Práctica 1 de Neurocomputación

Introducción a las Redes Neuronales Artificiales

Daniel Giménez Llorente
Ares Aguilar Sotos

Tabla de Contenidos

Tabla de Contenidos
1. Introducción
2. Neuronas de McCulloch-Pitts
    2.1 Descripción de Diseño
        2.1.1 Primera capa oculta
        2.1.2 Segunda capa oculta
        2.1.3 Capa de salida
    2.2 Ejemplo gráfico
3. Perceptron y Adaline
    3.1 Problema real 1
        3.1.1 Perceptron
        3.1.2 Adaline
    3.2 Puertas lógicas
    3.3 Problema real 2
        3.3.1 Predicciones del problema real 2
    3.4 Problemas con combinaciones lineales
Anexo A. Codificación de la Red Neuronal
    A.1. Codificación de la Neurona
    A.2. Codificación de la Capa
    A.3. Codificación de la Red Neuronal
Anexo B. Codificación del Clasificador
Anexo C. Manual de Usuario
    C.1. Makefile
        C.1.1 Modo de Uso
        C.1.2 Objetivos Disponibles
        C.1.3 Argumentos
    C.2. Neural-Network
        C.2.1 Casos de Uso
            C.2.1.1 Entrenamiento
            C.2.1.2 Predicción

1. Introducción

El sistema nervioso es un sistema de procesamiento de información en paralelo capaz de realizar una gran cantidad de tareas de forma rápida y eficiente bajo diversas condiciones de error. Las características que presenta son deseables para otros muchos sistemas en diversos campos de la ciencia, lo que hace del sistema nervioso un buen ejemplo a imitar. La Neurocomputación Artificial intenta proporcionar a los sistemas artificiales alguna de estas habilidades inspirándose en la forma en que los sistemas biológicos procesan la información.

En esta práctica nos marcamos como objetivo estudiar modelos básicos de redes neuronales artificiales: en primer lugar estudiaremos una red creada por nosotros mismos con neuronas de McCulloch-Pitts, para a continuación crear un clasificador que aprenda automáticamente creando redes neuronales según los modelos perceptrón y adaline.

A la hora de definir nuestra solución, hemos primado, fundamentalmente, dos variables: la rapidez del programa y la generalidad. Queremos, en primer lugar, una red neuronal tan rápida como sea posible, para permitir números elevados de iteraciones de entrenamiento. Por otra parte, nos pareció interesante intentar conseguir tanta generalidad como fuese posible en nuestra implementación de la red neuronal: intentamos que, en una misma librería de red neuronal, esté la posibilidad de crear todo tipo de redes, que la red sea independiente de las neuronas, y que toda ella esté, a su vez, aislada de la implementación del clasificador.

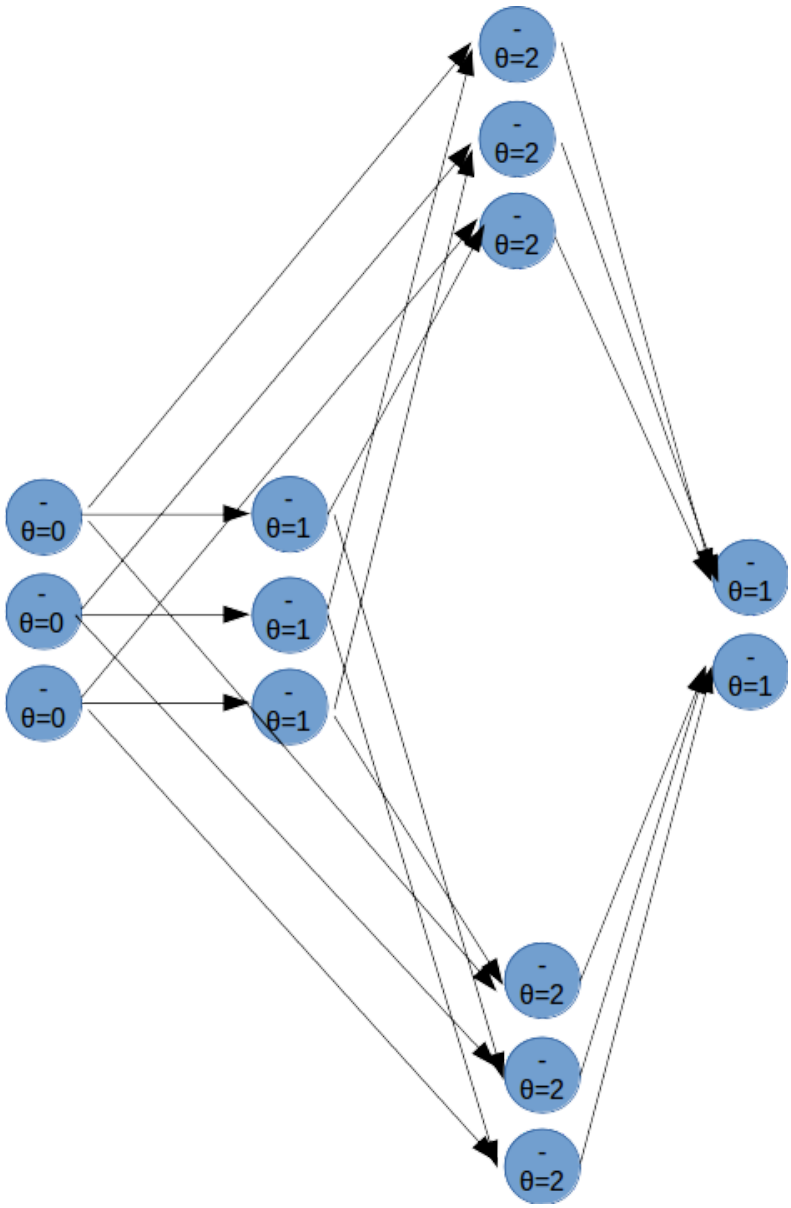
Para codificar nuestra solución hemos elegido el lenguaje de programación C, ya que es el más eficiente computativamente hablando, así como el más rápido de los que hemos considerado. Aunque otros lenguajes, como Octave, Python o Java, facilitan notoriamente la implementación con estructuras de datos orientadas al trabajo matricial, hemos preferido utilizar C, siendo nosotros mismos los que definieramos los objetos que compondrán la red neuronal, dejando en segundo plano la implementación de la red neuronal con matrices que en otros lenguajes resulta trivial. Aunque este lenguaje no está orientado a objetos, hemos intentado respetar los conceptos fundamentales de la programación orientada a objetos para conseguir mayor claridad y abstracción en nuestro código.

Asimismo, hemos diseñado un formato de archivo de red neuronal que permita almacenar los pesos de una red una vez se haya entrenado, y hemos implementado funciones estadísticas dentro de la misma red que permitan que el usuario defina condiciones de parada en base a estadísticos como el error cuadrático medio.

Todo lo anterior nos ha permitido experimentar con diversos modelos de redes y con la influencia de los parámetros de un clasificador, utilizando las bases de datos proporcionadas para la práctica.

2. Neuronas de McCulloch-Pitts

En esta sección vamos a explicar la red neuronal propuesta por nosotros para la resolución del problema. Como se puede observar en la imagen, la red cuenta con 4 capas, 2 de entrada y salida, y otras dos ocultas. Todos los enlaces tienen valor 1 y los umbrales se encuentran dentro de las neuronas. Los detalles de la codificación se encuentran en el Anexo A de este documento.



2.1 Descripción de Diseño

Hemos escogido este diseño por diversas razones. Se necesitan al menos dos capas ocultas para que funcione la red; debido a las características de las neuronas McCulloch-Pitts. Si recibe una conexión inhibidora la salida debe ser 0 y además los pesos positivos deben tener el mismo valor. Si solo tuviese una capa oculta, la salida estará directamente conectada con la entrada y la capa oculta. Debido a las características descritas anteriormente, sería imposible la creación de dicha red y que resolviese el problema de los estímulos presentados en dos direcciones diferentes.

2.1.1 Primera capa oculta

Esta capa sirve de reserva para la primera entrada. Pasa directamente desde la capa de entrada sin ningún tipo de modificación.

2.1.2 Segunda capa oculta

En esta capa se calcula si el segunda dato se corresponde con un estímulo en una dirección o en otra. Para ellos en las tres primeras neuronas de esta capa se comprueba si la entrada coincide con el dato anterior desplazado hacia arriba. Si este es el caso, alguna de las tres neuronas se activará. Lo mismo pasa con las tres siguientes; comparan si la entrada se corresponde con el datos anterior desplazado hacia abajo.

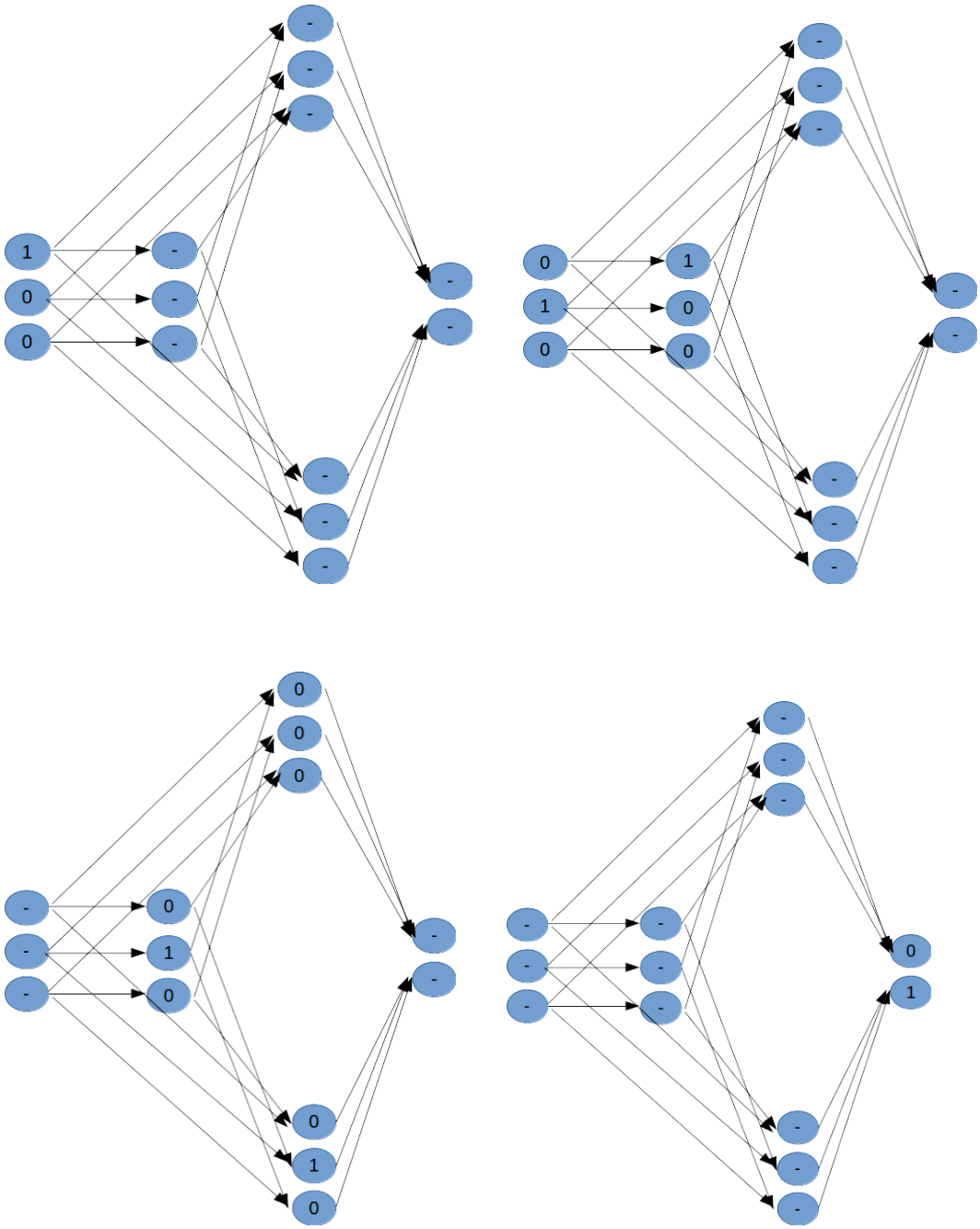
De esta forma, si se ha producido un desplazamiento hacia arriba alguna de las tres primeras neuronas valdrá 1. Si se ha producido hacia abajo, alguna de las 3 ultimas neuronas valdrá 1. Y por último, si no ha habido cambio, todas las neuronas de esta capa estarán a 0.

2.1.3 Capa de salida

La primera neurona de esta capa se corresponde con una puerta OR de las tres primeras neuronas de la capa anterior. La segunda se corresponde con una puerta OR de las tres siguientes. Por como se ha explicado en la capa anterior. Si se ha producido una subida la primera neurona estará a 1 y la otra 0 y viceversa

2.2 Ejemplo gráfico

Suponemos que tenemos una entrada que es (1,0,0) y el siguiente estímulo es (0,1,0)



Como se observa, en t=4 se produce la salida que nos indica que ha habido un descenso

Para ejecutar la simulación de esta red, se deberá llamar al makefile con el objetivo p1.2-mcculloch-pitts y, opcionalmente, con los parámetros INPUT_FILE y OUTPUT_FILE para los ficheros de datos de entrada y salida respectivamente.

3. Perceptron y Adaline

Para este problema, hemos desarrollado dos clasificadores, uno basado en el perceptron y otro basado en adaline. Por defecto se toma como conjunto de train el 60% de la muestra y el resto sirve para la validación. Además, en el caso de clasificar, se genera una archivo de estadísticas que contiene el numero de época, el porcentaje de acierto de train y el error mínimo cuadrático. De esta forma, se puede observar cómo ha ido mejorando el acierto en la clasificación. Los detalles de la codificación del clasificador se encuentran en el Anexo B de este documento, y la documentación del ejecutable final en el Anexo C.2.

3.1 Problema real 1

3.1.1 Perceptron

Hemos aplicado el clasificador basado en el perceptron al problema real 1, cambiando ciertos parámetros hasta llegar a la respuesta óptima. Gracias a la velocidad de C, hemos puesto que el número de épocas sea 1500.

Tras variar los distintos valores de alpha, hemos llegado a la conclusión que con 1500 épocas no hay ninguna relevancia significativa en valores entre el 0 y 1.

Como bien nos informa el programa, la tasa de acierto es de un 97-98 % en el train y de un 95% en el test. Las gráficas en este apartado no son significativas, ya que en la primera época se alcanza el 95% de acierto.

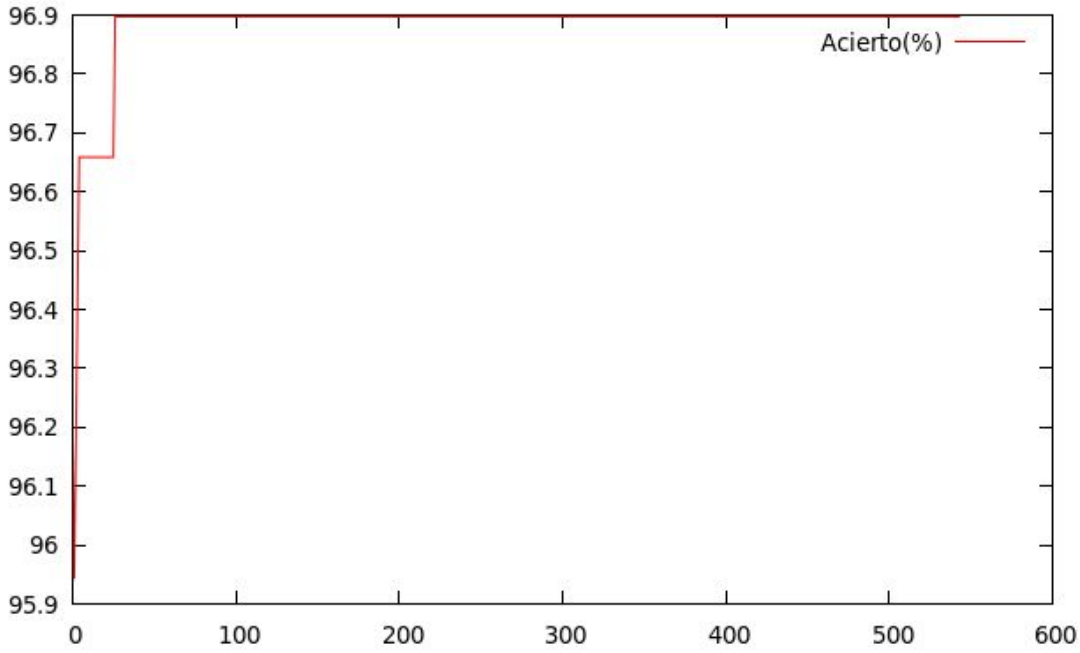
Para ejecutar esta simulación, se debe llamar al makefile con el objetivo p1.3.1-perceptron, que acepta los argumentos opcionales INPUT_FILE y OUTPUT_FILE que se refieren a los datos de entrada y fichero de salida con las predicciones. Si no se proporcionan, se utilizarán los ficheros por defecto.

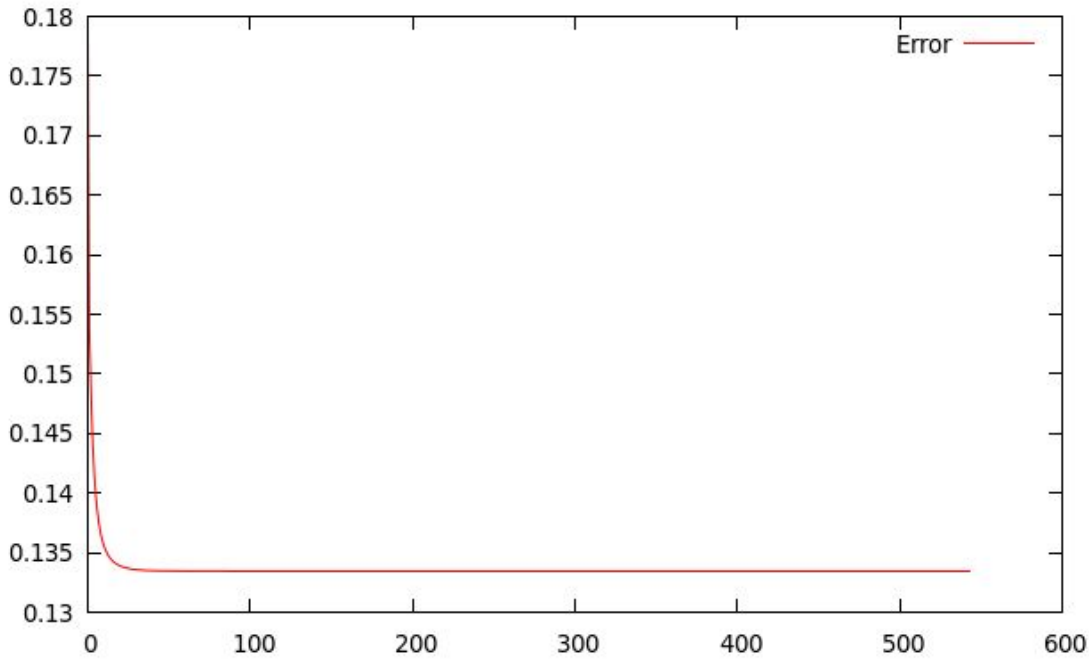
3.1.2 Adaline

En la siguiente tabla se muestra el error cuadrático medio y el porcentaje de acierto del adaline según el alpha

Alpha	**Error**	**Acierto(%)**
0.02	0.166624	95%
0.1	0.145618	95%
0.5	115.371471	70%

Como se puede observar, con un alpha entre el 0.02 y 0.1 se obtienen los mejores resultados, tanto de error como de train.



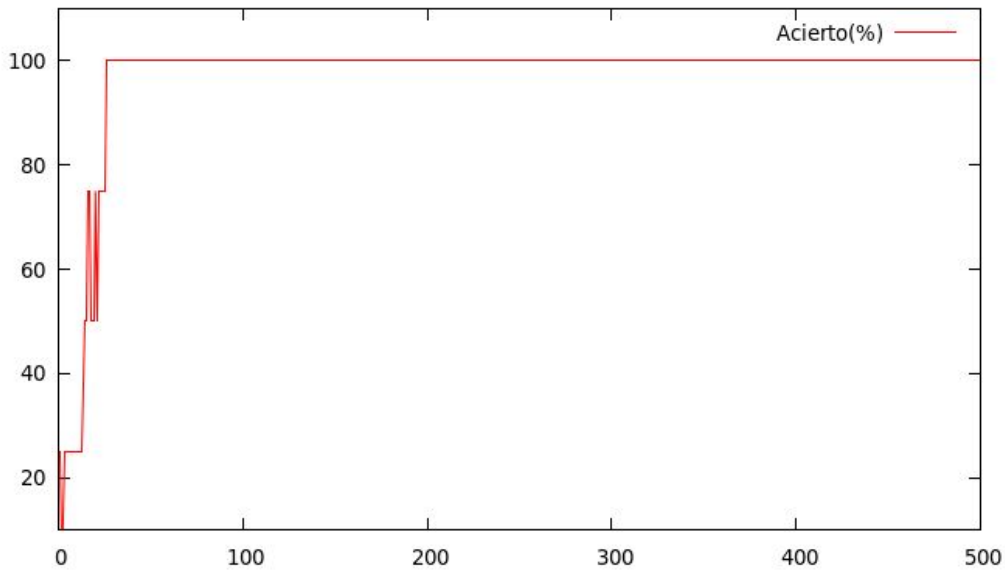


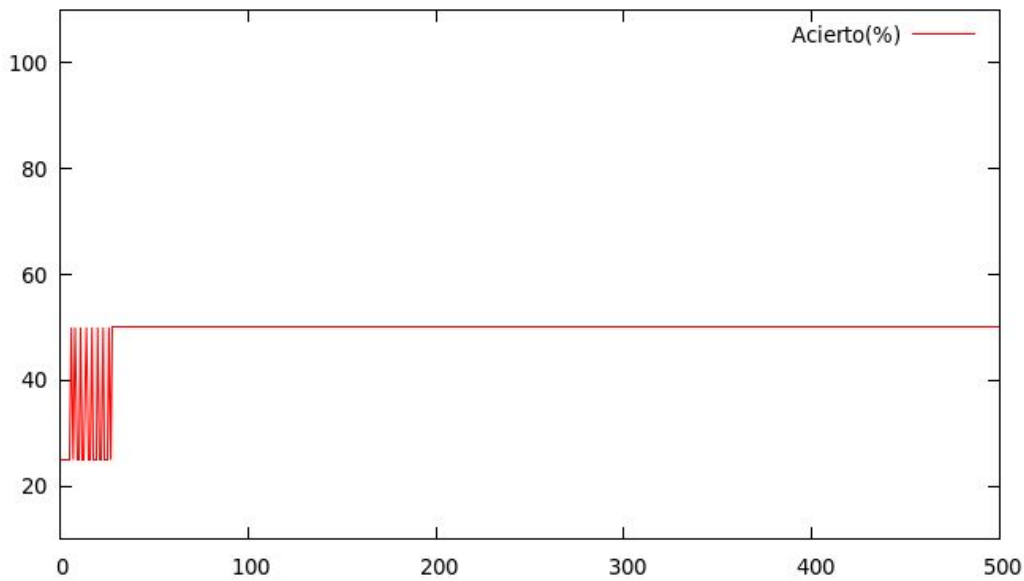
Como se observa en las gráficas el porcentaje de acierto sube rápidamente en las primeras épocas mientras que el error cuadrático medio disminuye.

Para ejecutar esta simulación, se debe llamar al makefile con el objetivo p1.3.1-adaline, que acepta los argumentos opcionales INPUT_FILE y OUTPUT_FILE que se refieren a los datos de entrada y fichero de salida con las predicciones. Si no se proporcionan, se utilizarán los ficheros por defecto.

3.2 Puertas lógicas

Después hemos añadido al programa que si recibe que el porcentaje de train es el 100%, tome también el 100% para la validación. Después de este cambio hemos analizado las diferentes puertas lógicas tanto con el perceptron como con adaline. Al entrenar con todas las posibles combinaciones de las puertas lógicas, al terminar de entrenar tiene 100% de acierto en las puertas lógicas NAND y NOR. Sin embargo la puerta lógica XOR no llega a ese valor. Esto se debe a que como la red neuronal no tiene capa oculta, simplemente está creando un hiperplano según los atributos para dividir el espacio. NAND y NOR si que se puede dividir por ese hiperplano mientras que XOR no.
La primera gráfica hace referencia a la tasa de acierto del NAND mientras que las segunda a XOR





Como se observa, la tasa de acierto en esa ejecución no superó el 50%.

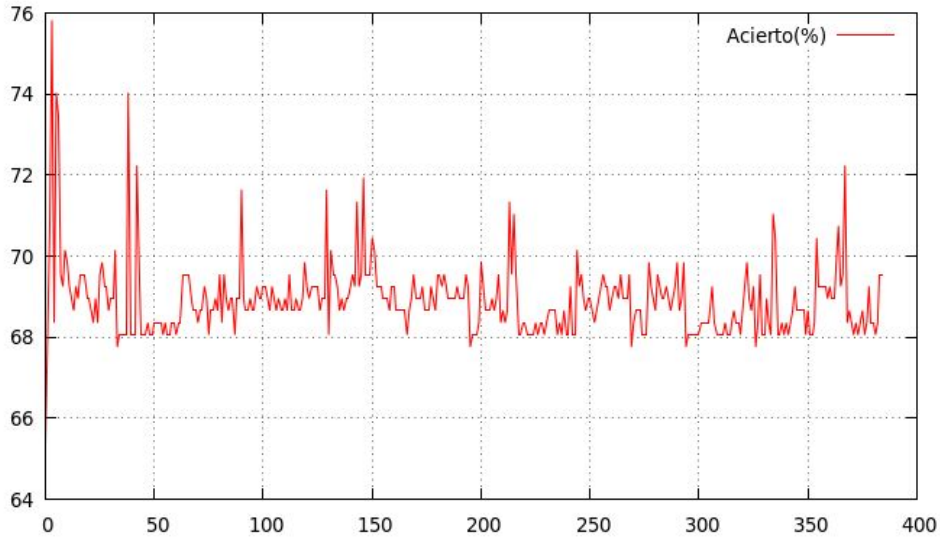
Para ejecutar estas simulaciones, se debe llamar al makefile con los objetivos p1.3.2-nand, p1.3.2-nor y p1.3.2-xor. Todos estos objetivos admiten los parámetros INPUT_FILE y OUTPUT_FILE (por defecto data/databases/[nand|nor|xor].txt y out/out.txt) para personalizar los ficheros de entrada o salida.

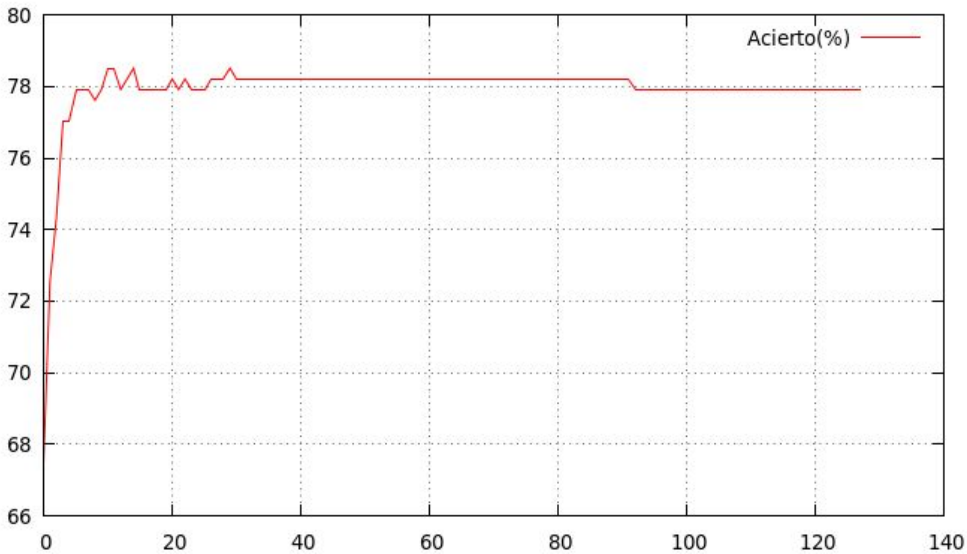
3.3 Problema real 2

Hemos aplicado, al igual que en el problema real 1, los sistemas de clasificadores de perceptron y adaline. Aquí se ha tomado 1500 épocas y un alpha de 0.02 de aprendizaje.
Con estos valores hemos alcanzado los siguientes resultados:

- Perceptron. Tiene una tasa de acierto que se mueve entre 65-75%
- Adaline. Tiene una tasa de acierto que se mueve entre 75-80%

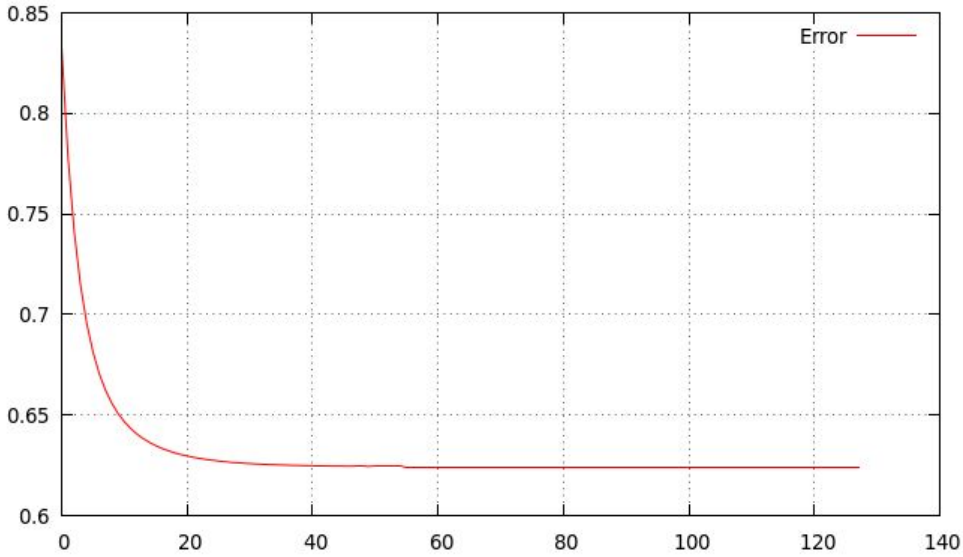
Tasa de acierto del perceptron y adaline, respectivamente:





Como se puede observar, parece funcionar mejor el adaline, que va reduciendo el error cuadrático medio. Además se ve que converge mejor con este que con el perceptron. Sin embargo se puede decir que no es problema separable, debido a las tasas de acierto y de error medio.

Error cuadrático medio del adaline:



Para ejecutar esta simulación, se debe llamar al makefile con los objetivos p1.3.3-perceptron y p1.3.3-adaline, que aceptan los argumentos opcionales INPUT_FILE y OUTPUT_FILE que se refieren a los datos de entrada y fichero de salida con las predicciones. Si no se proporcionan, se utilizarán los ficheros por defecto.

3.3.1 Predicciones del problema real 2

Hemos incorporado al sistema de clasificación que pueda predecir datos de una red neuronal que esté entrenada. Hemos generado los ficheros de predicción asociados tanto al perceptron como al adaline, y hemos observado que son parecidos en su mayoría. Las predicciones se encuentran dentro de la carpeta out.

Para ejecutar los clasificaciones y realizar las predicciones se debe llamar al makefile con los objetivos p1.3.3.1-perceptron-predict y p1.3.3.1-adaline-predict. Si se proporciona el parámetro INPUT_FILE, serán los datos de este archivo los que se clasificarán en lugar de los de data/databases/problema_real2_no_etiquetados.txt. Asimismo, si se proporciona OUTPUT_FILE, será aquí donde se escriban las predicciones. En caso contrario, se escriben en out/out.txt.

3.4 Problemas con combinaciones lineales

Hemos modificado algunos datos añadiendo columnas que se corresponden a combinaciones no lineales del resto de atributos. En el caso de las puertas lógicas, al crear nuevos atributos con combinaciones no lineales hemos reducido el número de aciertos. Esto se debe a que al introducir nuevos atributos, puede que no exista un hiperplano que no pueda separar estos nuevos atributos.
Por último en el problema_real2 tampoco hemos hallado una combinación que mejorase el resultado anterior.

Anexo A. Codificación de la Red Neuronal

En esta sección explicaremos las consideraciones de diseño a la hora de implementar la red neuronal genérica que hemos empleado en esta práctica. Dividiremos nuestra explicación en tres secciones: codificación de la neurona, de la capa y de la red total.

A.1. Codificación de la Neurona

Como estructura básica de la red neuronal, tenemos una neurona. La estructura que hemos utilizado se muestra a continuación:

```
typedef struct Neuron_ {
    double d;
    double d_new;               /* Neuron's value */
    double err;                 /* Error, for training purposes */
    int n_cons;                 /* Number of connections */
    double threshold;
    Connection * cons;          /* Neuron's connections */
} Neuron;
```

Las variables que se almacenan en esta estructura se definen por si mismas. Utilizamos dos variables, d y d_new, para almacenar los valores de la neurona antes y despues de actualizarla. Esto lo hacemos para tener acceso a ambos en todo momento, y sobreescribir el valor solo en el paso de actualización. La variable err la utilizaremos más adelante para el entrenamiento de la red.

Por otro lado, cada neurona almacena un array de conexiones con las neuronas que conectan a ella y sus correspondientes pesos. Para completar nuestra exposición, mostramos a continuación la estructura empleada para las conexiones.

```
typedef struct Connection_ {
    double weight;
    struct Neuron_ * from;
} Connection;
```

Como vemos, es una estructura muy simple que tan solo se limita a almacenar el peso de la conexión y la neurona de la que proviene.

Implementar de esta forma las conexiones en vez de almacenar la matriz de conexiones nos permite, por un lado, programar de forma más intuitiva la red, y por otro, ahorrar espacio al almacenar únicamente las posiciones con una conexión activa.

A.2. Codificación de la Capa

La capa es la estructura más simple entre las que trabajaremos. Esto se debe a que su implementación es inmediata, ya que las conexiones las hemos implementado ya a nivel neuronal.
Por esto, una capa es tan solo un array de neuronas, como se muestra a continuación.

```
typedef struct Neural_Layer_ {
    int n_neurons;               /* Number of neurons in this layer */
    Neuron * neurons;            /* Neurons in this layer */
} Neural_Layer;
```

Cabe destacar que la capa no se encarga del manejo de las neuronas, sino que actúa simplemente como un nivel de abstracción en nuestra red. Es decir, será la red neuronal la que reserve memoria para las neuronas, y cada capa almacenará el puntero a la neurona inicial de la capa y el número de neuronas almacenadas en la misma.

A.3. Codificación de la Red Neuronal

Para terminar el detalle de nuestra implementación, mostramos a continuación como hemos definido la estructura de red neuronal para después explicar los puntos más interesantes.

```
typedef struct Neural_Network_ {
    int n_neurons;             /* Total number of neurons in the net */
    nn_upd_neuron upd_neuron;  /* Function to compute the new value */
    nn_upd_weight upd_weight;  /* Function to compute the new weight */
    int n_layers;              /* Number of layers in the network */
    Neural_Layer * layers;     /* Layers array */
} Neural_Network;
```

Podemos ver inmediatamente que la implementación es muy sencilla: simplemente almacenamos un array de capas y el número total de neuronas (que nos permitirá trabajar con la red, cuando sea necesario, como un único array de neuronas).

Lo más interesante son las dos funciones que permiten la generalizacion en la red: upd_neuron y upd_weight. Los tipos empleados se muestran a continuación.

```
typedef void (*nn_upd_neuron)(Neuron *);
typedef void (*nn_upd_weight)(Neuron *, double , double);
```

Estas dos funciones permiten definir distintos tipos de neuronas con la misma red neuronal, ya que son ellas las que se encargan de actualizar los pesos de una conexion y los valores de la neurona. En el fichero neural_network_functions.c se han implementado estas dos funciones para los casos de las neuronas de McCulloch-Pitts, Perceptron y Adaline; pero el diseño de nuestra red admite que el usuario defina su propia neurona.

Anexo B. Codificación del Clasificador

La codificación del clasificador es bastante directa, ya que únicamente almacena la red neuronal, los conjuntos de datos y diferentes parámetros que se emplearán a la hora de entrenar la red. Además, el clasificador almacena estadísticas de errores en los conjuntos de datos que se podrán utilizar por el usuario para definir condiciones de parada basadas en estas estadísticas.

A continuación se muestra la estructura completa del clasificador neuronal.

```
typedef struct Classifier_ {

    /* Classifier Variables */
    Neural_Network * nn;
    Data           * data_training;
    Data           * data_generalization;
    Data           * data_validation;
    int              epoch;

    /* Classifier Parameters */
    double learning_rate;
    int   max_epochs;
    double max_accuracy;
    double max_mse;
    int bipolar;
    int function_transfer;

    /* Epoch Statistics */
    FILE * file_statistics;

    /*Output file*/
    FILE * predictions;
    /* Accuracy */
    double accuracy_training;
    double accuracy_generalization;
    double accuracy_validation;
    /* Mean Squared Error */
    double mse_training;
    double mse_generalization;
    double mse_validation;

} Classifier;
```

Anexo C. Manual de Usuario

En esta sección explicamos el funcionamiento tanto del programa de generación y simulación de redes neuronales como del Makefile que lo acompaña.

C.1. Makefile

El makefile empleado permite la ejecución de todas las simulaciones discutidas en esta memoria, admitiendo ciertos parámetros para los ficheros de entrada y salida. Detallamos a continuacion el modo de uso, los objetivos disponibles y los argumentos aceptados.

C.1.1 Modo de Uso

La llamada al makefile seguirá la siguiente sintaxis:

make TARGET [INPUT_FILE=/path/to/file] [OUTPUT_FILE=/path/to/file]

C.1.2 Objetivos Disponibles

En primer lugar, el makefile dispone de los siguientes objetivos genéricos:

compile: Compila todas las fuentes y genera el ejecutable bin/neural-network.
clean: Elimina los objetos y los archivos temporales.
help: Muestra la ayuda.

Por otro lado, proporcionamos los siguientes objetivos para realizar la simulaciones discutidas:

p1.2-mcculloch-pitts:
Ejecuta la simulación discutida en el apartado 2. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y OUTPUT_FILE, respectivamente. Si alguno no se proporciona, usará los ficheros por defecto (data/databases/McCulloch-Pitts.txt y out/out.txt).

p1.3.1-perceptron:
Ejecuta la simulación discutida en el apartado 3.1.1. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y OUTPUT_FILE, respectivamente. Si alguno no se proporciona, se utilizarán los ficheros por defecto (data/databases/problema_real1.txt y out/out.txt).

p1.3.1-adaline:
Ejecuta la simulación discutida en el apartado 3.1.2. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y OUTPUT_FILE, respectivamente. Si alguno no se proporciona, se utilizarán los ficheros por defecto (data/databases/problema_real1.txt y out/out.txt).

p1.3.2-nand:
Ejecuta la simulación discutida en el apartado 3.2 para la base de datos nand.txt. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y

OUTPUT_FILE, respectivamente. Si alguno no se proporciona, se utilizarán los ficheros por defecto (data/databases/nand.txt y out/out.txt).

p1.3.2-nor:

Ejecuta la simulación discutida en el apartado 3.2 para la base de datos nor.txt. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y OUTPUT_FILE, respectivamente. Si alguno no se proporciona, se utilizarán los ficheros por defecto (data/databases/nor.txt y out/out.txt).

p1.3.2-xor:

Ejecuta la simulación discutida en el apartado 3.2 para la base de datos xor.txt. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y OUTPUT_FILE, respectivamente. Si alguno no se proporciona, se utilizarán los ficheros por defecto (data/databases/xor.txt y out/out.txt).

p1.3.3-perceptron:

Ejecuta la simulación discutida en el apartado 3.3. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y OUTPUT_FILE, respectivamente. Si alguno no se proporciona, se utilizarán los ficheros por defecto (data/databases/problema_real2.txt y out/out.txt).

p1.3.3-adaline:

Ejecuta la simulación discutida en el apartado 3.3. Los ficheros de entrada y salida se pueden definir mediante INPUT_FILE y OUTPUT_FILE, respectivamente. Si alguno no se proporciona, se utilizarán los ficheros por defecto (data/databases/problema_real2.txt y out/out.txt).

p1.3.3.1-perceptron-predict:

Ejecuta la simulación discutida en el apartado 3.3.1. Primero entrena la red neuronal con la base de datos data/databases/problema_real2.txt y posteriormente clasifica los datos problema_real2_no_etiquetados.txt (o INPUT_FILE, si se proporciona). El fichero de salida con las predicciones se puede pasar por parámetro con OUTPUT_FILE o se utilizará out/out.txt.

p1.3.3.1-adaline-predict:

Ejecuta la simulación discutida en el apartado 3.3.1. Primero entrena la red neuronal con la base de datos data/databases/problema_real2.txt y posteriormente clasifica los datos problema_real2_no_etiquetados.txt (o INPUT_FILE, si se proporciona). El fichero de salida con las predicciones se puede pasar por parámetro con OUTPUT_FILE o se utilizará out/out.txt.

C.1.3 Argumentos

El makefile acepta los siguientes argumentos:

INPUT_FILE: Archivo del que se leerá. Para algunas simulaciones no es necesario porque las entradas fueron proporcionadas, pero es obligatorio en aquellos objetivos que no tienen una entrada prefijada.

OUTPUT_FILE: Archivo donde escribir las predicciones. Por defecto, siempre será out/out.txt

C.2. Neural-Network

La práctica genera un único ejecutable que admite distintos modos: generación de un clasificador, clasificación con un modelo ya entrenado o simulación de una red neuronal ya definida.

Todas las opciones se explican pasándole al programa el argumento --help, pero las detallamos a continuación según el caso de uso.

C.2.1 Casos de Uso

C.2.1.1 Entrenamiento

Este caso de uso contempla el entrenamiento de una red con datos previamente etiquetados y su posterior almacenamiento en un archivo.

Ejecución:

./neural-network -m MODE -i IN [-o OUT] [-n NETWORK -s] [-e EPOCH] [-t PERCENT]

Argumentos:

-m, --mode:
Puede ser “perceptron” o “adaline”, define el tipo de neurona que se empleará.

-i, --input-file:
Fichero con los datos de entrada. Parámetro obligatorio.

-o, --output-file:
Si se proporciona, escribe en el fichero OUT las predicciones, junto con la clase real de cada dato de test.

-s, --save:
Debe llamarse junto con -n. Si se proporciona, guarda en el archivo NETWORK los pesos finales de la red.

-n, --neural-network:
Si se llama sin el flag --save, carga la matriz inicial del archivo NETWORK.
Si se llama con el flag --save, guarda la matriz final en el archivo NETWORK.

-e, --max-epochs:
Si se proporciona, define una condición de parada por máximo número de épocas.

-t, --train-percent
Si se proporciona, sobreescribe el porcentaje por defecto de train por el proporcionado. Debe ser un entero. El porcentaje de test se calcula como 100-PERCENT. En el caso especial de que PERCENT sea 100, el porcentaje de test será también el total.

C.2.1.2 Predicción

Este caso de uso se contempla para predecir datos no etiquetados con una red que se ha guardado previamente según el caso de uso C.2.1.1, entrenamiento de una red neuronal.

Ejecución:

./neural-network -m MODE -n NETWORK -i IN -o OUT -f

Argumentos:

-m, --mode:

Puede ser “perceptron” o “adaline”, define el tipo de neurona que se empleará.

-n, --neural-network:

Fichero con la red neuronal ya entrenada a cargar.

-i, --input-file:

Fichero con los datos de entrada a clasificar. Parámetro obligatorio.

-o, --output-file:

Fichero donde escribir las clases predichas.