Juan Moreno y Diego Cano

MEMORIA PRÁCTICA 1

Apartado 2.1

Hemos realizado el diseño de la siguiente arquitectura propuesta:

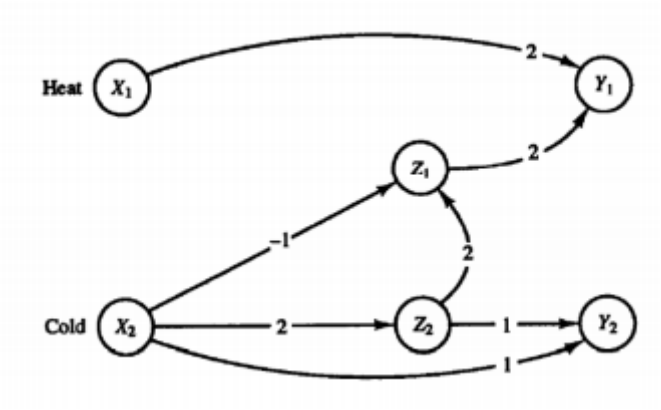

Figura 1. Red de McChulloch-Pitts (Frío Calor)

Para la realización del diseño hemos comenzado creando una red neuronal. Posteriormente creamos las capas y las neuronas y las añadimos de la siguiente forma:

- Capa de Entrada
    - Neurona X1 (tipo directa)
    - Neurona X2 (tipo directa)
- Capa oculta
    - Neurona Z1 (tipo McCulloch con umbral=2)
    - Neurona Z2 (tipo McCulloch con umbral=2)
- Capa de salida
    - Neurona Y1 (tipo McCulloch con umbral=2)
    - Neurona Y2 (tipo McCulloch con umbral=2)

Continuamos conectando las neuronas entre sí con los siguientes pesos:

X1 —> Y1 (peso = 2)
X2 —> Z1 (peso = -1)
X2 —> Z2 (peso = 2)
X2 —> Y2 (peso = 1)
Z1 —> Y1 (peso = 2)
Z2 —> Z1 (peso = 2)
Z2 —> Y2 (peso = 1)

Por último, por cada impulso obtenido del fichero *impulsos.txt*:

- Inicializamos las neuronas de entrada de la red (X1 y X2) a los valores recibidos
- Disparamos la red neuronal entera

- Inicializamos la red para poder propagar los valores posteriormente
- Finalmente, vamos propagando hasta que finalice.

El correcto funcionamiento de nuestro diseño se puede comprobar observando el fichero de salida producido llamado *salida_frio_calor.txt*. Al abrir este fichero se pueden observar los resultados, siendo idénticos a los de la siguiente tabla:

Tiempo	X1	X2	Z1	Z2	Y1	Y2
T0	0	0	0	0	0	0
T1	1	1	0	0	0	0
T2	1	1	0	1	1	0
T3	0	1	0	1	1	1
T4	1	0	0	1	0	1
T5	0	0	1	0	1	0
T6	0	0	0	0	1	0
T7	0	0	0	0	0	0

Figura 2. Tabla con resultados del problema Frío/Calor

Explicación de los resultados del ejemplo con nuestro diseño:

- Al llegar el primer impulso (0,0) las salidas de las demás neuronas valdrán todas 0.
- Al llegar el segundo impulso (1,1) las salidas de las demás neuronas valdrán todas 0 debido al retraso temporal que se impone
- Al llegar el tercer impulso (1,1) se producen las siguientes operaciones: z2=1*2 = 2>=umbral, z2=1 || z1=-1*0 = 0<umbral, z1=0 || y1=2*1= 2>=umbral, y1=1 || y2=1*1<umbral, y2=0
- Al llegar el cuarto impulso (0,1) se realizan las siguientes operaciones: z2=1*2=2>=umbral, z2=1 || z1=-1*1 + 2*1=1<umbral, z1=0 || y1=2*1+2*0=2>=umbral, y1=1 || y2=1*1+1*1=2 >=umbral, y2=1
- Así continúa hasta que se finaliza y termina de sacar todos los valores

Para cada estímulo la red calcula si se produce frío (0, 1) o calor (1, 0), la respuesta esperada será frío si el estímulo (0, 1) se aplica en dos pasos de tiempo, sin embargo será calor si se aplica algún estímulo de calor y también cuando se aplica un solo estímulo de frío en un paso de tiempo.

Funciones de salida de la red neuronal de McCulloch-Pitts para el sensor de Frío/Calor:

- y1(t) = { x1(t-1) } OR { x2(t-3) AND NOT x2(t-2) }
- y2(t) = x2(t-2) AND x2(t-1)

Apartado 4.1.1: Fronteras de decisión para problemas lógicos

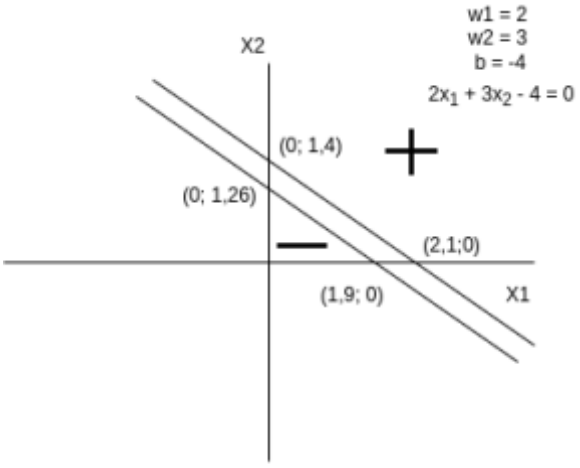

Figura 3. Frontera de decisión para el problema AND

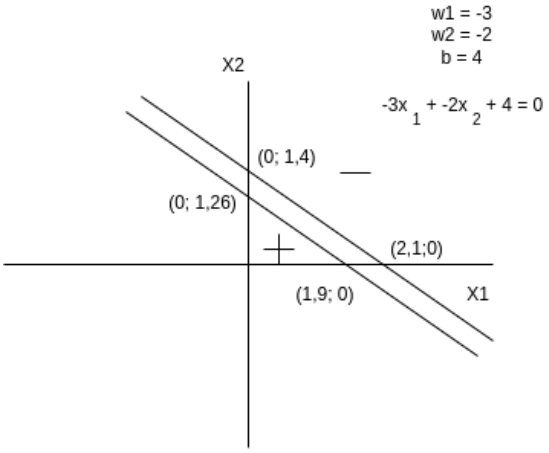

Figura 4. Frontera de decisión para el problema NAND

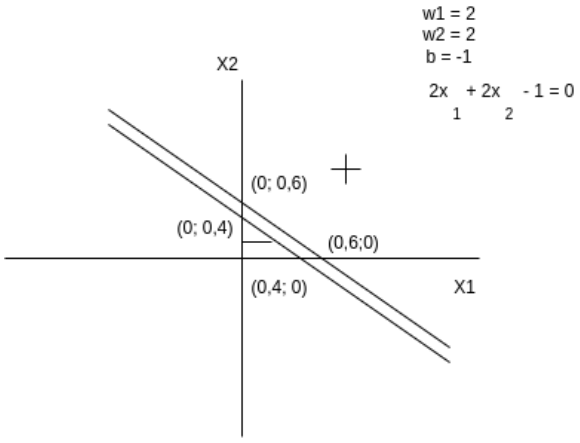

Figura 5. Frontera de decisión para el problema OR

XOR

Los pesos y sesgo obtenidos son [0, 0, 0], por lo tanto este problema no se puede solucionar. Esto se debe a que el problema XOR no es linealmente separable, aunque se podría solucionar utilizando una arquitectura distinta a la del perceptrón lineal. Con un perceptrón multicapa se podría resolver este problema y la mayoría de los que no son linealmente separables.

El resto de problemas si que se pueden solucionar ya que son linealmente separables y la arquitectura empleada es un perceptrón lineal. Para poder sacar las fronteras de decisión, hemos observado los pesos y sesgo finales de cada problema (w1, w2, b):

- AND —> [2, 3, -4]
- NAND —> [-3, -2, 4]
- OR —> [2, 2, -1]
- XOR —> [0, 0, 0]

Posteriormente, vemos la recta que cumple la respuesta positiva (> umbral) y también la que cumple la respuesta negativa (< - umbral). Ya que las entradas son binarias (valores entre 0 y 1), nos situamos en el primer cuadrante. En el fichero *main_pruebas.py* se puede observar un ejemplo que soluciona el problema AND e imprime por pantalla los pesos y sesgo finales.

Apartado 4.2.1

Descripción de la implementación del perceptrón
Hemos creado un fichero Python denominado *perceptron.py* que contiene los métodos de entrenamiento y de clasificación; además de funciones auxiliares para comprobaciones necesarias.

Diseño de la arquitectura de la red

1. Creamos la red neuronal.
2. Creamos la capa de entrada y la capa de salida.
3. Añadimos tantas neuronas (de tipo Directa) como entradas haya en el problema a la capa de entrada.
4. Añadimos la neurona de tipo sesgo a la capa de entrada.
5. Añadimos tantas neuronas (de tipo Perceptrón) como salidas haya en el problema a la capa de salida.
6. Conectamos la capa de entrada con la capa de salida con pesos y sesgo iniciales a 0.
7. Añadimos las capas a la red.
8. Inicializamos la red neuronal.
9. Comienzo del proceso de entrenamiento/clasificación.

Proceso de Entrenamiento
Pasos a seguir en cada época:

1. Comprobación de la condición de parada, es decir, que los pesos no cambien en la época.
2. Para cada ejemplo de entrenamiento aplicar los pasos a-e:
    a. Damos valor a las neuronas de entrada.
    b. Damos valor al sesgo (1.0).
    c. Establecemos las activaciones a las neuronas de entrada.
    d. Calculamos las respuestas de las neuronas de salida.
    e. Ajustamos pesos y sesgo si la salida de la red neuronal es distinta a lo esperado, sino se mantienen los pesos anteriores.

3. Una vez acabadas todas las épocas o que se haya cumplido la condición de parada, devolvemos los pesos resultantes.

Proceso de Clasificación
Para cada ejemplo de clasificación:
1. Damos valor a las neuronas de entrada.
2. Damos valor al sesgo.
3. Establecemos las activaciones a las neuronas de entrada.
4. Calculamos las respuestas de las neuronas de salida.
5. Comprobamos si las respuestas son iguales a lo esperado.
6. Devolvemos el porcentaje de aciertos de la red.
7. En la creación de la red para clasificación, hemos asignado los pesos y el sesgo obtenidos en la fase de entrenamiento.

Descripción de la implementación de adaline

Hemos creado un fichero Python denominado *adaline.py* que contiene los métodos de entrenamiento y de clasificación.

Diseño de la arquitectura de la red realizamos los siguientes pasos
1. Creamos la red neuronal.
2. Creamos la capa de entrada y la capa de salida.
3. Añadimos tantas neuronas (de tipo Directa) como entradas haya en el problema a la capa de entrada.
4. Añadimos la neurona tipo sesgo a la capa de entrada.
5. Añadimos tantas neuronas como salidas haya en el problema a la capa de salida.
6. Conectamos la capa de entrada con la capa de salida con pesos y sesgo iniciales a valores aleatorios pequeños entre 0 y 1.
7. Añadimos las capas a la red.
8. Inicializamos la red neuronal.
9. Comienzo del proceso de entrenamiento/clasificación.

Proceso de Entrenamiento
Pasos a seguir en cada época:
1. Para cada ejemplo de entrenamiento aplicar los pasos a-e:
    a. Damos valor a las neuronas de entrada.
    b. Damos valor al sesgo (1.0).

    c. Establecemos las activaciones a las neuronas de entrada.
    d. Calculamos las respuestas de las neuronas de salida que en este caso serán iguales al valor de entrada de éstas.
    e. Ajustamos pesos y el sesgo.
    f. Realizamos la comprobación de parada, es decir, si el peso más grande obtenido al ajustar es menor que la tolerancia especificada se para, sino se continúa.
2. Una vez acabadas todas las épocas o que se haya cumplido la condición de parada devolvemos los pesos resultantes.

Proceso de Clasificación

Para cada ejemplo de clasificación:

1. Damos valor a las neuronas de entrada.
2. Damos valor al sesgo.
3. Establecemos las activaciones a las neuronas de entrada.
4. Calculamos las respuestas de las neuronas de salida.
5. Comprobamos si las respuestas son iguales a lo esperado.
6. Devolvemos el porcentaje de aciertos de la red.
7. En la creación de la red para clasificación, hemos asignado los pesos y el sesgo obtenidos en la fase de entrenamiento.

Comparación del Error Cuadrático Medio entre Perceptrón y Adaline

Al realizar la siguiente instrucción: ***make ejecuta_perceptron***, se puede observar un ejemplo de ejecución del problema_real1 con el Modo 1 de lectura y con los siguientes parámetros:

- Número de épocas: 100
- Constante de aprendizaje: 1
- Umbral: 0.2

Al realizar la siguiente instrucción: ***make ejecuta_adaline***, se puede observar un ejemplo de ejecución del problema_real1 con el Modo 1 de lectura y con los siguientes parámetros:

- Número de épocas: 100
- Constante de aprendizaje: 0.01
- Tolerancia: 0.01

La comparación del ECM entre las dos redes se realizará con los parámetros anteriormente mencionados. Se usará un porcentaje de entrenamiento del 70%.
Al sacar la variación del error en perceptrón, podemos observar que durante las épocas iniciales, comienza a disminuir. Sin embargo, cuando lleva aproximadamente unas 10 épocas, muestra valores que van subiendo y bajando continuamente. Podemos concluir que el ECM tiene una tendencia decreciente, pero llega un punto en el que se vuelve algo irregular y variable. Esto se debe a que durante el entrenamiento habrá épocas en las que no se tiene interés en cambiar algunos pesos para determinados ejemplos. por lo tanto provoca errores en épocas posteriores. Es el propio comportamiento de la red el que produce esa variación del error.

Si sacamos la variación del error en Adaline, podemos ver un comportamiento muy distinto al del Perceptrón. Es cierto que en las primeras épocas tienen tendencias similares. Sin embargo, en Adaline, al llegar aproximadamente a la época número 6, el valor del ECM se mantiene casi idéntico durante las demás épocas. Los pesos en Adaline se ajustan siempre de la misma forma y eso hace que se acabe manteniendo constante.

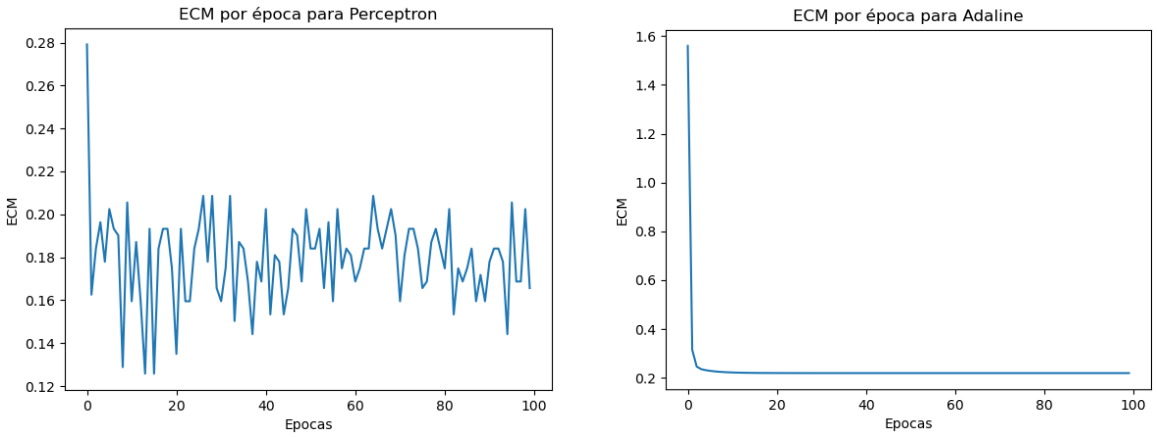

Figura 6. Comparación del ECM entre Perceptrón y Adaline

Gráficas del Error Cuadrático Medio con cambio de parámetros en ambas redes

1. Para el Perceptrón, mantendremos el número de épocas a 100, la constante de aprendizaje a 1 (puede ser mayor que 0 o menor o igual a 1) e iremos variando el umbral de las neuronas de salida.

2. Para Adaline, mantenemos el número de épocas a 100, mantendremos la tolerancia a un valor relativamente bajo para que podamos observar el comportamiento de la red de forma más clara (0.01), e iremos variando la constante de aprendizaje (usaremos valores no muy grandes para que el aprendizaje termine de converger y no muy pequeños para que no sea muy lento).

Estos experimentos han sido realizados sobre el *problema_real1* con el Modo 1 de lectura y utilizando un porcentaje de entrenamiento del 70%.

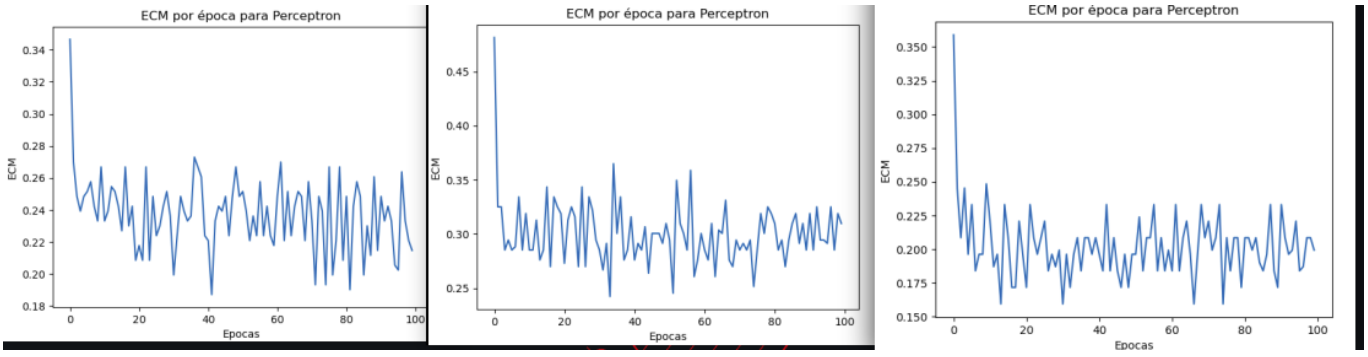

Figura 7. ECM Perceptrón con umbrales = (0.2, 0.1, 0.01) respectivamente

Como se puede observar en el comportamiento de las 3 gráficas, si se disminuye ligeramente el valor umbral de las neuronas de salida, el ECM se va ajustando a lo largo de las épocas. En la primera gráfica (umbral = 0.2) tenemos el mismo comportamiento descrito anteriormente, mientras que al disminuirlo en las gráficas 2 y 3, se puede observar una mejoría en las irregularidades del cálculo del error cuadrático medio. Además, los valores en la gráfica 3 se van manteniendo en una zona más inferior. Al hacer el umbral más pequeño, se está permitiendo que las neuronas de salida se activen un mayor número de veces y así se realizarán comprobaciones de salida más exactas que permitirán que los pesos se ajusten más rápido y de una forma más regular.

Valores finales de ejecución decididos: épocas = 100, constante de aprendizaje = 1, umbral = 0.01. Para cambiar estos valores fácilmente hay que localizarse en el fichero *main.py* y en la zona de ejecución del perceptrón, están nombrados y asignados los valores de los parámetros.

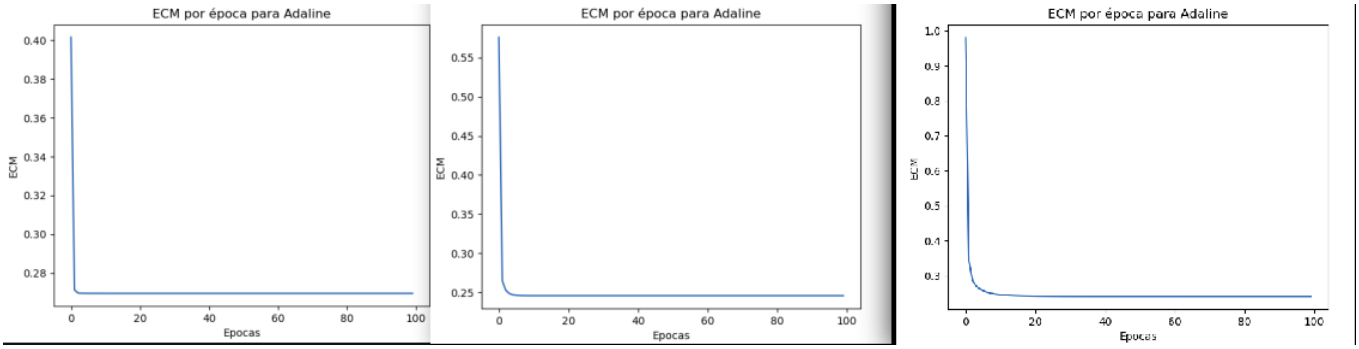

Figura 8. ECM Adaline con constante de aprendizaje = (0.1, 0.05, 0.01) respectivamente

En el caso de Adaline, se puede observar que al disminuir la constante de aprendizaje, se produce una regularización del ECM. Con los valores más altos (0.1 y 0.05), se aprecia un cambio brusco en las épocas iniciales y posteriormente se mantiene. Sin embargo, para el valor de 0.01, el cambio se produce de forma más progresiva y posteriormente el error se vuelve a mantener igual. Hemos considerado 0.01 el valor óptimo ya que al aumentarlo, el aprendizaje es menos probable que vaya a converger.

Valores finales de ejecución decididos: épocas = 100, constante de aprendizaje = 0.01, tolerancia = 0.01. Para cambiar estos valores fácilmente hay que localizarse en el fichero *main.py* y en la zona de ejecución del adaline, están nombrados y asignados los valores de los parámetros.

Apartado 4.3.1

La implementación de las redes neuronales para la resolución del *problema_real2* ha sido la misma que en los apartados anteriores. Hemos utilizado los parámetros óptimos descritos en el apartado anterior y en la carpeta raíz de la práctica se puede observar el directorio *predicciones* el cual incluye dos .txt para las predicciones realizadas sobre el *porblema_real2_no_etiquetado* con Perceptrón y Adaline. Para cambiar cualquier parámetro

en este apartado, habría que localizarse en el fichero *main_pruebas.py.* Para probar con los parámetros deseados, únicamente habría que modificarlos y eliminar la carpeta predicciones. Posteriormente, habría que ejecutar la siguiente instrucción: ***python3 main_pruebas.py*** y de esta forma generará una nueva carpeta con las predicciones obtenidas.

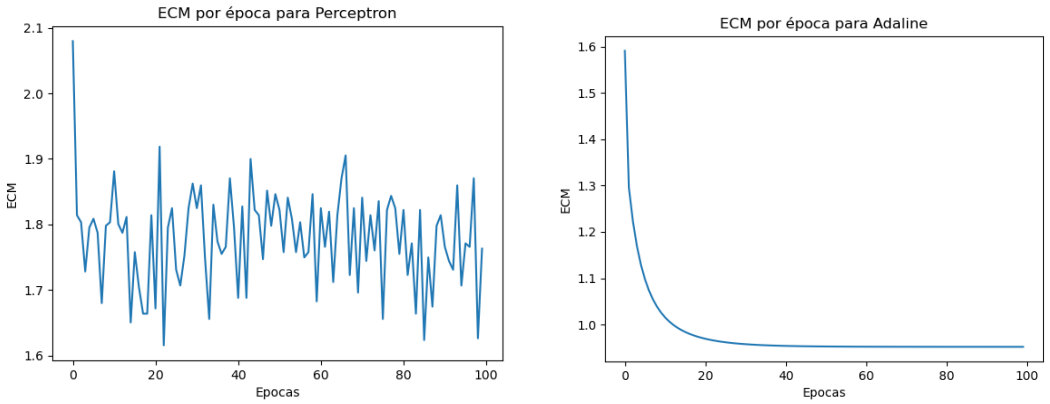

Figura 9. ECM *problema_real2* para Perceptron y Adaline

Como podemos observar, gracias a las observaciones realizadas y al cambio de parámetros de apartados anteriores, obtenemos gráficas bastante acordes a comportamientos buenos de las redes. Es cierto que se tiene una tendencia parecida (irregularidad del perceptrón a lo largo de las épocas, o mantenibilidad de Adaline). Pero esto quiere decir que se están resolviendo los problemas de forma correcta y se están intentando optimizar.