

Memoria Práctica 1

Neurocomputación - Ingeniería Informática

13 de marzo de 2021

Alejandro Santorum Varela
Sergio Galán Martín



Ciudad Universitaria de Cantoblanco
28049 Madrid
Tlf.: +34 91 497 50 00
Fax: +34 91 497 50 00
informacion.general@uam.es
<http://www.uam.es>

Índice

1 Introducción	1
2 Estructura de la carpeta de entrega	1
3 Descripción librería de Redes Neuronales	3
4 Descripción librería de lectura de datos	5
5 Neuronas de McCulloch-Pitts	6
5.1 Descripción e implementación	6
5.2 Ejercicio: implementando un circuito electrónico	7
6 Algoritmo de aprendizaje Perceptrón	9
6.1 Descripción e implementación	9
6.2 Ejercicio: entrenamiento con los problemas lógicos	11
7 Algoritmo de aprendizaje Adaline	12
7.1 Descripción e implementación	12
7.2 Ejercicio: entrenamiento con los problemas lógicos	14
8 Problema real 1	15
8.1 Algoritmo de aprendizaje Perceptrón	16
8.2 Algoritmo de aprendizaje Adaline	19
9 Problema real 2	23
9.1 Algoritmo de aprendizaje Perceptrón	23
9.2 Algoritmo de aprendizaje Adaline	24
10 Conclusiones	25

1. Introducción

El objetivo de esta práctica es diseñar e implementar distintos modelos neuronales artificiales tradicionales.

Para comenzar implementaremos, con un diseño orientado a objetos, ciertas **clases** que nos permitirán **conformar las distintas redes neuronales de las prácticas de esta asignatura**. La sección 3 explica dicho diseño e implementación.

A continuación nos centraremos en las **neuronas de McCulloch-Pitts**, que son unas estructuras neuronales artificiales sin proceso de aprendizaje, por lo que su funcionamiento depende principalmente del diseño del ingeniero (especificación de neuronas, conexiones, pesos, umbrales, etc.).

Finalmente, introduciremos dos algoritmos de aprendizaje de redes neuronales: **Perceptrón y Adaline**. Se implementarán ciertas redes neuronales que entrenen utilizando dichos algoritmos de aprendizaje y se probarán en algunos problemas didácticos (tablas de verdad de puertas lógicas) y en algunos problemas reales de mayor dificultad.

2. Estructura de la carpeta de entrega

La siguiente ilustración representa el contenido y la estructura de la carpeta de entrega. Con más detalle, dentro de la carpeta entregada podemos encontrar:

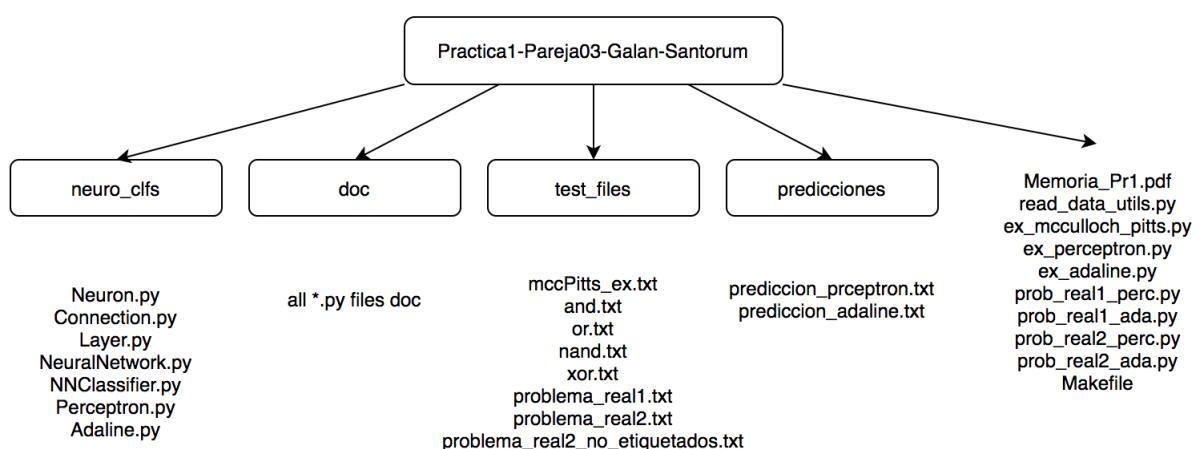


Figura 2.1: Estructura de la carpeta de la entrega

- **neuro_clfs**: directorio que contiene el módulo que implementa todas las clases necesarias para la elaboración de redes neuronales tradicionales. Para mayor información de cada una de estas clases, por favor, visitar la sección 3.

- **doc**: directorio donde podemos encontrar la documentación de todos los ficheros de Python de la entrega, que han sido generados utilizando la herramienta `pydoc`.
- **test_files**: directorio que posee todos los ficheros de datos que se requieren para la ejecución de esta práctica.
- **predicciones**: carpeta que contiene las predicciones del `problema_real2_no_etiquetados`. Para mayor detalle visitar la sección 9.
- *Scripts* de Python que resuelven los problemas de la práctica:
 - `read_data_utils.py`: fichero que contiene la implementación de las funciones de lectura de datos. Explicado con mayor detalle en la sección 4.
 - `ex_mcculloch_pitts.py`: fichero que resuelve el ejercicio 1 de la práctica con una red de McCulloch-Pitts. Se debe ejecutar con el nombre del fichero con las entradas binarias correspondientes. Véase sección 5.
 - `ex_perceptron.py`: fichero que resuelve los problemas lógicos utilizando el algoritmo de aprendizaje Perceptrón. Véase sección 6, sobre todo para conocer los **parámetros de entrada necesarios**.
 - `ex_adaline.py`: fichero que resuelve los problemas lógicos utilizando el algoritmo de aprendizaje Adaline. Véase sección 7, sobre todo para conocer los **parámetros de entrada necesarios**.
 - `prob_real1_perc.py`: El fichero `ex_perceptron.py` resuelve el problema real 1, devolviendo todas las predicciones, la frontera de decisión obtenida y el ECM. No obstante, este *script* no nos permite **variar los hiperparámetros** (utilizando *GridSearch*) para obtener el **mejor resultado** posible. Con este objetivo en mente, se ha implementado el *script* `prob_real1_perc.py`, que permite tanto ejecutar el algoritmo Perceptrón en el problema real 1 con unos parámetros dados, o realizar una búsqueda de los mejores hiperparámetros. Para seleccionar el funcionamiento, es importante conocer el **formato de los argumentos de entrada**. Véase sección 8.1.
 - `prob_real1_ada.py`: El fichero `ex_adaline.py` es capaz de resolver el problema real 1, devolviendo todas las predicciones, la frontera de decisión obtenida y el ECM. No obstante, este *script* no nos permite **variar los hiperparámetros** (utilizando *GridSearch*) para obtener el **mejor resultado** posible. Con este objetivo en mente, se ha implementado el *script* `prob_real1_ada.py`, que permite tanto ejecutar el algoritmo Adaline en el problema real 1 con unos parámetros dados, o realizar una búsqueda de los mejores hiperparámetros. Para seleccionar el funcionamiento, es importante conocer el **formato de los argumentos de entrada**. Véase sección 8.2.
 - `prob_real2_perc.py`: Este fichero permite resolver el problema real 2. Para ello, y **dependiendo de los argumentos de entrada**, se realizará una búsqueda de hiperparámetros utilizando el **fichero con datos etiquetados**.

Una vez tengamos los mejores hiperámetros podremos llamar de nuevo a este *script* especificando estos valores como entrada. Se obtendrá como salida un fichero de texto en la carpeta **predicciones** con las **predicciones del problema real 2 del fichero no etiquetado**. Véase sección 9.1.

- `prob_real2_ada.py`: Análogo al *script* anterior pero esta vez utilizando el algoritmo de aprendizaje Adaline. Obsérvese y téngase cuidado con los argumentos de entrada, ya que los hiperparámetros del algoritmo Perceptrón son diferentes que los del algoritmo Adaline, por lo que los argumentos de entrada de los *scripts* mencionados. Para mayor información, sección 9.2.
- `Makefile` con los comandos especificados por el enunciados de la práctica. Adicionalmente, contiene otros comandos que permiten realizar tareas útiles: generación del `pydoc`, lista de requisitos e instalación de los mismos.

3. Descripción librería de Redes Neuronales

Uno de los objetivos de esta práctica es diseñar e implementar distintos modelos neuronales artificiales tradicionales. Para facilitar el desarrollo de esta práctica y las posteriores, lo más razonable es plantear el diseño de una librería que permita gestionar los valores internos de las neuronas, agruparlas en capas y realizar sinapsis. A continuación se expone el diagrama UML propuesto por el equipo docente, en el que hemos basado el diseño e implementación de nuestra librería.

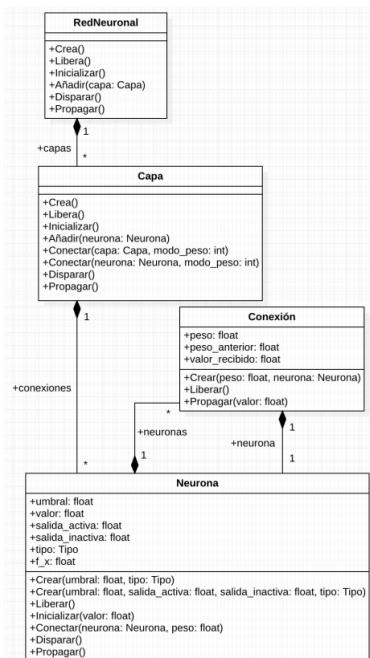


Figura 3.1: Especificación UML para la librería de manejo de redes neuronales

Ahora explicamos ligeramente cada clase, y sus decisiones de diseño más relevantes.

■ Clase Connection:

- Fichero Connection.py
- Atributos: el peso actual, el peso anterior, la neurona de llegada de la conexión y el valor recibido de la conexión.
- Métodos importantes: `propagate()` calcula el valor de entrada de la neurona a la que esta conexión está conectada, multiplicando el peso actual por el valor recibido de la conexión; `any_weight_update()` comprueba si en la última actualización se modificaron los pesos; y `update_weight(term)`, que actualiza el peso actual dado un término: $w = w + term$.

■ Clase Neuron:

- Fichero Neuron.py
- Atributos: valor, activación, conjunto de conexiones que parten de esta neurona, valor de salida activa, valor de salida inactiva y tipo (McCulloch, Directa, Bias, etc).
- Métodos importantes: `trigger()` calcula el valor de activación de la neurona a partir del valor actual de la misma; `propagate()` propaga el valor de activación de la neurona a las conexiones salientes; `any_weight_update()` comprueba si en la última actualización se modificaron los pesos de alguna conexión; `connect(Neuron n, weight)` conecta la neurona actual con otra, creando una conexión con un peso dado.

■ Clase Layer:

- Fichero Layer.py
- Atributos: conjunto de neuronas que posee esta capa.
- Métodos importantes: `add(Neuron n)` añade una neurona a la capa; `trigger()` calcula el valor de activación de las neuronas de la capa; `propagate()` propaga el valor de activación de las neuronas de la capa a las conexiones salientes de las mismas; `connectLayer(Layer l, weight_mode)` conecta la capa actual con otra capa dada con ciertos pesos; `any_weight_update()` comprueba si en la última actualización se modificaron los pesos de alguna neurona y/o conexión de la capa.

■ Clase NeuralNetwork:

- Fichero NeuralNetwork.py
- Atributos: conjunto de capas que conforma la red neuronal.

- **Métodos importantes:** `add(Layer 1)` añade una capa a la red; `any_weight_update()` comprueba si en la última actualización se modificaron los pesos de alguna neurona y/o conexión de la red; `trigger()` calcula el valor de activación de todas las neuronas de la red; `propagate()` propaga el valor de activación de las neuronas de la red a las conexiones salientes de las mismas; `get_output()` calcula el valor de salida de la red en este instante de tiempo. Si la red tiene varias salidas, devuelve un `array` con todos los valores de salida.
- Clase `NNClassifier` (abstracta):
 - Fichero `NNClassifier.py`
 - Atributos: ninguno.
 - **Métodos importantes:** `train(xtrain, ytrain)` entrena la red neuronal dado un conjunto de datos de entrenamiento (`xtrain` valores de entrada de entrenamiento, `ytrain` valores objetivo de entrenamiento). Las clases que hereden de esta clase deberán implementar su algoritmo de entrenamiento explícitamente. `predict(xtest)` calcula las salidas predichas por la red después de una fase de entrenamiento dadas unas entradas. Las clases que hereden de esta clase deberán implementar su algoritmo de predicción basándose en su fase de explotación. `score(xtest, ytest)` calcula la precisión de la red dados unos datos de `test`. Finalmente, `error(ytrue, ypred, metric)` calcula el error de la red dados unos objetivos y unas predicciones. El error puede ser el error de precisión ($1 - \% \text{precisión}$), o el Error Cuadrático Medio:

$$ECM(y, t) = \frac{1}{m} \sum_{i=1}^m (y_i - t_i)^2, \text{ donde } m = \text{número de ejemplos.}$$

- Observación: de esta clase heredarán la mayoría de las redes neuronales que veremos a lo largo de las prácticas, como por ejemplo la red Perceptrón (sección 6) o la red Adaline (sección 7).

4. Descripción librería de lectura de datos

Para poder leer los datos suministrados de manera **sencilla** y **modular** y poder generar el **conjunto de entrenamiento** (*training set*) y el **conjunto de prueba** (*test set*), soportando además los diferentes modos de lectura de *datasets* sugeridos en el enunciado de la práctica, hemos implementado varias funciones auxiliares que se pueden encontrar en el fichero `read_data_utils.py`. La primera función es `parse_read_mode()`, que nos permite obtener el modo de lectura solicitado por el usuario, además de los parámetros adicionales que se deben suministrar para cada uno de los modos.

Tras ello se encuentran **tres** funciones, una correspondiente a cada modo de lectura de *datasets*. En el **modo 1** (`read1(data_file, perc)`) tanto el conjunto de entrena-

miento como el de prueba se obtienen a partir de un único fichero (`data`), especificando adicionalmente un porcentaje (`perc`) que indicará la proporción de ejemplos que se usarán como conjunto de entrenamiento, dejando el resto como conjunto de prueba. Además, los ejemplos concretos que irán a cada conjunto se elegirán aleatoriamente, lo que variará ligeramente las salidas a cada ejecución. El **modo 2** (`read2(data_file)`) es algo más simple que el anterior, siendo el conjunto de entrenamiento y el de pruebas el mismo, el conjunto de todos los ejemplos del fichero (`data_file`). Es decir, en este modo se entrenará la red neuronal con todos los ejemplos del fichero suministrado y se comprobará la precisión de su predicción con los mismos ejemplos. Es importante destacar que, aunque este modo genere una proporción de aciertos muy alta, **es probable que sea poco generalizable**, y al suministrarle nuevos datos su precisión baje considerablemente. En el **modo 3** (`read3(train_file, test_file)`) se suministran dos ficheros independientes, uno de ellos conteniendo el conjunto de entrenamiento (`train_file`) y otro el conjunto de prueba (`test_file`), con lo que **mitigamos en cierta medida** el problema mencionado en el modo 2.

Una vez los conjuntos de entrenamiento y de prueba han sido generados en base a los parámetros suministrados por estas funciones, esos conjuntos serán introducidos en los diferentes modelos de redes neuronales implementados, se entrenarán con el conjunto de entrenamiento y se comprobará su eficacia con el conjunto de prueba usando **diferentes métricas** (porcentaje de aciertos, error cuadrático medio...).

5. Neuronas de McCulloch-Pitts

5.1. Descripción e implementación

En una neurona típica de McCulloch–Pitts la activación es binaria, es decir, su salida es 0 o 1. Las conexiones excitadoras que llegan a una misma neurona tienen el mismo peso, el umbral se escoge de forma que una sola conexión inhibidora fuerce a que la salida de la neurona sea 0 en ese paso y las salidas tardan un intervalo de tiempo en llegar a las neuronas receptoras.

Todas estas propiedades hacen que las neuronas de McCulloch–Pitts sean útiles para modelar fenómenos que requieran **funciones lógicas y retrasos temporales**.

Implementaremos esta red creando neuronas del tipo `McCulloch`. Para que la activación sea binaria estableceremos el valor de la salida activa a 1, y el valor de la salida inactiva a 0. Deberemos conectar dichas neuronas teniendo en cuenta los pesos de las conexiones, recordando que el umbral se escoge de forma que una sola conexión inhibidora fuerce a que la salida de la neurona sea 0.

Comentar que las neuronas de McCulloch–Pitts no tienen proceso de aprendizaje, por lo que la funcionalidad de estas redes vendrá determinada por el diseño inicial del ingeniero.

5.2. Ejercicio: implementando un circuito electrónico

Para exemplificar las redes que incorporan neuronas de McCulloch-Pitts, intentaremos resolver el primer problema de la práctica, que consiste en imitar el comportamiento del circuito electrónico de la figura 5.1.

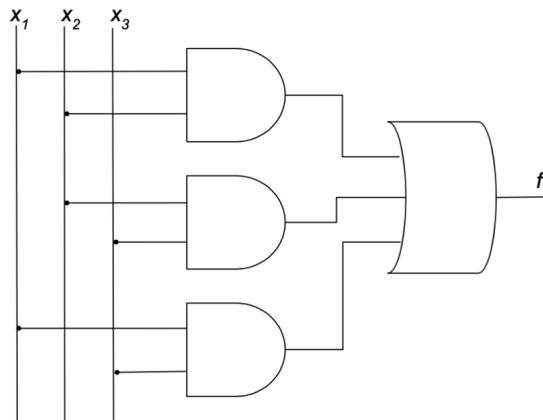


Figura 5.1: Función lógica a implementar con neuronas de McCulloch-Pitts

No es difícil implementar la función lógica f si tenemos en mente la codificación de funciones más simples (función *and* y *or*) mediante redes de McCulloch-Pitts:

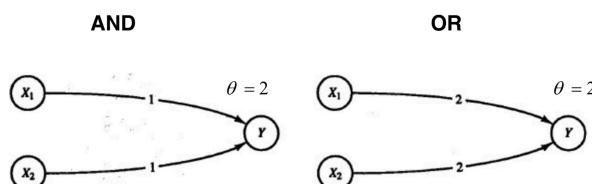


Figura 5.2: Funciones *and* y *or* usando redes de McCulloch-Pitts

En el circuito 5.1 podemos ver que hay tres puertas lógicas *and* (una para cada par de entradas), las cuales a su vez están conectadas a una puerta lógica *or* de 3 entradas. Su salida es la función lógica f .

El diseño propuesto de red neuronal de McCulloch-Pitts es la descrita en la representación de la figura (5.3).

Las neuronas x_1 , x_2 y x_3 son neuronas directas, es decir, su valor de activación es directamente el valor de entrada. Por el contrario, las neuronas a_{12} , a_{13} , a_{23} e Y son neuronas de McCulloch-Pitts, cuya salida activa tiene valor 1 y su salida inactiva 0.

La funcionalidad del circuito se basa principalmente en el diseño de las conexiones, sus pesos y los umbrales de las neuronas. Tomando como referencia las puertas *and* y *or* (5.2), podemos implementar dicho circuito. Las conexiones a las neuronas de la capa oculta (a_{12} , a_{13} , a_{23}) tendrán peso 1 cuando queramos conectarlas con la capa de entrada, y cero en caso contrario.

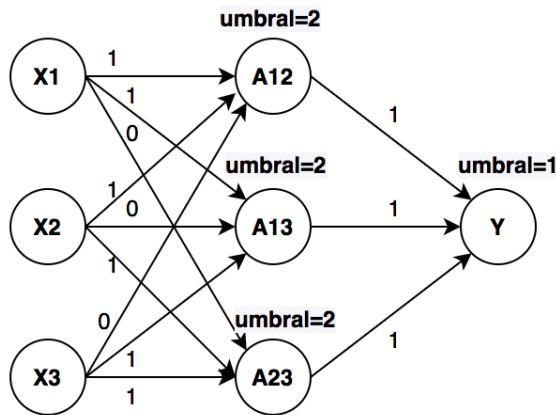


Figura 5.3: Red de McCulloch-Pitts que implementa la función lógica f

Observar que no eran necesarias las conexiones cuyo peso es cero, ya que nunca van a trasmitir una "señal". No obstante se ha incluido para resaltar el hecho de que la neurona de llegada ejecuta una función *and* de dos puertas.

Si tenemos un vector de entrada con al menos dos unos, por ejemplo $(x_1, x_2, x_3) = (1, 1, 0)$, la neurona **a12** se activará (su salida será 1) ya que $x_1 \cdot 1 + x_2 \cdot 1 + x_3 \cdot 0 = 1 + 1 = 2$ y el umbral de **a12** es $\theta = 2$. El comportamiento es análogo para las neuronas **a13** y **a23**.

Finalmente, para la misma entrada de ejemplo anterior, la salida de la neurona **Y** también será 1 ya que $a_{12} \cdot 1 + a_{13} \cdot 1 + a_{23} \cdot 1 = 1 \cdot 1 + 0 \cdot 1 + 0 \cdot 1 = 1$ y el umbral es $\theta = 1$. La neurona **Y**, teniendo en cuenta los pesos de las conexiones entrantes y el umbral, simula una puerta lógica *or* de tres entradas.

No es complicado notar que esta red **necesita dos pasos de tiempo para producir una salida a partir de una entrada**. Adicionalmente, tanto analizando la función lógica f de 5.1 como la red neuronal de McCulloch-Pitts 5.3, podemos ver que $f(t+2) = 1$ si hay al menos dos entradas (**x1**, **x2**, **x3**) activadas, es decir, si al menos dos entradas tienen valor 1 en tiempo t . En caso contrario, la salida será cero.

Para corroborar lo dicho hasta el momento se ha programado el *script* de Python `ex_mcculloch_pitts.py`, que recibe como argumento de entrada un fichero con las entradas **x1**, **x2** y **x3** y produce, después de dos pasos de tiempo, la salida esperada:

```

(base) SantorumPC:assignment1 santorum$ python3 ex_mcculloch_pitts.py test_files/mccPitts_ex.txt
+-----+
| x1 | x2 | x3 | a12 | a13 | a23 | y |
+-----+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 0 |
| 1 | 1 | 0 | 0 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
+-----+
(base) SantorumPC:assignment1 santorum$ 
```

Figura 5.4: Salida de `ex_mcculloch_pitts.py` con las entradas especificadas

Podemos ejecutar el mismo comando mostrado anteriormente o, si se desea, utilizar el Makefile de soporte:

```
(base) SantorumPC:assignment1 santorum$ make ayuda_mc
===== AYUDA PARA EL PROGRAMA: RED DE MCCULLOCH-PITTS =====
·Nombre del programa ejecutable de Python3: ex_mcculloch_pitts.py
·Posibles parámetros de entrada:
  - (obligatorio) ruta fichero de lectura donde se especifican las entradas del ejercicio
  ·EJEMPLO DE USO: python3 ex_mcculloch_pitts.py test_files/mccPitts_ex.txt
=====
(base) SantorumPC:assignment1 santorum$
```

Figura 5.5: Comando de ayuda `make ayuda_mc`

```
(base) SantorumPC:assignment1 santorum$ make ejecuta_mc
+-----+
| x1 | x2 | x3 | a12 | a13 | a23 | y |
+-----+
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 | 0 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 0 | 0 | 1 |
| 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| 0 | 0 | 0 | 0 | 0 | 0 | 1 |
+-----+
(base) SantorumPC:assignment1 santorum$
```

Figura 5.6: Comando de ejecución de ejemplo `make ejecuta_mc`

Se añade, junto con el resto de ficheros de prueba *txt*, el fichero *mccPitts_ex.txt*, con las entradas **x1**, **x2** y **x3** especificadas en el enunciado de la práctica.

6. Algoritmo de aprendizaje Perceptrón

6.1. Descripción e implementación

El **primer** algoritmo de aprendizaje de redes neuronales que vamos a implementar en esta práctica es el **algoritmo de aprendizaje Perceptrón**. Durante el aprendizaje, los patrones (entradas) son introducidos a la red neuronal varias veces de forma iterativa. El algoritmo varía los pesos de las conexiones de la red conforme a una constante de proporcionalidad de aprendizaje α (constante de aprendizaje), que suele ser $\alpha \leq 1$, con el objetivo de mejorar la clasificación de la red, es decir, reducir la diferencia entre las salidas objetivo y las salidas predichas.

Con hipótesis específicas, se puede probar formalmente que los pesos convergen durante el entrenamiento. A continuación exponemos el algoritmo básico Perceptrón: red neuronal con una capa y una única neurona de salida.

Paso 0: Inicializar todos los pesos y sesgos (por simplicidad a cero)

Establecer la tasa de aprendizaje α ($0 < \alpha \leq 1$)

Paso 1: Mientras que la condición de parada sea falsa, ejecutar pasos 2-6

Paso 2: Para cada par de entrenamiento ($s:t$), ejecutar los pasos 3-5:

Paso 3: Establecer las activaciones a las neuronas de entrada

$$x_i = s_i \quad (i=1 \dots n)$$

Paso 4: Calcular la respuesta de la neurona de salida:

$$y_in = b + \sum_i x_i w_i$$

$$f(y_in) = \begin{cases} 1 & \text{si } y_in > \theta \\ 0 & \text{si } -\theta \leq y_in \leq \theta \\ -1 & \text{si } y_in < -\theta \end{cases}$$

Paso 5: Ajustar los pesos y el sesgo si ha ocurrido un error para este

patrón: Si $y \neq t$

$$w_i(\text{nuevo}) = w_i(\text{anterior}) + \alpha t x_i$$

$$b(\text{nuevo}) = b(\text{anterior}) + \alpha t$$

Si no

$$w_i(\text{nuevo}) = w_i(\text{anterior})$$

$$b(\text{nuevo}) = b(\text{anterior})$$

Paso 6: Comprobar la condición de parada: si no han cambiado los pesos en el paso 2: parar; en caso contrario, continuar.

Figura 6.1: Algoritmo de aprendizaje Perceptrón

En esta práctica no buscaremos ampliar el algoritmo a redes multicapa, aunque eso no quiera decir que sea difícil. Adicionalmente, podemos generalizar el algoritmo en el paso 5 para poder tener varias neuronas de salida (cuando $y_j \neq t_j$):

$$w_{ij} = w_{ij} + \alpha t_j x_i$$

$$b_j = b_j + \alpha t_j$$

$$\forall j \in \{0, \dots, m\}, \quad \forall i \in \{1, \dots, n\},$$

donde n = número de entradas, m = número de salidas.

En cuanto a la implementación, se ha programado una clase `Perceptron` que hereda ciertos métodos de la clase abstracta `NNClassifier`, explicada con detalle en la sección 3.

Es importante notar que la clase `Perceptron` tiene que especificar los métodos `train` y `predict`, que se diseñan y se implementan siguiendo el pseudocódigo (6.1). Adicionalmente, se ha mejorado el paso 5 para permitir tener varias salidas, cuyo número deberá ser especificado al constructor de la clase.

6.2. Ejercicio: entrenamiento con los problemas lógicos

Mostraremos a continuación las salidas de ejecutar nuestra implementación del Perceptrón a los problemas lógicos cuyos datos se encuentran en los ficheros `and.txt`, `or.txt`, `nand.txt` y `xor.txt` suministrados con el enunciado de la práctica.

```
(base) SantorumPC:assignment1 santorum$ python3 ex_perceptron.py 2 test_files/and.txt
+-----+
| x1 | x2 | t (target) | y (predicted) |
+-----+
| 1.0 | 1.0 | 1.0 | 1 |
| 1.0 | 0.0 | -1.0 | -1 |
| 0.0 | 1.0 | -1.0 | -1 |
| 0.0 | 0.0 | -1.0 | -1 |
+-----+
MSE Loss: 0.0
Decision boundary: 2.0·X1 + 3.0·X2 + -4.0 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 6.2: Aprendizaje función AND usando perceptrón

```
(base) SantorumPC:assignment1 santorum$ python3 ex_perceptron.py 2 test_files/or.txt
+-----+
| x1 | x2 | t (target) | y (predicted) |
+-----+
| 1.0 | 1.0 | 1.0 | 1 |
| 1.0 | 0.0 | 1.0 | 1 |
| 0.0 | 1.0 | 1.0 | 1 |
| 0.0 | 0.0 | -1.0 | -1 |
+-----+
MSE Loss: 0.0
Decision boundary: 2.0·X1 + 2.0·X2 + -1.0 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 6.3: Aprendizaje función OR usando perceptrón

```
(base) SantorumPC:assignment1 santorum$ python3 ex_perceptron.py 2 test_files/nand.txt
+-----+
| x1 | x2 | t (target) | y (predicted) |
+-----+
| 0.0 | 0.0 | 1.0 | 1 |
| 0.0 | 1.0 | 1.0 | 1 |
| 1.0 | 0.0 | 1.0 | 1 |
| 1.0 | 1.0 | -1.0 | -1 |
+-----+
MSE Loss: 0.0
Decision boundary: -3.0·X1 + -2.0·X2 + 4.0 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 6.4: Aprendizaje función NAND usando perceptrón

```
(base) SantorumPC:assignment1 santorum$ python3 ex_perceptron.py 2 test_files/xor.txt
+-----+
| x1 | x2 | t (target) | y (predicted) |
+-----+
| 1.0 | 1.0 | -1.0 | 0 |
| 1.0 | 0.0 | 1.0 | 0 |
| 0.0 | 1.0 | 1.0 | 0 |
| 0.0 | 0.0 | -1.0 | 0 |
+-----+
MSE Loss: 1.0
Decision boundary: 0.0·X1 + 0.0·X2 + 0.0 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 6.5: Aprendizaje función XOR usando perceptrón

Como podemos ver, el Perceptrón es capaz de aprender las funciones AND, OR y NAND, aunque no es capaz de aprender la función XOR. Añadimos la representación gráfica de las rectas de separación encontradas por el Perceptrón para los tres primeros casos (en el XOR no ha encontrado ninguna recta válida).

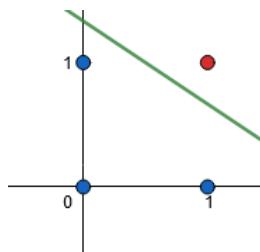


Figura 6.6: Recta AND usando perceptrón

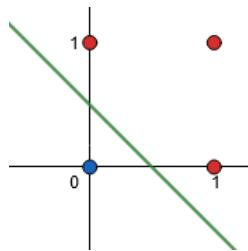


Figura 6.7: Recta OR usando perceptrón

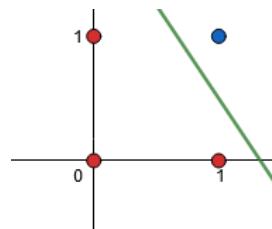


Figura 6.8: Recta NAND usando perceptrón

7. Algoritmo de aprendizaje Adaline

7.1. Descripción e implementación

El **segundo** algoritmo de aprendizaje de redes neuronales que vamos a implementar en esta práctica es el **algoritmo de aprendizaje Adaline**. De manera similar al Perceptrón, los patrones (entradas) son introducidos durante el aprendizaje a la red neuronal varias veces de forma iterativa. La principal diferencia entre ambos es la regla de actualización de los pesos, que en este caso es la que se conoce como **regla delta**, dependiente también de una constante de aprendizaje α , habitualmente pequeño.

Como con el Perceptrón, se puede probar matemáticamente la convergencia del método a unos pesos concretos que **minimizarán el ECM** de todas las posibles separaciones con rectas. Además, si el problema es **linealmente separable**, podemos asegurar que el Adaline encontrará una solución con 100 % de acierto.

Paso 0: Inicializar todos los pesos y sesgos (valores aleatorios pequeños)
Establecer la tasa de aprendizaje α .

Paso 1: Mientras que la condición de parada sea falsa, ejecutar pasos 2-6

Paso 2: Para cada par de entrenamiento ($s:t$) bipolar, ejecutar los pasos 3-5:

Paso 3: Establecer las activaciones a las neuronas de entrada

$$x_i = s_i \quad (i=1 \dots n)$$

Paso 4: Calcular la respuesta de la neurona de salida:

$$y_in = b + \sum_i x_i w_i$$

Paso 5: Ajustar los pesos y el sesgo:

$$w_i(\text{nuevo}) = w_i(\text{anterior}) + \alpha(t-y_in)x_i$$

$$b(\text{nuevo}) = b(\text{anterior}) + \alpha(t-y_in)$$

Paso 6: Comprobar la condición de parada: si el cambio de peso más grande en el paso 2 es menor que una tolerancia especificada: parar; en caso contrario, continuar.

Figura 7.1: Algoritmo de aprendizaje Adaline

Análogamente al Perceptrón podemos modificar el algoritmo para añadir un número arbitrario de capas de salida calculando sus valores de salida y_{in_j} en el paso 4 y modificando el paso 5:

$$w_{ij} = w_{ij} + \alpha(t_j - y_{in_j})x_i,$$

$$b_j = b_j + \alpha(t_j - y_{in_j}),$$

$$\forall j \in \{0, \dots, m\} \quad \forall i \in \{0, \dots, n\},$$

donde

n = número de entradas, m = número de salidas.

En cuanto a la implementación, se ha implementado la clase Adaline que hereda ciertos métodos de la clase abstracta `NNClassifier`, explicada con detalle en la sección 3.

Es importante notar que la clase `Adaline` tiene que implementar los métodos `train` y `predict`, que se diseñan siguiendo el pseudocódigo (7.1). Adicionalmente, se ha mejorado el paso 5 para **permitir tener varias salidas**, cuyo número deberá ser especificado al constructor de la clase.

7.2. Ejercicio: entrenamiento con los problemas lógicos

Mostraremos a continuación las salidas de ejecutar nuestra implementación del Adaline a los problemas lógicos cuyos datos se encuentran en los ficheros `and.txt`, `or.txt`, `nand.txt` y `xor.txt` suministrados con el enunciado de la práctica.

```
(base) SantorumPC:assignment1 santorum$ python3 ex_adaline.py 2 test_files/and.txt
+---+---+---+---+
| x1 | x2 | t1 | y1 |
+---+---+---+---+
| 1.0 | 1.0 | 1.0 | 1 |
| 1.0 | 0.0 | -1.0 | -1 |
| 0.0 | 1.0 | -1.0 | -1 |
| 0.0 | 0.0 | -1.0 | -1 |
+---+---+---+---+
MSE Loss: 0.0
Decision boundary: 0.93·X1 + 0.99·X2 + -1.49 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 7.2: Aprendizaje función AND usando adaline

```
(base) SantorumPC:assignment1 santorum$ python3 ex_adaline.py 2 test_files/or.txt
+---+---+---+---+
| x1 | x2 | t1 | y1 |
+---+---+---+---+
| 1.0 | 1.0 | 1.0 | 1 |
| 1.0 | 0.0 | 1.0 | 1 |
| 0.0 | 1.0 | 1.0 | 1 |
| 0.0 | 0.0 | -1.0 | -1 |
+---+---+---+---+
MSE Loss: 0.0
Decision boundary: 1.04·X1 + 0.99·X2 + -0.49 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 7.3: Aprendizaje función OR usando adaline

```
(base) SantorumPC:assignment1 santorum$ python3 ex_adaline.py 2 test_files/nand.txt
+---+---+---+---+
| x1 | x2 | t1 | y1 |
+---+---+---+---+
| 0.0 | 0.0 | 1.0 | 1 |
| 0.0 | 1.0 | 1.0 | 1 |
| 1.0 | 0.0 | 1.0 | 1 |
| 1.0 | 1.0 | -1.0 | -1 |
+---+---+---+---+
MSE Loss: 0.0
Decision boundary: -1.1·X1 + -1.04·X2 + 1.54 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 7.4: Aprendizaje función NAND usando adaline

```
(base) SantorumPC:assignment1 santorum$ python3 ex_adaline.py 2 test_files/xor.txt
+---+---+---+---+
| x1 | x2 | t1 | y1 |
+---+---+---+---+
| 1.0 | 1.0 | -1.0 | 1 |
| 1.0 | 0.0 | 1.0 | 1 |
| 0.0 | 1.0 | 1.0 | -1 |
| 0.0 | 0.0 | -1.0 | -1 |
+---+---+---+---+
MSE Loss: 2.0
Decision boundary: 0.12·X1 + 0.01·X2 + -0.01 = 0
(base) SantorumPC:assignment1 santorum$
```

Figura 7.5: Aprendizaje función XOR usando adaline

Como podemos ver, el Adaline también aprende las funciones AND, OR y NAND, aunque tampoco es capaz de aprender la función XOR. Añadimos la representación gráfica de las rectas de separación encontradas por el Adaline (en este caso sí se ha encontrado una recta para la función XOR pero no muy útil).

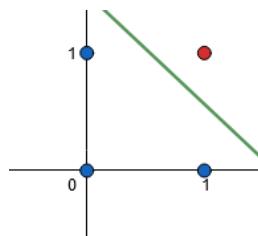


Figura 7.6: Recta función AND usando adaline

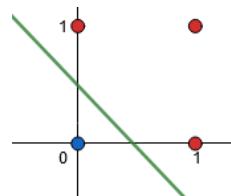


Figura 7.7: Recta función OR usando adaline

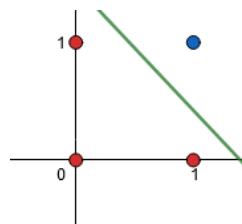


Figura 7.8: Recta función NAND usando adaline

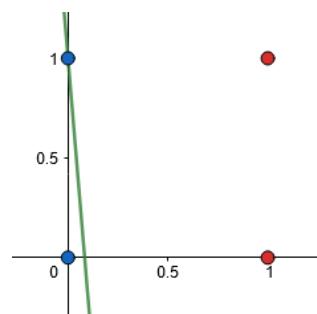


Figura 7.9: Recta función XOR usando adaline

8. Problema real 1

Los *scripts* `ex_perceptron.py` y `ex_adaline.py` pueden ser ejecutados utilizando el fichero de datos del problema real 1. No obstante, estos programas tienen como salida la tabla de datos, las salidas objetivo y las salidas predichas tras la ejecución del algoritmo. Esto no es práctico para resolver el problema real 1 (y próximamente el 2), ya que estaríamos más interesados únicamente en el ECM o en la precisión de clasificación.

Por lo tanto, extenderemos la funcionalidad de ambos algoritmos para adaptarlos a problemas más exigentes que los lógicos (que además no conseguirán proporcionar una solución óptima que clasifique correctamente el 100 % de los datos) y se le añadirá cierta funcionalidad para ver gráfica y numéricamente la evolución del ECM a través de las épocas de cada algoritmo y la influencia de los parámetros escogidos en el resultado final.

Finalmente, los nuevos *scripts* orientados a los problemas reales tendrán la posibilidad de ejecutarse realizando una búsqueda de los mejores hiperparámetros o *Grid Search*.

8.1. Algoritmo de aprendizaje Perceptrón

Hemos extendido el código del Perceptrón usado para resolver las funciones lógicas **añadiéndole optimización de hiperparámetros** mediante búsqueda en rejilla con el comando `python3 prob_real1_perc.py -hyper` (esto es, la búsqueda de parámetros óptimos dentro de un espacio de parámetros predeterminado $\alpha \in [0,01, 0,05, 0,1, 0,5, 1]$, $umbral \in [0, 0,25, 0,5, 1]$) e implementando la parada del algoritmo tras un número máximo de épocas `max_epoch` (ya que al no ser el problema linealmente separable no va a conseguir clasificar el 100 % de los datos correctamente).

Tras ejecutar la optimización de hiperparámetros hemos encontrado que la pareja de α y `threshold` que **minimiza el ECM manteniendo una alta probabilidad de acierto** es $\alpha = 0,5$ y `threshold = 0` (como podemos ver en la figura 8.1), siendo estos los valores que hemos establecido como **predeterminados** si se ejecuta el programa sin especificar manualmente dichos valores. Adicionalmente, en la imagen 8.3 podemos ver que usando los mejores hiperparámetros obtenemos un $ECM = 0,27 \pm 0,02$ y una **precisión de clasificación sobre el 97 %**.

Si se quiere especificar manualmente algún parámetro se deberá llamar al programa con el comando:

```
python3 prob_real1_perc.py -a alpha -th umbral -nreps num_reps -nep max_epoch,
```

donde `num_reps` es el número de iteraciones del algoritmo de aprendizaje del Perceptrón (tras lo cual calcularemos el ECM medio y la precisión media de las iteraciones) y `max_epoch` es el máximo número de épocas en cada iteración del algoritmo.

```
(base) SantorumPC:assignment1 santorum$ python3 prob_reali_perc.py -hyper
Alpha: 0.01 Threshold: 0 ---> mse: 0.34743 , acc: 0.957
Alpha: 0.05 Threshold: 0 ---> mse: 0.32914 , acc: 0.959
Alpha: 0.1 Threshold: 0 ---> mse: 0.34286 , acc: 0.957
Alpha: 0.5 Threshold: 0 ---> mse: 0.32457 , acc: 0.959
Alpha: 1 Threshold: 0 ---> mse: 0.60343 , acc: 0.925
Alpha: 0.01 Threshold: 0.25 ---> mse: 0.24114 , acc: 0.907
Alpha: 0.05 Threshold: 0.25 ---> mse: 0.26514 , acc: 0.922
Alpha: 0.1 Threshold: 0.25 ---> mse: 0.29829 , acc: 0.925
Alpha: 0.5 Threshold: 0.25 ---> mse: 0.25714 , acc: 0.954
Alpha: 1 Threshold: 0.25 ---> mse: 0.33371 , acc: 0.95
Alpha: 0.01 Threshold: 0.5 ---> mse: 0.28457 , acc: 0.895
Alpha: 0.05 Threshold: 0.5 ---> mse: 0.25143 , acc: 0.921
Alpha: 0.1 Threshold: 0.5 ---> mse: 0.24229 , acc: 0.93
Alpha: 0.5 Threshold: 0.5 ---> mse: 0.31771 , acc: 0.937
Alpha: 1 Threshold: 0.5 ---> mse: 0.34629 , acc: 0.942
Alpha: 0.01 Threshold: 1 ---> mse: 0.288 , acc: 0.889
Alpha: 0.05 Threshold: 1 ---> mse: 0.29257 , acc: 0.902
Alpha: 0.1 Threshold: 1 ---> mse: 0.256 , acc: 0.913
Alpha: 0.5 Threshold: 1 ---> mse: 0.28229 , acc: 0.931
Alpha: 1 Threshold: 1 ---> mse: 0.39886 , acc: 0.927
+-----+-----+-----+-----+-----+
| Thresholds \ Alphas | 0.01 | 0.05 | 0.1 | 0.5 | 1 |
+=====+=====+=====+=====+=====+
| 0 | 0.34743 +- 0.16 | 0.32914 +- 0.117 | 0.34286 +- 0.183 | 0.32457 +- 0.147 | 0.60343 +- 0.578 |
+-----+-----+-----+-----+-----+
| 0.25 | 0.24114 +- 0.053 | 0.26514 +- 0.105 | 0.29829 +- 0.096 | 0.25714 +- 0.099 | 0.33371 +- 0.153 |
+-----+-----+-----+-----+-----+
| 0.5 | 0.28457 +- 0.032 | 0.25143 +- 0.056 | 0.24229 +- 0.064 | 0.31771 +- 0.086 | 0.34629 +- 0.227 |
+-----+-----+-----+-----+-----+
| 1 | 0.288 +- 0.049 | 0.29257 +- 0.071 | 0.256 +- 0.087 | 0.28229 +- 0.081 | 0.39886 +- 0.3 |
+-----+-----+-----+-----+-----+
+-----+-----+-----+-----+-----+
| Thresholds \ Alphas | 0.01 | 0.05 | 0.1 | 0.5 | 1 |
+=====+=====+=====+=====+=====+
| 0 | 0.957 +- 0.02 | 0.959 +- 0.01 | 0.957 +- 0.02 | 0.959 +- 0.02 | 0.925 +- 0.07 |
+-----+-----+-----+-----+-----+
| 0.25 | 0.907 +- 0.01 | 0.922 +- 0.02 | 0.925 +- 0.03 | 0.954 +- 0.02 | 0.95 +- 0.02 |
+-----+-----+-----+-----+-----+
| 0.5 | 0.895 +- 0.01 | 0.921 +- 0.01 | 0.93 +- 0.02 | 0.937 +- 0.02 | 0.942 +- 0.03 |
+-----+-----+-----+-----+-----+
| 1 | 0.889 +- 0.01 | 0.902 +- 0.02 | 0.913 +- 0.03 | 0.931 +- 0.02 | 0.927 +- 0.05 |
+-----+-----+-----+-----+-----+
(base) SantorumPC:assignment1 santorum$
```

Figura 8.1: Búsqueda de rejilla de hiperparámetros P1 Perceptrón

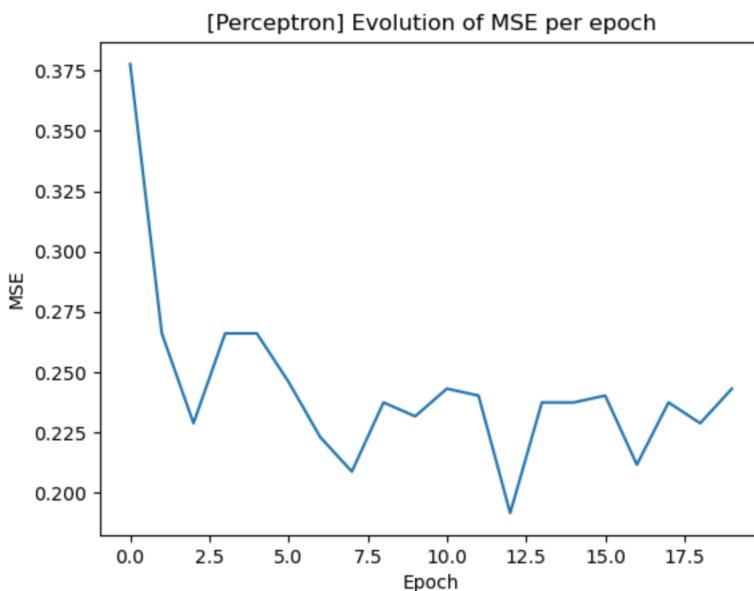


Figura 8.2: Evolución ECM por época para alfa = 0.5 th = 0

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real1_perc.py
Executing Perceptron algorithm with parameters:
    Alpha = 0.5
    Threshold = 0.0
    Number of repetitions = 10
    Number of maximum epochs = 20
====> Mean Squared Error: 0.27429 +- 0.136
====> Mean Accuracy: 0.966 +- 0.02
(base) SantorumPC:assignment1 santorum$ █
```

Figura 8.3: ECM y porcentaje de acierto para alfa = 0.5 th = 0

Procedemos a mostrar el ECM medio para los diferentes valores probados al ejecutar el programa con el parámetro `-hyper` (Figura 8.1), para tras ello mostrar la evolución del ECM a cada época con los valores óptimos encontrados (Figura 8.2), además de mostrar el ECM final y el porcentaje de acierto que tiene la red neuronal entrenada con esos parámetros sobre el conjunto de prueba (Figura 8.3). También se proporciona la opción de fijar un valor de α para probar diferentes valores del *threshold* y viceversa (obteniendo los resultados que se pueden ver en las figuras 8.4, 8.5, 8.6, 8.7), que se podrá obtener con las llamadas `python3 prob_real1_perc.py -hyper -a alpha` y `python3 prob_real1_ada.py -hyper -th threshold`.

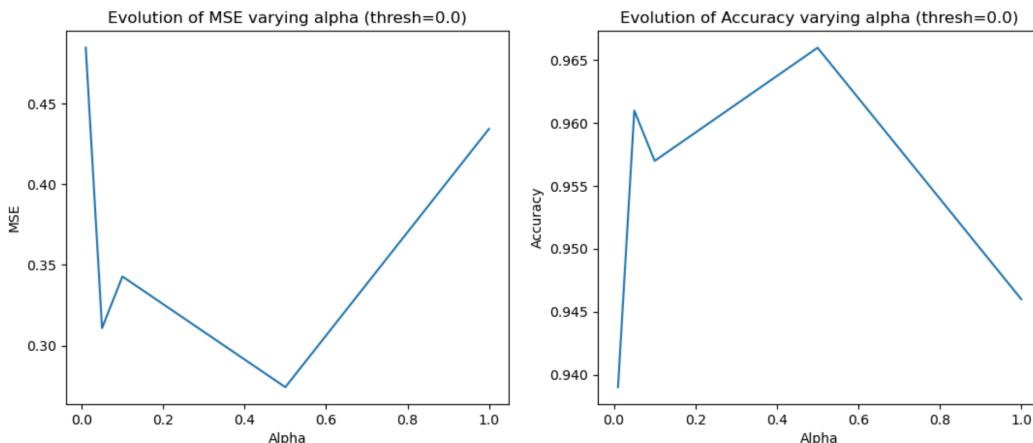


Figura 8.4: Gráfico ECM y porcentaje de acierto variando alfa con th = 0

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real1_perc.py -hyper -th 0
Alpha: 0.01 Threshold: 0.0 ----> mse: 0.42957 , acc: 0.947
Alpha: 0.05 Threshold: 0.0 ----> mse: 0.53943 , acc: 0.933
Alpha: 0.1 Threshold: 0.0 ----> mse: 0.38629 , acc: 0.962
Alpha: 0.5 Threshold: 0.0 ----> mse: 0.33371 , acc: 0.958
Alpha: 1 Threshold: 0.0 ----> mse: 0.43886 , acc: 0.945
+-----+
| Threshold \ Alphas | 0.01 | 0.05 | 0.1 | 0.5 | 1 |
+-----+
| 0 | 0.42957 +- 0.426 | 0.53943 +- 0.293 | 0.38629 +- 0.112 | 0.33371 +- 0.182 | 0.43886 +- 0.319 |
+-----+
| Threshold \ Alphas | 0.01 | 0.05 | 0.1 | 0.5 | 1 |
+-----+
| 0 | 0.947 +- 0.05 | 0.933 +- 0.04 | 0.962 +- 0.01 | 0.958 +- 0.02 | 0.945 +- 0.04 |
+-----+
(base) SantorumPC:assignment1 santorum$ █
```

Figura 8.5: Tabla ECM y porcentaje de acierto variando alfa con th = 0

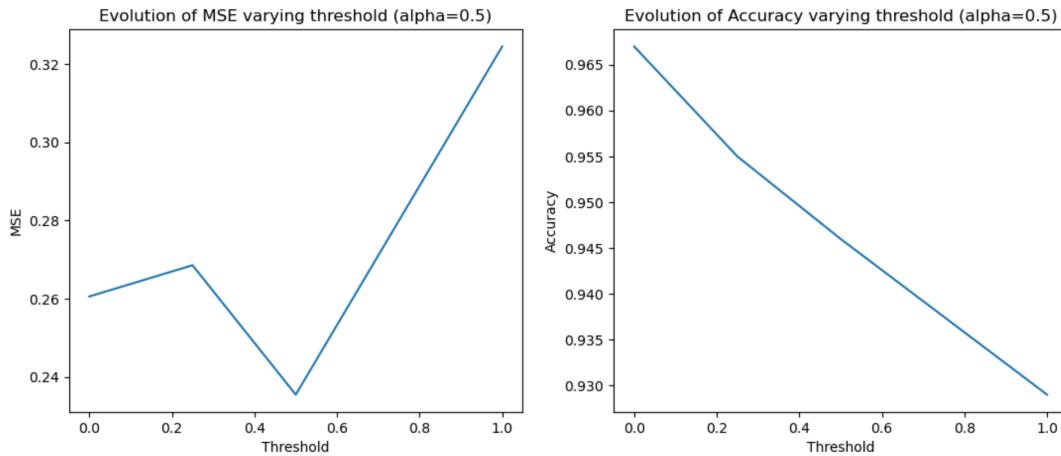


Figura 8.6: Gráfico ECM y porcentaje de acierto variando th con alfa = 0.5

```
(base) SantorumPC:assignment1 santorum$ python3 prob_reali_perc.py -hyper -a 0.5
Alpha: 0.5 Threshold: 0 ---> mse: 0.26057 , acc: 0.967
Alpha: 0.5 Threshold: 0.25 ---> mse: 0.26857 , acc: 0.955
Alpha: 0.5 Threshold: 0.5 ---> mse: 0.23543 , acc: 0.946
Alpha: 0.5 Threshold: 1 ---> mse: 0.32457 , acc: 0.929
+-----+-----+-----+-----+
| Alpha \ Thresholds | 0 | 0.25 | 0.5 | 1 |
+-----+-----+-----+-----+
| 0.5 | 0.26057 +- 0.084 | 0.26857 +- 0.135 | 0.23543 +- 0.065 | 0.32457 +- 0.12 |
+-----+-----+-----+-----+
+-----+-----+-----+-----+
| Alpha \ Thresholds | 0 | 0.25 | 0.5 | 1 |
+-----+-----+-----+-----+
| 0.5 | 0.967 +- 0.01 | 0.955 +- 0.02 | 0.946 +- 0.02 | 0.929 +- 0.02 |
+-----+-----+-----+-----+
(base) SantorumPC:assignment1 santorum$
```

Figura 8.7: Tabla ECM y porcentaje de acierto variando th con alfa = 0.5

Como podemos concluir de todas estas capturas, el Perceptrón, aunque obtiene un **porcentaje de acierto bastante alto** con los parámetros encontrados, **no siempre reduce el ECM a cada época**. Además, podemos ver que para un *threshold* fijo el alfa genera variaciones aparentemente aleatorias de ECM y porcentaje de acierto (eso sí, al estar el umbral a 0 justo la gráfica del ECM y del porcentaje de acierto son completamente opuestas ya que o bien clasifica correctamente o incorrectamente) y que para un alfa fijo, variar el *threshold* se traduce en una variación aparentemente aleatoria del ECM pero un constante decrecimiento en el porcentaje de acierto (ya que aumenta el número de clases predichas con 0 que son directamente erróneas).

8.2. Algoritmo de aprendizaje Adaline

Para el Adaline hemos tenido que realizar menos modificaciones en comparación con el caso de las funciones lógicas, ya que el Adaline sí termina aunque el problema no sea linealmente separable. Al igual que con el Perceptrón, hemos implementado la funcionalidad de optimización de hiperparámetros al llamar a

```
python3 prob_reali_ada.py -hyper
```

con $\alpha \in [0,005, 0,01, 0,05, 0,1]$ y tolerancia $\in [0,001, 0,005, 0,01, 0,05]$, en el que nos mostrará el ECM. Es importante observar que durante todo este ejercicio omitiremos la precisión salvo en el caso final ya que, como hemos visto en teoría, el algoritmo Adaline busca minimizar el ECM mediante la regla delta y el descenso del gradiente. Por lo tanto, menor ECM se corresponde con mayor precisión y viceversa. La figura 8.8 es un extracto de una búsqueda de hiperparámetros exhaustiva para diferentes valores de los parámetros alfa y tolerancia.

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real1_ada.py -hyper
Alpha: 0.005 Tolerance: 0.001 --> mse: 0.36343
Alpha: 0.01 Tolerance: 0.001 --> mse: 0.33829
Alpha: 0.05 Tolerance: 0.001 --> mse: 0.30629
Alpha: 0.1 Tolerance: 0.001 --> mse: 0.37486
Alpha: 0.005 Tolerance: 0.005 --> mse: 0.35657
Alpha: 0.01 Tolerance: 0.005 --> mse: 0.32
Alpha: 0.05 Tolerance: 0.005 --> mse: 0.34057
Alpha: 0.1 Tolerance: 0.005 --> mse: 0.37943
Alpha: 0.005 Tolerance: 0.01 --> mse: 0.29714
Alpha: 0.01 Tolerance: 0.01 --> mse: 0.256
Alpha: 0.05 Tolerance: 0.01 --> mse: 0.26971
Alpha: 0.1 Tolerance: 0.01 --> mse: 0.336
Alpha: 0.005 Tolerance: 0.05 --> mse: 0.384
Alpha: 0.01 Tolerance: 0.05 --> mse: 0.37029
Alpha: 0.05 Tolerance: 0.05 --> mse: 0.33371
Alpha: 0.1 Tolerance: 0.05 --> mse: 0.38171
+-----+-----+-----+-----+
| Tols \ Alphas | 0.005 | 0.01 | 0.05 | 0.1 |
+=====+=====+=====+=====+
| 0.001 | 0.36343 +- 0.175 | 0.33829 +- 0.065 | 0.30629 +- 0.096 | 0.37486 +- 0.134 |
+-----+-----+-----+-----+
| 0.005 | 0.35657 +- 0.094 | 0.32 +- 0.072 | 0.34057 +- 0.155 | 0.37943 +- 0.102 |
+-----+-----+-----+-----+
| 0.01 | 0.29714 +- 0.109 | 0.256 +- 0.12 | 0.26971 +- 0.065 | 0.336 +- 0.102 |
+-----+-----+-----+-----+
| 0.05 | 0.384 +- 0.097 | 0.37029 +- 0.1 | 0.33371 +- 0.114 | 0.38171 +- 0.21 |
+-----+-----+-----+-----+
(base) SantorumPC:assignment1 santorum$
```

Figura 8.8: Búsqueda de rejilla de hiperparámetros P1 Adaline

Como podemos apreciar, los valores que minimizan el ECM son $\alpha = 0,01$ y tol = 0,01. De manera similar, también permitimos fijar uno de los valores para probar con varias opciones del otro con las llamadas `python3 prob_real1_ada.py -hyper -a alpha` y `python3 prob_real1_ada.py -hyper -tol tolerancia`. Los resultados de fijar uno de los dos parámetros encontrados y variar el otro se pueden ver en las figuras 8.9, 8.10, 8.11 y 8.12.

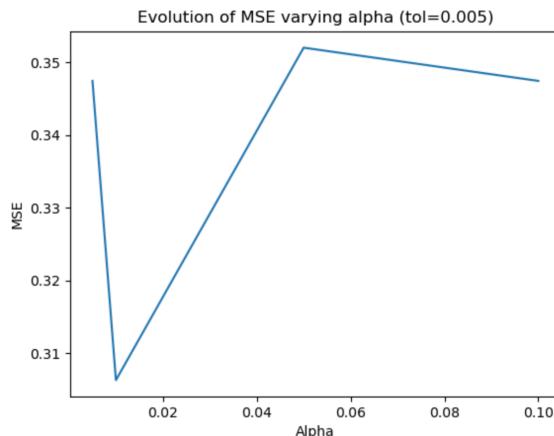


Figura 8.9: Gráfico ECM variando alfa con tol = 0.01

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real1_ada.py -hyper -tol 0.005
Alpha: 0.005 Tol: 0.005 --> mse: 0.34743
Alpha: 0.01 Tol: 0.005 --> mse: 0.30629
Alpha: 0.05 Tol: 0.005 --> mse: 0.352
Alpha: 0.1 Tol: 0.005 --> mse: 0.34743
+-----+
|   Tols \ Alphas | 0.005      | 0.01      | 0.05      | 0.1      |
+=====+=====+=====+=====+
| 0.005 | 0.34743 +- 0.167 | 0.30629 +- 0.089 | 0.352 +- 0.106 | 0.34743 +- 0.139 |
+-----+
(base) SantorumPC:assignment1 santorum$
```

Figura 8.10: Tabla ECM variando alfa con tol = 0.01

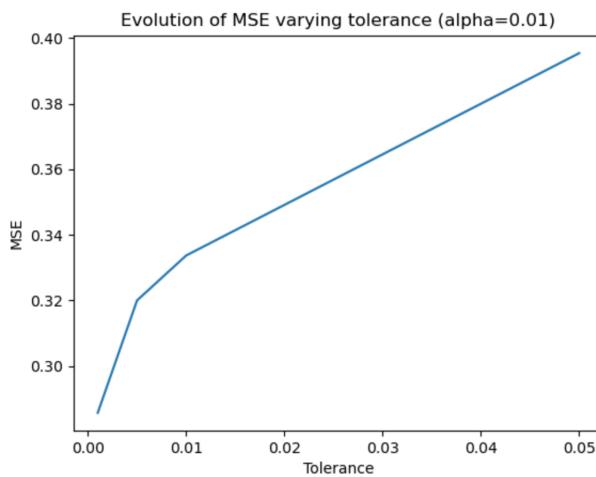


Figura 8.11: Gráfico ECM variando tol con alfa = 0.01

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real1_ada.py -hyper -a 0.01
Alpha: 0.01 Tol: 0.001 --> mse: 0.28571
Alpha: 0.01 Tol: 0.005 --> mse: 0.32
Alpha: 0.01 Tol: 0.01 --> mse: 0.33371
Alpha: 0.01 Tol: 0.05 --> mse: 0.39543
+-----+
|   Alpha \ Tols | 0.001      | 0.005      | 0.01      | 0.05      |
+=====+=====+=====+=====+
| 0.01 | 0.28571 +- 0.064 | 0.32 +- 0.127 | 0.33371 +- 0.132 | 0.39543 +- 0.13 |
+-----+
(base) SantorumPC:assignment1 santorum$
```

Figura 8.12: Tabla ECM variando tol con alfa = 0.01

Tras ello, hemos establecido los **valores por defecto de la ejecución normal del programa** para ser los que **menor ECM generan en la ejecución** con la opción `-hyper`, obteniendo un **porcentaje de acierto bastante alto, sobre el 97 %** (Figura 8.13). Si se quiere especificar unos valores de alfa y tolerancia específicos, se deberá ejecutar `python3 prob_real1_ada.py -a alpha -tol tolerance -nreps num_reps`, donde `num_reps` es el número de iteraciones del algoritmo Adaline que se deseen. También hemos graficado la evolución del ECM para los valores de los hiperparámetros encontrados, lo que se puede apreciar en la figura 8.14.

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real1_ada.py
Executing Adaline algorithm with parameters:
    Alpha = 0.01
    Tolerance = 0.01
    Number of repetitions = 10
====> Mean Squared Error: 0.256 +- 0.124
====> Mean Accuracy: 0.967 +- 0.02
(base) SantorumPC:assignment1 santorum$
```

Figura 8.13: ECM y porcentaje de acierto con alfa = tol = 0.01

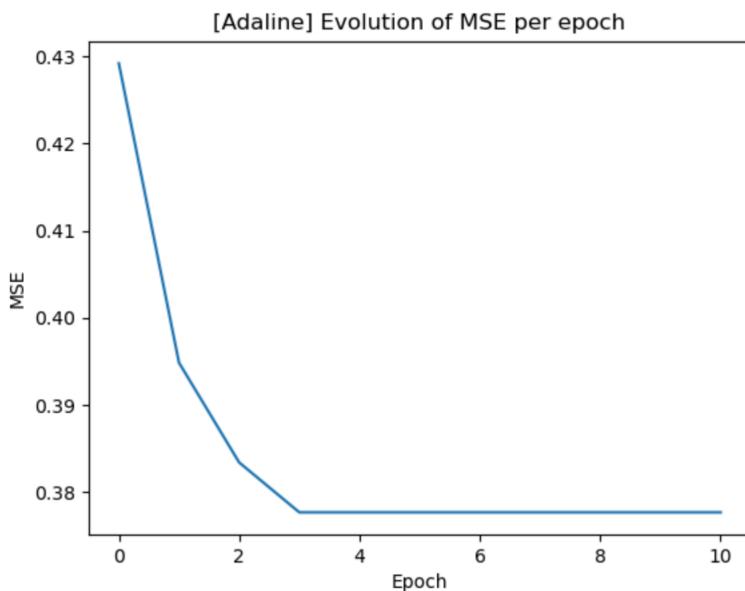


Figura 8.14: Evolución ECM por época para alfa = tol = 0.01

Como podemos observar en todas estas capturas, en este caso el ECM se reduce o se mantiene a cada época (ya que la regla delta del Adaline está construida con ese objetivo). También podemos apreciar que tanto el ECM y el porcentaje de aciertos generados por ambos algoritmos casi indistinguibles (probablemente debido a que el **problema real 1 esta muy cerca de ser linealmente separable**). Por último, en las gráficas en las que fijamos uno de los parámetros y variamos el otro, podemos observar que en el caso en el que variamos alfa, el ECM varía de forma aparentemente aleatoria pero que en el caso en el que variamos la tolerancia el ECM crece de manera directamente proporcional, debido a que el permitir un margen más amplio para considerar que el algoritmo ha convergido nos podemos quedar más lejos del mínimo real.

Recordemos para el problema real 1 disponemos de un Makefile que ejecuta los programas `prob_real1_perc.py` y `prob_real1_ada.py` **con los hiperparámetros que mejores resultados han dado**.

9. Problema real 2

La única diferencia entre los dos problemas reales es, básicamente, **el modo de lectura de los ficheros de datos que usaremos**, ya que en el primero disponíamos de una batería de ejemplos, todos con su clasificación correcta, y ahora disponemos de un conjunto de entrenamiento y un conjunto de predicción sin las etiquetas correctas de los datos. Entrenaremos los algoritmos de manera idéntica al problema real 1 y en las secciones siguientes nos limitaremos a ilustrar los resultados de las ejecuciones y extraer conclusiones de las mismas, añadiendo con la entrega de la memoria las predicciones generadas del fichero de datos sin etiquetar. Un detalle importante a destacar es que también ofrecemos una precisión estimada del algoritmo. Esto se ha implementado con otra red neuronal en la que se entrena con el 75 % de los datos etiquetados y se prueba con el 25 % restante (modo 2 de lectura). Al no ser exactamente la misma red con la que hacemos predicción con los datos sin etiquetar la precisión real puede variar, pero con esto nos hacemos una **idea orientativa del resultado**.

9.1. Algoritmo de aprendizaje Perceptrón

Ejecutamos una búsqueda en rejilla para obtener los mejores hiperparámetros, utilizando para ello el conjunto de datos etiquetados, dividiéndolo en *trainset* y *testset*.

Thresholds \ Alphas	0.01	0.05	0.1	0.5	1
0	2.89143 ± 0.093	2.42286 ± 0.057	2.74857 ± 0.034	2.27429 ± 0.044	2.50857 ± 0.085
0.1	1.41857 ± 0.156	1.69714 ± 0.389	1.88143 ± 0.317	2.34286 ± 0.452	2.72286 ± 1.097
0.3	1.48 ± 0.165	1.45857 ± 0.196	1.64857 ± 0.305	2.10714 ± 0.668	2.39143 ± 0.883
0.5	1.48714 ± 0.151	1.51 ± 0.091	1.52571 ± 0.183	1.86714 ± 0.456	1.86571 ± 0.733
1	1.52429 ± 0.099	1.46286 ± 0.155	1.43714 ± 0.152	1.61714 ± 0.258	1.74286 ± 0.465
Thresholds \ Alphas	0.01	0.05	0.1	0.5	1
0	0.639 ± 0.14	0.697 ± 0.11	0.656 ± 0.08	0.716 ± 0.06	0.686 ± 0.1
0.1	0.484 ± 0.05	0.548 ± 0.1	0.621 ± 0.03	0.664 ± 0.06	0.635 ± 0.13
0.3	0.397 ± 0.05	0.506 ± 0.06	0.557 ± 0.04	0.634 ± 0.09	0.631 ± 0.11
0.5	0.323 ± 0.08	0.466 ± 0.04	0.546 ± 0.06	0.617 ± 0.08	0.646 ± 0.1
1	0.306 ± 0.06	0.472 ± 0.06	0.582 ± 0.05	0.592 ± 0.08	0.609 ± 0.09

Figura 9.1: *Grid search* de hiperparámetros para el Prob2 Perceptrón

Nos quedamos con los parámetros $\alpha = 0,5$ $threshold = 0$, obteniendo una **precisión orientativa del 74 %** con el algoritmo Perceptrón.

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real2_perc.py
Executing Perceptron algorithm with parameters:
  Alpha = 0.5
  Threshold = 0.0
  Number of repetitions = 10
  Number of maximum epochs = 20
Estimated accuracy: 0.738 +- 0.12
The predictions have been stored in: predicciones/prediccion_perceptron.txt
(base) SantorumPC:assignment1 santorum$
```

Figura 9.2: Estimación de la precisión Perceptrón con una red neuronal auxiliar

9.2. Algoritmo de aprendizaje Adaline

Igual que con el algoritmo Perceptrón, ejecutamos una búsqueda en rejilla para obtener los mejores hiperparámetros, utilizando para ello el conjunto de datos etiquetados, dividiéndolo en *trainset* y *testset*.

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real2_ada.py -hyper
Alpha: 0.005 Tolerance: 0.001 ---> mse: 1.82571
Alpha: 0.01 Tolerance: 0.001 ---> mse: 1.72571
Alpha: 0.05 Tolerance: 0.001 ---> mse: 1.93143
Alpha: 0.1 Tolerance: 0.001 ---> mse: 2.02857
Alpha: 0.005 Tolerance: 0.005 ---> mse: 1.78571
Alpha: 0.01 Tolerance: 0.005 ---> mse: 1.78286
Alpha: 0.05 Tolerance: 0.005 ---> mse: 1.99429
Alpha: 0.1 Tolerance: 0.005 ---> mse: 2.07143
Alpha: 0.005 Tolerance: 0.01 ---> mse: 1.78286
Alpha: 0.01 Tolerance: 0.01 ---> mse: 1.88571
Alpha: 0.05 Tolerance: 0.01 ---> mse: 1.94
Alpha: 0.1 Tolerance: 0.01 ---> mse: 2.00857
Alpha: 0.005 Tolerance: 0.05 ---> mse: 2.34286
Alpha: 0.01 Tolerance: 0.05 ---> mse: 2.09714
Alpha: 0.05 Tolerance: 0.05 ---> mse: 2.07429
Alpha: 0.1 Tolerance: 0.05 ---> mse: 1.93714
+---+ +---+ +---+ +---+
| Tols \ Alphas | 0.005 | 0.01 | 0.05 | 0.1 |
+=====+=====+=====+=====+
| 0.001 | 1.82571 +- 0.23 | 1.72571 +- 0.298 | 1.93143 +- 0.396 | 2.02857 +- 0.38 |
+-----+-----+-----+-----+
| 0.005 | 1.78571 +- 0.22 | 1.78286 +- 0.217 | 1.99429 +- 0.299 | 2.07143 +- 0.271 |
+-----+-----+-----+-----+
| 0.01 | 1.78286 +- 0.312 | 1.88571 +- 0.306 | 1.94 +- 0.129 | 2.00857 +- 0.265 |
+-----+-----+-----+-----+
| 0.05 | 2.34286 +- 0.3 | 2.09714 +- 0.154 | 2.07429 +- 0.422 | 1.93714 +- 0.315 |
+-----+-----+-----+-----+
(base) SantorumPC:assignment1 santorum$ python3 prob_real2_ada.py -hyper
```

Figura 9.3: Búsqueda de rejilla de hiperparámetros para el Prob2 Adaline

Nos quedamos con los valores $\alpha = 0,01$ tolerancia = 0,001, obteniendo una **precisión estimada del 79 %**. En el caso del problema real 2 vemos que el Adaline nos ofrece una precisión sensiblemente superior al Perceptrón, probablemente debido a que este problema ya no está tan cerca de ser linealmente separable como el problema real 1.

```
(base) SantorumPC:assignment1 santorum$ python3 prob_real2_ada.py
Executing Adaline algorithm with parameters:
    Alpha = 0.01
    Tolerance = 0.001
    Number of repetitions = 10
Estimated accuracy: 0.788 +- 0.03
The predictions have been stored in: predicciones/prediccion_adaline.txt
(base) SantorumPC:assignment1 santorum$
```

Figura 9.4: Estimación de la precisión Adaline con una red neuronal auxiliar

10. Conclusiones

Llegamos al final de la práctica 1 de la asignatura de Neurocomputación. A lo largo del camino hemos podido **establecer contacto** con varios **modelos de redes neuronales y algoritmos de aprendizaje**: redes de **McCulloch-Pitts**, algoritmo **Perceptrón** y algoritmo de aprendizaje **Adaline**.

Adicionalmente, hemos podido emplear varias de estas implementaciones en problemas reales. En el primero, parece que hemos obtenido un resultado bastante bueno, **sobre el 97 % de precisión**. En el segundo aún no sabemos los resultados reales al no disponer de las clases reales del fichero de prueba, no obstante estimamos que la precisión estará **rozando el 80 %**.

Finalmente, en el corazón del código de esta práctica se sitúa la **librería de manejo de redes neuronales**. Esta nos permitirá crear fácilmente nuevos modelos en las siguientes prácticas de esta asignatura.