



Tutorial para el Diseño de una Red Neuronal con JRedesNeuronales

Índice:

- 8.1 .. *Introducción*
- 8.2 .. *Diseño de la arquitectura.*
- 8.3 .. *Diseño del Comportamiento*
 - 8.3.1 .. *Diseño de la Dinámica de la red*
 - *diseño de ejecutar en Red*
 - *diseño de actualizar en Neurona*
 - 8.3.2 .. *Diseño del Aprendizaje de la red*
 - *diseño de aprender en Red*
 - *diseño de aprender en sinapsis*
 - 8.3.3 .. *Las redes online.*
- 8.4 .. *Diseño de los patrones*
- 8.5 .. *Empaquetado de la red.*

Proyecto fin de carrera:

Desarrollo de un Marco de Trabajo para el diseño de Redes Neuronales

Autor: Alfonso Ballesteros González

Director: Enrique Domínguez Merino

Ingeniería Informática - Universidad de Málaga

8.1 Introducción

Este tutorial es una pequeña ayuda para el ayudar al diseñador a comenzar a desarrollar redes neuronales con este marco de trabajo.

En él, explicare paso a paso, como vamos a crear una red, modelando sus tres aspectos: arquitectura, dinámica y aprendizaje, además del diseño de los patrones que admitirá la red.

Cada aspecto es modelado especificando una serie de elementos generales que intervienen en el funcionamiento del marco de trabajo.

El diseño de una red se estructura en cuatro pasos que se explicaran más detalladamente en los siguientes apartados.

8.2 Diseño de la arquitectura.

8.3 Diseño del Comportamiento.

8.3.1 -->Dinámica

8.3.2 -->Aprendizaje

8.4 Diseño de los Patrones.

Para ilustrar este tutorial vamos a desarrollar una red de ejemplo, un perceptrón, Una explicación a grandes rasgos del proceso de diseño será el siguiente. Primero, crearemos tres clases Java a partir de las plantillas, la clase para especificar la red heredara de la clase Red y en el deberemos redefinir el constructor y los métodos ejecutar y aprender.

La plantilla de la red debe quedar algo así:

```
public class RedEjemplo_Red extends Red {  
    public RedEjemplo_Red()      { /* Arquitectura */ }  
    public ejecutar(Patrón P)    { /* Dinámica a nivel de red */ }  
    public Aprender(ListaPatrones Ps) { /* Aprendizaje a nivel de red */ }  
}
```

La clase que represente la neurona, heredará de la clase Neurona y en ella deberemos redefinir el método actualizar.

La plantilla de la clase neurona debe quedar algo así:

```
public class Neurona_RedEjemplo extends Neurona {  
    public actualizar() { /* dinámica a nivel de neurona */ }  
}
```

Finalmente para la clase que represente una sinapsis, heredara de la clase Sinapsis y en ella deberemos redefinir el método aprender.

```
public class Sinapsis_RedEjemplo extends Sinapsis {  
    public aprender(double[] parámetros){  
        /* Aprendizaje a nivel de sinapsis */  
    }  
}
```

La arquitectura se definirá en el constructor de la red, la dinámica de la red en el método ejecutar de la red y en la actualizar de la neurona y el aprendizaje en el método aprender de la red y de la sinapsis.

En resumen, en la clase hija de la red tenemos que definir los métodos:

```
public Perceptron_red()
```

Que define la arquitectura de la red

```
public Patrón ejecutar(Patrón p)
```

Que se encarga de definir la dinámica de la red. Toma como parámetro un patrón de entrada y devuelve otro patrón procesado.

```
public void aprender(ListaPatrones ps)
```

Se encarga de definir como aprende la red la red lista de patrones Ps

Además debemos definir en las clases correspondientes como se actualizan las sinapsis y las neuronas y eso es todo lo que hay que hacer para diseñar la red.

8.2 Diseño de la arquitectura.

El diseño de la arquitectura de la capa se hace implementado la función constructora de la red_hija en las cuales podemos poner el número de parámetros que deseemos como en los siguientes ejemplos:

```
public Perceptron_Red()
public Perceptron_Red(int nneuronas1 ,int nneuronas2 )
```

Hay dos modos de realizar estas funciones dependiendo de los métodos que usemos:

1) Modo Rápido:

La idea es diseñar cada capa con las funciones, digamos macroscópicas, esto es , aquellas que manejan a todas las neuronas y sinapsis de las capas. Por ejemplo, llamando a una función, crear la capa y todas sus neuronas y con otra unimos las capas con un determinado tipo de unión ya sea lineal (una neurona de cada capa con otra de cada capa) o total (esto es, Cada neurona esta unida a todas las Neuronas de las otras capas mediante sinapsis).

Esto permite una mayor rapidez en el diseño ya que con pocas líneas realizamos todo el diseño de la arquitectura de la red.

Tenemos dos opciones la primera es usar la funciones de la red

```
public void crearCapaEntrada(int NNeuronas, String Tipo_Neuronas)
public void crearCapaSalida(int NNeuronas, String tipoNeurona,int NMaxCapas)
public void crearCapaOculta(int NNeuronas, String tipoNeurona)
```

NNeuronas es el número de neuronas que tendrá la capa.
NMaxCapas es el número máximo de capas.
tipo_Neuronas es el tipo de la clase que representa a las neuronas de nuestra red.

La segunda opción es usar los constructores de la Capa y aplicarlos sobre las relaciones de la red CapaEntrada y CapaSalida.

```
public Capa(int Numero_Neuronas, int orden_de_la_Capa, String tipo_Neuronas)
donde
```

Numero_Neuronas es el número de neuronas que tendrá la capa.
orden_de_la_Capa es el orden de la capa que representa su ordenación frente a las otras capas CapaEntrada tiene orden 0 y CapaSalida tiene el mayor orden.
tipo_Neuronas es el tipo de la clase que representa a las neuronas de nuestra red

Para unir capas usaremos este método de la red.

```
public int unirCapas(Capa Capa1, Capa Capa2, int tipo_union, String ClaseSinapsis)
    donde
```

Capa1, Capa2 Se unen estas dos capas.

tipo_union Es el tipo de unión que tendrán las neuronas: Lineal o Total.

tipo_Neuronas es el tipo de la clase que representa a las Sinapsis con que se unirán las neuronas de cada Capa de nuestra red.

Un Ejemplo para crear un perceptrón podría ser el siguiente código:

```
public Perceptron_Red(int nneuronas1 ,int nneuronas2 ) {
    //Creo La capa de entrada y salida
    CapaEntrada = new Capa(nneuronas1,0,"Neurona_Bipolar");
    CapaSalida = new Capa(nneuronas2,2,"Neurona_Bipolar");

    //Uno La capa de entrada y salida de forma total con
    //sinapsis_perceptron que la definiremos después
    unirCapas(CapaEntrada,CapaSalida,Total,"Sinapsis_Perceptron");
}
```

Este modo de diseñar redes tiene la ventaja de que es muy rápido. En apenas tres líneas de código, ya hemos diseñado la red, pero tiene la desventaja de que tiene restricciones:

- Las Neuronas de una capa tienen que ser todas iguales (de la misma clase).
- Las Sinapsis que unen una capa con otra capa tienen que ser todas iguales.
- Solo podemos unir las Capas de dos maneras Linealmente o Totalmente no hay otras opciones.

Con los métodos anteriores podemos tener casi todas las redes cubiertas pero existen algunas redes que necesitan más flexibilidad, por ejemplo aquellas cuya arquitectura se genera dinámicamente como las redes ART.

Para modelar este tipo de redes se le añaden al modelo una serie de métodos que pretenden añadir la flexibilidad necesaria.

2) *Modo Flexible:*

Algunas redes neuronales necesitan más flexibilidad que la que proveen los anteriores métodos. Estos métodos manejan las neuronas una a una, así como las sinapsis

Para crear neuronas dinámicamente en la red usaremos estos métodos. La diferencia entre ellos es que uno define el estado y la Salida deseada y el otro deja indefinidos estos valores.

```
public void addNeurona (int ordenC,String TipoN, double Estado, double Sal_Deseada)
public void addNeurona(int ordenCapa,String TipoNeurona)
```

int ordenC es el orden de la capa en la que crearemos la neurona.

String TipoN es el nombre de la clase a la que pertenecerá la neurona.

double Estado es el estado que tendrá la neurona

double Sal_Deseada es la salida deseada que tendrá la neurona.

Para unir neuronas usamos este método de la clase red.

```
public void unirNeuronas(int ordenCapaOrigen,int idNeuronaOrigen, int ordenCapafin ,
int idNeuronafin, String ClaseSinapsis)
```

ordenCapaOrigen es el orden de la capa en la que está la neurona desde la que sale la sinapsis.

idNeuronaOrigen es el identificador de la neurona desde la que sale la sinapsis

ordenCapaDestino es el orden de la capa en la que está la neurona a la que llega la sinapsis

idNeuronaDestino es el identificador de la neurona a la que llega la sinapsis

ClaseSinapsis es el nombre de la clase a la que pertenecerá la sinapsis.

Para unir una neurona a todas las neuronas de una capa usamos este método de la clase capa.

```
public void unirCapaNeurona(int ordenCapa,int idNeurona, int ordenCapaNeurona ,  
String ClaseSinapsis)
```

ordenCapa es el orden de la capa en la que está la neurona desde la que salen la sinapsis.

ordenCapaDestino es el orden de la capa en la que está la neurona desde la que sale la sinapsis

idNeuronaDestino es el identificador de la neurona a la que llegarán la sinapsis.

ClaseSinapsis es el nombre de la clase a la que pertenecerá la sinapsis.

8.3 Diseño del Comportamiento.

El diseño del comportamiento de una red neuronal se puede dividir en dos partes la dinámica de la computación y el aprendizaje de la red.

En el marco de trabajo estos dos comportamientos se diferencian en dos partes, para uno en la red y el otro en la sinapsis o la neurona. Para definir la dinámica debemos definir el método ejecutar de la red, ejecuta la red con un patrón, y actualizar de la neurona, actualizar el estado de la neurona cuando se le solicita. Para definir el aprendizaje tenemos que definir el método aprender de la clase red, que define los pasos de la red para aprender una lista de patrones, y aprender de la clase sinapsis, que define como actualiza sus pesos una sinapsis.

Mención aparte tienen las redes de aprendizaje online, que van aprendiendo patrones según se van ejecutando en ellas. Estas redes se pueden modelar tomando los dos comportamientos en el mismo método y de esto se habla en un pequeño apartado al final del tutorial.

8.3.1 Diseño de la Dinámica de Computación

En este apartado deberemos especificar el comportamiento de una red neuronal cuando le envían un patrón.

Para ello deberemos redefinir el método abstracto de la red

```
public Patrón ejecutar(Patrón p);
```

Además deberemos redefinir el método abstracto de la neurona

```
public abstract void actualizar();
```

8.3.1.1 El diseño del método de la red ejecutar

Primero nos centraremos en el diseño del método de la red ejecutar. Para ello vamos a seguir el siguiente guión:

A) Especificamos que hace la red en cada paso:

Lo que debemos hacer es una lista de tareas que la red neuronal hace con cada patrón que le enviamos. Por ejemplo, un perceptrón :

1. Le enviamos el patrón a la capa de entrada
2. Las neuronas de la capa de salida consultan el valor de las de entrada y se actualizan

B) Traducimos esto a las funciones que tiene nuestro marco:

El marco tiene 3 métodos principales de la clase capa para esto, que son:

- a. void actualizarNeuronasAleatorias (int NumVeces)
- b. void actualizarNeuronas()
- c. void enviarPatrón (Patrón P)

El ultimo método enviarpatrón carga el patrón P en la capa que llamemos. Los métodos a y b lo que hacen es actualizar las neuronas de la capa a la que estemos llamando. Las neuronas se actualizan según se determine en el método actualizar de las clases neurona hija. La diferencia entre ambos es que actualizarNeuronas actualiza a todas las neuronas de la capa, sirve para representar el sincronismo, y actualizarNeuronasAleatorias actualiza NumVeces Neuronas aleatoriamente y sirve para representar el comportamiento asíncrono.

Si algunas de las tareas de la lista implican que las neuronas deben hacer algo que no es actualizarse entonces debemos modelar ese comportamiento en la neurona haciendo una neurona_hija

- a. void ejecutarNeuronas(String Tipo, String Metodo)
- b. void ejecutarNeuronasAleatorias (String Tipo, String Método, int NumVeces)

Los dos métodos ejecutarNeuronas y ejecutarNeuronasAleatorias llaman a un método determinado de las neuronas de la capa llamante la diferencia entre ambas es la misma que la que hay entre los métodos a y b arriba explicado. ejecutarNeuronas modela comportamiento síncrono y ejecutarNeuronasAleatorias modela comportamiento asíncrono.

La única restricción es que el método no debe tener parámetros. El método que se llama debe ser definido en una neurona hija de la principal
De la siguiente manera

```
package RedesNeuronales;
public class Neurona_Especial extends Neurona {
    public void metodo_especial () {
        // Aquí se espera la Implementacion del usuario
        double pot = math.sqrt(potencial_Adelante());
    }
}
```

y la llamada al método sería así :

```
CapaEntrada.ejecutarNeuronas("Neurona_Especial", "metodo_especial");
```

- C) Escribimos este funcionamiento en el método public Patrón ejecutar(Patrón p)
Por ejemplo para el perceptrón

```
public Patrón ejecutar(Patrón p) {
    CapaEntrada.enviarPatron(p);
    CapaSalida.actualizarNeuronas();
```

```
return CapaSalida.conseguirEstado ();
}
```

Con esto terminamos la dinámica de la red hija y nos queda definir como es el comportamiento de la neurona.

8.3.1.2 El diseño del método actualizar de la clase neurona

Para definir el comportamiento de la neurona, deberemos en redefinir el método actualizar.

```
public abstract void actualizar()
```

La tarea que debe desarrollar este método es el de pasar la neurona de un estado a otro nuevo según dicte el algoritmo de la red neuronal.

Para ello la neurona cuenta con una serie de métodos para acceder a todos los valores que te puedan ser necesarios .

Estos métodos devuelven el potencial sináptico de las sinapsis de neuronas que llegan desde capas de adelante, Atrás y lateralmente usando esta función

$$\text{Potencial} = \sum (PesoSinapsis_i * EstadoNeurona_i)$$

```
public double potencial_Adelante()
public double potencial_Atras()
public double potencial_Lateral()
```

Estos métodos devuelven la norma cuadrática entre las sinapsis y los estados de las neuronas de las que salen desde capas de adelante, Atrás y lateralmente usando esta función. Se suele usar en redes competitivas o no supervisadas.

$$\text{Potencial} = \sqrt{\sum (PesoSinapsis_i - EstadoNeurona_i)^2}$$

```
public double normaCuadraticaAd()
public double normaCuadraticaAtr()
public double normaCuadraticaLat()
```

Un ejemplo podría ser el estado del perceptrón:

```
public void actualizar() {
    // primero hallamos el potencial
    double pot = potencial_Adelante();
    if (pot<0) Estado = -1.0;
    else if (pot>0) Estado = 1.0;
}
```

8.3.1 Diseño del Aprendizaje de la red

En este apartado vamos a diseñar el modo en que nuestra red aprende y lo vamos a traducir a nuestro marco de trabajo.

Para ello tendremos que redefinir dos métodos

El primero pertenece a la red hija y se refiere al modo en que la red toma los patrones de la lista y los va usando para actualizar los pesos las sinapsis.

```
public void aprender(ListaPatrones_ps)
```

Además de este debemos redefinir el método aprender de la sinapsis

```
public void aprender(double[] parámetros)
```

Para ello seguimos el guión del paso anterior pero teniendo en cuenta que las sinapsis son las que aprenden y las neuronas se actualizan.

8.3.2.1 El diseño del método de la clase Red: Aprender

- A) Especificamos que hace la red en cada paso como en el paso anterior, con estas salvedades

Las sinapsis aprenden cuando les decimos que se actualicen y también que al método aprender le entra una lista de patrones.

Un ejemplo podría ser este: un perceptrón

Para cada patrón

- Enviamos el patrón a la capa de entrada.
 - Actualizamos las neuronas de la capa de salida para tener los resultados predichos
 - Les decimos a las sinapsis que unen las capas que se actualicen.

- B) La anterior lista de tareas la pasamos a código para actualizar las sinapsis tenemos la siguiente función.

```
public void aprenderSinapsis(int capaOrigen, int capaDestino, double[]
parámetros , int numVeces)
```

donde

int capaOrigen es la capa de donde salen las sinapsis.

int capaDestino es la capa a la que llegan las sinapsis.

double[] parámetros son los posibles parámetros que le tengamos que pasar a

las sinapsis para que aprenda.

int numVeces si numVeces = 0 se actualizaran todas las sinapsis de la capa
si es mayor que cero se actualizaran numVeces sinapsis aleatorias

un ejemplo podría ser este perceptrón.

```
public void Aprender(ListaPatrones Ps) {
    int i=0; int j =0; Patrón p;

    while(i<Ps.NumeroPatrones)
    {
        p= Ps.Patron(i);
        //1. Envío al principio y al final los valores de entrada y de Salida_deseada
        CapaEntrada.EnviaPatron(p);
        CapaSalida.EnviaSalida(p);
        //2. Realizo una ejecución de la red
        CapaEntrada.ActualizarSinapsisAdelante();
        CapaSalida.ActualizarNeuronas();
        //3. Le digo a las sinapsis que se actualicen para ese patrón i
        double[] sinParametros = new double [0];
        aprenderAdelante(0,2, sinParametros,0);
        i++;
    }
}
```

8.3.2.1 El diseño del método de la clase Red: Aprender

En este paso, especificaremos como se comportara una sinapsis individual cuando se le ordene que aprenda, esto es, que actualice su peso.

Consistirá en aplicar la función con la que se actualiza lo pesos el algoritmo de aprendizaje.

El modo en que se actualizan las sinapsis, la definimos nosotros mismos en la clase sinapsis_hija, redefiniendo el método **public void Aprender(double [] parámetros)**

Para ello tendremos una serie de funciones que nos darán todos los datos que necesitemos como :

public double neuronaEntradaEstado()
Devuelve el estado de la neurona que llega, que entra, a la sinapsis

public double neuronaSalidaEstado()
Devuelve el estado de la neurona de sale de la sinapsis

public double neuronaEntradaSalida()
Devuelve la salida deseada de la neurona que llega, que entra, a la sinapsis

public double neuronaSalidaSalida()

Devuelve la salida deseada de la neurona de sale de la sinapsis.

un ejemplo podría ser este que es el método de aprendizaje del perceptrón

$$W = W + (Z - Y) * X * \eta \text{ con}$$

x = Salida deseada.

y = Salida tras la actualización.

x = entrada.

W = peso.

η = parámetro de regulación.

public void Aprender(double [] parámetros){

double X = neuronaEntradaEstado();

double Y = neuronaSalidaSalida();

double Z = neuronaSalidaEstado();

double η = 0.05;

Peso = Peso + ($Z - Y$) * X * η ;

}

8.3.3 Redes de Aprendizaje online

Las redes online son redes que siempre están aprendiendo. Cuando ejecutamos un patrón para que lo clasifique lo incluye dentro de su aprendizaje.

Las redes con aprendizaje online no se pueden ajustar al la manera de diseñar que se ha explicado antes aunque si se pueden ajustar al marco de trabajo.

La forma de modelar este tipo de redes es hacer que el método ejecutar haga las dos tareas ejecutar un patrón y aprenderlo y el método aprender no haga mas que mostrar los patrones para que los ejecute, como en el siguiente código:

public void aprender(ListaPatrones Ps) {

Patron p; *Patron* q; *int* i=0;

while(i<Ps.NumeroPatrones) {

p= Ps.patron(i);

q= ejecutar(p);

i++;

} }

Un ejemplo de este tipo de redes son las ART1 y ART2 modeladas en los ejemplos.

8.4 Diseño de los Patrones

Las redes neuronales funcionan con patrones, esto es, las listas de valores de entrada o de salida que se corresponden a los estados de las neuronas. La manera que el marco de trabajo usa los patrones es mediante las clases Patrón y ListaPatrones que los manejan. La mejor forma de crear patrones en el marco de trabajo es desde un fichero.

El fichero debe ser de texto y tener este formato

```
Numero_de_Patrones
Num_ValoresEntrada1 NumValores_Salida1  Valor_Entrada11 ValorEntrada12 ...
Valores_Salida11 Valores_Salida12 ...

....

Num_ValoresEntradaN NumValores_SalidaN  Valor_EntradaN1 ValorEntradaN2 ...
Valores_SalidaN1 Valores_SalidaN2 ...
```

Esto es, primero el número de patrones que hay en el fichero. Después, en cada línea un patrón, Los dos primeros números indican la longitud del patrón en la entrada y en la Salida_deseada.

La opción de introducir salida deseada es para redes supervisadas, en el caso de redes no supervisadas los patrones serian igual solo que NumValores_Salida será 0 en todos los patrones y no habrá valores de salida.

Para más detalles sobre este punto en el apéndice C hay una guía sobre el formato de los ficheros de patrones

8.5 Empaquetamiento de la red neuronal.

Java proporciona una herramienta muy útil para guardar los diversos ficheros class resultantes de compilar nuestro código fuente java en uno solo, los ficheros jar.

Una vez que tenemos nuestras clases terminadas, preparadas y compiladas en ficheros class, podemos añadirlas al modelo para que puedan ser usadas sin necesidad de importar librerías en el sistema java.

Para ello, debemos convertirlas en ficheros jar, debemos unir las clases de las redes ya compiladas con la herramienta jar.exe, junto con las clases esenciales del marco de trabajo. Para facilitar esta tarea, se ha realizado un batch script, “crearJar.bat” que automatiza este trabajo, solo debemos configurarlo adecuadamente y el nos creará el fichero jar con las clases del modelo.

El proceso para crear un fichero jar, es muy fácil. Solo se debe crear un directorio en el directorio crearjar del marco de trabajo, al igual que las redes que ya están realizadas, y guardar en el los ficheros class resultantes de la compilación de los ficheros java que modelan nuestra red, editar el fichero crearJar.bat y ejecutarlo.