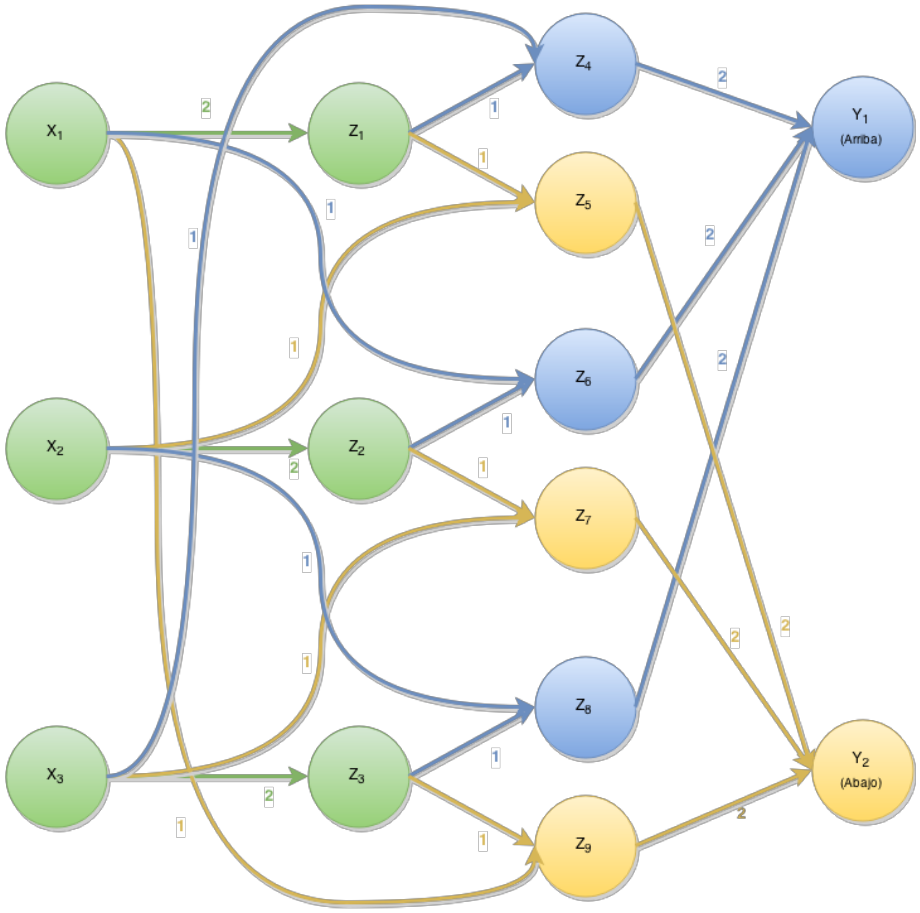
Práctica 1

Neurocomputación

Daniel Garnacho Martín
Pablo Molins Ruano

1. Neuronas de McCulloch - Pitts

Para resolver el ejercicio propuesto utilizamos una red como la de la siguiente imagen, donde todos los umbrales de activación son 2:



Es una red con dos capas ocultas. La primera (Z_1 a Z_3) sirve para retrasar a t+1 el impulso que haya recibido una neurona de entrada en t ya que necesitaremos propagar esa señal para poder conocer si ha habido movimiento. La segunda capa es un poco más complicada. Cada neurona de esta capa (Z_4 hasta Z_9) recibe input de la primera capa oculta y de la capa de entrada, haciendo un AND entre ambas. De las seis, tres se dedican a detectar movimiento ascendente y tres movimiento descendiente. Por ejemplo, $Z_4(t)=X_3(t-1)$ *AND* $X_1(t-2)$, $Z_6(t)=X_1(t-1)$ *AND* $X_2(t-2)$...

¿Podría existir una red con menos neuronas? Parece bastante improbable. El problema se divide en seis casos distintos, ya que hay que comprobar si de las tres entradas dos han estado activas en momentos distintos, por lo que las seis neuronas de las segunda capa oculta no pueden reducirse ya que cada una comprueba uno de esos casos. Tampoco puede

reducirse la primera capa oculta porque al servir para propagar información se necesitan tantas neuronas como información vaya a propagarse. Las de entrada y salida, obviamente, tampoco pueden disminuirse.

¿Y una con menos capas? Tampoco. La primera capa oculta es necesaria para propagar la información en el tiempo y la segunda es necesaria para realizar la comprobación de que se cumple la condición. En la segunda capa oculta se calculan todos los AND en los que el problema se descompone mientras que en la de salida se calculan los OR. Si eliminamos la segunda capa, tendríamos que hacer el AND y el OR en el mismo nivel, lo cual es imposible.

En las siguiente tablas mostramos algunos ejemplos de la evolución de la red. En todas las tablas solo se muestran las neuronas que disparan, para facilitar su identificación (el resto de celdas es 0). Empezamos con el ejemplo de un impulso descendente que no sigue después:

	X_1	X_2	X_3	Z_1	Z_2	Z_3	Z_4	Z_5	Z_6	Z_7	Z_8	Z_9	Y_1	Y_2
0	1													
1		1		1										
2					1			1						
3														1

Ahora, un caso más complicado. Primero van cuatro impulsos descendentes, luego se congelan y por último llega un ascendente:

	X_1	**X_2**	**X_3**	**Z_1**	**Z_2**	**Z_3**	**Z_4**	**Z_5**	**Z_6**	**Z_7**	**Z_8**	**Z_9**	**Y_1**	**Y_2**
0	1													
1		1		1										
2			1		1			1						
3	1					1				1				1
4		1		1								1		1
5		1			1			1						1
6	1				1									1
7			1	1					1					
8						1	1						1	

2. Perceptrón y Adaline

Diseño

Al igual que con el ejercicio anterior, hemos utilizado Python como lenguaje de programación. Esta decisión se debe principalmente a que es un lenguaje que permite un desarrollo mucho más rápido. Por un lado, facilita enormemente las tareas más frecuentes, como la entrada y salida a ficheros o el manejo de ciertas estructuras de datos básicas, como las listas. Por otro lado, su orientación a objetos permite hace un diseño bastante modular.

El diseño que hemos implementado está fuertemente influenciado por la herramienta Weka (http://www.cs.waikato.ac.nz/~ml/weka/index.html). Nuestro diseño gira en torno a Clasificadores, Particiones e Instancias.

Una Instancia es el conjunto mínimo de datos relacionados que definen un caso completo de nuestro problema. Es decir, es un conjunto de entradas asociadas a una salida esperada, la cual se recoge en la Instancia si esta es conocida. El conjunto de todas las instancias de nuestro problema se pasa a un Particionador, que las divide en distintos subconjutos para entrenamiento, validación y test en función de una estrategia específica. El Clasificador se entrena con las instancias separadas para tal fin por el particionador, utiliza el conjunto de validación si el algoritmo de clasificación lo necesita, se estima con el conjunto de test su exactitud y por último se explota.

La ventaja de esta aproximación es que se pueden intercambiar estrategias de particionado y clasificadores sin demasiado esfuerzo, ya que ambas trabajan sobre una abstracción de los datos común (las instancias). En este caso solo hemos programado una estrategia de particionado (división porcentual) y los dos clasificadores pedidos (el perceptrón y adaline) pero así hemos dejado la puerta abierta a cuando en futuras prácticas se necesario o conveniente implementar validación cruzada como particionado u otros tipos de redes como clasificadores.

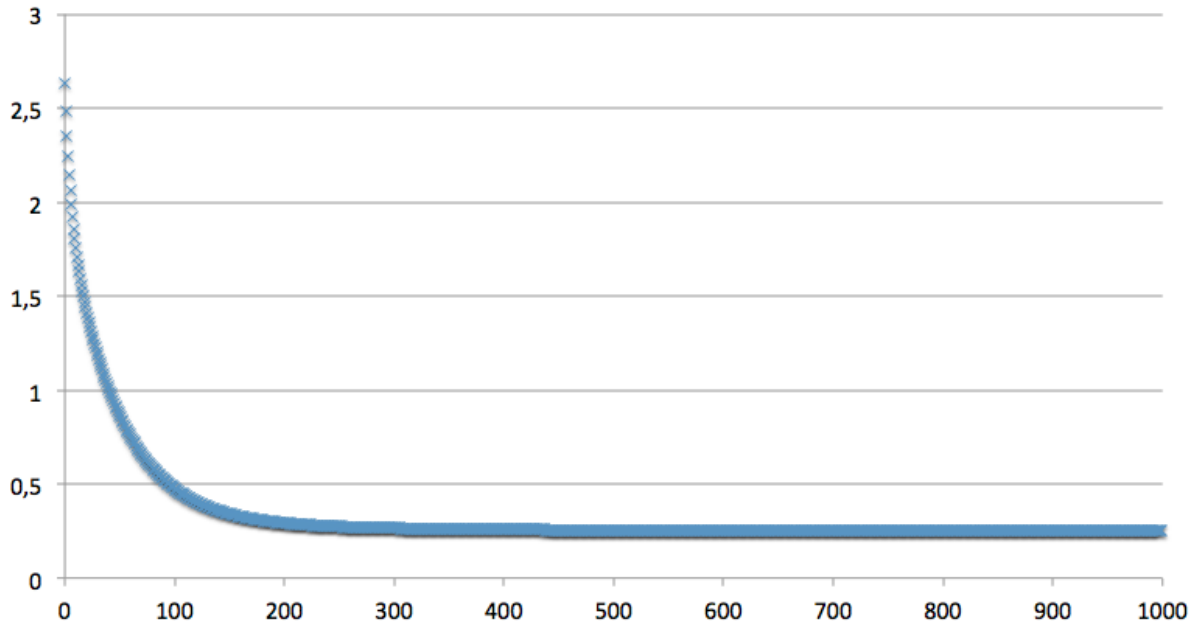
Otros módulos importantes en nuestro diseño son el módulo de lectura de los datos de fichero (en la carpeta src/RW), el módulo principal y la funcionalidad de debug. El módulo de lectura sencillamente se encarga de leer todos los casos del fichero de entrada siguiendo el formato especificado en el enunciado y crear con él un conjunto de instancias. El módulo principal sencillamente va llamando a otros módulos para hacer toda la ejecución entera. Con la funcionalida de debug nos referimos a que cada clasificador tiene como parámetro de debug un booleano que indica si debe ir registrando el error en cada época o si solo debe indicarlo al final. Si el debug está a True, el clasificador abre un fichero (debugAdaline.txt o debugPerceptron.txt) donde almacena la época y el error.

Por último, a la hora de elegir la condición de parada hemos implementado varios sistemas. En ambas redes tenemos dos comprobaciones básicas. Primero, que la red se detenga cuando logre una época con error 0. Segundo, un número máximo de épocas que ambos clasificadores tiene definido como parámetro. Para todas las pruebas hemos dejado fijado ese máximo a 500. Para Adaline, además, hemos implementado un sistema que detiene el entrenamiento cuando la diferencia entre el error cuadrático medio de una época y la siguiente es demasiado bajo, lo cual indica que la red ya está muy próxima al límite de su capacidad de mejora. En concreto, detenemos el entrenamiento cuando la diferencia es menor a $1 \cdot 10^{-7}$.

Evaluación de la implementación

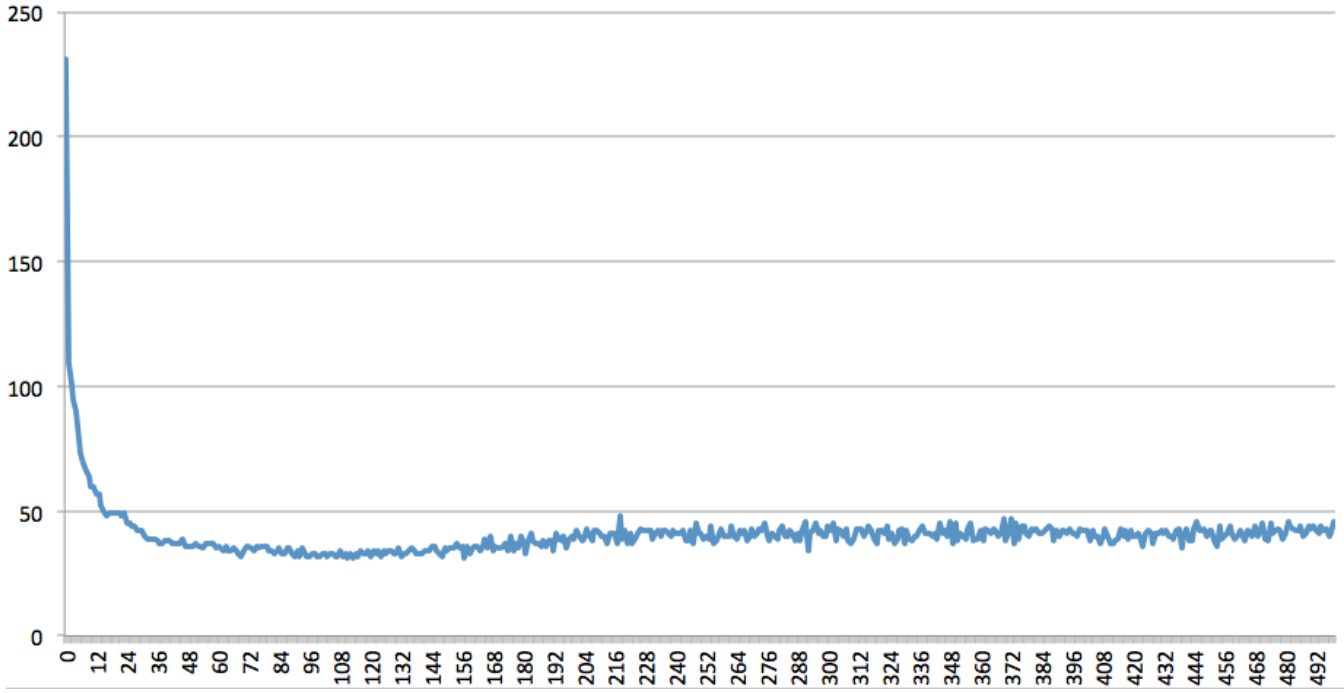
Para evaluar nuestra implementación, realizamos varias pruebas. Primero, creamos dos casos de prueba bastante sencillos (las funciones lógicas AND y OR) para comprobar que ambas redes clasificaban correctamente los casos más simples. Los ficheros de entrada que probamos están disponibles en la carpeta data y existe un ejecutable para cada prueba. En ambos casos comprobamos satisfactoriamente que ambas redes clasificaban correctamente todos los casos.

La siguiente prueba que hicimos fue con el caso de Problema Real 1 y la red Adaline. La teoría nos dice que Adaline es una red que minimiza el error cuadrático medio, por lo que cabe esperar que sea estrictamente decreciente. En este gráfico podemos ver la evolución del error cuadrático medio en cada época, el cual es, efectivamente, decreciente.



*Gráfica sobre la evolución del error cuadrático medio a lo largo del tiempo para una red **Adaline**. En el **eje X**, la **época** a la que corresponde y en el **eje Y**, el valor del **error cuadrático medio**.*

Con el perceptrón hicimos la misma prueba, solo que esta vez no nos fijamos en el error cuadrático medio, sino sencillamente en el número de errores absoluto por época. Según la teoría debíamos esperar que tenga una época de rápida bajada para después estancarse en unas fluctuaciones si el problema no es linealmente separable o llegar a 0, si lo es. Como parece que el problema no lo es, lo que esperabamos es que el error empezara a fluctuar a partir de una época y eso fue exáctamente lo que pasó:



*Gráfica sobre la evolución del error a lo largo del tiempo para una red **Perceptrón**.*
*En el **eje X**, la **época** a la que corresponde y en el **eje Y**, el valor del **error**.*
Se puede observar que al ser un problema no separable linealmente llega un punto en el que error empieza a fluctuar, sin lograr bajar de un umbral.

Cuestiones del enunciado

Evaluación del rendimiento de las redes

Primero hemos evaluado el rendimiento de ambas redes con los problemas reales. En la siguiente tabla se muestra el error medio en la fase de test de ambas redes para ambos problemas. Para evitar el efecto negativo que puede tener el componente aleatorio de inicialización, lo que se muestra es el promedio de 10 ejecuciones:

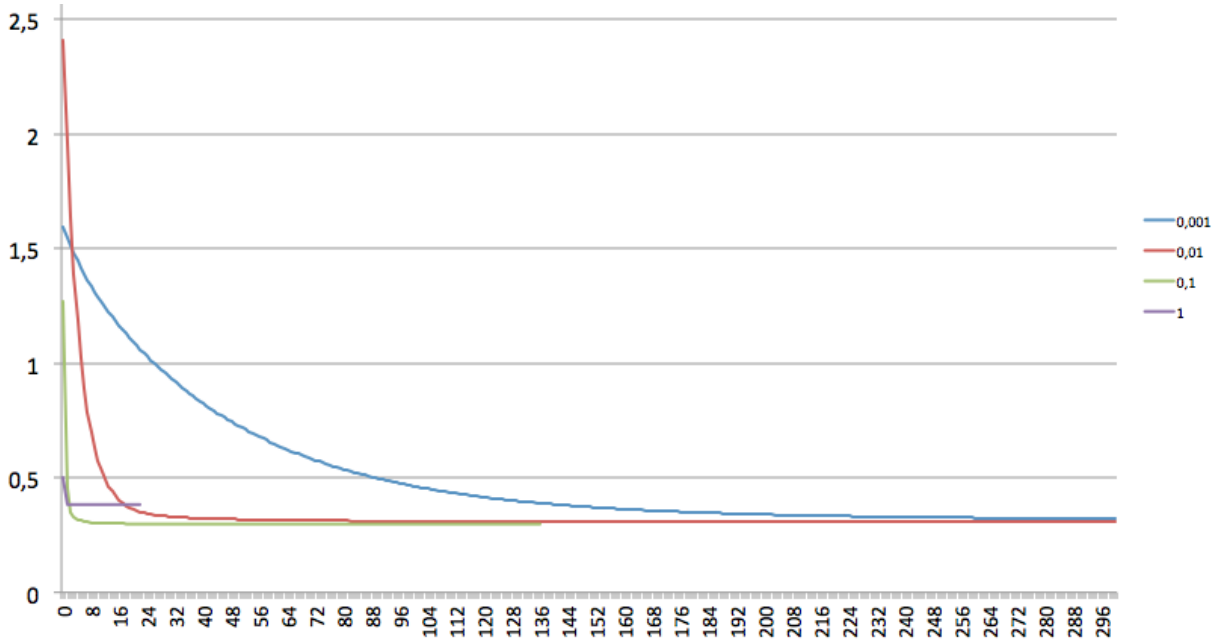
Problema	Error medio Perceptrón	Error medio Adaline
Problema Real 1	5,3588%	4,4019%
Problema Real 2	39,5393%	28,6215%

En los casos linealmente separables, el error medio no es buena métrica de la idoneidad de cada red, pues ambas siempre logran un error del 0%, por lo que hemos buscado otra. En concreto, nos hemos quedado con el número de épocas necesarias para completar el entrenamiento. Mientras que Adaline entrena de media (de nuevo para 10 ejecuciones) en 185 épocas, el perceptrón necesita 340.

Da igual qué métrica analicemos, Adaline siempre es superior, logrando menor error consumiendo menos tiempo de entrenamiento.

Exploración de los parámetros de la redes

Buscando afinar lo mejor posible el parámetro de aprendizaje (alpha) de cada red, realizamos varios entrenamientos utilizando el mismo problema y variando dicho parámetro. Utilizamos como datos el Problema Real 1 y probamos para valores de alpha de 0.001, 0.01, 0.1 y 1. La siguiente gráfica recoge el resultado para la red Adaline:



Evolución del error en función de la época y el alpha elegido para Adaline.
*Cada **serie** se corresponde con un valor de **alpha** distinto.*
*El **eje X** es el **error cuadrático medio** y el **eje Y** es la **época**.*

Como se puede ver, un valor muy bajo hace que la red tarde mucho en entrenar, siendo mejor valores altos. Sin embargo, un valor de 1 hace que la red termine de entrenar demasiado rápido no llegando a la mejor configuración. Así, decidimos dejar alpha establecida a 0.1.

Repetimos lo mismo con el Perceptrón y obtuvimos la misma conclusión, con una evolución bastante parecida.

NAND, NOR, XOR

Como estamos trabajando con dos redes monocapa solo podemos aspirar a entrenar perfectamente problemas separables linealmente como el NAND o el NOR, pero jamás podremos dar con estos clasificadores una solución al problema XOR.

Efectivamente, cuando ejecutamos estos 3 casos, mientras que el NAND y el NOR siempre logran un error de 0% en ambas redes, el XOR se queda en un 50% de error, ya que da igual con qué frontera de división se intente, que el problema XOR siempre deja como mínimo a un caso de cada clase a cada lado de la frontera de división, forzando ese 50% de error.

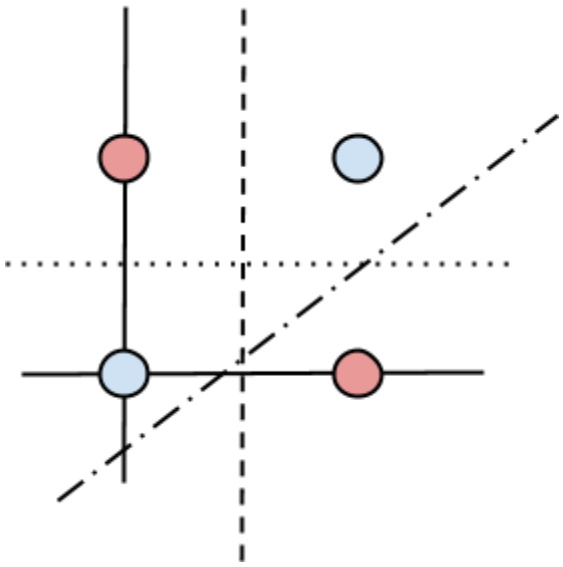


*Diagrama representando el **XOR**. Como se ve, los datos no son linealmente separables, por lo que ni Adaline ni el Perceptrón pueden resolverlo.*

Clasificación de los datos no etiquetados del Problema Real 2

En la carpeta data se incluyen los dos ficheros solicitados con las clasificaciones hechas por las dos redes.

Combinaciones no lineales de los atributos

Para esta sección de la práctica generamos dos ficheros de entrada de nuevos: xor_combinacion.txt y problema_real2_combinacion.txt.

En ambos casos hemos añadido una nueva columna al comienzo de cada línea del fichero que es el primer atributo original por el segundo.

En el caso de XOR, esta sencilla manipulación permite que tanto Adaline como el Perceptrón logren clasificarlo correctamente con un 0% de error. Curiosamente, al añadir esta nueva dimensión al problema, este pasa a ser linealmente separable.

En el caso del problema real 2, la situación no es tan idílica, aunque también se logra una mejora. En concreto, pasamos del 39.5393% de error del Perceptrón que teníamos antes a un 26.2969%. En el caso de Adaline, la mejora es notablemente menor. Pasamos del 28.6215% que teníamos a un 27.5491%, que aunque es una mejora, no es demasiado espectacular.