**Anotaciones guía 1 – Práctica**

Drive videos guía 1: <https://drive.google.com/drive/folders/1ArsG-8Ubf5ocHJEZiFmXq1KcH72uno-P>

**RECORDAR QUE EN 2023 CAMBIARON LA GUIA. SOLO EL EJERCICIO 1 EXPLICADO EN ESTE WORD SERIA PARTE DE LA GUIA 1 DE 2023**

La guía 1 (2022) está dividida en dos partes: los dos primeros ejercicios corresponden a perceptrón simple (ejs. 1 y 2, que entran en la evaluación 1) y los otros dos a perceptrón multicapa (ejs. 3 y 4 para la evaluación 2).

*Algo para recordar sobre la velocidad de aprendizaje “μ”:  
Si* ***μ es chico****, más pequeños son los cambios en los pesos sinápticos, pero más lento es el aprendizaje.* ***Si μ es grande****, resulta en grandes cambios en los pesos sinápticos, pero puede provocar inestabilidad (oscilaciones).* Debemos pensar que si yo conceptualmente pongo una *velocidad de aprendizaje* ***muy grande***, la neurona se va a aprender el ejemplo que le acabo de mostrar pero se va a olvidar todo lo que aprendió antes, porque va a modificar tanto los pesos que va a perder o se va a olvidar todo lo que aprendió de los ejemplos anteriores. Y si yo le pusiera una *velocidad aprendizaje* ***muy chica***, conceptualmente va a aprender muy lento, le voy a tener que mostrar muchísimas veces las cosas para que pueda aprender, así que hay un compromiso en esa velocidad de aprendizaje que luego vamos a analizar.

Comentarios en la clase:

-En inteligencia artificial se busca que una computadora haga cosas que son propias del ser humano (capacidades como la creatividad, etc.).

-Inteligencia computacional es una sub área de la inteligencia artificial. La IA es todo (maching learning, redes neuronales, redes profundas, etc.).

-En la **fórmula para actualizar los pesos**, se multiplica por las entradas al final porque van a tener distintas escalas, por ejemplo, temperatura, humedad, etc.

Otra cosa que se suele hacer es normalizar las entradas al comienzo y usar un mismo gamma (velocidad de aprendizaje) para todas.

-Hay veces que los datos de entrenamiento los dividimos en entrenamiento y otra parte para validación, para validar con datos que no se usaron en el entrenamiento, y luego otro grupo para pruebas al final.   
Ahora no vamos a usar eso, solo usamos un mismo grupo de datos para entrenamiento y validación y otro grupo para pruebas.

-No hay una sola recta que resuelva el problema, hay infinitas, por lo tanto, no hay un solo grupo de pesos.

El OR es un problema muy fácil de resolver, entonces lo debería resolver con 100% de efectividad, mientras que el XOR no, ahí tendremos que ver hasta qué porcentaje de efectividad se puede llegar usando un solo perceptrón simple (en el archivo “Anotaciones importantes” explico eso en detalle).

-Y la recta cambia cada vez que cambian los pesos, así que la gráfica hay que actualizarla en cada patrón, no en cada época.

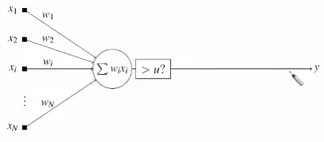
**Introducción**

-Se comenzó con un repaso sobre la ***neurona biológica***: entradas que llegan a través de las dendritas, se procesan de alguna forma en el núcleo y se produce o no una salida que viaja a través del axón.

-Luego se pasó al ***modelo de neurona*** que vamos a usar, donde tendremos un vector de entradas (Xi) que son mediciones relacionadas a algo, por ejemplo, distintas propiedades de un vino (color, acidez, etc.) y el problema podría ser determinar qué tipo de vino es (blanco, tinto). Esas entradas están conectadas al cuerpo de la neurona a través de unos pesos sinápticos (Wi).

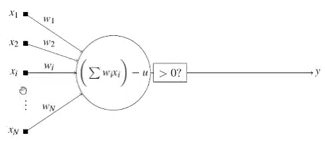
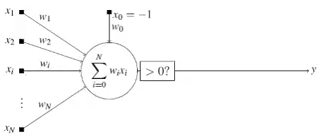
El procesamiento que se hace en el núcleo es una suma ponderada de las entradas con los pesos.

Luego, si esa suma ponderada es mayor a un umbral “u”, la neurona da una salida +1 y sino una salida -1, en el ejemplo del vino podría ser 1 -> vino blanco, -1 -> vino tinto.



Si incluimos el umbral dentro de la suma ponderada (se lo restamos a la suma), simplificamos un poco el modelo y el umbral con el que debemos comparar ahora pasa a ser 0 (imagen de abajo a la izquierda). Y se puede simplificar aún más si el umbral lo transformamos en un peso más (imagen de abajo a la derecha). Para eso, en vez de empezar la suma desde i=1, empezamos desde i=0, y añadimos una entrada X0 = -1 que queda fija siempre y el peso W0 asociado a esa entrada va a ser el valor de “u” (del umbral).

Así ahora podemos hacer una suma más sencilla, y el W0 (el valor del umbral) también va a poder ser aprendido por la neurona como los otros pesos.

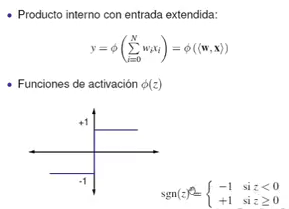
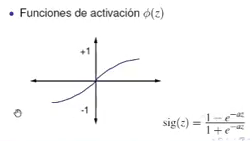
 

Matemáticamente esto lo podemos plantear como la siguiente ecuación, donde la suma ponderada es igual al producto interno entre dos vectores, que son el vector de entradas “**x**” y el vector de pesos “**w**”. El resultado de ese producto interno, que es un número, es el argumento de una *función de activación no lineal* ф.

Recordemos que la salida de la sumatoria nos va a dar algo lineal, que lo pasamos por una función no lineal ф, por ejemplo, la función signo, y esa función ф es la que nos da +1 si el argumento es mayor o igual a 0, o -1 si el argumento es menor a 0, lo cual será la salida final que nos interesa para hacer la clasificación.

Por ejemplo, teniendo vectores “**x**” y “**w**” columnas, la salida lineal luego será z = **x**T**w**, donde sería transpuesto porque fila por columna me daría un número, ese “z” lo pasamos por la función no lineal, por ejemplo, la función signo, y obtenemos una salida final “y” que será +1 o -1.

En el caso de la imagen de abajo a la izquierda es la función signo (sgn), pero también se podría usar otra función como función no lineal, por ejemplo, si queremos una función de activación más suave, que vaya de -1 a +1 de forma más suave, como en la imagen de la derecha donde tenemos la función sigmoidea, en vez de ser un escalón brusco como en la función signo. Eso sirve cuando necesitamos que sea suave y derivable.

La clave es cómo hago para determinar los valores del vector de pesos “w” de forma que, cuando le muestre un patrón de entradas que corresponden a un vino blanco por ejemplo, esa neurona me indique la salida correcta.

Esto se logra con un algoritmo de aprendizaje basado en ejemplos.

Para entrenar voy a tener un conjunto de valores para el vector de entradas “**x**” y la etiqueta asociada “yd” (salida correcta que debería dar con esas entradas), si era un vino blanco o tinto en el ejemplo. Es decir, tenemos pares (**x**, yd), que se denominan “**patrones**” (cada fila del archivo será un patrón). Y la idea del algoritmo de aprendizaje es ir mostrándole a la neurona uno por uno esos patrones y fijarme, dado los pesos que tiene, la salida que nos da, y si la salida no coincide con lo que nos debería haber dado (con la salida deseada “yd”), hacemos una *corrección* sobre los pesos para que la próxima vez que le muestre esas entradas el resultado cambie y pueda obtener el valor correcto (o nos vayamos acercando con las iteraciones).

Ahora vemos el *algoritmo del perceptrón simple*.

1. Se inicializa el vector de pesos “**w**” con valores al azar entre [-0.5; 0.5] por ejemplo.
2. Para cada ejemplo de entrenamiento, obtenemos la salida de la neurona (y), si esa salida fue correcta no hacemos nada y si fue incorrecta adaptamos o corregimos los pesos.
3. Volvemos al paso 2 y hacemos lo mismo hasta que la neurona no cometa errores o hasta que los errores sean menores a cierto porcentaje que nos interese.

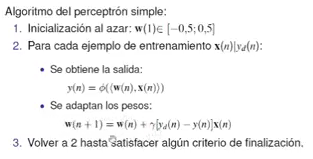
Como vemos en la imagen, la función de adaptación de pesos nos dice que el nuevo peso w(n+1) será igual al peso actual w(n) más un factor de corrección, que vale esa constante “γ” (gamma) por la “yd” (salida deseada) menos la “y” (salida real), y eso multiplicado por el vector de entradas “**x**”.

Iterando muchas veces esos pasos, los pesos se irán adaptando hasta obtener un resultado correcto.

La constante “γ” se llama *tasa de aprendizaje*, la cual permite regular que tanto se van a ir acomodando los pesos en cada iteración, por lo tanto, *que tanto va a ir aprendiendo la neurona*.

Hay que tener cuidado porque una tasa de aprendizaje alta podría hacer que la neurona se aprenda mucho lo último pero se olvide todo lo anterior (en uno de los vídeos de Dot CSV sobre redes neuronales mostraba eso).

Si achico la tasa de aprendizaje, los pasitos que dé van a ser más chicos, es decir, modifico menos los pesos en cada paso, pero eso hace que el aprendizaje sea más lento, porque cuánto más chica es la tasa de aprendizaje, menos modifico los pesos.



El resultado sigue siendo un vector, así que está bien.

**Ejercicio 1 – Parte 1 - Implementación**

La extensión “***csv***” en los archivos indica que son archivos de texto con *valores separados por coma*, donde tenemos un ***patrón*** por fila: x1, x2, x3, …, yd.

Es decir*, un patrón es un conjunto de entradas con su correspondiente salida* (una fila completa).

Debemos implementar el algoritmo que acabamos de ver en la introducción, teniendo en cuenta que lo querríamos usar para distintos problemas, sin saber ahora la dimensión del vector “x”, por eso dice con una *cantidad variable de entradas*.

-Lectura de los patrones de entrenamiento: serían las entradas y salidas deseadas.

El formato en el archivo de texto sería: x1, x2, x3,..., xn, yd

Al final se le pone la salida deseada con esas entradas (yd).

Así tendríamos varias filas en ese archivo, con valores para las entradas y valores para la salida deseada.

El programa debería poder leer un archivo así y separar lo que es “X” y lo que es “yd”, por ejemplo, poner en una matriz todos los vectores con los valores de “x” en cada fila y luego un vector con la salida deseada (yd) para cada uno de esos patrones de entradas.

Además, *recordar* que añadimos una entrada X0 = -1 para el umbral, por lo cual, en el archivo se debe escribir el -1 al comienzo de cada fila, y sino directamente generarlo desde el código como hicimos, donde en cada iteración “i” de un for lo que hacemos es poner ese -1 al comienzo y luego las demás entradas correspondientes a la fila “i”. Y, por otro lado, la salida correspondiente que se encuentra al final de la fila en el archivo.

-Cuando lo tenemos organizado como un vector de entradas, cada “patrón” sería una fila de datos:

-1 x1, x2, x3, …, xn, yd

Pero un patrón podría ser una imagen o lo que sea.

-Criterio de finalización del entrenamiento: supongamos que tenemos un archivo con 50 ejemplos que se los vamos presentando y corrigiendo los pesos. Una vez que termina el archivo se vuelve al comienzo y así se pasa varias veces y se van corrigiendo los pesos. Cada vez que se pasan todos los patrones es una época.

¿Cuántas veces pasamos los patrones? Mientras no se cumpla cierta condición de finalización. La básica es lo que indica el punto siguiente con el número máximo de épocas de entrenamiento.

-Se llama “*época*” al pasaje de todos los patrones una vez (es decir, todo el archivo), si tenía 50 patrones, una vez que pase los 50 completé una época de entrenamiento. Yo podría decir por ejemplo que el *número máximo de épocas* sea 100, así si pase 100 veces los patrones corto el algoritmo.

Y además, se suele combinar con otro tipo de parada, por ejemplo, si el error ya es 0 no tiene sentido seguir sumando épocas, seguir pasándole patrones.

*En vez de error 0, algo más realista sería poner un umbral de error*. Por ejemplo, *si el porcentaje de errores que obtuvo en una época es menor al 3% (depende del problema) ya me es suficiente así que corto el algoritmo*.

Sería un umbral de error aceptable.

-Otra cosa que debe poder elegir el programa es la *tasa de aprendizaje*. *Ese parámetro “γ” (gamma) que explicamos antes regula que tanto se aprende en cada patrón*. En general “γ” es un valor chico.

En la teoría se explicó el problema que pasa si es un valor grande, que se puede aprender lo último que vio pero olvidarse todo lo anterior.

Para determinar el valor de “γ” se corre el modelo muchas veces y se ve cuál nos da mejores resultados.

Y hay estrategias donde conviene cambiar la tasa de aprendizaje durante el proceso, por ejemplo, empezar con un valor más grande y luego achicarla cuando nos estamos acercando al resultado que queremos. Eso tampoco lo vamos a hacer.

-Y el último ítem sobre las pruebas se refiere a que lo que nos interesa en estos algoritmos es que el entrenamiento se dé de tal forma que la red aprenda a generalizar. Con “*generalizar*” queremos decir que podemos tener un sistema que, dado un conjunto de patrones, me los clasifique perfecto, entonces para los ejemplos que le di para entrenar funciona perfecto, pero ¿Qué pasará cuando le doy una muestra que no la tuvo para el entrenamiento? ¿Va a funcionar bien o mal? No lo sabemos de antemano.

Eso se llama *“****capacidad de generalización****”, que la red aprenda características de cada clase de manera que cuando venga una muestra desconocida (que no haya estado como parte del entrenamiento) la red sea capaz de clasificarla de forma adecuada.*

¿Cómo podemos garantizar que el método tenga esa capacidad de generalizar? para verificar eso lo que se hace es, luego de haber terminado el entrenamiento, se hace una etapa de pruebas con un conjunto de valores que no se hayan usado en el entrenamiento, y ahí veo que tal funciona.

Por ejemplo, corto el entrenamiento cuando llego a un error del 3% (el umbral de error que haya definido usar), y luego si pruebo con los datos desconocidos y me da un error de 3% también, significa que está bien, que aprendió bien, tiene la capacidad de generalizar. Pero si el error en esos datos que el algoritmo no conocía me da 80% significa que no aprendió bien, no funciona con datos que no vio antes.

-Lo que debemos hacer en este ejercicio es un código de entrenamiento:

1- Inicializamos los pesos “w” con valores al azar, por ejemplo, entre -0.5 y 0.5.

2- Bucle para cada época

Bucle para cada patrón:

2.a- Se calcula la salida.

2.b- Se actualizan los pesos.

Cierro bucle

3- Repetimos el bucle mientras el número de épocas sea menor al número de épocas máximas, y mientras se cumpla el otro criterio de error (esto sería la condición de salida, el cierre de bucle para cada época).

Esas condiciones del punto 3 quiere decir que, cuando termina una época, yo tengo que probar con todos los patrones para calcular el error, es decir, con los pesos como están, sin actualizarlos en este bucle, paso todos los patrones por la red y voy contando cuantos errores tuve.

Entonces, si modificamos un poquito el algoritmo para tener en cuenta eso nos quedaría lo siguiente, y además añadimos una rutina de prueba o test al final, con otros datos que aún no vio el modelo y con los pesos ya actualizados que obtuvimos.

Los datos de entrenamiento en realidad se los subdivide en *entrenamiento* y *validación, para que la validación también se haga con datos que la red aún no vio* (estos últimos para usarlos en el punto 3 de abajo, pero no lo hacemos en la práctica, usamos los mismos datos tanto en el entrenamiento como en la validación para ver el error), y después aparte tenemos un grupo de datos de *prueba* que esos no se usan en el entrenamiento.

1. Inicializar los pesos.
2. -------Mientras la condición de parada sea falsa:

-Para cada patrón:

2.1- Calculamos la salida lineal (producto interno).

2.2- Se aplica la no linealidad (función signo u otra, para tener como salida final +1 o -1).

2.3- Actualizamos los pesos.

%Verificación:

Para cada patrón (sin actualizar los pesos y usando datos ya usados en el entrenamiento)

3.1- Calculamos la salida lineal.

3.2- Se aplica la no linealidad (función signo u otra, para tener como salida final +1 o -1).

3.2- Contamos los errores (cuando se alcance cierto porcentaje de error aceptable corto)

-------Evaluamos la condición de salida (cierre bucle 2).

3- Rutina de prueba o test (con datos que no se usaron para el entrenamiento):

-Para cada patrón: (con los pesos finales ya entrenados)

-Calculo la salida.

-Calculo un índice de desempeño en esa época (error, porcentaje de error u otro).

Entonces tenemos dos rutinas separadas (entrenamiento serían los puntos 1 y 2, y rutina de prueba el 3). Vamos a necesitar dos archivos, uno con los datos de entrenamiento y otro para las pruebas.

Con el archivo de entrenamiento hacemos la primera rutina, lo validamos y una vez que consideramos que se completó adecuadamente, vamos a usar el archivo de prueba (con los pesos ya entrenados) y ver como se desempeña.

-Otra cosa que se suele hacer, si la persona que cargo los datos los cargo ordenados (ya clasificados en orden) es *ordenarlos de forma aleatoria*, para que durante el entrenamiento la red aprenda mejor, porque si no le van a llegar todos los datos de una categoría, luego todos los de otra categoría y así.

-Algo que comentó el profe en la clase es que sería mejor hacer una actualización de los pesos en modo “batch” (o “por lotes”) que significa hacer la actualización según un promedio una vez que se pasó por todos los patrones, no actualizar en cada patrón. Pero no se suele hacer porque lleva mucho más tiempo.

Y otra opción intermedia es usar mini baches, por ejemplo, tomar 10 patrones y ahí actualizar.

Y otro detalle es que se suelen normalizar las entradas para que todas estén en la misma escala, porque puede traer problemas.

*(Esas cosas no lo vamos a hacer, pero nos explicó para que sepamos que existen).*

**Ejercicio 1 - Parte 2 – Verificación del funcionamiento**

Los archivos “trn” son para entrenamiento y los “tst” para test (prueba).

Esta parte se corresponde a los incisos a) y b), donde se debe verificar el funcionamiento del programa.

1. En la teoría vimos detallado el **OR**. En la gráfica de abajo se representó la función OR con entradas x1, x2 y salida “y”, tanto en una tabla de verdad como en el plano.

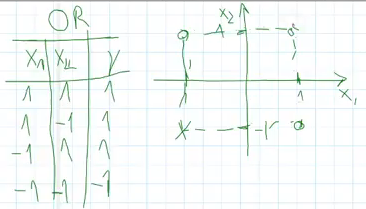
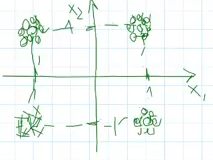
Lo que simulamos en este inciso a) es una nube de puntos alrededor de las coordenadas (1,1), (1,-1), (-1,1) y (-1,-1), donde la función OR nos da +1 o -1, como se marcan los puntos en la imagen de abajo a la derecha, y esos puntos estarán con una dispersión del 5% como dice el enunciado, ya que lo que se representaría con el archivo de datos es una medición real de algo, donde no siempre nos va a dar justo +1 o -1, sino que al ser una medición real nos pueden dar distintos valores: 0.2, 0.8, -1.2, etc.

Entonces en esos archivos de texto tendremos algo como:

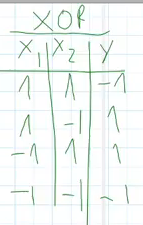
0.98, 1.01, 1

1.2, 0.4, 1

Y así con distintos valores de x1, x2, yd

Lo mismo para el caso del **XOR**



La idea en el ***inciso a)*** es que usemos el archivo de entrenamiento del OR en nuestro perceptrón simple, y una vez que se llega al criterio de parada lo probamos con el archivo de prueba y reportamos los resultados. Y luego hacemos lo mismo pero con el archivo de entrenamiento y de prueba del XOR.

*Es probable que con el XOR nos encontremos con alguna dificultad que requiera que reveamos el criterio de parada, para eso hay que ver cuánto sería el error esperable.*

***Inciso b)*** para ver cómo hace para separar en una categoría o la otra, tenemos que ver la fórmula que implementa el perceptrón simple. En este caso que tenemos dos entradas, lo que hacía el perceptrón simple era hacer una función no lineal ф que arroja valores +1 o -1 (por ejemplo, usando la función signo).

En la imagen de abajo se representó esa función ф(z), que da 1 con valores de z ≥ 0 y da -1 con z < 0, donde “z” sería el resultado lineal que nos da el producto interno o la suma ponderada.

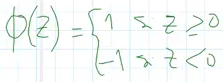
Entonces ¿*dónde está el límite o frontera para decir si es de una clase o la otra*? En z = 0, ya que un poquito más grande que 0 daría +1 y un poquito más chico que 0 daría -1.

Ahora, sabemos que “z” era z = x1w1 + x2w2 – w0.

Si igualamos eso a 0 y despejamos para obtener por ejemplo x2 en función de x1 tenemos:

x2 = w0/w2 – w1/w2\*x1

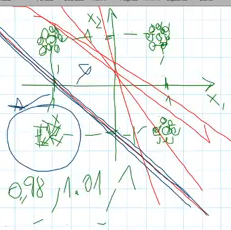
Entonces los puntos x1, x2 que satisfacen esa ecuación son aquellos donde z = 0, por lo tanto, son la *frontera de decisión entre una clase y la otra*. ¿Y qué curva describe esa ecuación? *Una* ***recta****, de pendiente negativa w1/w2 y ordenada al origen w0/w2*.



*Esto quiere decir que el perceptrón simple traza una recta en función de x1 y x2, es decir que, para un lado de la recta todos los puntos x1, x2 van a dar +1 y para el otro lado de la recta todos los puntos van a dar -1 (como se explicó en la teoría).*

Al ir entrenando el perceptrón simple, la pendiente (-w1/w2) y la ordenada al origen (w0/w2) de la recta dependen de los valores de los pesos, entonces *si en el entrenamiento voy cambiando los pesos, la recta se va a ir moviendo durante el entrenamiento a medida que van pasando las iteraciones* (rectas rojas de la imagen de abajo), se actualiza la recta luego de cada patrón, porque es donde se van actualizando los pesos, y llegará un momento donde la recta quedará en una zona donde encontremos una solución del problema (recta azul de la imagen, donde ahí se cumplen las condiciones para el OR), como en la teoría cuando se hablaba del OR y se dibujaba esto con la recta.

*La red que tenga parámetros w0, w1 y w2 que permitan obtener esa recta adecuada, es una solución al problema.*



Entonces, en este inciso se pide graficar esto que estuvimos viendo. Graficar una distribución de los puntos y que a medida que vamos iterando en el entrenamiento vayamos dibujando las rectas para ir viendo cómo se van moviendo, hasta que en algún momento se estabiliza como en la azul de la imagen de arriba y ahí llegamos a la solución del problema.

Una recomendación es que entre recta y recta pongamos alguna pausa para que no pase tan rápido, y otra recomendación es que vayamos dibujando la recta cada presentación de uno o dos patrones con una pausa, para ir viendo cómo se mueve. No hacer la gráfica una vez por época, porque quizás en una época ya se entrenó el modelo y no vemos la diferencia.

-***Recordar*** que en el caso del OR podemos crear esa recta sin problema, pero en el XOR vamos a tener dos puntos que den +1 y los otros dos puntos enfrentados -1 (si vemos los dibujos de arriba), entonces el perceptron simple no puede resolver el problema del XOR con una recta al 100% de efectividad.

*(Explicado en otro archivo con anotaciones importantes sobre el XOR, en la carpeta de esta misma guía).*

-***Importante:***

Una vez que funciona el algoritmo, hay que ver cosas como cuánto error me podría dar como máximo, analizar distintos valores de la tasa de aprendizaje, etc.

***Cosas que se comentaron en la evaluación***:

-Hay *infinitas posibles soluciones*.

Partiendo del mismo punto, patrones y orden, se llegaría a la misma solución (misma recta), pero si hay algo de aleatoriedad ya no.

Y *no hay una solución “optima*” o mejor que las demás, porque si estoy midiendo un error y muchas rectas son solución, todas son igual de válidas en este algoritmo que tenemos, porque todas son equivalentes y dan el mismo error.

-Podríamos suponer que una recta que pase en el medio de las nubes de puntos (igual distancia de todos) sería la “mejor”, pero es una suposición, porque puede haber distinta varianza en los datos.

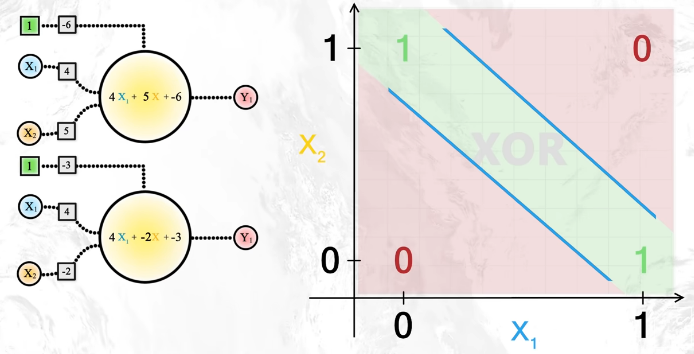
-La **hipótesis** principal que hacemos es que los datos son significativos del problema, por eso es importante el dataset que armamos.

-En el XOR preguntarnos: ¿Qué es lo mejor que puede hacer esta arquitectura? ¿Puede resolver con 100% de efectividad? ¿Con 70%? Y así...

-En este vídeo de **Dot CSV**: <https://www.youtube.com/watch?v=MRIv2IwFTPg> hace toda una explicación de qué es una neurona, muy similar a lo que estuvimos viendo en teoría, y al final muestra la separación con la recta en un problema AND, un OR y el XOR, diciendo que este último no se puede resolver con una única neurona, ya que no podemos separar la región en dos con una recta y cumplir con el problema al 100%, es decir, es imposible separar de forma lineal las dos clases. *El OR es “linealmente separable”, pero el XOR no*.

La solución sería añadir una segunda neurona, y de esa forma tendríamos dos rectas que podríamos usar para separar ambos grupos, como en la imagen de abajo.

Todo eso también se explica en los vídeos de teoría 006 y 007 de la teoría, donde se usan tres neuronas, dos para separar la región de esa forma con dos rectas y una tercera que de +1 en toda la región del centro que es donde nos interesa.



Lista de vídeos de **Dot CSV** sobre Redes neuronales:

<https://www.youtube.com/playlist?list=PL-Ogd76BhmcB9OjPucsnc2-piEE96jJDQ>

**Ejercicio 2 - Introducción**

En este ejercicio vamos a analizar como *estimar mejor la capacidad de generalización*.

Dijimos que tenemos un conjunto de datos para el entrenamiento y otro conjunto de datos para las pruebas. Este es un *problema que se da cuando tenemos un conjunto chico de datos*, donde puede pasar que los que dejamos apartados para usar en la etapa de pruebas sean justo todos casos fáciles, entonces hacemos la prueba y funciona perfecto, pero justo no había ningún caso “confuso” para el algoritmo, algún valor en las fronteras o casos donde el algoritmo pueda tener más probabilidades de fallar, así que cuando lo use en la vida real y aparezcan casos de ese tipo seguramente haya problemas.

*Puede ser que la muestra que yo tome para hacer la prueba no sea* ***representativa*** *realmente de la complejidad del problema, o al revés, que lo que no sea representativo sean todos los datos que quedaron para el entrenamiento* y por lo tanto entrene al clasificador con casos fáciles pero cuando vaya a probar con el conjunto de prueba donde quedaron los datos más difíciles me encuentre con que falla demasiado.

Entonces, eso hace que haya un sesgo generado por la elección de los datos de entrenamiento y prueba.

*Esto es importante cuando yo quiero comparar clasificadores*, porque quiero saber cuál me conviene usar, entonces ¿cómo decido cuál es mejor? Si hago la división de datos en entrenamiento y prueba, puede ser que esos datos que quedaron para la prueba le favorezcan más a un clasificador que a otro, pero no porque dicho clasificador sea mejor que el otro, entonces ahí hay un sesgo inducido por la selección de los datos.

¿Cuál es la forma de resolver esto? En la teoría vamos a ver distintos ejemplos de “***validación cruzada***”.

Para este ejercicio vamos a usar una validación cruzada sencilla, donde ***la idea es***, en vez de tener los datos de entrenamiento y prueba ya separados como en el ejercicio 1, ahora tendríamos todo junto, y un esquema podría ser decir que dejamos el 80% de los datos para entrenar y el 20% lo dejamos para probar, pero si hacemos una sola vez eso, podríamos tener el problema que mencionamos antes, de que nos hayan quedado los datos fáciles en la parte de pruebas.

Entonces, para esta forma sencilla de validación cruzada lo que se hace es repetir ese proceso “n” veces, por ejemplo, 10 veces, donde cada vez tomo al azar 80% de los datos y los coloco en un grupo para entrenar y el 20% restante en otro grupo para probar, pero después repito lo mismo ahora sacando otro 80% para entrenar y 20% para probar, y eso lo vuelvo a repetir “n” veces, siempre tomando los datos al azar.

Haciendo esto, al final del proceso entreno al clasificador con un grupo de 80% de los datos y pruebo con el grupo de 20%, anoto ese resultado. Luego entreno al clasificador con otro 80% de datos y lo pruebo con otro 20% de datos y anoto el resultado. Y así sucesivamente las “n” veces.

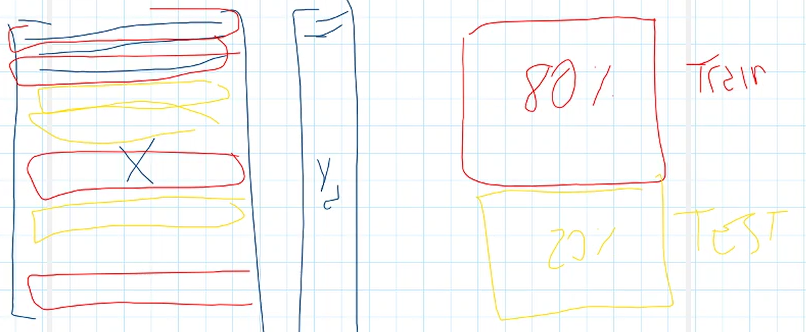
Al terminar el proceso, tengo los resultados de las pruebas sobre las “n” particiones que hicimos, y puedo tener mucho mejor caracterizado el desempeño del clasificador ya que puedo sacar por ejemplo la ***media*** de todas las pruebas, la *varianza*, *desviación*, etc.

*Eso se llama esquema de* ***validación cruzada por hold down****. El “hold down” (ver como se escribe) quiere decir que me reservo una parte para las pruebas, y la validación cruzada está en que hago varias repeticiones del mismo experimento (obteniendo varias particiones distintas).*

Lo normal es que las particiones que uno hace las grabe en el disco, porque *hacemos estas cosas cuando queremos comparar clasificadores*, y si quiero comparar dos clasificadores los debería haber entrenado con los mismos datos, porque si no de nuevo puedo tener el problema que mencionamos al comienzo de poder estar beneficiando a uno de los clasificadores sin querer por usar ciertos datos. Entonces lo mejor es tener esas particiones grabadas y así podemos ir probando distintos clasificadores con los mismos datos y poder compararlos.

¿*Cómo se implementa esta partición*?

La imagen de abajo representaría que tenemos una matriz de entradas “X” donde tendríamos un patrón por cada fila, y un vector “yd” con las correspondientes salidas deseadas. De esas entradas tomo al azar el 80% y las guardo en un conjunto para entrenar (el rojo) y el 20% restante en otro conjunto para probar (el amarillo). Guardo eso y luego repito la misma operación pero tomando al azar otro 80% para entrenar y 20% para probar, lo guardo y así sucesivamente “n” veces.

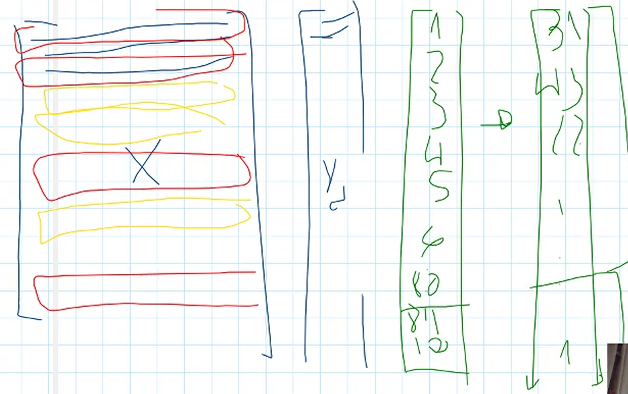


Ahora, la forma eficiente para hacer eso no es ir armando nuevas matrices con los datos ordenados de forma distinta, sino que sería mejor trabajar con un ***vector de índices***.

En la imagen de abajo, el verde sería un vector de índices donde los números corresponden al patrón 1, 2,..., 100 (supongamos que son 100 para tener 80% y 20% fácil).

*Ese vector de índices lo puedo ir ordenando de distintas formas* y sacar el 80% inicial para entrenamiento y el 20% restante para pruebas, incluso *puedo* ***guardar solo el vector de índices*** *y así no requiero guardar todos los datos*. Así, por ejemplo, en el segundo vector de índices, cuando quiero buscar los datos de entrenamiento busco en la matriz con los datos el patrón 31, 43, 22, ... y los uso para entrenamiento, y luego busco los de prueba.

Luego cuando vaya a usar otro clasificador, simplemente acceso a los datos de acuerdo a los vectores de incides que guarde antes, así lo entreno y pruebo igual que al otro clasificador y los puedo comparar.



-Este ejercicio 2 sería un *equivalente al OR y XOR, pero en 3D.*

*(Las tablas no dan que sea un OR o XOR, el equivalente es porque el inciso “a” es* ***no linealmente separable*** *(como el XOR) y el inciso “b” sí es* ***linealmente separable*** *(como el OR), por el cambio que se hizo del -1 en el tercer renglón como indica en enunciado). Acá no usamos rectas, sino planos para separar las categorías (no hace falta graficarlos).*

*Mostrar cosas extras* como el ***error en cada época*** o gráficas, para ir viendo cómo funciona, si aprende y demás.

***Inciso a)***

Cantidad de particiones = cuántas veces voy a repetir el proceso.

En la tabla 1 vemos algo similar a lo que estuvimos viendo con el OR y XOR, pero ahora en tres dimensiones (x1, x2, x3). Vamos a simular de nuevo que son mediciones verdaderas, que no nos dan justo 1 o -1, por eso en las imágenes de las gráficas no se ven puntos, sino “nubes” de puntos rojos y negros, que representan la dispersión de los datos.

Esa tabla de verdad es un equivalente en 3D a un XOR (*porque es no linealmente separable por un plano*).

La consigna pide realizar la validación cruzada del perceptron simple con 5 particiones de entrenamiento y prueba (5 veces repetimos el proceso de ir separando los datos en grupo de entrenamiento y grupo de prueba), con una relación de 80/20, es decir, 80% de datos para el entrenamiento y 20% para la prueba.

***Inciso b)***

Dice que se modificó el tercer renglón de la tabla, poniendo que la salida deseada sea +1, donde la gráfica (b) representaría ese cambio (el punto en esas coordenadas que antes era rojo ahora es negro), y tenemos archivos con distintos porcentajes de dispersión en los datos.

Se pide realizar la validación cruzada del perceptrón simple ahora con 10 particiones de entrenamiento y prueba, con relación 80/20.

Así que, con esos conjuntos de datos, debemos correr el programa que hayamos creado para generar particiones y así para cada partición usamos una parte para entrenar y otra parte para probar, y al final *reportamos los resultados* de las pruebas en los 10 casos, con una *media y desvío*.

-Otras cosas una vez terminado el ejercicio:

Ver en qué afecto ese cambio que se hizo del +1 y detalles al usar más particiones.

También podríamos probar usar otros valores que no sean 80% para entrenamiento y 20% para prueba, por si encontramos algo interesante.

***Cosas que se dijeron en clases***:

-Validación cruzada: me sirve para tener más confianza en que un modelo es mejor que otro, pero no me da los parámetros o pesos “w” óptimos (porque en cada partición voy a tener los “w”, por eso no es lo que busco), sino que lo uso para decidir si un modelo “A” es mejor que otro modelo “B”, y luego con el mejor busco los “w” óptimos haciendo las correcciones de los pesos y demás.

Un modelo nuevo no sería otro perceptrón simple con algún cambio, sino que sería otra arquitectura, por ejemplo, un perceptrón multicapa, o comparar uno de dos capas contra uno de tres capas y así.

-*En cada nueva partición se vuelven a inicializar los pesos*, no se usan los pesos del anterior.

-Validación cruzada por “**hold down**” es el que usamos nosotros. *El “hold down” (ver como se escribe) quiere decir que me reservo una parte para las pruebas, y la validación cruzada está en que hago varias repeticiones del mismo experimento (obteniendo varias particiones distintas).*

Hay otro tipo de validación cruzada es **k-fold**, que permite asegurarse que todos los patrones se puedan usar para la etapa de pruebas alguna vez, porque el hold down no me garantiza eso, puede que haya patrones que nunca hayan caído en la parte de prueba.

Después en el ejercicio 4 por ejemplo usamos otra variante de validación cruzada que es leave\_k\_out y leave\_one\_out.

-Los generadores de números aleatorios en realidad nunca son aleatorios, son funciones estadísticas, pero a fines prácticos acá nos sirve (usamos una función random para los índices).

-Las nubes de puntos es porque al tener mediciones no siempre vamos a tener las coordenadas justas (1, 1, 1), (1, -1, 1) y demás, sino que las mediciones van a estar con cierta variación alrededor de esas coordenadas.

-Inciso b)

El yd = 1 que dice ya está modificado en el archivo, no lo tenemos que modificar nosotros.

-Funciones para leer y escribir archivo CSV: csvwrite y csvread.

Para leer, usando “load” funcionaba igual.

Tener cuidado porque al usar csvread nos creaba una dimensión de más y eso nos dio problemas que demoramos un montón en solucionar, hasta que el profe nos dijo que usemos la función “squeeze” para eliminar las dimensiones redundantes que era lo que nos daba problemas (en la función de entrenamiento nos pasaba eso).

**Comentarios durante la evaluación de estos dos ejercicios:**

-En el archivo de test del XOR en el ejercicio 1, como hay 112 (-1) y 88 (+1) en las salidas deseadas, también afecta para que nos dé cosas como 0.30 (30%) y no 0.25 (25%) como esperaríamos, porque no hay la misma cantidad de datos de cada categoría, entonces puede que la recta separe bien pero hay más datos de una de las categorías mal clasificadas o algo por el estilo.

-*Más particiones significa más confianza en el resultado, no mejor resultado*.

Si uso una sola partición puede pasar que me dé por ejemplo 20% de error o que me de 60% de error, todo depende de los datos que entren en esa partición. Por eso al usar varias particiones luego sacamos una media de todas y eso nos da más confianza en el resultado.

-En el ejercicio 2, en vez de rectas tendríamos *planos* que separan las categorías (no hace falta graficarlas). El inciso “a” es no linealmente separable y el inciso “b” sí es linealmente separable, por el cambio que se hizo del -1 en el tercer renglón como indica en enunciado.