.epochs = 7
net=train(net, x);
```

Tras el entrenamiento podemos observar cómo se ha modificado la posición de cada una de las clases según se observa en la figura 3.6.

```
w=net.IW\{1\};
delete(circles);
plot3(w(:,1), w(:,2), w(:,3),'ob');
```

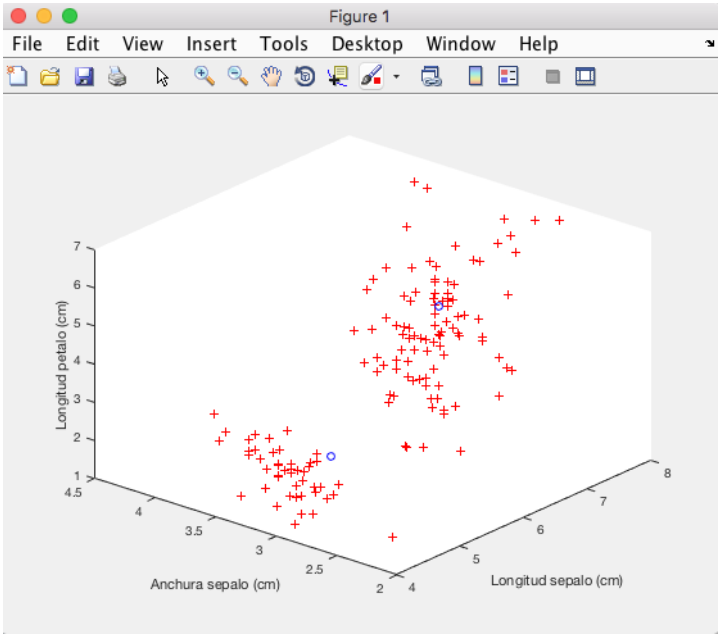

Figura 3.6: Posiciones finales de las dos clases (en círculo azul) tras el entrenamiento de la red competitiva.

Por último, si quisiéramos clasificar un patrón de entrada cualquiera a partir de la red, una vez que esta ha sido ya entrenada, lo podríamos hacer como sigue. Imaginemos que tenemos un patrón de entrada con valor $(6.4; 3.1; 5.5; 1.7)$:

```
patron=[6.4; 3.1; 5.5; 1.7];
sal=net(patron);
clase=vec2ind(sal)
```

Obteniendo como clase a la que le corresponde dicho patrón de entrada la clase 1 de las dos en que clasificaría la red competitiva creada.

3.2 ¿Qué son las SOM (Self Organizing Maps)?

Los Self Organizing Maps (SOM) o mapas auto-organizados es una arquitectura de red neuronal propuesta por T. Kohonen (profesor de la Universidad de Helsinki) en 1982. La propuesta de esta arquitectura se encuentra basada en las evidencias observadas a nivel cerebral. El objetivo de esta arquitectura de red es descubrir los rasgos comunes, regularidades, correlaciones o referencias similares entre los datos de entrada. Es decir, el objetivo consiste en que la red neuronal agrupe los datos que se le presenten a la entrada teniendo en cuenta los rasgos comunes que presenten las características que definen cada uno de estos datos de entrada. Resuelve pues un problema de agrupamiento o **clusterización**. De ahí el nombre de este tipo de redes, son las neuronas las que deben auto-organizarse en función de los datos que se le proporcionen con objeto de agruparse en torno a datos similares. De esta forma, se pretende que, una vez entrenada, cuando se proporcione a la red un nuevo dato de entrada, sólo una de las neuronas o bien un grupo alrededor de esta que sea la más similar al nuevo dato de entrada se active ante dicho dato. El tipo de aprendizaje en este tipo de redes es un aprendizaje no supervisado, puesto que a priori se desconoce a qué conjunto debe clasificarse cada uno de los datos. De hecho lo que se busca es que la propia red interprete estos datos y realice un agrupamiento teniendo en cuenta las similitudes existentes entre dichos datos.

3.2.1 Arquitectura de una red SOM

La arquitectura de una red de tipo SOM se encuentra formada por dos capas de neuronas (capa de entrada y capa de salida). La capa de entrada es la que recibe la información (datos de entrada). El número de neuronas de la capa de entrada dependerá en consecuencia del número de características o valores que tengamos para cada dato o patrón de entrada. Por ejemplo si los datos de entrada a la red se componen de un vector de 10 elementos, el número de neuronas en esta capa de entrada será de 10. El número de neuronas de la capa de salida se debe elegir (M). Estas neuronas son las encargadas de procesar la información a partir de los datos de entrada. Normalmente las neuronas de la capa de salida se agrupan en forma de mapas en dos dimensiones. Esta arquitectura se muestra en la figura 3.7.

Cada neurona de la capa de entrada se encuentra conectada hacia adelante con cada neurona de la capa de salida. Como se observa, entre cada neurona de la capa de entrada (i) con cada neurona de la capa de salida (j) existirá un peso asociado w_{ij}. Así, para la neurona j de la capa de salida, tendremos un vector de pesos asociado a sus entradas de un tamaño igual al número de neuronas N de la capa de entrada $W_j = (w_{1j}, w_{2j}, \ldots, w_{Nj})$.

A medida que se van introduciendo nuevos datos a la red a través de la capa de entrada, estos pesos se van modificando durante el proceso de aprendizaje.

Las dos ideas básicas en las que se basa el mapa auto-organizado de Kohonen son:

- Primero, la forma de adaptar estos pesos a medida que se realiza el aprendizaje.

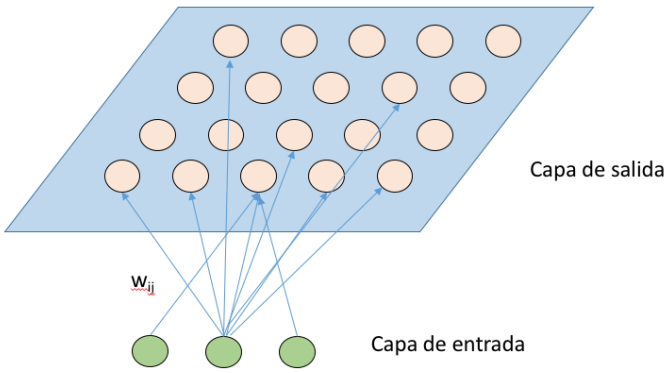

Figura 3.7: Arquitectura de una Red SOM

- Segundo, la relación topológica que presentan las neuronas de la capa de salida.

Como observamos en la figura 3.7, las neuronas de la capa de salida no presentan conexiones entre sí. Sin embargo, a pesar de no tener ninguna conexión entre estas neuronas, estas neuronas sí que presentan influencia sobre otras adyacentes, teniendo en cuenta la vecindad de las mismas (relación topológica entre ellas).

En las redes SOM es importante por lo tanto la topología en la que se representan las neuronas de la capa de salida. Las topologías más frecuentemente utilizadas son la rectangular y la hexagonal. En la topología rectangular se considera que una neurona es vecina de cuatro u ocho neuronas, mientras que en la hexagonal una neurona es vecina de seis neuronas (ver figura 3.8).

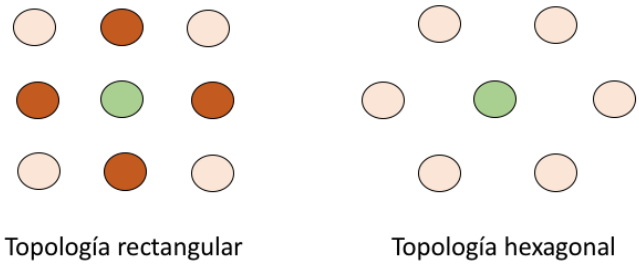

Figura 3.8: Topologia rectangular (con 4 u 8 vecinos) y hexagonal de una Red SOM

Teniendo en cuenta una topología, podemos asumir que existen un conjunto de neuronas adyacentes o vecinas a la neurona j dentro de una vecindad. Esta topología y el número de neuronas vecinas puede permanecer fijo desde un primer momento en que

se define la arquitectura de la red. Este número de neuronas de una cierta vecindad influye en el ajuste de los pesos y en la capacidad de generalización de la red.

3.2.2 Funcionamiento de una red SOM

Inicialmente, al crear la red neuronal, los pesos de las neuronas se establecen de forma aleatoria. Cuando se introduce un nuevo dato de entrada a la red, se selecciona la neurona de la capa de salida que resulta ganadora para este dato de entrada. Se considera como neurona ganadora aquella que presenta un vector de pesos más similar al dato de entrada (obviamente son dos vectores de igual dimensión), es decir, se selecciona como neurona ganadora aquella neurona que se encuentra más cercana (su vector de pesos) al dato de entrada. De esta forma es preciso establecer una función de distancia que nos estime y cuantifique la distancia entre estos dos vectores (el dato que se presenta a la entrada $X = (x_1, x_2, \ldots, x_N)$, y el vector de pesos de la neurona j de la capa de salida W_j).

Obviamente se pueden elegir diferentes funciones de distancia, y el modelo de red neuronal se comportará de una forma diferente.

Una posibilidad consiste en elegir como función de distancia, el producto escalar de dos vectores:

$$d(X, W_j) = \sum_{i=1}^{N} x_i w_{ij}$$

Con esta medida de distancia, la salida de cada neurona de la capa de salida será la habitual en las redes unidireccionales utilizadas en un proceso supervisado.

No obstante, una de las funciones más utilizadas como estimación de la distancia entre dos vectores es la distancia euclídea:

$$d(X, W_j) = \sqrt{\sum_{i=1}^{N} (x_i - w_{ij})^2}$$

A continuación, cuando se han generado las distancias entre el dato de entrada y cada uno de los vectores de pesos de cada neurona de la capa de salida, se establece una competición entre las mismas con objeto de seleccionar aquella neurona que presente una salida más pequeña (esto es una distancia menor al dato de entrada). Esta neurona es la neurona ganadora para la entrada seleccionada. Aquella neurona que ha resultado ganadora es la que presenta un vector de pesos a su entrada más similar con el patrón o dato empleado.

En la etapa de aprendizaje, una vez que se ha determinado la neurona ganadora se produce una adaptación de los pesos asociados a cada neurona en la capa de salida. Para ello se usa una regla de aprendizaje de tipo Hebb, es decir, se refuerza más el peso de aquellas neuronas que han respondido mejor ante la entrada X en dicho instante k, y de forma proporcional a esta entrada. Esto se puede hacer según la siguiente regla

de adaptación de pasos una vez que se ha pasado a la red un patrón de entrada en el instante k:

$$w_{ij}(k+1) = w_{ij}(k) + \alpha(k)N_j(k)(x_i(k) - w_{ij}(k))$$

donde w_{ij} es el peso que conecta la neurona i de la capa de entrada con la neurona j de la capa de salida, $\alpha(k)$ es la tasa de aprendizaje en el instante k, N_j es el entorno de vecindad sobre el cual se desea adaptar los pesos, y x_i es la característica i de la entrada o patrón, en dicho instante k.

3.2.2.1 Entorno de vecindad

En relación con el entorno de vecindad, si sólo se quieren adaptar los pesos de la neurona ganadora, se tendría:

$$N_j = \begin{cases} 1 & \text{neurona ganadora} \\ 0 & \text{otro caso} \end{cases}$$

Esta relación topológica del entorno de vecindad de una neurona ganadora tiene mucha importancia en el funcionamiento de la red de tipo SOM. Como se ha comentado anteriormente, se puede definir para cada neurona un conjunto de células o neuronas a las que se trata como vecinas de una dada, de acuerdo a algún tipo de estructura espacial (en el caso de un plano bidimensional, se han indicado anteriormente las principales). Pero estos vecindarios también pueden hacerse no bidimensionales en estructuras tridimensionales o de cualquier otro tipo. Este entorno de vecindad es uno de los parámetros a definir en la red de tipo SOM. Es preciso resaltar que este entorno de vecindad sólo se utiliza en la fase de entrenamiento de la red y con objeto de adaptar los pesos. Una vez esta haya sido entrenada, deja de actuar ya que la red ya ha ajustado estos pesos y proporciona la salida en función de estos pesos ajustados.

Una tercera posibilidad (además de adaptar los pesos únicamente de la neurona ganadora, o bien los de la neurona ganadora y los de su entorno de vecindad), consiste en que esta adaptación de pesos dependa de la distancia de las neuronas vecinas a la neurona ganadora, y en función de estas distancias adaptar estos pesos, con lo que nos encontraríamos con un factor que estimaría esta distancia de vecindad $d(c_i, c_j)$ entre dos neuronas vecinas c_i y c_j dentro del entorno. Teniendo en cuenta esta posibilidad, los pesos de las neuronas se podrían actualizar durante la fase de aprendizaje según la siguiente expresión:

$$w_{ij}(k+1) = w_{ij}(k) + \begin{cases} \frac{\alpha(k)}{d(c_i,c_j)}(x_i(k) - w_{ij}(k)) & \text{para } c_i \text{ neurona ganadora} \\ & \text{y } d(c_i, c_j) < \theta \\ 0 & \text{neuronas fuera} \\ & \text{del entorno de vecindad} \end{cases}$$

siendo θ el umbral o valor que indica la pertenencia al entorno de vecindad de una neurona ganadora. Como se ha indicado previamente, el concepto de vecindad es importante en las redes de tipo SOM. Cuando se han introducido los patrones de forma

sucesiva tras una etapa de aprendizaje, la neurona ganadora y sus vecinas representarán (a través de sus pesos de entrada) datos parecidos, ya que estos pesos han sido desplazados de forma sucesiva por los datos de entrada a medida que estos han sido introducidos a la red en la fase de aprendizaje. De esta forma, tras el proceso de aprendizaje, que es completamente no supervisado, las neuronas dentro de un vecindario ocupan posiciones cercanas dentro del espacio de entrada. Por lo tanto tras la etapa de aprendizaje, se puede observar cómo están representados geométricamente los prototipos (a través de los pesos) que representan a las clases en que queremos clasificar los datos de entrada, para dichos datos de entrada puesto que neuronas cercanas geométricamente tendrán vectores de pesos similares tras este aprendizaje.

3.2.2.2 Tasa de aprendizaje

El otro parámetro interesante a tener en cuenta en las redes de tipo SOM es la tasa de aprendizaje. La tasa de aprendizaje en el esquema original de Kohonen era un valor decreciente con el tiempo. Esto hace que al principio del entrenamiento haya ajustes mayores en los pesos, y estas adaptaciones son menores conforme va pasando el tiempo a medida que se desarrolla el aprendizaje. Los dos esquemas que habitualmente se utilizan para realizar esto son:

- La tasa de aprendizaje se decrementa una cantidad fija y constante por cada ciclo completo de aprendizaje (se entiende por ciclo de aprendizaje cada vez que se le introducen a la red todos los datos de entrada).

  $$\alpha(k+1) = \alpha(k) - \beta$$

  Este parámetro β puede depender del número de iteraciones que se indiquen a la red para el aprendizaje:

  $$\beta = \frac{\alpha(0)}{Iteraciones}$$

- La tasa de aprendizaje se decrementa mediante un proceso logarítmico. De esta forma se decrementa mucho al principio, y se reduce paulatinamente hasta que alcanza valores muy pequeños. Mediante este esquema se da mucha más importancia a la adaptación de los pesos en las primeras iteraciones conformando a partir de esta, la estructura global de la red. Tras esta estructura global, se produce el ajuste fino hasta alcanzar el equilibrio (fin del proceso de aprendizaje).

3.2.3 Aprendizaje de una red SOM

Una vez presentadas las características funcionales de la red tipo SOM, podemos presentar el esquema de aprendizaje, a través de los siguientes pasos:

1. Dado un número de patrones de entrada, se identifica el número de características para cada uno de estos patrones de entrada. Esto configura el tamaño de neuronas de la capa de entrada.

2. Se selecciona el tamaño de neuronas (M) de la segunda capa.

3. Se realiza el aprendizaje: determinación de los vectores de pesos que enlazan las neuronas de la capa de entrada con la capa segunda.

   a) Se inicializan los pesos con valores pequeños aleatorios.

   b) Se presenta de forma consecutiva un patrón de entrada a la red, de entre los existentes en el entrenamiento (etapa k).

   c) Se propaga el patrón de entrada a la segunda capa.

   d) Se selecciona la neurona ganadora en la segunda capa (aquella que presenta una menor distancia con el patrón de entrada, esto es menor distancia entre el patrón de entrada y el vector de pesos asociado de la neurona).

   e) Se actualizan los vectores de pesos (conexiones) entre la capa de entrada y las neuronas dentro de la vecindad de la neurona ganadora. Esto se realiza teniendo en cuenta las expresiones previas así como las diferentes posibilidades.

   f) Una vez introducidos todos los patrones se aumenta el ciclo de aprendizaje. El número de etapas o ciclos de aprendizaje puede ser fijo, o bien, cuando el valor de aprendizaje va disminuyendo en cada etapa hasta que $\alpha(k) < Umbral$.

3.2.4 Ejemplo de uso de una red SOM mediante Matlab

Matlab proporciona un conjunto de comandos que se pueden utilizar para la creación y aprendizaje de redes de tipo SOM. Además de estos comandos y funciones que pueden ser invocados desde la línea de comandos, Matlab también incorpora una herramienta que integra estas aplicaciones. Esta herramienta es *nctool*. La herramienta *nctool* está incorporada dentro de la herramienta global *nnstart*, por lo que puede ejecutarse a partir de ésta o bien de forma completamente independiente.

Como se observa en la figura 3.9, es posible abrir a partir de *nnstart* la herramienta de clusterización (*nctool)*. Ejecutando esta aparece la ventana que se muestra en la figura 3.10.

Lo primero que haremos será cargar los datos con los que entrenaremos la red. Para ello usaremos los datos almacenados en '*SimpleClusters*', accesibles mediante el botón '*Load Example Data Set*'. También podríamos haber utilizado otros datos que tuviéramos cargados previamente en Matlab. Como podemos ver, hemos cargado un conjunto de patrones (en este caso 1000), cada uno de ellos con 2 características (tenemos en definitiva una matriz de 2 filas y 1000 columnas).

A continuación, definimos el número de neuronas de la segunda capa. Esta herramienta permite utilizar capas de dos dimensiones, seleccionando el número de neuronas de lado. (por ejemplo, si se seleccionan 6 neuronas, tendremos una capa bidimensional de 6x6 y así sucesivamente). En el ejemplo que estamos considerando, probaremos con tamaños de capas bidimensionales de 6, 10 y 15 neuronas de lado respectivamente. La topología

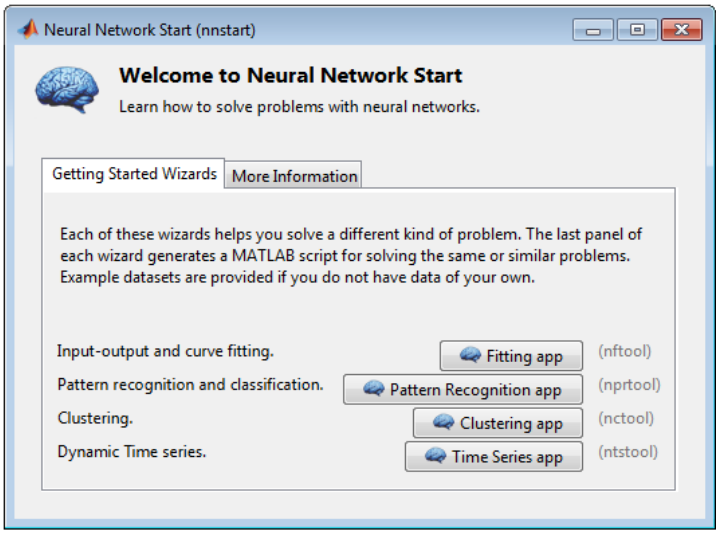

Figura 3.9: Herramienta *nnstart* en Matlab

de la red por defecto es una topología hexagonal (estas topologías pueden modificarse desde línea de comandos).

Una vez definida la arquitectura de la red, tan sólo resta por entrenarla. Para ello pulsamos el botón '*Train*', con lo que se procederá al entrenamiento de la red de acuerdo a los parámetros especificados y los que tenga establecidos la herramienta por defecto. Tras el entrenamiento podremos visualizar los resultados de la red ya entrenada.

Si queremos observar algunos detalles del entrenamiento lo podemos hacer a partir de un conjunto de gráficas:

- *SampleHits*: En este gráfico podremos observar el número de patrones de entrada que han resultado ganadores en cada una de las neuronas de la red. Como se observa en la figura 3.12, la neurona que ha resultado más veces ganadora lo ha hecho en 31 ocasiones.

- *InputPlanes*: En este gráfico podemos observar el valor de los pesos para cada una de las características de los patrones de entrada. En el ejemplo que estamos considerando trabajamos con patrones con dos características (patrones de dos dimensiones), por lo tanto tendremos dos gráficos distintos. En cada uno de ellos (figura 3.13) se representa con un color el valor del peso asociado a cada neurona para dicha característica: valores más oscuros indican pesos mayores. Por ejemplo, si tuvieramos una gradación similar para cada una de las dos características esto podría indicar que estas dos características presentan una fuerte correlación entre sí.

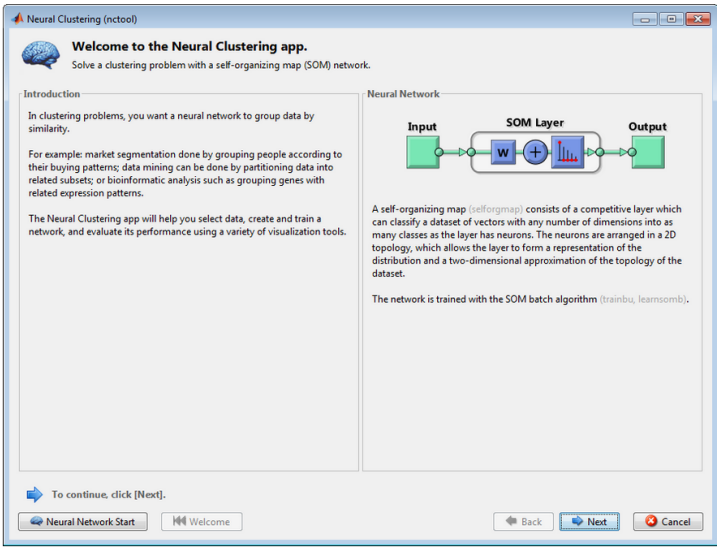

Figura 3.10: Herramienta *nctool* en Matlab

- Distancias entre neuronas: En el gráfico *plotsomnd* (ver figura 3.14) podemos observar la distancia entre los vectores de cada neurona una vez entrenada. En el gráfico observamos en rojo la distancia entre dos neuronas vecinas (representadas cada una de ellas en un hexagono azul), y en color la distancia entre sus vectores de pesos asociados (cuando más oscuro más cerca se encuentran estos vectores de pesos, es decir menor distancia entre ellos). Esto indicaría que tras el proceso de entrenamiento estas dos neuronas representan clases que se encuentran muy cercanas entre sí (vectores de pesos muy similares).
- Posiciones de los pesos. Por último podemos representar las posiciones de los pesos en un espacio de dos dimensiones. En el ejemplo que estamos tratando, el número de características para cada patrón de entrada es de dos unidades, por lo tanto es lo mismo que representar la posición en el plano de donde se encuentran cada una de las neuronas (caracterizada cada una de ellas por su vector de pesos asociado). De esta manera podemos visualizar a la vez los patrones de entrada y la posición de las neuronas tras el entrenamiento como se observa en la figura 3.15.

Si queremos hacer uso de la línea de comandos en Matlab, para diseñar, entrenar y utilizar una red de tipo SOM, esto lo podemos hacer mediante el comando `selforgmap`.

3.3 Implementando una SOM en Python

En esta sección veremos la forma de implementar una red SOM mediante Python. En concreto haremos uso de la librería *NumPy* para implementar directamente este tipo

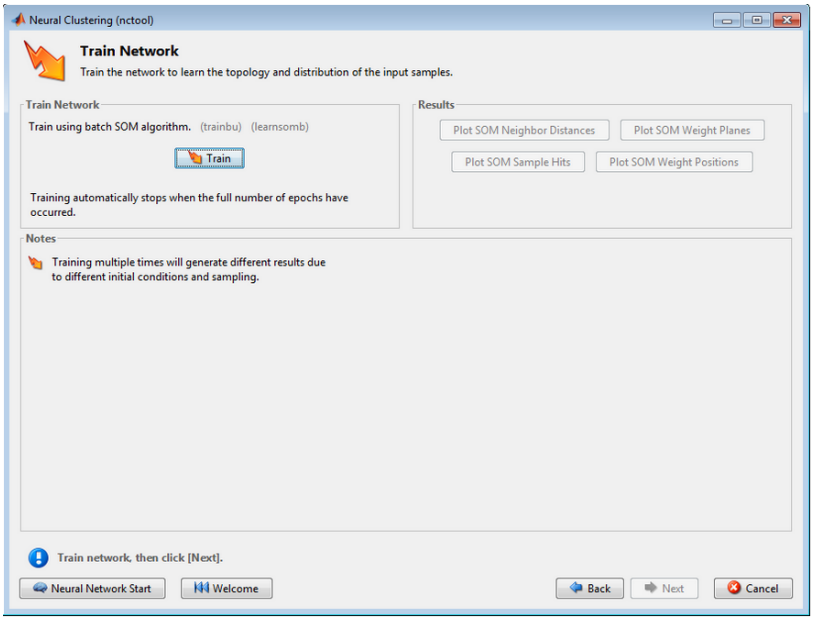

Figura 3.11: Entrenamiento de la red SOM

de redes. Para utilizar una red SOM bidimensional utilizaremos un array 3D. Es esta tercera dimensión del array la que usaremos para almacenar los pesos de cada una de las neuronas (celdas).

Necesitaremos crear tres funciones para el manejo de la red:

1. Una función que nos calcule la celda/neurona más cercana a un dato de entrada. Es decir que nos devuelva qué neurona tiene su vector de pesos asociado más cercano al dato de entrada.
2. Una función para actualizar los pesos en función de la celda/neurona que resulte ganadora.
3. Una función para el entrenamiento de la red. Esta función engloba a las dos anteriores e irá llamando a las previas en el proceso de entrenamiento.

Una posibilidad de tales funciones lo encontramos en el siguiente código:

```
1 # Funcion que devuelve la celda mas cercana a la entrada
2 # Parametros de entrada: array 3D con la red SOM y valor de
    entrada (2D)
3 # Parametros de salida: coordenadas de la celda mas cercana al
    valor x
```

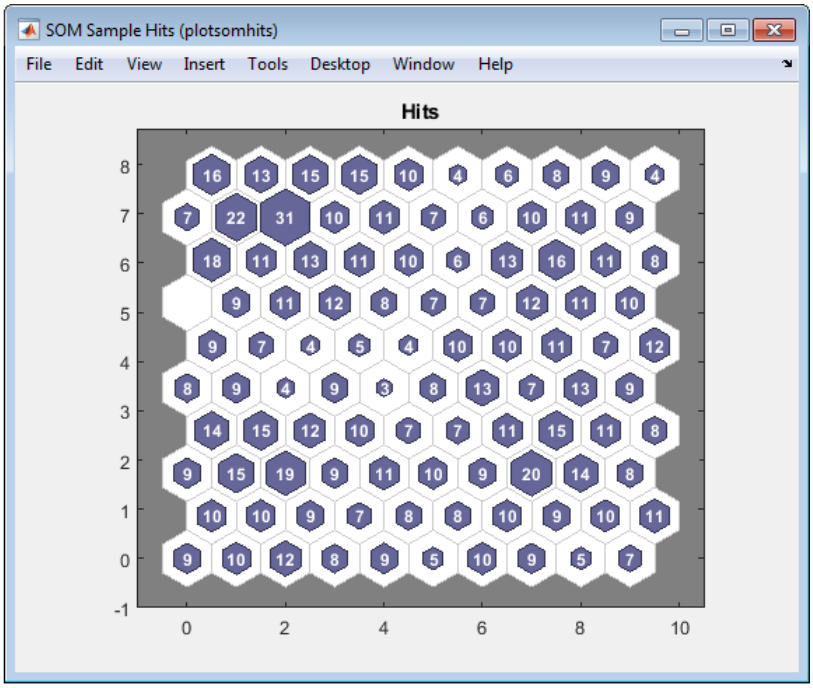

Figura 3.12: Número de patrones ganadores para cada neurona

```
4  def find_BMU(SOM,x):
5      distSq = (np.square(SOM - x)).sum(axis=2)
6      return np.unravel_index(np.argmin(distSq, axis=None), distSq.
           shape)
7
8  # Actualizacion de los pesos de las neuronas de la red SOM a
       partir de
9  # la neurona ganadora
10 def update_weights(SOM, train_ex, learn_rate, radius_sq,
11                     BMU_coord, step=3):
12     g, h = BMU_coord
13     # si el radio es cercano a cero, entonces solo la neurona
           ganadora BMU se
14     # modifica
15     if radius_sq < 1e-3:
16         SOM[g,h,:] += learn_rate * (train_ex - SOM[g,h,:])
17         return SOM
18     # se modifican las neuronas dentro de la vecindad de la
           ganadora
19     for i in range(max(0, g-step), min(SOM.shape[0], g+step)):
```

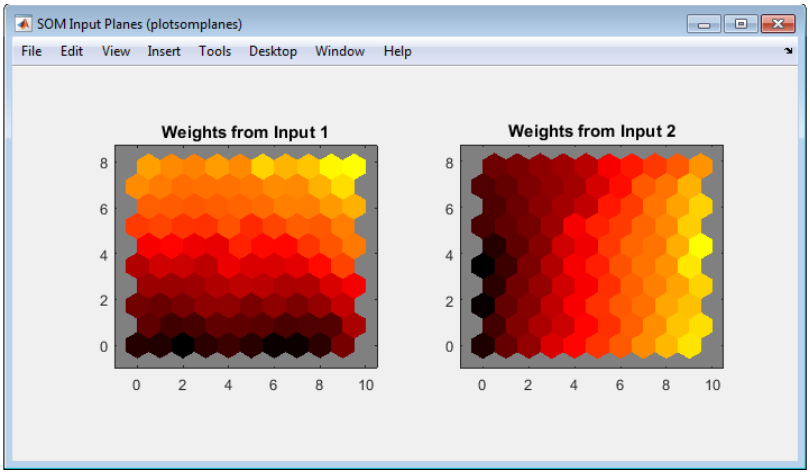

Figura 3.13: Pesos de cada neurona para cada característica de los patrones de entrada

```
20          for j in range(max(0, h-step), min(SOM.shape[1], h+step)):
21              dist_sq = np.square(i - g) + np.square(j - h)
22              dist_func = np.exp(-dist_sq / 2 / radius_sq)
23              SOM[i,j,:] += learn_rate * dist_func * (train_ex - SOM
                    [i,j,:])
24      return SOM
25
26  # Rutina principal para el entrenamiento de la red SOM.
27  def train_SOM(SOM, train_data, learn_rate = .1, radius_sq = 1,
28                lr_decay = .1, radius_decay = .1, epochs = 10):
29      learn_rate_0 = learn_rate
30      radius_0 = radius_sq
31      for epoch in np.arange(0, epochs):
32          rand.shuffle(train_data)
33          for train_ex in train_data:
34              g, h = find_BMU(SOM, train_ex)
35              SOM = update_weights(SOM, train_ex,
36                                   learn_rate, radius_sq, (g,h))
37          # Actualizacion de la tasa de aprendizaje y el radio
38          learn_rate = learn_rate_0 * np.exp(-epoch * lr_decay)
39          radius_sq = radius_0 * np.exp(-epoch * radius_decay)
40      return SOM
```

Veamos, en primer lugar, como configurar una red SOM con los datos que hemos utilizado en el apartado anterior mediante Matlab. Aprovecharemos los datos que hemos utilizado en Matlab en la sección anterior, y los almacenaremos a un fichero en

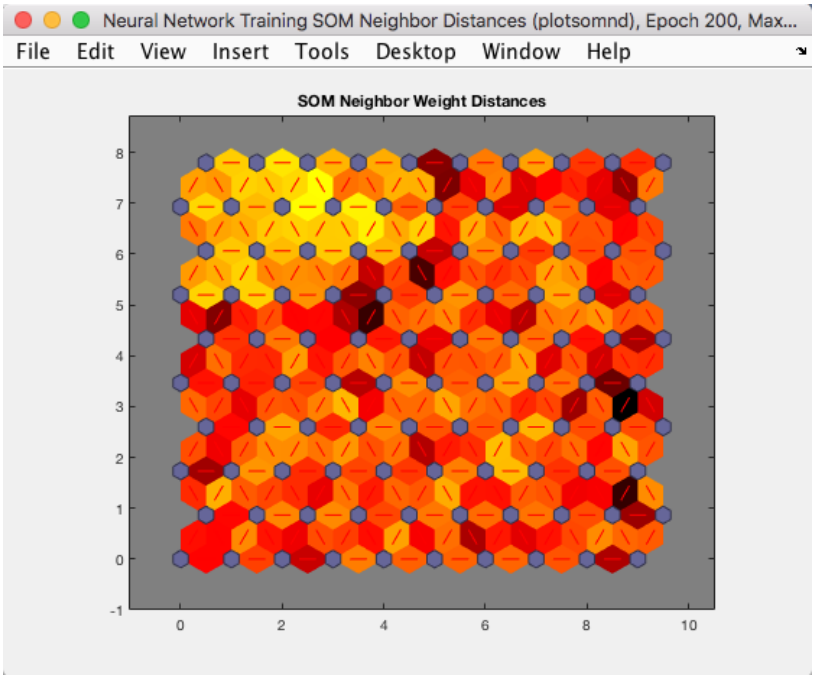

Figura 3.14: Distancia entre los vectores de pesos entre neuronas vecinas

formato csv para su utilización desde el programa que realizaremos. De esta manera podemos leer desde Python estos datos que son los que usaremos para el entrenamiento:

```
import numpy as np
import matplotlib.pyplot as plt

# Leemos los datos de entrenamiento
with open('data.csv', 'r') as f:
    data = np.loadtxt(f, delimiter=",")

train_data=np.transpose(data)
```

Así, ya tendremos los datos de entrenamiento, almacenados en la variable *train_data*. Si queremos construir una red SOM podemos hacerlo directamente mediante la creación de un array de valores, y almacenando en cada una de estas celdas o neuronas de la red, los pesos de forma aleatoria. En el siguiente código podemos encontrar el resultado de la programación con suficientes comentarios.

```
###### CREAMOS LA MISMA RED QUE EN MATLAB POR COMPARAR
m=8
n=8
```

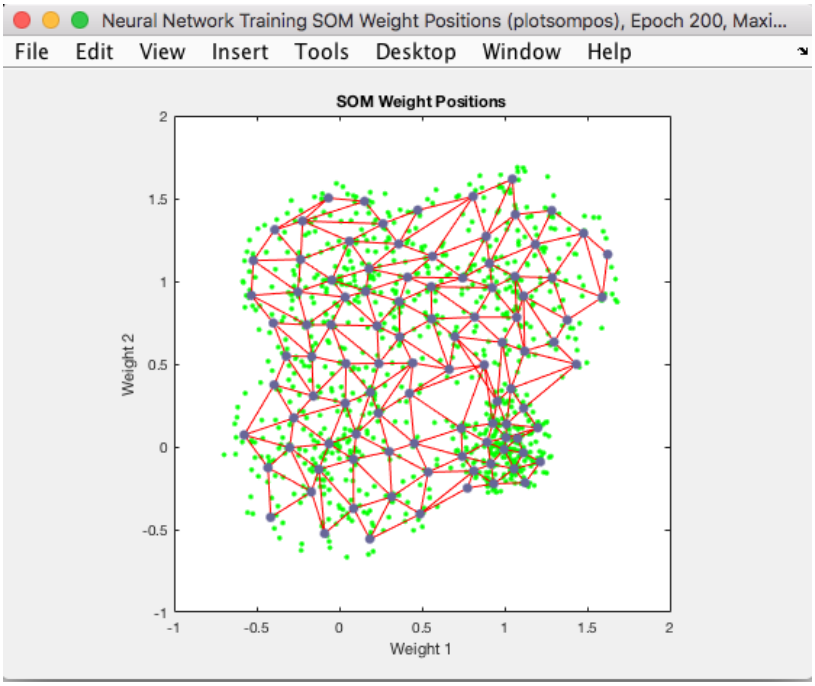

Figura 3.15: Posiciones de los vectores de pesos para cada una de las neuronas

```
4
5  # Construimos la red SOM: mxnx2 con valores tipo float aleatorios
6  rand = np.random.RandomState(0)
7  red_som = np.random.random_sample((m,n,2))
8
9  # print(red_som)
10
11 # Construida la red SOM junto con los datos de entrenamiento,
12 # se realiza el entrenamiento de la red, con objeto de que las
       celdas se agrupen
13 red_som_entrenada = train_SOM(red_som, train_data, epochs=10)
14
15 # Dibujamos la red una vez esta se encuentra entrenada.
       Representamos los pesos
16 plt.scatter(red_som[:,:,0],red_som[:,:,1])
17 plt.show()
```

Tras el entrenamiento podemos visualizar cómo se han agrupado las neuronas en función de los datos que hemos utilizado para el entrenamiento. Esta agrupación de las neuronas la visualizamos en la figura 3.16, donde se puede observar los diferentes con-

juntos que aparecen con los datos de entrada utilizados. La posición de las neuronas se ha representado en circulos azules, mientras que la posición de los patrones o datos de entrada se encuentra representada por puntos de color naranja.

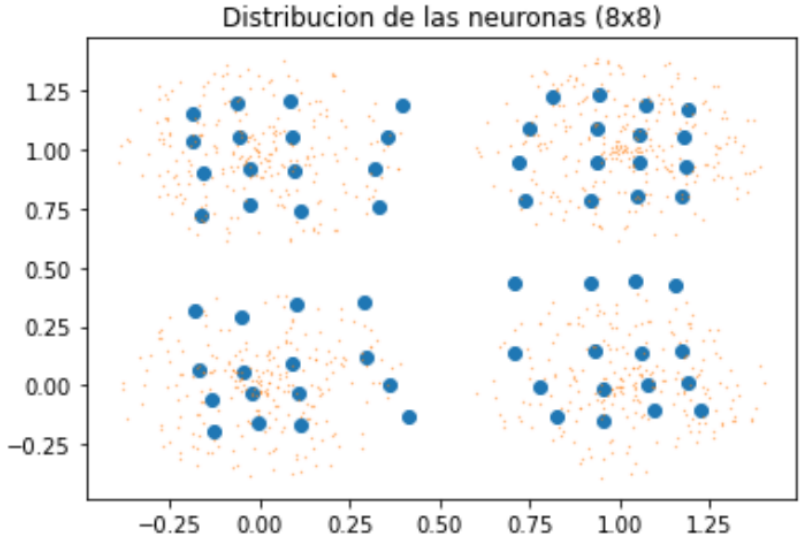

Figura 3.16: Distribución de las neuronas en la red SOM tras el entrenamiento. Con puntos se han representado los datos de entrada

3.3.1 Ejemplo de entrenamiento y adaptación de los pesos de las neuronas

Vamos a exponer otro ejemplo de cómo la red actualiza los pesos de cada una de las celdas o neuronas, de forma que observemos cómo valores de entrada similares tienden a agruparse en celdas o neuronas próximas entre sí. Usaremos las mismas funciones para el entrenamiento (selección de la celda ganadora, y actualización de los pesos) que se han utilizado previamente.

Para ello utilizaremos como datos de entrada colores. Vamos a crear de forma aleatoria un total de 3000 colores diferentes. En realidad cada dato o patrón de entrada será un vector de 3 componentes (color RGB):

```
1 # Creamos los DATOS DE ENTRENAMIENTO: 3000 valores aleatorios
2 n_x = 3000
3 rand = np.random.RandomState(0)
4 # Inicializamos los datos de entrenamiento: cada dato es un color
      RGB
5 train_data = rand.randint(0, 255, (n_x, 3))
```

A continuación construimos una red de tipo SOM de 10x10 celdas.

```
# COMPROBACION DE LA RED SOM
# Creamos una red SOM de 10x10 celdas
m = 10
n = 10
# Construimos la red SOM: mxnx3 con valores RGB tipo float aleatorios
red_som = rand.randint(0, 255, (m, n, 3)).astype(float)
```

Podemos representar tanto los datos de entrenamiento (3000 valores, cada uno de ellos es un color tipo RGB), junto con los pesos iniciales de cada una de las celdas que componen la red (10x10) en dos gráficos diferentes, obteniendo el resultado representado en la figura 3.17.

```
# Representamos tanto los datos de entrenamiento (array de 3000 valores)
# como los pesos iniciales de cada una de las celdas de la red (mxn)
fig, ax = plt.subplots(
    nrows=1, ncols=2, figsize=(12, 3.5),
    subplot_kw=dict(xticks=[], yticks=[]))
ax[0].imshow(train_data.reshape(50, 60, 3))
ax[0].title.set_text('Datos de entrenamiento')
ax[1].imshow(red_som.astype(int))
ax[1].title.set_text('Pesos iniciales en la red SOM')
```

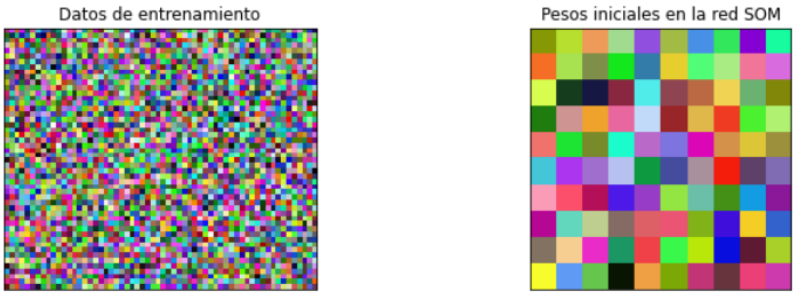

Figura 3.17: Datos (colores) usados como entrenamiento, y pesos iniciales (colores) de cada una de las celdas de la red

Por último podemos realizar el entrenamiento de la red, parando cada cierto número de iteraciones para poder observar, en función del número de iteraciones que hayamos realizado en el entrenamiento, cómo se van aproximando los vectores de pesos de las neuronas o celdas. En las gráficas podemos observar cómo se van aproximando y reuniendo los colores a media que avanza el entrenamiento.

```
# Construida la red SOM junto con los datos de entrenamiento,
# se realiza el entrenamiento de la red, con objeto de que las
    celdas se agrupen
# Se representa el resultado con 1, 5, 10 y 20 epochs para
    comprobar la evolucion de los
# pesos de cada celda en funcion de las iteraciones del
    entrenamiento
fig, ax = plt.subplots(
    nrows=1, ncols=4, figsize=(15, 3.5),
    subplot_kw=dict(xticks=[], yticks=[]))
total_epochs = 0
for epochs, i in zip([1, 4, 5, 10], range(0,4)):
    total_epochs += epochs
    red_som = train_SOM(red_som, train_data, epochs=epochs)
    ax[i].imshow(red_som.astype(int))
    ax[i].title.set_text('Epochs = ' + str(total_epochs))
```

En la figura 3.18 se observan los pesos de cada una de las neuronas/celdas de la red SOM (colores) tras 1 iteración, 5 iteraciones, 10 iteraciones y 20 iteraciones. Como se observa los colores se van aproximando y agrupando a medida que se produce el entrenamiento de la red, de forma que colores semejantes tienden a posicionarse unos al lado de otros a medida que avanza este entrenamiento.

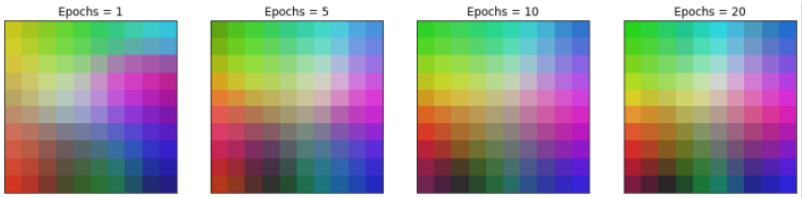

Figura 3.18: Evolución de los pesos de las neuronas a medida que se van produciendo iteraciones en el entrenamiento de la red

3.4 Usando un paquete para SOM

Podemos encontrar múltiples paquetes para realizar el procesamiento (entrenamiento y visualización) de redes tipo SOM mediante Python. Uno de los más sencillos que podemos encontrar es el paquete *MiniSom*. La documentación sobre su uso así como algunos ejemplos interesantes los podemos encontrar en https://github.com/JustGlowing/minisom.

Lo primero que tendremos que hacer es instalar dicho paquete dentro del entorno de Python que estemos manejando. Si estamos utilizando el navegador Anaconda, no será posible instalar este paquete a través de la interfaz como hemos realizado en otras

ocasiones, y lo tendremos que hacer a través de la línea de comandos. Para ello lo primero que haremos será abrir una Terminal dentro del entorno en el que queramos instalar este paquete. Sobre esta ventana terminal, teclearemos:

```
pip install minisom
```

Con esto tendremos instalado el paquete *MiniSom* en nuestro entorno Python y podremos hacer uso de sus funcionalidades.

Veamos con un ejemplo concreto cómo podemos utilizar este paquete. Lo utilizaremos para clasificar los datos ya conocidos (*iris_dataset*) que hemos utilizado en algunas ocasiones previas en Matlab. En primer lugar, procederemos a la carga de los datos. Recordemos que estos datos se corresponden con un total de 150 valores (patrones) correspondientes a 3 tipos de flores diferentes. Una vez que carguemos los datos, separaremos las clases (3 tipos de flores) de los valores que son los que usaremos para el entrenamiento de la red SOM.

```
# Importamos los datos de iris_dataset
from minisom import MiniSom
import numpy as np
import pandas as pd

# Leemos los datos y los dividimos en la entrada (data) y salida (target)
columns = ['longitudSepalo', 'anchuraSepalo', 'longitudPetalo', 'anchuraPetalo',
           'clase']
data = pd.read_csv('https://archive.ics.uci.edu/ml/machine-learning-databases/iris/iris.data',
                   sep=',', names=columns, header=None, engine='python')
target = data['clase'].values
#Convertimos las etiquetas a numeros
targetValues=pd.factorize(target)[0]
data = data[data.columns[:-1]]

# Normalizamos los datos
data = (data - np.mean(data, axis=0)) / np.std(data, axis=0)
data = data.values
```

Para crear una red de tipo SOM, debemos seleccionar el número de neuronas (x, y) en función de la estructura que queramos tener. A partir de esto, el método MiniSom, nos permite establecer diferentes parámetros como pudieran ser entre otros:

- Distancia de activación
- Función de vecindad

- Tasa de aprendizaje
- Topología de la red

También podemos indicarle la forma en que deseamos inicializar los pesos de cada una de las neuronas. Una razonable alternativa es inicializarlos mediante las direcciones principales (PCA - Principal Component Analysis) de estos datos de entrada. Esto lo podemos hacer a través del método *pca_weights_init*. O bien de forma totalmente aleatoria. En cualquier caso, tras inicializar los pesos de las neuronas, procedemos al entrenamiento, indicando el número de iteraciones.

```
# Creamos la red SOM y entrenamos
n = 8
m = 8
som = MiniSom(n, m, data.shape[1], sigma=1.5, learning_rate =0.5,
              neighborhood_function='gaussian',
              activation_distance='euclidean', random_seed=0)
som.pca_weights_init(data)
som.train(data,1000,verbose=True)
```

Una vez que la red está entrenada, podemos observar algunos resultados interesantes sobre esta. Vamos a representar en primer lugar la distancia entre los pesos finales (una vez entrenada la red) de cada una de las neuronas. Esto lo podemos hacer ya que el método *distance_map()* nos proporciona este mapa de distancias (euclideas) entre los pesos de las neuronas. Recordemos, que para el caso que estamos considerando, los pesos de cada neurona es un vector de cuatro componentes puesto que cada uno de los datos de entrada presenta 4 características o valores.

```
# Visualizamos algunos resultados
import matplotlib.pyplot as plt

# En primer lugar representamos la distancia euclidea entre los
    pesos con las
# celdas vecinas
plt.figure(figsize=(9,9))
plt.pcolor(som.distance_map().T, cmap='Oranges')
plt.colorbar()
plt.title('Distancia de los pesos entre neuronas')
```

Con el código previo, veríamos la figura 3.19.

Dado que conocemos la clasificación correcta para los datos de entrada (que recordemos no hemos empleado en el entrenamiento de la red, ya que este aprendizaje se ha efectuado de forma no supervisada), podemos querer visualizar cuales son las neuronas ganadoras para cada uno de los datos de entrada y representar sobre la neurona

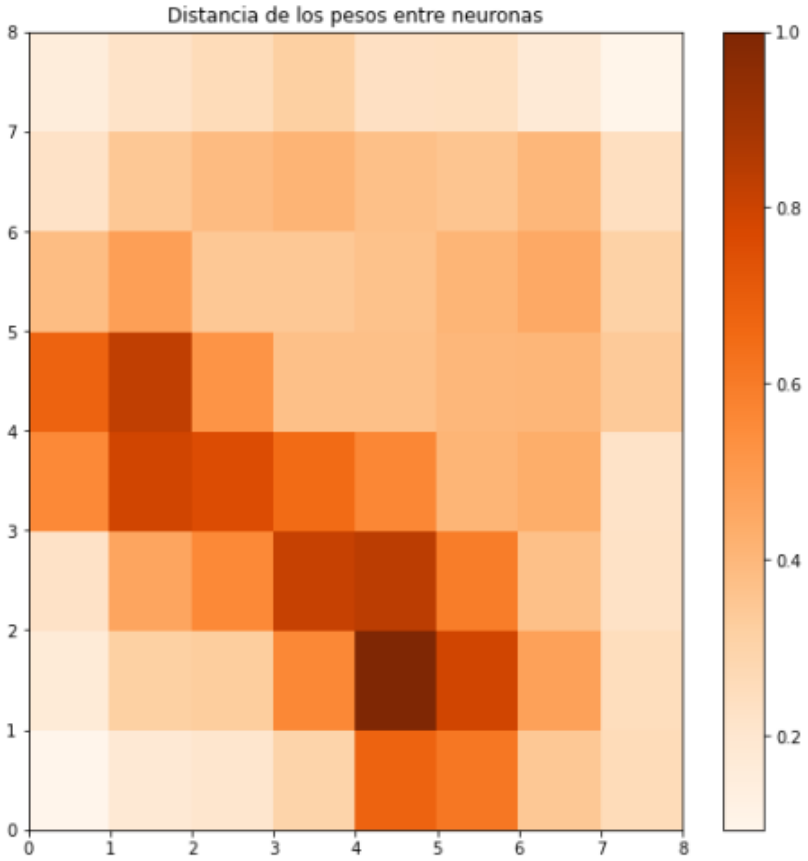

Figura 3.19: Distancia euclidea entre los pesos de las neuronas vecinas una vez entrenada la red SOM

ganadora la clase de dicho patrón o dato que nos gustaría haber obtenido. Esto lo podemos hacer en este caso dado que a pesar de que hemos efectuado un entrenamiento no supervisado, sí que conocemos las clases de cada dato de entrada. La neurona ganadora tras el entrenamiento ante un dato de entrada lo podemos obtener mediante el método *winner*. Esto lo podemos hacer con el siguiente código, obteniendo la figura 3.20.

```
# # En segundo lugar representamos la respuesta de la red a cada
    patron de entrada
markers = ['o', 's', 'D']
colors = ['C0', 'C1', 'C2']
for cnt, xx in enumerate(data):
    win = som.winner(xx) # obtenemos la neurona ganadora
    plt.plot(win[0]+0.5, win[1]+0.5, markers[targetValues[cnt]],
```

```
7                 markerfacecolor='None',
8                 markeredgecolor=colors[targetValues[cnt]],
                      markersize=12,
9                 markeredgewidth=2)
10
11 plt.show()
```

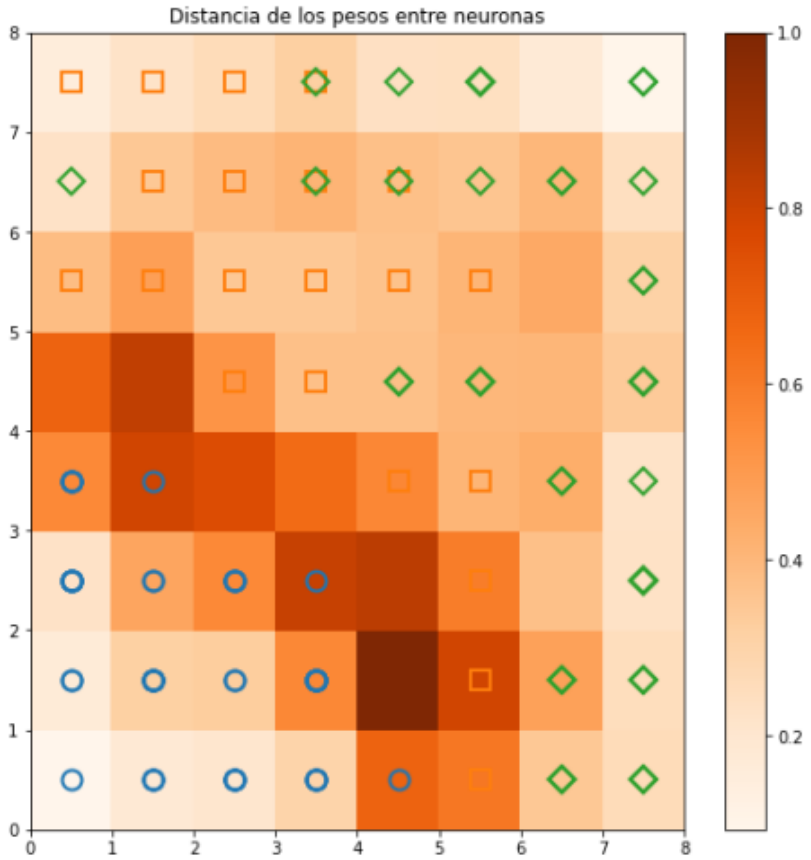

Figura 3.20: Representación de las clases ganadoras en cada neurona con los datos de entrada

En la figura 3.20, se ha representado la clase ganadora para cada neurona. Se representa cada clase con un símbolo diferente (circulo azul, rombo verde, cuadrado naranja). Por ejemplo, en la neurona que se encuentra en la primera fila y cuarta columna, observamos que esa neurona ha resultado ganadora con dos clases diferentes. Si queremos observar en una misma figura qué neuronas son ganadoras (tras el entrenamiento) para todos los patrones de entrada podemos utilizar el código siguiente. En el mismo se añade un offset aleatorio con objeto de poder pintar varios puntos sobre una misma neurona en caso de que esta resulte ganadora en varias ocasiones. Como en este caso conocemos

la clase de salida, representamos además en diferente color estas clases. Obtendríamos la información representada en la figura 3.21. En esta figura cada uno de los 150 datos de entrada, lo hemos representado en la neurona ganadora y le hemos asignado un color diferente en función de la clase a la que hemos asignado tras el entrenamiento (azul para la clase 'iris-setosa', naranja para la clase 'iris-versicolor', y verde para la clase 'iris-virginica').

```
# Ahora dibujamos las neuronas que han sido ganadoras para cada entrada
# Se incorpora un offset aleatorio para evitar solapamientos al pintarlas con objeto
# de que podamos ver las veces que ha sido ganadora cada celda
# Creamos los valores (x,y) de las neuronas ganadoras para cada dato de entrada
w_x, w_y = zip(*[som.winner(d) for d in data])
w_x = np.array(w_x)
w_y = np.array(w_y)

plt.figure(figsize=(10, 9))
plt.pcolor(som.distance_map().T, cmap='bone_r', alpha=.2)
plt.colorbar()


for c in np.unique(targetValues):
    idx_target = targetValues==c
    plt.scatter(w_x[idx_target]+.5+(np.random.rand(np.sum(idx_target))-.5)*.8,
                w_y[idx_target]+.5+(np.random.rand(np.sum(idx_target))-.5)*.8,
                s=50, c=colors[c], label=np.unique(target)[c])
plt.legend(loc='upper right')
plt.grid()
plt.title('Neuronas ganadoras y clase de salida')
plt.show()
```

Por último de una forma sencilla podemos ver únicamente el número de veces que cada neurona ha sido ganadora una vez que la red ha sido entrenada. Esto lo podemos realizar con el código siguiente, obteniendo la figura 3.22.

```
# Por ultimo podemos ver cuales son las neuronas mas ganadoras (activadas) ante
# los datos presentados
plt.figure(figsize=(8,8))
frequencies = som.activation_response(data)
plt.pcolor(frequencies.T, cmap='Blues')
plt.colorbar()
```

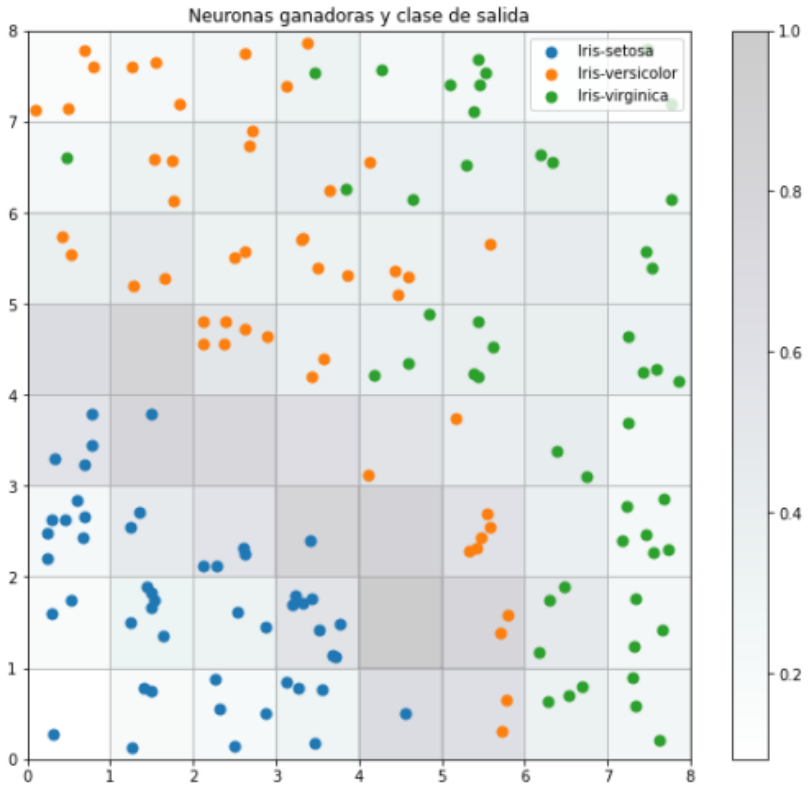

Figura 3.21: Representación de las neuronas ganadoras junto con la clase correcta

```
plt.title('Neuronas mas activadas')
plt.show()
```

3.5 Algoritmo LVQ

El algoritmo LVQ (Linear Vector Quantization) es un procedimiento para entrenamiento de capas de neuronas competitivas de forma supervisada. Una red LVQ tiene una primera capa competitiva y una segunda capa lineal. La capa competitiva tiene por cometido clasificar los patrones de entrada en una forma similar a como se ha comentado en apartados previos, mientras que la capa final lineal transforma las clases de la capa competitiva en clasificaciones definidas por el usuario, habitualmente un número más reducido que el utilizado en la capa competitiva. De esta forma podemos hablar de subclases (como las clases de la capa competitiva) y clases objetivo que son las clases finales. Este algoritmo se utiliza cuando conocemos las clases a las que queremos asociar cada una de los datos de entrada. Podemos resolver este problema mediante una red unidireccional como en capítulos previos, o bien, crear esta clasificación mediante

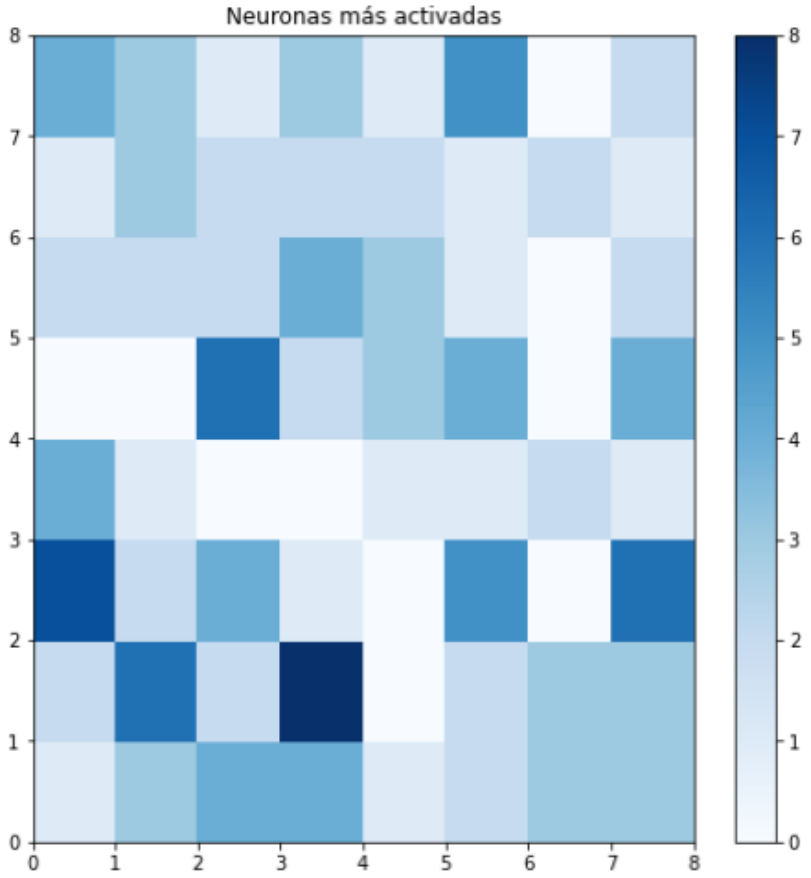

Figura 3.22: Número de veces que cada neurona del mapa ha resultado ganadora

la unión de una capa competitiva y una capa lineal (unidireccional). En este caso se utiliza un entrenamiento de tipo supervisado.

Tanto la capa competitiva como la capa de salida lineal presentan una neurona por cada una de las clases.

En Matlab es posible crear una red de tipo LVQ mediante el comando *lvqnet*. La sintaxis de este comando es *lvqnet(hiddenSize,lvqLR,lvqLF)*, siendo:

hiddenSize : Número de neuronas en la primera o capa competitiva

lvqLR : Razón de aprendizaje LVQ. Por defecto 0.01

lvqLF : Función de aprendizaje. Por defecto *learnlv1*

3.5.1 Entrenamiento de la red

Para entender el funcionamiento de la red LVQ, supongamos que tenemos un patrón de entrada de tres componentes, $p1 = [2; -1; 0]$, y la clasificación de este patrón es a la tercera de un total de cuatro clases posibles $t1 = [0; 0; 1; 0]$. Cuando se presenta este patrón de entrada a la red, las neuronas de la primera capa compiten para detectar cual de estas neuronas presenta unos pesos a su entrada con una distancia menor al patrón de entrada (lo más parecida posible). Esta neurona que resulta ganadora en la capa competitiva ofrece un 1 a su salida, mientras que el resto de neuronas ofrecen un 0 a la salida. Cuando la salida de las neuronas de la primera capa se pasan a la segunda capa, el único 1 en la neurona previa selecciona la clase asociada con la entrada. De esta forma la red asociaría el patrón de entrada $p1$ con la salida $t1$ (obviamente esta asignación podría ser correcta o no dependiendo del valor asignado a la salida).

El proceso de aprendizaje consiste en desplazar los pesos de la neurona que ha resultado ganadora en la capa competitiva más cerca del patrón de entrada si la asignación final de salida es correcta y mover estos en dirección contraria si la asignación final es incorrecta. Estos desplazamientos y correcciones de los pesos de la neurona ganadora pueden realizarse mediante el algoritmo de backpropagation. Este algoritmo se encuentra implementado en Matlab bajo el método *learnlv1*.

Matlab proporciona otra regla de aprendizaje *learnlv2* que se puede utilizar siempre después de ajustar los parámetros mediante el procedimiento *learnlv1*. La principal diferencia de este algoritmo consiste en que ahora se usan los pesos de las dos neuronas que se encuentren más cerca de la neurona ganadora de forma tal que el más próximo al patrón de entrada se aproxima aún más a este, mientras que el segundo más próximo se aleja de este. Esto conviene solo hacerlo tras un entrenamiento de la red con el procedimiento inicial *learnlv1*.

3.5.2 Diseño de una red LVQ mediante Matlab

Como se ha comentado anteriormente, Matlab proporciona un conjunto de funciones para la utilización de las redes LVQ. Veamos un ejemplo de uso para la clasificación de un conjunto de patrones como los utilizados previamente. En primer lugar cargaremos los patrones a clasificar (en la variable x), junto con la clasificación, en este caso conocida, para dichos patrones (t).

```
1 [x,t] = iris\_dataset;
```

Estos patrones se corresponden con los mismos datos que se utilizaron en secciones previas. Aquí, además, se hace uso de una clasificación conocida de los mismos. Como se puede observar se dispone de un total de 150 patrones, cada uno de ellos con 4 características (matriz de $4x150$ valores), y se dispone de una clasificación de cada uno de estos patrones en 3 clases diferentes (t es una matriz de $3x150$).

Para la creación de la red de tipo LVQ, es preciso indicar el número de neuronas de la capa competitiva. Elegiremos 10 como neuronas de esta capa.

```
net=lvqnet(10);}
```

Con el comando anterior se crea una red de tipo LVQ con 10 neuronas en la capa competitiva. Como se observa en la figura 3.23, dado que aún no le hemos indicado el número de clases (a través de la clasificación de los patrones), el número de neuronas en la capa de salida lineal es 0.

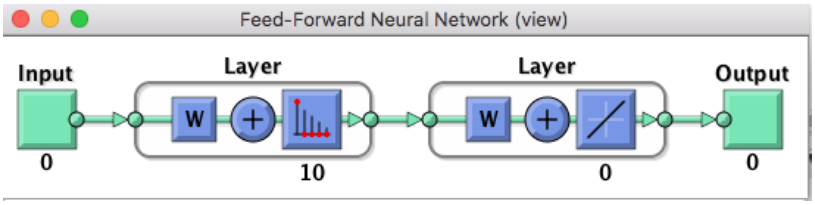

Figura 3.23: Red LVQ creada sin indicar la clasificación de los patrones de entrada.

A continuación seleccionamos el número de iteraciones de entrenamiento y procedemos a entrenar la red creada:

```
net.trainParam.epochs = 50;
net = train(net,x,t);
view(net)
```

Como se observa en la figura 3.24, la red creada ya presenta a su salida 3 neuronas, ya que hay 3 clases distintas en las que se clasifican los patrones de entrenamiento. Esto lo ha conocido la red cuando hemos empleado el método *'train'* y le hemos indicado las clases de salida.

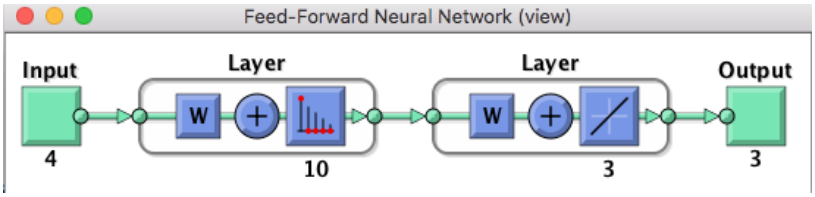

Figura 3.24: Red LVQ creada tras el enternamiento.

Una vez entrenada la red, podemos calcular la salida de esta ante un conjunto de patrones de entrada. Lo razonable hubiera sido quedarnos con un conjunto de patrones de entrada que no hubiéramos utilizado para el entrenamiento. Esto no lo hemos hecho dado que disponemos de pocos datos (únicamente 150) y hemos preferido utilizar todos los datos para el entrenamiento. No obstante, si le pasamos a la red de nuevo estos patrones (que hemos usado previamente en el entrenamiento), la salida de la red sería la proporcionada a través del comando *net*.

```
y=net(x);
classes=vect2ind(y);
```

Si comparamos la salida de la red una vez entrenada (que hemos almacenado en la variable y), con la salida que debería dar (variable t), podemos observar los errores de clasificación utilizando los mismos datos tanto para entrenamiento como para test.

```
classesT=vect2ind(t);
plot(classes, 'r+');
plot(classesT, 'bo');
```

Como se observa en la figura 3.25, podemos encontrar los patrones en los que el resultado del entrenamiento es erróneo.

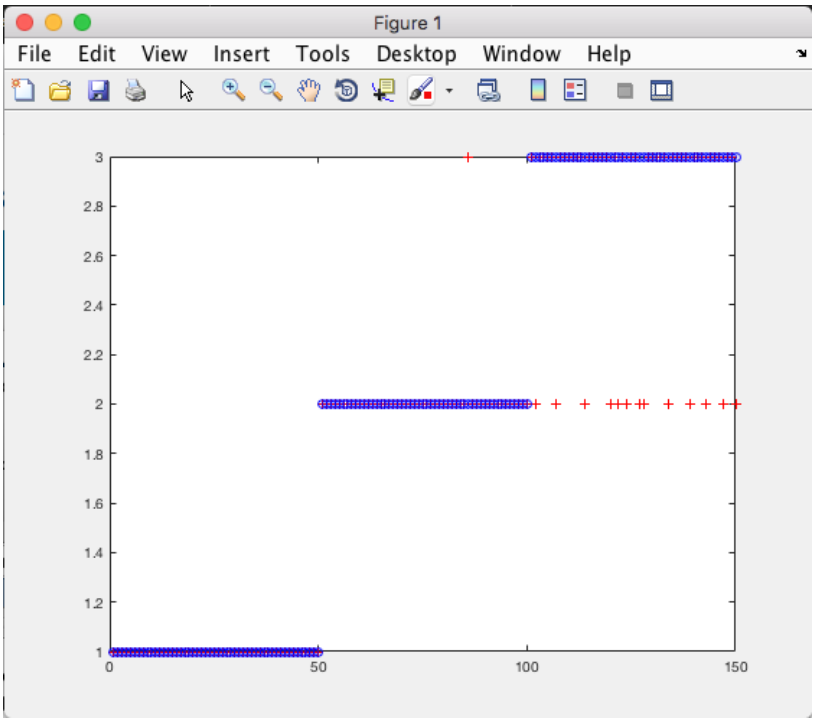

Figura 3.25: Clases reales para cada uno de los 150 patrones de entrada (rojo), y clases que proporciona la red entrenada (azul).