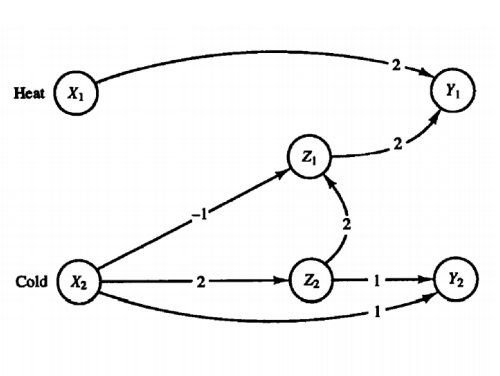
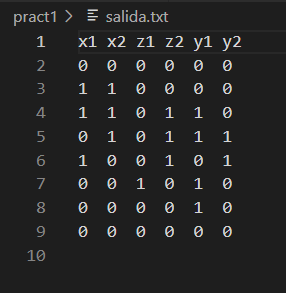
**Ej 2.1**

El diseño de la red es el siguiente donde las neuronas se dividen en una capa de las entradas(x1, x2), una capa oculta(z1, z2) y una capa de salida(y1, y2) con el peso de las conexiones que se aprecian. Las neuronas de la capa de entrada se tratan de neuronas directas donde su entrada se pasa directamente a través de la conexión, mientras que el resto de las neuronas del resto de capas se tratan de neuronas McCulloch-Pitts que cuentan con un umbral de valor 2.



El diseño es correcto ya que por un lado está la capa de entrada cuya función será transmitir las entradas a las capas posteriores, algo que se consigue al estar está constituida de neuronas directas y el resto de las neuronas de las siguientes capas al ser de McCulloch-Pitts su disparo cambia y saca 1 cuando se supera el umbral de 2 o 0 en caso contrario. Todas estas capas se añaden a la red neuronal, formándose así el diseño correcto de la red. Como prueba de su correcto funcionamiento ante la entrada de frio-calor que se nos proporciona desde el enunciado de la práctica se van obteniendo las sucesivas líneas de la siguiente imagen tras inicializar, propagar y disparar en la red neuronal, las cuales se corresponden con la solución esperada.



Con una entrada saca el valor detectado de ella, si es calor, tras el siguiente disparo, si es frio tras el segundo disparo siguiente. Se basa en estudios de fisiología sobre la percepción que tenemos tras aplicar ciertos estímulos de frio o calor.

La secuencia que seguimos es inicializar toda la red, inicializar la entrada con su valor correspondiente, dispara la capa, inicializa la capa disparada y se propaga hacia siguiente capa. A diferencia respecto otras redes neuronales, todas las capas de McCulloch-Pitts se disparan y propagan a la vez, por lo tanto, se puede llamar el método *dispara*, *inicializar* y *propagar* de la clase RedNeuronal que itera todas las capas, pero para redes como puede ser Perceptrón o Adaline, habría que seguir la secuencia de que, dispara una capa, inicializa esa capa, propaga hacia siguiente capa, dispara esa siguiente capa, inicializa, propaga etc.

**Salida de fichero para Perceptron y Adaline**

Para poder leer más rápido los resultados, en el fichero output o la salida estándar del terminal, primero se imprimirá la predicción tras aplicar el Test, luego los pesos que tienen las conexiones a las neuronas de salida y finalmente el % de acierto en Test en función de predicho y lo real.

**Ej 4.1.1**

Para implementación del perceptrón y Adaline, se han utilizado los mismos criterios de parada como han explicado en la clase de teoría, pero como hay casos como XOR que genera un bucle infinito puesto que la red no pueda resolver, se ha implementado otro criterio de parada llamado *época*, que cuando llega la época indicada, se para el entrenamiento. Por otra parte, los pesos iniciales para Perceptron siempre será 0 mientras que para Adaline, los pesos será una distribución uniforme entre -0.5 y 0.5.

En perceptrón se utiliza una tasa de aprendizaje igual a 1 y umbral a 0.2 como lo habíamos visto en clase.

En Adaline se utiliza una tasa de aprendizaje igual a 0,3 puesto que 0,1 <= 3 neuronas \* 0,3 <= 1, el umbral es igual a 0.2 y la tolerancia igual a 0.22(mejor tolerancia para el caso del AND utilizando tasa de aprendizaje 0.3). Esta tolerancia es tan alta porque el criterio de parada para Adaline es que cuando el cambio de peso más grande es menor que la tolerancia, pero el cambio de peso es , donde suele ser un número mayor que 0.1 puesto equivale -1 o 1 y más de lo mismo.

Existe un cambio de nombre respecto los métodos indicados por UML del enunciado, que son los métodos *conectar* de la clase Capa, ya que en Python no se puede saber qué tipo de clase es el atributo *capa* o *neurona*. Por lo tanto, para resolver este problema, el método *conectar* para las neuronas se va a llamarse como *conectar\_neurona*.

Tanto el script de perceptón como adaline esta dividido en 5 partes/métodos, la primera es *make\_red* donde se crea todas las capas y neuronas necesarias, *train* donde se realiza el entrenamiento tal cual explicado en la clase de teoría, *test* donde se escribe los resultados en el fichero indicado o la salida estándar, *score* para evaluar el porcentaje de acierto y *get\_weights* para sacar los pesos de las conexiones a la capa de salida.

AND frontera decisión:

Perceptron -> 2\*x1 + 3\*x2 - 4 = 0

Adaline -> 0,8\*x1 + 1\*x2 -1,5 = 0

OR frontera decisión:

Perceptron -> 2\*x1 + 2\*x2 -1 = 0

Adaline -> 1.2\*x1 +1\*x2 -0.5 = 0

NAND frontera decisión:

Perceptron -> -3\*x1 -2\*x2 + 4 = 0

Adaline -> -1.4\*x1 -1.1\*x2 + 1.7 = 0

XOR frontera decisión:

Perceptron -> -x1 + 0\*x2 -1 = 0

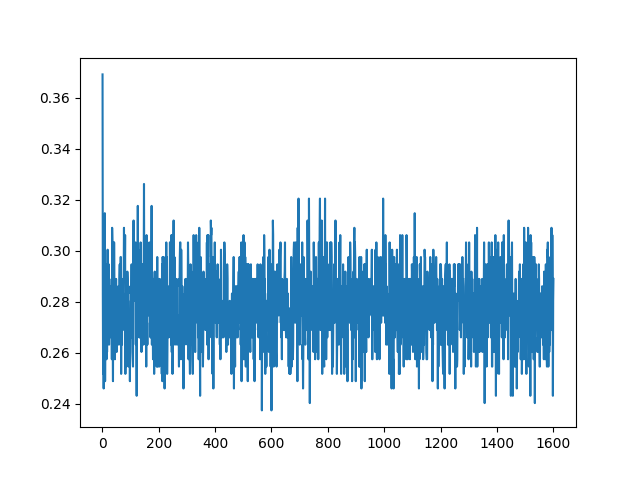
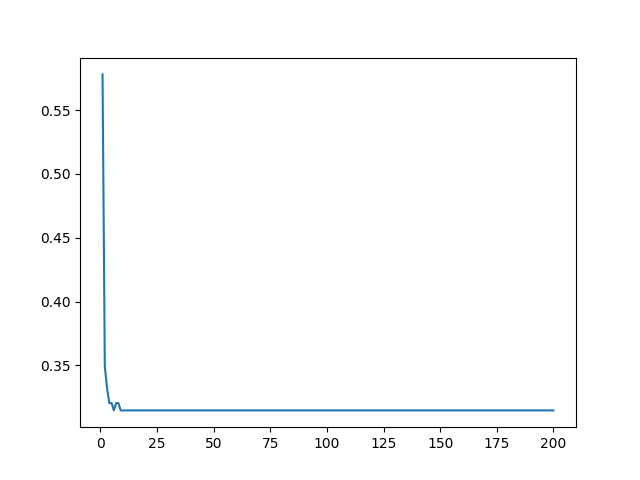
Adaline -> -0,23\*x1 -0,1\*x2 -0,5= 0

Los problemas lógicos son solucionables en todos los casos salvo en el caso de XOR, ya que no es linealmente separable. Se podría solucionar con ciertas redes neuronales de más de una capa, como el perceptrón multicapa, ya que con redes de una capa no se pueden resolver problemas que no sean linealmente separables.

**Ej 4.2.1**

La implementación de ambas redes ya está explicada en la sección anterior donde se respeta todos los pasos vistos de la clase de teoría, pero añadiendo un nuevo criterio de parada llamado *época*, por otra parte, hubo un cambio de nombre en el método conectar puesto que Python no puede saber qué tipo de atributo le está pasando.

Para el error cuadrático medio obtenido durante el entrenamiento se observa que para el perceptrón varía bastante a lo largo de las épocas en el entorno de entre el 0.32 y 0.24, mientras que en el caso de adaline se mantiene bajo y constante sin apenas cambios. Estos valores se aprecian en las siguientes gráficas:



ECM de umbral Adaline ECM de umbral Perceptron

Antes de meter con la modificación de solo un parámetro, hay que destacar que el valor de alpha para Adaline no puede ser muy alto, puesto que el cambio de peso tiene formula de , esto quiere decir que suponiendo que tenemos 9 entradas como el problema que queremos resolver, alpha = 1, salida = 1 y suponiendo que tanto los pesos iniciales como entrada inicial también son 1, el valor de , que son sumatorio de pesos por , valdrá 9+1(bias) = 10, por lo tanto, . Esto quiere decir que si alpha = 1, el cambio de peso será 1\*-9\*1 = -9 que es un valor demasiado alto, ya que para siguiente sumatorio , estará sumando 1\*-9 + 1\*-9 + 1\*9…, y resultado de será cada vez más grande. Para simular lo comentado, se ha creado un fichero llamado problemaSolo1.txt en la carpeta data y se ha ejecutado para alpha = 1 y alpha = 0.01, el resultado obtenido está en la carpeta output con el nombre de problemaSolo1\_alpha1.txt y problemaSolo1\_alpha0\_01.txt y se puede ver que los pesos de las neuronas de salida para alpha1 tiene valores gigantescos tras ejecutar muchísimas épocas. Por esa razón, se va a limitar que alpha será un número pequeño.

Otra cosa importante es que como estamos ejecutando en el modo1 donde se escoge 0.25 como proporción de Test, cada simulación se obtendrá un acierto diferente, por lo tanto, aunque que exista un hiperparámetro con mayor acierto, no quiere decir que es realmente óptimo, puesto que existe otros hiperparámetros que obtiene resultados muy parecidos.

**Variación en función de umbral.**

Para facilitar la implementación y en obtener mejores hiperparámetros, se realiza primero la variación de umbral, ya que el umbral suele ser un valor común para obtener el mejor resultado. Para el valor de época, se ha decidido utilizar 500 puesto que no es un número muy grande ni pequeño y para el valor de alpha, se ha decidido utilizar 0.1 por mismas razones. En el caso de Adaline que requiere torelancia, se ha puesto 0.01 como valor.

Los resultados se evalúan en función de % de acierto para Test y se pueden visualizar en siguientes gráficas. Para el caso de Perceptrón, el mejor umbral encontrado es 0.1 y para Adaline 0.06.

Ilustración 1 Acierto en funcion de umbral, Perceptron

Ilustración 2 Acierto en función de umbral Adaline

**Variación en función de alpha.**

Una vez tenemos el valor de umbral, se va a iterar el valor de alpha, con el fin de averiguar con que tasa de aprendizaje se aprende mejor. El valor de entrenamiento vamos a seguir con 500.

Los resultados se pueden ver en siguientes gráficas. Para Perceptrón el mejor alpha es 0.2 y para Adaline 0.1

Ilustración 3 Acierto de funcion de alpha, Perceptrón

Ilustración 4 Acierto de funcion de alpha, Adaline

Como dijimos al principio, al utilizar el modo1, existe una variación en el resultado de acierto. Por ejemplo, en Percepton, el % de acierto para alpha 0.1, posiblemente fue afectado por los datos que hay en Train y Test. Por otra parte, aunque en la gráfica de Adaline no se ve muy bien, en el alpha = 0.1, el resultado vale 96% mientras que otros son mucho más inferiores, esto concuerda bastante con la teoría puesto que en general, alpha suele tomar 0,1 <= n entrada \* alpha <= 1, como hay 9 entradas, alpha valdrá mas o menos 0.1.

**Variación en función de época.**

Normalmente, a medida de que aumente la época, se debería reducir el error del Train, pero es muy posible que la red aprenda tanto y empieza a subir el error del Test. En nuestro caso, por cuestiones de tiempo, no se ha simulado con unas épocas muy grandes, pero existe casos donde se baja ese porcentaje de acierto. Para simular esto, se ha usado el mejor umbral y alpha obtenidos en las secciones anteriores.

Los resultados obtenidos son las siguientes.

Ilustración 5 Porcentaje acierto en funcion epocas, Perceptron

Ilustración 6 Porcentaje acierto en funcion epocas, Adaline

En conclusión, los mejores hiperparámetros obtenidos para Perceptron es umbral = 0.1, alpha = 0.2 y época = 700. Para Adaline, umbral = 0.06, alpha = 0.1 y época = 400.

Todos los datos se pueden obtener en la carpeta output/parameter\_tuning/problema1 y para facilitar la visualización, se puede ver los resultados en el Excel proporcionado llamado test.xslx.

**Ej 4.3.1**

La implementación del perceptrón y del adeline está explicada en la sección 4.1.1 y la técnica para obtener los hiperparámetro será el mismo aplicado en la sección 4.2.1, la única diferencia es que en el Test, el acierto siempre valdrá 0 puesto que las etiquetas en las clases están mal colocadas, para ello, se intenta mejorar el acierto en Train para utilizarse como modelo predictor.

Todos los datos se pueden obtener en la carpeta output/parameter\_tuning/problema2 y para facilitar la visualización, se puede los resultados ver en el Excel proporcionado llamado test.xslx(recuerda que esta vez los resultados son acierto para datos del Train).

Perceptrón: umbral = 0.01, alpha = 0.1, época = 300.

Adaline: umbral = 0.01, alpha = 0.001, época = 700 y tolerancia = 0.001 puesto que alpha es un número demasiado pequeño y se parará la ejecución de forma inmediata ya que el cambio de peso es menor que esa tolerancia.

Nota: si desea comparar los resultados del problema real 2 no etiquetado con alguien script automático, recuerde eliminar las últimas del fichero donde se imprime los pesos de las conexiones a las neuronas de salida y el porcentaje de acierto que es 0% en este caso.

**Gráficas**

Ilustración 1 Acierto en funcion de umbral, Perceptron

Ilustración 2 Acierto en función de umbral Adaline

Ilustración 3 Acierto de funcion de alpha, Perceptrón

Ilustración 4 Acierto de funcion de alpha, Adaline

Ilustración 5 Porcentaje acierto en funcion epocas, Perceptron

Ilustración 6 Porcentaje acierto en funcion epocas, Adaline