

---

# Redes Neuronales

*Práctica de Modelos de la Computación II*

---

Juan Pedro Bolívar Puente  
*raskolnikov@es.gnu.org*

---

Ingeniería Informática, 2º A

*Curso 07/08*

# Índice general

<b>1. Introducción</b>	<b>3</b>
1.1. Las redes neuronales . . . . .	3
1.2. Objetivos de la práctica . . . . .	4
1.3. Planteamiento del problema . . . . .	4
<b>2. Diseño</b>	<b>6</b>
2.1. Introducción . . . . .	6
2.2. Metodología . . . . .	6
2.3. Arquitectura . . . . .	7
2.3.1. Caso de estudio . . . . .	7
2.3.2. Pruebas . . . . .	7
2.3.3. Conclusiones . . . . .	8
2.4. Aprendizaje . . . . .	9
2.4.1. Caso de estudio . . . . .	9
2.4.2. Pruebas . . . . .	10
2.4.3. Conclusiones . . . . .	11
2.5. Red final . . . . .	11
<b>3. Implementación</b>	<b>13</b>
3.1. Introducción . . . . .	13
3.2. Generalidad . . . . .	13
3.3. Eficiencia . . . . .	15
3.3.1. El Pipeline . . . . .	15
3.3.2. Materialización . . . . .	16
3.4. El programa . . . . .	17
3.4.1. Instalación . . . . .	17
3.4.2. Uso . . . . .	18
3.4.3. Documentación . . . . .	19

## ÍNDICE GENERAL

---

3.4.4. Licencia . . . . .	19
3.5. Mejoras . . . . .	19
3.5.1. Redes recurrentes . . . . .	19
3.5.2. Detección automática de capas y ajuste del nivel de paralelización	19
3.5.3. Enfoque estático para la genericidad: persiguiendo la eficiencia	20

# 1

## Introducción

### 1.1 Las redes neuronales

---

Las redes neuronales son un modelo de computación muy interesante, en cuanto a que, al contrario que las máquinas Turing-compatibles que estamos acostumbrados a programar, no indicamos al sistema de forma explícita los pasos que tiene que seguir en el cómputo de su solución, indicando directamente las relaciones existentes entre todos los datos de entrada, sino que la red aprende, modificando paulatinamente sus parámetros internos para responder correctamente a una serie de ejemplos.

De esta forma, se trata de emular a la naturaleza, modelando una red de “neuronas”, que son idealizaciones hipotéticas dónde cada nodo es una transformación funcional sobre una serie de entradas, creando una topología todo lo compleja que queramos. De esta forma, hay que realizar dos pasos claros para tratar construir una red neuronal que resuelva un problema:

1. Diseñar la arquitectura. Es decir, qué tipo de transformación funcional aplicarán las neuronas, cuantas de ellas habrá, como se conectarán entre sí y que parámetros de entrada proporcionaremos a la red.
2. Entrenar a la red, aplicando un algoritmo de aprendizaje según el cual se modifiquen iterativamente parámetros internos de la red -pesos de las conexiones, bías de las neuronas, etc.- de tal forma que el método converja hacia la contestación correcta por parte de la red de los casos que se le proponen en su entrenamiento.

El proceso, que como se verá combina el análisis teórico a priori con la comprobación empírica y estadística, será descrito con más detalle en el capítulo próximo.

### 1.2 Objetivos de la práctica

---

Con la siguiente práctica se pretende lograr una primera aproximación real a las redes neuronales, haciendo ejercicio de los contenidos teóricos estudiados en la asignatura. Para ello, se deberá resolver un problema mediante una red neural, realizando:

- El diseño de la arquitectura y el entrenamiento de la misma, con el fin de obtener una red neuronal que resuelva con eficacia suficiente nuestro problema, utilizando para ello las herramientas teóricas y empíricas adecuadas.
- La implementación de la red como un programa en un lenguaje turing-compatible, de tal forma que proveamos una aplicación que se pueda utilizar directamente para dar respuestas a nuestro problema.

### 1.3 Planteamiento del problema

---

Existen numerosos problemas en los que no conocemos con precisión las relaciones entre las distintas variables, o tal vez sean demasiados los parámetros, o los algoritmos exactos sean inviables por su ineficiencia. Ejemplos típicos pueden ser el reconocimiento de elementos en señales, como pueden ser rostros en imágenes. Es fácil reconocer una cara en una fotografía, pero cuando tratamos de indicar con precisión a un ordenador cómo diferenciar un rostro de otro, nos damos cuenta de que la subjetividad o la experiencia o mecanismos “desconocidos” de que nuestro cerebro realiza en el proceso, no son para nada intuitivos ni fáciles de cuantificar y describir en un programa directamente y, en gran parte, son aún desconocidos para la ciencia. Muchos otros ejemplos existen, como por ejemplo la toma de decisiones bursátiles, la predicción de sucesos y un largo etcétera.

En nuestro caso, el problema consiste en la detección de cáncer de mama en pacientes. Para ello, debemos definir el problema en términos de una serie de parámetros cuantificados, en este caso son:

1. Espesor de la masa.
2. Uniformidad del tamaño de las células.
3. Uniformidad de la forma de las células.
4. Adhesión marginal.
5. Tamaño de las células epiteliales.

### 1.3. PLANTEAMIENTO DEL PROBLEMA

---

6. Núcleos aislados.
7. Cromatina suave.
8. Núcleos normales.
9. Mitoses.

Los cuales no tienen mucho sentido fuera del ámbito médico, pero por fortuna ya han sido estudiados y cuantificados por los médicos para que nosotros aportemos nuestro granito de arena en la automatización de diagnóstico construyendo una red neuronal. Nuestro sistema debe responder *benigno* (0.0) o *maligno* (1.0), según la muestra de entrada, proporcionada en los términos de los parámetros anteriores, normalizados en valores entre 0 y 1, tenga o no cáncer.

Los datos que utilizaremos en el entrenamiento se han obtenido de la *Wisconsin Breast Cancer Database*. En [1] puede encontrar más información y referencias sobre el caso de estudio y los datos que usaremos.

# 2

## Diseño

### 2.1 Introducción

---

El diseño de una red neuronal, requiere de un proceso que combine el conocimiento y aplicación de los resultados teóricos de la disciplina con el trabajo empírico, con el fin de decidir la red que mejor se adecúe a nuestro problema. Así, procederemos a la elección, por un lado, de la arquitectura, basándonos en la contrastación de los resultados de distintas arquitecturas siguiendo una metodología empírica específica y después haremos lo propio con el algoritmo de aprendizaje. Esta metodología es importante dado el carácter probabilístico del método de aprendizaje -que encuentra mínimos locales y depende de la situación inicial.

### 2.2 Metodología

---

Tanto para la elección de la arquitectura como para la elección del método de entrenamiento utilizaremos *validación cruzada* con un factor  $n$  de 10.

Este método consiste en dividir el conjunto de prueba en  $n$  trozos de igual tamaño. Sea  $P$  el conjunto de  $n$  conjuntos de prueba se deberá entrenar la red con todas las combinaciones usando  $P - p_i$  como conjunto de entrenamiento y  $p_i$  como conjunto de validación. Para evaluar la eficacia de la red usaremos la media de las tasas de fallos en validación de cada una de las  $n$  pruebas.

## 2.3 Arquitectura

---

### 2.3.1 Caso de estudio

Para la resolución del problema utilizaremos una red de perceptrones multicapa, conectados acíclicamente mediante *feedforward*. Como función de activación se usará la *función sigmoide*:

$$f(x) = \frac{1}{1 + e^{-x}} \quad (2.1)$$

Claramente, por la definición del problema, la red debe tener una capa de entrada con nueve nodos -uno por cada parámetro de los descritos anteriormente- y una capa de salida con un solo nodo, cuya salida emita si existe o no cáncer.

Establecidos estos parámetros a priori -por definición de la práctica- decidiremos mediante la corroboración empírica de los resultados tanto el número de capas ocultas a utilizar, como el número de nodos por capa. En concreto, reduciremos nuestro estudio a las siguientes arquitecturas:

1. Con una capa oculta.
  - a) Con 5 nodos en la capa oculta.
  - b) Con 10 nodos en la capa oculta.
  - c) Con 15 nodos en la capa oculta.
2. Con dos capas ocultas.
  - a) Con 5 nodos en cada capa oculta.
  - b) Con 10 nodos en cada capa oculta.

### 2.3.2 Pruebas

Las pruebas han sido realizadas con el software JavaNNS [2]. La base de datos obtenida <sup>1</sup> ha sido dividida en 10 trozos de 45 casos cada una.

Cada red ha sido entrenada utilizando la metodología de validación cruzada con los 10 conjuntos de casos. Finalmente tomamos la media de las tasas de errores de validación en los 10 entrenamientos como medida de la eficacia de la red.

El entrenamiento utilizado ha sido el de retropropagación con los siguientes parámetros:

---

<sup>1</sup>Puede consultarse en el fichero cancer .nor adjunto.



## 2.3. ARQUITECTURA

Figura 2.1: Error de validación de las distintas arquitecturas.

- Tasa de aprendizaje:  $\eta = 0,002$
- Tolerancia de error:  $d_{max} = 0,1$
- Ciclos de aprendizaje: 10000

La tabla 2.3.2 refleja los datos obtenidos para las redes de una capa oculta y la tabla 2.3.2 hace lo propio con las de dos. La gráfica 2.1 nos permite comparar la evolución de la red con las distintas arquitecturas probadas.

$i$	Error de validación		
	5 nodos	10 nodos	15 nodos
0	0.021270851294199627	0.01945723957485623	0.020818810992770723
1	0.019854662153455944	0.02006758451461792	0.020488214492797852
2	0.011333371533287897	0.010540370146433513	0.009815755817625258
3	0.004746770858764649	0.00419209102789561	0.005027639203601413
4	0.01309681600994534	0.014297074741787381	0.012932133674621583
5	0.05839768515692817	0.05819496048821343	0.060096099641588
6	0.025800601346993152	0.001935586002137926	0.0016678424345122444
7	0.014699208736419677	0.014977227316962348	0.015156735314263238
8	0.026421279377407497	0.02739789221021864	0.027005132039388022
9	0.00535668863190545	0.004030687279171414	0.004638457298278808
$\mu$	0.020097793509930736	0.017509071330229439	0.017764682090944714

Cuadro 2.1: Resultados de los entrenamientos de las arquitecturas con una capa oculta.

### 2.3.3 Conclusiones

Observamos que la menor tasa media de error de validación se da con la red de dos capas ocultas y diez nodos por capa, aunque realmente los valores son parecidos entre las distintas arquitecturas. Elegiremos por tanto una red de este tipo para nuestro programa final.

## 2.4. APRENDIZAJE

$i$	Error de validación	
	5 x 2 nodos	10 x 2 nodos
0	0.019298096497853596	0.01959741645389133
1	0.01906248993343777	0.02024976544910007
2	0.009838303592469956	0.00944445530573527
3	0.0035957694053649902	0.004090438617600335
4	0.012596702575683594	0.012613874011569553
5	0.06134229765997993	0.062032397588094076
6	0.0019037400682767232	0.0011065036058425903
7	0.011175958315531413	0.011795600255330404
8	0.025809698634677462	0.026357854737175834
9	0.003982770111825731	0.003615072700712416
$\mu$	0.016860582679510121	0.016728830602433943

Cuadro 2.2: Resultados de los entrenamientos de las arquitecturas con dos capas ocultas.

## 2.4 Aprendizaje

### 2.4.1 Caso de estudio

Una vez elegida la arquitectura, es menester disertar sobre el método de entrenamiento que finalmente usaremos para obtener los valores de los pesos de los enlaces y los bias de los perceptrones.

En concreto, nuestro estudio se centrará en la retropropagación con y sin momentum. Para hacerlo más didáctico, trataremos de observar cómo afecta la aplicación de un momentum. Es de esperar que una tasa de aprendizaje grande converja rápido pero contenga oscilaciones indeseables y el momentum atenue esas oscilaciones.

#### 1. Retropropagación:

- a) Tasa de aprendizaje:  $\eta = 0,002$
- b) Tasa de aprendizaje:  $\eta = 0,2$

#### 2. Retropropagación con momentum:

- a) Tasa de aprendizaje:  $\eta = 0,002$ , momentum:  $\mu = 0,5$
- b) Tasa de aprendizaje:  $\eta = 0,2$ , momentum:  $\mu = 0,5$

## 2.4.2 Pruebas

Las pruebas han sido realizadas sobre los conjuntos de datos anteriores, siguiendo la misma metodología, y también con el software JavaNNS. Los entrenamientos se han realizado con 10000 ciclos.

Los resultados obtenidos para el algoritmo de *backpropagation* se reflejan en la tabla 2.4.2, y cuando se añade el momentum en la tabla 2.4.2.

$i$	Error de validación	
	$\eta = 0,002$	$\eta = 0,02$
0	0.01959741645389133	0.023274676005045573
1	0.02024976544910007	0.02053376038869222
2	0.00944445530573527	0.005169603890842861
3	0.004090438617600335	0.003399343623055352
4	0.012613874011569553	0.01141665776570638
5	0.062032397588094076	0.07476975123087565
6	0.0011065036058425903	0.001297944618595971
7	0.011795600255330404	0.008620789978239271
8	0.026357854737175834	0.021949769390953913
9	0.003615072700712416	0.0031754202312893336
$\mu$	0.016728830602433943	0.017360771712329654

Cuadro 2.3: Resultados de los entrenamientos realizados con retropropagación.

$i$	Error de validación	
	$\eta = 0,002$	$\eta = 0,02$
0	0.020721244812011718	0.021710952123006184
1	0.019953446918063694	0.01993919875886705
2	0.006105400456322564	0.00992592904302809
3	0.0017760425806045532	0.0012154945896731483
4	0.012125474876827663	0.003920455442534553
5	0.06485695838928222	0.08531966739230686
6	0.0011047507325808207	0.0013104829523298475
7	0.009995554553137885	0.007999180422888861
8	0.021759606732262505	5.945018596119351E-4
9	0.0033366603983773124	0.016570259465111627
$\mu$	0.016173514044947095	0.016850612204935816

Cuadro 2.4: Resultados de los entrenamientos realizados con retropropagación con momentum.

## 2.4. APRENDIZAJE

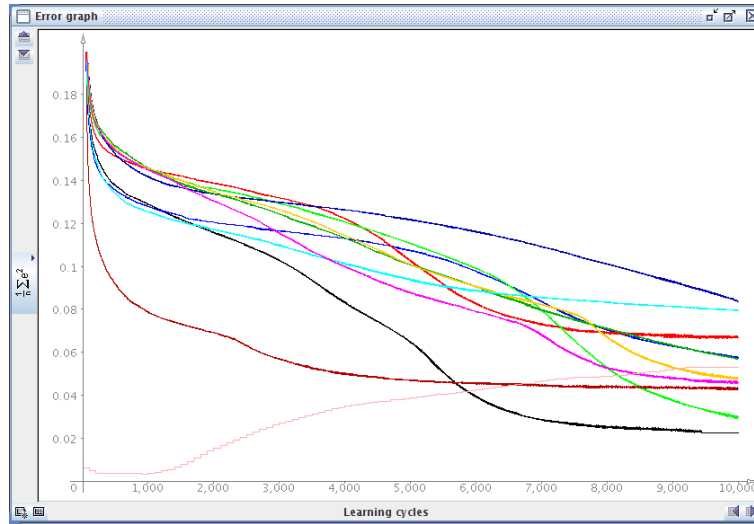


Figura 2.2: Evolución del error durante el proceso de aprendizaje con retro-propagación y  $\eta = 0,02$

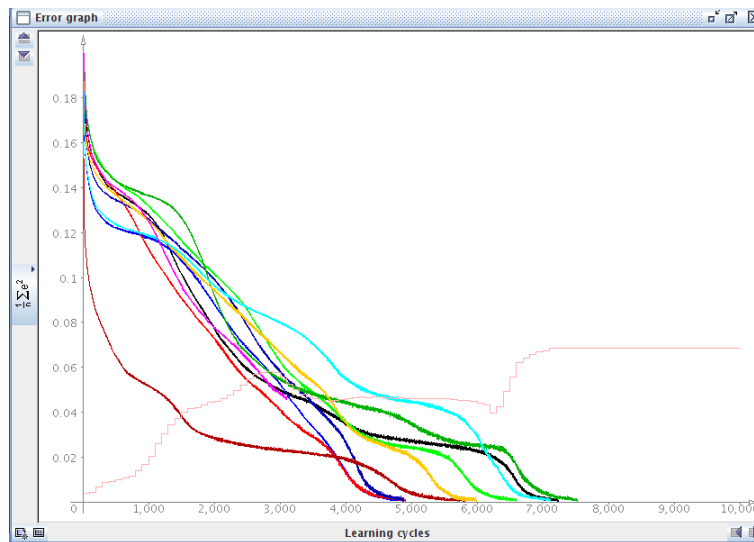


Figura 2.3: Evolución del error durante el proceso de aprendizaje con retro-propagación-momentum y  $\eta = 0,02$

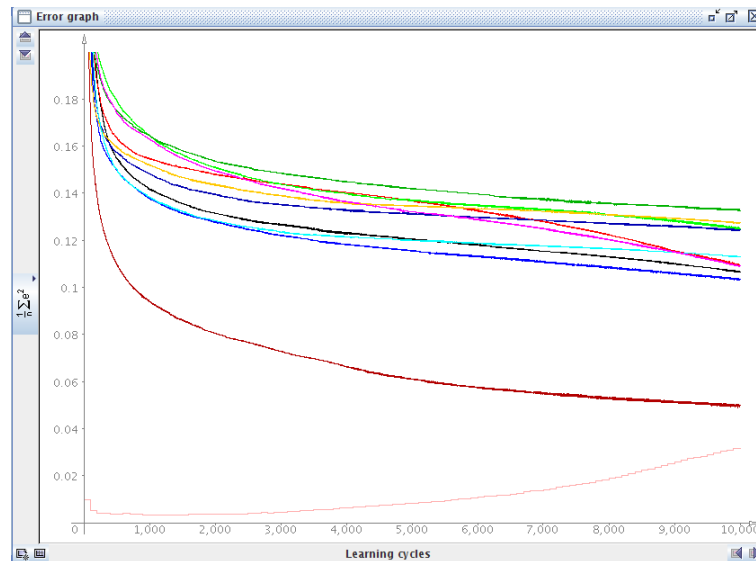


Figura 2.4: Evolución del error durante el proceso de aprendizaje con retro-propagación-momentum y  $\eta = 0,002$

### 2.4.3 Conclusiones

Tras las pruebas realizadas, observamos que la retropropagación con momentum produce mejores resultados que cuando no existe momentum.

Por otro lado, una tasa de aprendizaje mayor produce, como era esperable, una convergencia más rápida, como podemos observar comparando las figuras 2.2 y 2.4. Sin embargo, podemos contrastar<sup>2</sup> que este método produce rápidamente sobre-aprendizaje, repercutiendo negativamente en los resultados finales. En este sentido, parece más conveniente usar un  $\eta$  menor, que proporcione una aprendizaje más lento pero más estable.

Además, comparando 2.2 y 2.3 observamos que el momentum acelera bastante el proceso de aprendizaje.

Finalmente, el menor error medio de validación entre las 10 pruebas lo tiene la retropropagación con momentum y una tasa  $\eta = 0,002$ , que será el mecanismo de aprendizaje que usaremos finalmente.

<sup>2</sup>Usando, a parte de los datos de las tablas proporcionadas, los datos completos del proceso de aprendizaje de los ficheros .log adjuntos.

## 2.5 Red final

Para la construcción final de la red hemos utilizado la arquitectura y método de aprendizaje descritos en los dos apartados anteriores. Para el aprendizaje final hemos dividido el conjunto de datos en dos ficheros, uno de aprendizaje con el 80 % de los casos y otro de validación con el 20 % restante. Diremos además que hemos utilizado 10000 ciclos de aprendizaje, ya que tras varias pruebas hemos constatado que es un valor cercano al punto donde comienza el sobre-aprendizaje, aunque éste punto no es siempre el mismo a causa del carácter probabilístico de la configuración inicial durante el aprendizaje.

El aspecto final de la red dentro del programa JavaNNS puede observarse en la figura 2.5. La figura 2.6 representa la evolución de la tasa de fallos, tanto en validación como en aprendizaje, durante el proceso de educación de la red.

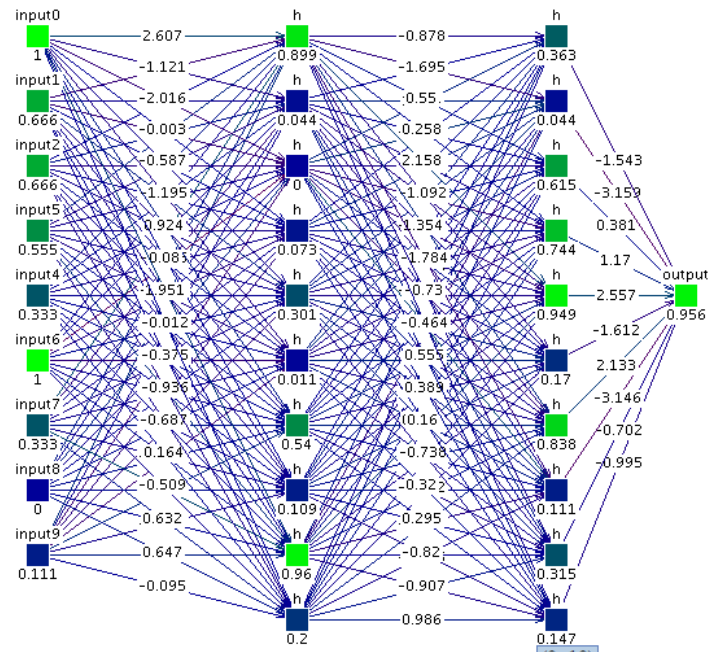


Figura 2.5: Diagrama de la red en JavaNNS tras el proceso de aprendizaje.

Comentar también que el error en aprendizaje final es de 0,05943158432677552 y el de validación 0,010698234641944969. Su media es  $\mu = 0,035064909484360245$ , con lo que la fiabilidad esperada por nuestro programa final será:

$$(1,0 - \mu) \cdot 100 = 96,493 \%$$

2.2

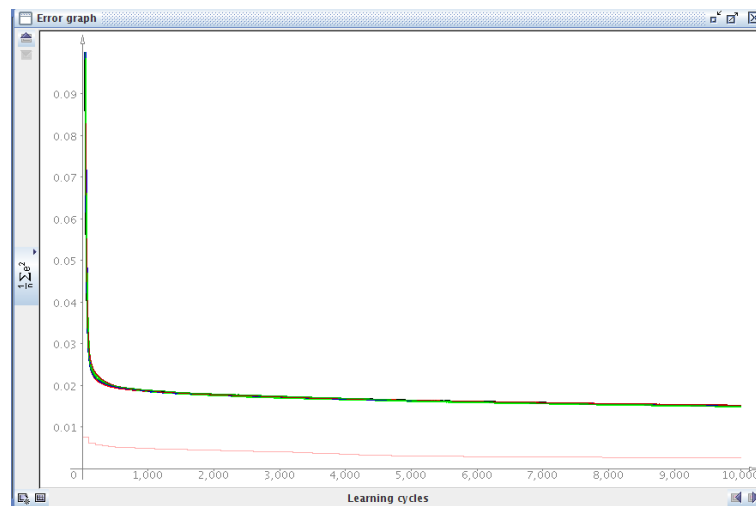


Figura 2.6: Evolución del error durante el proceso de aprendizaje con de la red final.

# 3

## Implementación

### 3.1 Introducción

---

La implementación de una red neuronal en un computador tradicional -turing-compatible- es bastante interesante, en cuanto a que al ser un modelo de computación con un enfoque radicalmente diferente, confluyen decisiones y enfoques muy interesantes que pueden tomarse a la hora de implementarse.

Así, mientras que la tarea anterior estaba bastante delimitada por el guión de prácticas, ahora se nos abre un abanico de posibilidades que, desde mi perspectiva personal, enriquece bastante el desarrollo de la práctica.

En este sentido, discutiremos dos aspectos importantes: por un lado, la búsqueda de generalidad, es decir, una implementación que provea de las abstracciones adecuadas para que pueda adaptarse a cualquier red neuronal y, por otro lado, la eficiencia, aprovechando adecuadamente los recursos del ordenador.

### 3.2 Generalidad

---

No es difícil percatarse de que la *orientación a objetos* es un paradigma adecuado para enfocar el diseño de nuestro programa. Así, en busca de generalidad, podemos representar cada neurona como un objeto. Además, trataremos de satisfacer los siguientes requisitos:

- Soportar redes heterogéneas. Con esto nos referimos a la posibilidad de incluir distintos tipos de neuronas en la red que no sean necesariamente perceptrones, o que en los propios perceptrones puedan alterarse las funciones de activación y de salida del nodo.
- Soportar redes no sólo basadas en el modelo *feedforward* por capas,



sino que puedan, por ejemplo, incluirse “accesos directos” que se salten capas o bifurcaciones entre capas “paralelas”. También se deseará la posibilidad de que la topología de la red pueda contener ciclos y en definitiva ser tan libre como podamos imaginar un grafo de nodos.

En la figura 3.1 observamos el diagrama de clases UML del programa.

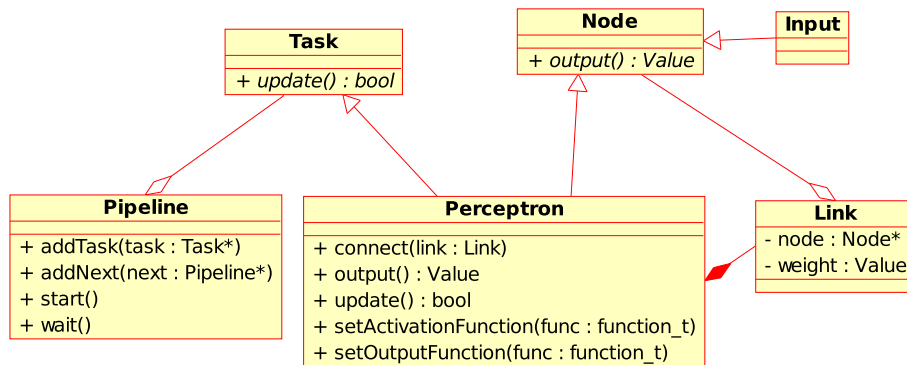


Figura 3.1: Diagrama de clases UML de la estructura interna.

Para soportar el primer requisito usaremos polimorfismo, reduciendo el concepto de “nodo” a lo más básico: *un nodo es algo que emite un valor de salida*. De esta forma, la interfaz `Node` puede implementarse de varias formas. Nosotros en nuestra implementación, por ser meramente didáctica, sólo hemos incluido perceptrones (clase `Perceptron`) y nodos de entrada (clase `Input`) -que sirven para introducir en el sistema los valores proporcionados por el usuario.

El nombre de la interfaz `Node` puede ser engañosa, puesto que invita a pensar que es una clase que en si misma implementa la base de un nodo del grafo de la red. Sin embargo, hemos preferido dejar a las clases que heredan conceptualizar ellas su modelo de arquitectura. De esta forma observamos que es la clase `Perceptron` la que provee un método `connect()` para conectarle una entrada, que puede ser un nodo cualquiera, no sólo un perceptron. De esta forma llevamos al extremo la capacidad de heterogeneidad: cada modelo puede proporcionar su propia visión de las conexiones -por ejemplo, no necesariamente dirigidas o ponderadas- y estos modelos pueden usarse en común siempre que sus modelos de conexionado intersequen en la interfaz `Node`.

Por otro lado, observamos que en un `Perceptron` podemos asignar cualquier función unaria  $Value \rightarrow Value$  como función de activación o de salida.

Además, la función `connect ( )` permite además crear redes recursivas y otras lindezas varias.

A parte de todo esto, los perceptrones necesitan ejecutarse, y es por esto que los definimos como una tarea -implementa Task. Ahora veremos que ventaja tiene esto y comentaremos las otras clases del diagrama.

## 3.3 Eficiencia

---

A simple vista puede parecer, debido a la sencillez con la que se implementan en un computador las redes neuronales *feedforward* estudiadas, que los computadores son un medio idóneo para su implementación. Mi opinión (desde la humildad de un indocto recién iniciado en esta materia) es que no es así.

Y opino así porque, mientras que los ordenadores trabajan de forma iterativa, una red neuronal es intrínsecamente paralela y/o concurrente. En este sentido, cada neurona puede entenderse como una unidad de cálculo independiente, que sirve datos según le van llegando. Por tanto, nuestra implementación debe tener en cuenta los siguientes factores:

1. La naturaleza de una red neuronal es paralela.
2. Simular la paralelización de una red neuronal mediante los mecanismos de concurrencia sólo añadiría coste a nuestra implementación. Sin embargo, casi todos los ordenadores personales que actualmente se venden incluyen procesadores multicore, que son realmente paralelos y pueden tratar de explotarse para mejorar la eficiencia de la red.

### 3.3.1 El Pipeline

A pesar de lo dicho, el nivel de paralelismo de una red neuronal tiene restricciones, en cuanto a que una neurona no puede servir un valor antes de que sus entradas estén estables, ni antes de que las neuronas siguientes terminen de consumir el último valor emitido.

Pensemos en una red multicapa *feedforward*. En este caso, cada capa debe computarse una detrás de otra. ¿No nos conduce eso a un método iterativo? La respuesta es no, en cuanto a que consideremos:

1. Que no existen dependencias de entradas y salidas entre los nodos de una misma capa. Por tanto, una capa puede computarse en para-

lelo, agrupando conjuntos de nodos si se quiere con una granularidad arbitraria.

2. Que la red neuronal no tiene por qué estar orientada a computar un único vector de entrada, sino que lo más probable es que compute muchos vectores de entrada uno tras otro.

En este segundo punto entra el concepto de *pipeline*. El *pipeline* es un método utilizado por los microprocesadores para acelerar la ejecución de instrucciones. En un procesador, la ejecución de una instrucción requiere de varios pasos más pequeños que deben ejecutarse uno tras otro, pero que utilizan partes del procesador independientes entre sí. Lo que hace entonces el procesador es que, una vez una instrucción ha abandonado el primer paso de para pasar al segundo, el procesador comienza a computar *a la vez* que el segundo paso de esa instrucción, el primer paso de la instrucción siguiente.

Aplicado esto a las redes neuronales, podemos decir que, si el sistema operativo distribuye correctamente las hebras que computarían cada capa de la red, mientras que la segunda capa está computando el primer vector de entrada, la primera capa puede ir adelantando trabajo y computar el primer vector de entrada. Así, si tenemos que el tiempo que se tarda en ejecutar cada capa es  $t$ , y existen  $n$  capas, mientras que un modelo iterativo tardaría  $m \cdot n \cdot t$  en computar  $m$  vectores de entrada, un modelo mediante pipeline tardaría  $m \cdot t + n$ .

#### 3.3.2 Materialización

Este concepto lo implementamos nosotros a través de la clase `Pipeline`. Esta clase, cumple las dos formas de paralelización que comentábamos antes, es decir, permite que una capa adelante trabajo antes de la siguiente, pero también permite que varias neuronas de la misma capa se ejecuten en hebras distintas. La forma de conseguir esto es reduciendo el concepto a su más sencilla y versátil expresión.

Así, pensemos en la clase `Pipeline` como una tarea que forma parte de una cadena de pipelines, es decir, constituye ese paso atómico que describíamos antes, que se ejecuta como parte de un proceso mayor. Cuando reducimos el concepto de este paso atómico, observamos que podemos entenderlo como *una tarea que requiere de que otra/s tareas se ejecuten antes, y que no puede ejecutarse antes de que se haya ejecutado la tarea siguiente*.

En nuestra implementación, la clase `Pipeline` tiene una función `addNext()` que añade un pipeline siguiente. De esta forma, si hacemos `a.setNext(b)`,

una vez lancemos los pipelines mediante sus métodos `start()`, *b* esperará a que se ejecute *a* y luego *a* a que se ejecute *b*, etc. Observad que hemos asumido que *a* y *b* comparten datos, por tanto, realmente, mientras se ejecute *b* no se ejecuta *a* y vice versa. Visto así, no existe ganancia de paralelismo. Sin embargo, si añadimos una etapa *c* como siguiente de *b*, puede estar ejecutándose *a* y *c* simultáneamente. Más aún, en su lugar podemos añadir la etapa *c* como siguiente de *a*, en cuyo caso esta se ejecutaría a la vez que *b*, como en el caso de dos neuronas de la misma capa.

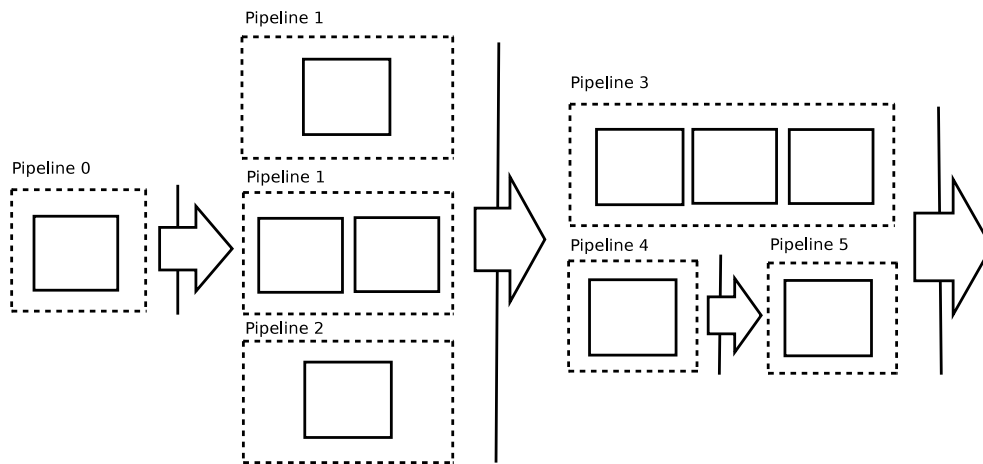


Figura 3.2: Ejemplo de funcionamiento de los pipelines.

La figura 3.2 representa una estructura más compleja de como pueden combinarse los pipelines, reflejo de la versatilidad de nuestra implementación. En ese gráfico las líneas punteadas representan pipelines y las líneas verticales dividen puntos de sincronía como los descritos, dónde la flecha indica en el sentido en el que avanza la cadena. Varios pipelines apilados verticalmente en el dibujo representa que se ejecutan en paralelo sin existir dependencias entre ellos.

Del diagrama cabe mencionar también el significado de las cajitas que hay dentro de los pipelines. Los pipelines, no son las tareas en sí, sino que son “portadores de tareas”. Esto nos permite definir las tareas independientemente de como se estructure su ejecución finalmente. Así, una tarea, en nuestro caso de redes neuronales, consiste en *la actualización de una neurona*. También hay dos tareas especiales, `InputTask` y `OutputTask`, que se encargan de leer los datos de entrada e inyectar en la red y devolver al usuario las salidas emitidas por la red. En general, una tarea puede ser cualquier objeto que herede de `Task` e implemente su método `update()`. Notar que un pipeline puede contener varias tareas, que se ejecutarán una tras otra en un mismo ciclo del mismo. Así, por ejemplo, podemos meter

todas las neuronas de una misma capa en el mismo pipeline.

### 3.4 El programa

---

Una vez descrito como está hecho y cuales son los aspectos más relevantes de su diseño, mostremos el programa en sí y su funcionamiento. Al programa lo hemos bautizado *yann -Yet Another Neural Network* y está implementado en C++. Su funcionamiento ha sido comprobado sobre una máquina Debian Sid con `gcc 4.3.1`.

#### 3.4.1 Instalación

Para el sistema de compilación e instalación del programa se utiliza GNU Autotools, lo que hace que su compilación sea más cómoda y portable en cualquier entorno Unix. Para compilar el programa debe ejecutar, como usuario:

```
$ ./configure
$ make
```

Y para instalarlo, como superusuario:

```
$ make install
```

Para evitar que se instale en la ruta por defecto puede usar la opción `--prefix` durante la ejecución de `./configure`. Para más opciones de compilación e instalación ejecute `./configure --help`.

#### 3.4.2 Uso

Como se ha visto, el programa es bastante genérico, por lo que la definición de la red neuronal debe proveerse a través de un fichero. A través de este fichero, podemos describir *cualquier red neuronal de perceptrones no recurrente*. La red no puede ser recurrente, ya que el modelo de pipelines que hemos descrito no admite recurrencia. En la carpeta `examples` puede encontrar el fichero `network.yann` que implementa la red que hemos estudiado en el capítulo 2. La sintaxis de estos ficheros `.yann` no es compleja y debería servir el estudio del fichero de ejemplo para construir otras redes.

Cuando se ejecuta el programa sobre una red, este quedará esperando a que introduzcamos tuplas de valores de entrada, y por cada una de ellas

emitirá un vector de respuesta. Este proceso continuará hasta que envíemos el carácter EOF (Ctrl+D en Bash). Este comportamiento lo hace idóneo para ser utilizado en combinación con otras herramientas de la línea de comandos mediante *pipes*.

Además, tal y como especifica la ayuda del programa, este admite otros argumentos:

```
yann - yet another neural network.  
(c) 2008 Juan Pedro Bolívar Puente.
```

uso:

```
yann fichero_de_red [opciones]
```

opciones:

-i, --input <fichero>	Lee la entrada desde fichero en lugar de desde la entrada estándar.
-o, --output <fichero>	Emite la salida sobre un fichero en lugar de sobre la salida estándar.
-h, --help	Muestra esta ayuda.

### 3.4.3 Documentación

En la carpeta doc del paquete del programa puede encontrarse este documento acompañado de sus fuentes en L<sup>A</sup>T<sub>E</sub>X.

### 3.4.4 Licencia

El programa es software libre mediante la licencia GPLv3+. De esta forma, todo su código o parte de él pueden ser utilizados en otros proyectos de software que tengan licencia GPL versión 3 o superior. Puede leer la licencia completa en el fichero COPYING.

## 3.5 Mejoras

---

El programa está aún en una fase temprana de desarrollo y ha sido desarrollado en buena parte bajo la presión de los exámenes, por lo que no debe ser considerado exento de errores. Durante el desarrollo de la práctica hemos pensado también muchas mejoras que no ha habido tiempo de implementar. A continuación las resumimos:

### 3.5.1 Redes recurrentes

Como hemos comentado anteriormente, en la versión actual del programa, si bien pueden describirse redes recurrentes, estas fallarán al ejecutarse a no ser que diseñemos muy adecuadamente las capas.

Sería interesante implementar un método para la evaluación de las redes neuronales que realice un recorrido en profundidad del grafo que la describe, de tal forma que no sólo se soportarían redes recurrentes sino que además para cualquier tipo de red ya no haría falta especificar las capas para que el programa evalúe en el orden correcto la red.

Además, esta mejora no es de mucha dificultad.

### 3.5.2 Detección automática de capas y ajuste del nivel de paralelización

En la versión actual del programa hay que especificar las capas a mano en el fichero de descripción de la red, de tal forma que además el programa creará un pipeline por cada capa definida. Mediante una modificación del algoritmo de detección de grafos bipartitos no sería muy complejo detectar automáticamente las capas y, además, poder recibir un parámetro de “nivel de paralelización” por la línea de comandos, de tal forma que se ajuste automáticamente el número de capas por pipeline que se incluirán, o si incluso una capa se partirá entre varios pipelines.

Tal vez un poco más compleja que la anterior, esta tarea tampoco presenta una dificultad excesiva.

### 3.5.3 Enfoque estático para la genericidad: persiguiendo la eficiencia

En la versión actual del programa, podemos decir que hemos utilizado un *enfoque dinámico* para la búsqueda de la generalización, en cuanto a que es en tiempo de ejecución cuando se construye la red y puede modificarse esta en tiempo de ejecución. Las ventajas de este enfoque son muchas y no necesitan ser enumeradas, sin embargo, puede plantearse la genericidad en otros términos.

El mecanismo actual *dinámico* puede entenderse como un *intérprete* de redes neuronales. Sin embargo, un enfoque estático construiría un *compilador* de redes neuronales. Esto permitiría, por encima de todo, optimizar enormemente la red, realizando simplificaciones algebraicas y todo tipo de triquiñuelas para mejorar la eficiencia. En mi visión sobre como debería

funcionar este sistema, lo visualizo como una especie de `lex` para redes neuronales: le pasaríamos la descripción de una red neuronal a nuestro programa y éste nos devolvería un fichero en C que podemos enlazar con nuestro programa para utilizar la red dentro del mismo como una función opaca.

Aunque realizar esta tarea completamente requeriría un estudio más profundo de la problemática, sí que podrían realizarse primeras aproximaciones con una dificultad moderada. Se me ocurre, por ejemplo, que realizar un compilador de redes multicapa via *feedforward* como las estudiadas podría ser sencillo, e incluso se podrían implementar algunas optimizaciones, como la expansión de bucles en beneficio de la eficiencia temporal y detrimento de la espacial u alguna otra. Realmente lo veo muy interesante ya que permitiría construir redes realmente eficientes, y es una pena que la idea se me haya ocurrido cuando ya estaba enfangado en los exámenes.



# Bibliografía

- [1] *Wisconsin Breast Cancer Database*  
`ftp://ftp.ics.uci.edu/pub/machine-learning-databases/breast-cancer-wi`
- [2] *JavaNNS*  
`http://www.ra.cs.uni-tuebingen.de/software/JavaNNS/`