**Anotaciones – Ejercicios 3 y 4 (video 2022) - Perceptrón multicapa**

***Cosas dichas en clases sobre el ejercicio 3***:

-Ya no nos alcanza un vector de pesos, porque tenemos varias neuronas. Vamos a tener matrices de pesos, donde cada matriz estaría asociada a una capa. *Las filas representan al índice de neuronas y las columnas al índice de entrada.*

-El tamaño de esas matrices puede ser distinto en cada capa, ya que dependerá de la cantidad de neuronas que tenga la capa y la cantidad de entradas a la capa. Serían las matrices WI, WII, WIII, etc.

-Teniendo la matriz de pesos y vector de entradas, al hacer el producto matriz vector nos dará la salida lineal “z” que ahora será un vector de salidas lineales de toda la capa (salida lineal de cada neurona de la capa), que luego la podemos pasar por una función de activación no lineal para obtener la salida lineal “y” (vector de salidas no lineales para toda la capa).

-Recordar que el **sesgo** va en todas las capas, como una entrada a cada una de las neuronas.

-Como función de activación no lineal usamos la sigmoidea, porque es diferenciable y derivable, mientras que la signo no. Recordar que la sigmoidea es la que va de -1 a +1, que es simétrica, porque en algunos lados aparece la sigmoide que va de 0 a +1.

-Se podría tener más de una salida al final de la red, si tengo más de dos clases para clasificar.

Por ejemplo, si tengo tres salidas, puedo decir que una sea 1 y las otras -1 (o 0).

Ahí el error va a ser un vector de errores, si tengo más de una salida.

-Recordar que el “***delta***” o “***δ***” es el ***gradiente de error local instantáneo***.

(Explicado en la teoría 02 – Perceptrón multicapa).

*Algo para recordar sobre la velocidad de aprendizaje “μ”:  
Si* ***μ es chico****, más pequeños serán los cambios en los pesos sinápticos, por lo tanto, más lento es el aprendizaje.* ***Si μ es grande****, resulta en grandes cambios en los pesos sinápticos, pero puede provocar inestabilidad (oscilaciones).* Debemos pensar de 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 muchas veces las cosas para que pueda aprender, así que hay un compromiso en esa velocidad de aprendizaje que luego vamos a analizar.

**De la clase del 2023**:

Pseudo código de retro propagación (algoritmo de entrenamiento):

Para cada época (bucle general)

Para cada patrón (etapa de entrenamiento)

-Paso hacia adelante

-Calculo el error al final de la red (si tengo una salida es un valor y si tengo más de una salida es un vector de errores)

-Paso hacia atrás

-Actualizo los pesos

Para cada patrón (etapa de validación)

-Paso hacia adelante

-Calculo el error

-Evalúo el desempeño

Evalúo si sigo en el bucle o corto (por cantidad de épocas o error aceptable).

Luego el código para el algoritmo de pruebas sería casi lo mismo que la etapa de validación.

***Recordar*** *que en el entrenamiento por cada patrón que se pasa se hace propagación hacia adelante, propagación hacia atrás y actualización de pesos, luego viene otro patrón y se repite lo mismo y así. No es que se actualizan los pesos por época, sino por cada patrón.*

**RECORDAR QUE HAY DOS ERRORES:**

Uno es el que yo mido al final de la red, que la red lo va a tratar de ir minimizando (yd - y)2, y otro error de clasificación (en la etapa de validación).

Sirve graficar para ir viendo si se va reduciendo el error interno de la red para saber si va aprendiendo, porque si solo veo el error de clasificación puede parecer estancado durante varias épocas pero que en realidad la red sí está aprendiendo y podemos ver eso a través del error interno de la red.

Y normalmente *nos interesa tener los dos tipos de error*, porque *el error interno que se usa para corregir los pesos puede cambiar mucho durante el pasaje de patrones sin que el error de clasificación cambie*, por ejemplo, supongamos que una salida debía dar +1 pero como nos dio -0.2 al aplicar la función signo la salida nos da -1, quizás en la siguiente iteración nos da -0.18 y en la siguiente -0.14, y así quizás pasan muchas iteraciones, muchos patrones, mientras se va corrigiendo hasta que nos de algo positivo como 0.1 para que pase a ser +1 cuando le aplique la función signo, o sea que puede pasar mucho tiempo en que ese -1 va a seguir siendo contado como un error en la clasificación pero que en realidad fue mejorando a lo largo de las iteraciones, entonces si yo contara solo la cantidad de errores, todos esos casos serían iguales, y sin embargo si uso el error numérico interno, por ejemplo la norma del vector error o el error cuadrático, me voy a dar cuenta que fue cambiando ese valor, que fue mejorando, fue aprendiendo.

*Ese error interno nos va a mostrar si la red está o no aprendiendo, pero el problema de ese error es que no es interpretable por el humano*, por ejemplo, me dice que tiene un error cuadrático medio de 0.8, pero ¿qué quiere decir? No me dice nada, entonces por eso usamos el *porcentaje de errores o aciertos* que sí lo podemos interpretar. Sin embargo, ese error de porcentaje de errores o aciertos tiene el problema que dijimos recién, que puede parecer estancado y en realidad la red sí está aprendiendo pero no lo refleja dicho valor.

Entonces en general, *para cada época se puede* ***reportar los dos valores***, el error cuadrático medio y el error de clasificación *(se los puede ir mostrando con una gráfica).*

Ir teniendo cuidado con el tamaño de las matrices y vectores que vamos obteniendo, para no equivocarnos.

*REVISAR* LAS DIMENSIONES QUE DEBEN TENER LOS DELTA, ENTRADAS Y PESOS, PARA SABER EXPLICAR ESO.

Por ejemplo, si hago un producto matriz vector, que las dimensiones sean adecuadas, algo como 3x2 y 2x1, para que se puedan multiplicar, y obtendría un vector de 3x1.

Abajo dejo un ejemplo que detallo más en la pág. 9.

Un detalle es que en la fórmula del error cuadrático, en el pdf multiplica al comienzo por ½, eso simplemente escala el resultado (lo divide a la mitad). Entonces cuando lo usamos para calcular el error cuadrático en la validación no nos importa, mientras que cuando se usa en la otra parte donde después se saca la derivada ahí si conviene usar el ½ porque luego se cancela con el ^2 al derivar.

**Importante**:

Algo que se hace al retro propagar, es a la matriz de pesos quitarle la columna de sesgo y transponerla, y esa es la que se va a usar en la fórmula cuando retro propagamos.

*(Abajo explico más y en la pág. 9 deje un ejemplo aplicando eso y más detallado con una imagen de las neuronas y demás).*

La estructura de una matriz de pesos de una capa será algo como:

[w10 w11 w12 w13

w20 w21 w22 w23]

Donde la primera fila son los pesos que van a la neurona 1 (primer sub índice) provenientes de las entradas 0, 1, 2, 3 (segundo sub índice). Y lo mismo para la segundo fila de la matriz que son los pesos que van hacia la neurona 2.

Entonces, cuando retro propagamos el delta, para ir calculando los deltas de cada capa a la matriz de pesos se le quita la primer columna que es del sesgo porque vamos a calcular un delta para cada neurona de la capa (calculamos todos los delta de una capa a la vez) y ese sesgo no es una neurona, y también se transpone la matriz para hacer el producto.

Sin embargo, ese peso del sesgo también se entrena, y como está incluido en las entradas (X) y salidas de las capas, también se actualiza luego cuando se obtiene el incremento de pesos al multiplicar la velocidad de aprendizaje por el delta por la entrada.

Entonces, al hacer la retro propagación transponemos esa sub matriz de pesos que nos queda y la multiplicamos por los deltas, y esa es la parte clave de la retro propagación. Y lo mismo para las capas anteriores.

Otro detalle es que si para el error usamos (yd - y)2 luego en las derivadas que aparecen en la regla de la cadera, nos queda un -1 cuando se hace la derivada parcial de “e” con respecto a “y”, ya que e = (yd - y). Y ese -1 es el que cancela el “-“ que hay en el gradiente (porque vamos en dirección opuesta al gradiente de error) al sumarlo para actualizar los pesos.

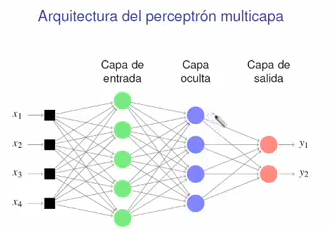
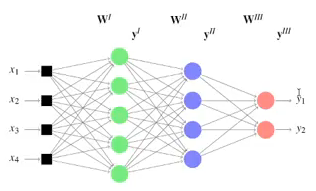
Pero un posible error es que en vez de usar (yd - y) usemos (y - yd), entonces ahí cuando hagamos la derivada parcial de “e” con respecto a “y” nos da +1 y no se cancela el “-“, entonces es como que vayamos en la dirección del gradiente en vez de en la dirección opuesta.

Si tenemos algún error recordar revisar detalles como que hayamos añadido el sesgo, luego quitado el sesgo y transponer, etc. Y como última opción probar paso por paso las cosas, poniendo a mano los pesos y en papel y lápiz ver que nos daría la salida y así encontrar donde está el error.

**EJERCICIO 3 – Introducción**

Como se ve en las imágenes de abajo, las entradas (vector **x**) alimentan a la capa de entrada o primera capa de neuronas, mediante una **matriz de pesos** **W**I, luego las salidas de la capa de entrada **y**I (vector) alimentan a la capa oculta con otra matriz de pesos **W**II, es decir, las salidas de la capa I son entradas para la capa II, y las salidas de la capa oculta alimentan a la capa de salida o última capa de neuronas mediante otra matriz de pesos **W**III. Puede haber más capas, este es un ejemplo con tres capas.

En este ejemplo, la salida es de dimensión 2 (dos salidas al final de la red y1 e y2), pero podría ser cualquier número de salidas.

En la imagen de arriba a la derecha vemos la notación que vamos a usar para las entradas, las matrices de pesos y las salidas. Los súper índices indican la capa en la que nos encontramos, por ejemplo, en la salida yIII el “III” es porque sería la salida de la capa 3, que sería un vector con la salida de toda la capa, pero también en este caso la podemos llamar directamente “y”, porque es la salida final de la red.

***¿Cómo funciona el perceptrón multicapa?***

Al igual que en el perceptrón simple, tenemos dos formas de funcionamiento, una durante el entrenamiento y otra durante la prueba.

Durante el entrenamiento se hacen dos cosas: una es el paso o propagación hacia adelante, que consiste en calcular la salida de cada capa, progresivamente hasta llegar a la última capa y ver la salida final de la red. Y luego sigue la propagación hacia atrás, la cual consiste en ir calculando todos los valores que voy a necesitar para poder actualizar los pesos o parámetros de la red.

Lo que hacemos es *medir el desempeño con alguna función de costo* al final de la red (utilizando la salida deseada y la salida obtenida) y hacer una optimización, donde se van moviendo los pesos para ir mejorando ese error o medida de desempeño.

***Propagación hacia adelante***:

Simplemente se va haciendo la sumatoria de entradas por pesos y calculando la salida lineal “v” de cada neurona de cada capa y se las pasa por la función de activación no lineal ф para obtener la salida no lineal de cada neurona (usamos la función sigmoidea). Luego esas salidas “**y**” de una capa pasan a ser entradas de la capa siguiente y se realiza lo mismo hasta llegar a la salida de la última capa o salida final.

*Una forma de verlo* es hacer un *producto interno*, para una capa hacemos un producto interno entre un vector de pesos (que une a una neurona con cada entrada) y el vector de entradas. Como se ve en la imagen de abajo, para la capa I hace el producto interno del vector de pesos de la neurona “j” de la capa I (**w**jI), con el vector de entradas “**x**”, y así al ir variando “j” obtiene la salida lineal “vj” para cada neurona “j”. Luego para la capa II sería el producto interno del vector de pesos “j” de la capa II, con el vector de entradas que ahora será “yI” (las salidas de la capa anterior), y así con las demás.

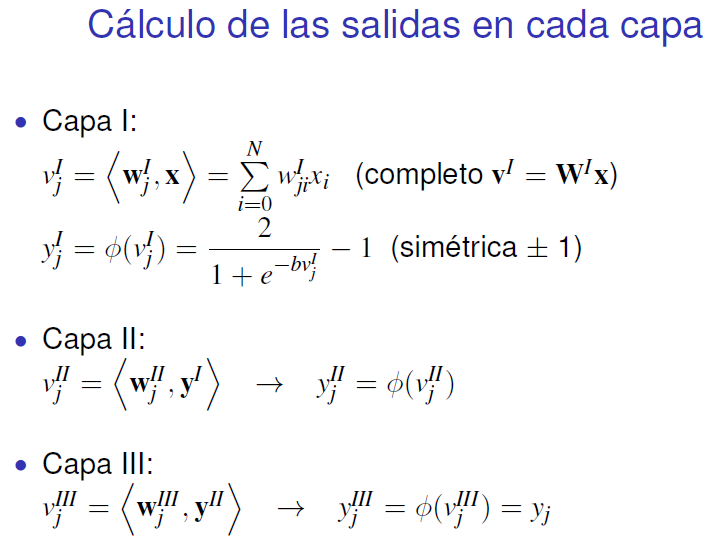
***Otra opción mejor***, como se muestra al final de la primera fórmula de la imagen, *si a los pesos de una capa los voy poniendo como filas de una matriz de pesos,* puedo hacer **v**I = **WIx**, que sería un *producto matriz por vector*, *entre la matriz de pesos* ***W*** *para esa capa y el vector de entradas “****x****” de dicha capa*, lo cual nos daría un vector con la *salida lineal “****v****” de todas las neuronas de esa capa*.

Lo bueno de ir guardando los pesos en matrices “W” es que podemos calcular la salida lineal de toda la capa (vector) haciendo una sola operación de producto matriz vector: **v**I = **W**I**x**

Luego esa salida lineal se pasa por la función de activación no lineal ф y se obtiene la salida lineal **y**I de la capa, que pasa a ser la entrada a la capa siguiente.

**Recordar** que una de las entradas que vamos a tener en todas las neuronas de todas las capas será la correspondiente al **sesgo** o bias.

De esa manera vamos calculando y propagando las salidas hacia adelante.



*Esa propagación hacia adelante la vamos a hacer en dos instancias*, primero durante el entrenamiento, donde se propaga hacia adelante, se propaga hacia atrás y se ajustan los pesos. Y después, una vez que tengamos entrenada la red, durante el uso cuando tengo un nuevo patrón que quiero clasificar se lo doy a la red y hago el paso hacia adelante.

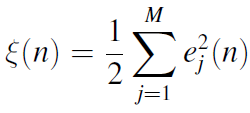
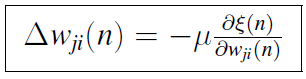
Lo mismo para la prueba y para la validación, si quiero probar el desempeño le voy dando patrones de test, hago la propagación hacia adelante, veo que salida me da y la comparo con la salida deseada para obtener estadísticas del desempeño. Similar a lo que hacíamos en el perceptrón simple.

***Propagación hacia atrás***:

*Al final tendremos la función de costo ξ(n) (o error) que va a medir el desempeño*, y lo que vamos a hacer es actualizar los pesos “w” como indica la imagen de abajo, donde el incremento o variación de pesos Δwji(n) que luego le vamos a sumar al peso actual, va a ser igual a menos la tasa de aprendizaje “μ” por la derivada de esa función de error con respecto a los pesos. Esa derivada sería el gradiente de error, y el signo negativo del comienzo indica que vamos a hacer una actualización de los pesos en el sentido opuesto al gradiente de error, es decir, en el sentido opuesto al que crece el error.

Entonces *aplicando la regla de la cadena* vamos haciendo todos los pasos que explicamos en la teoría y que están en el pdf, donde se hacen simplificaciones usando los gradientes “δ”.

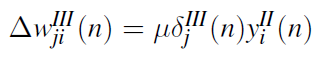
Y terminamos obteniendo la fórmula de la derecha que es la fórmula para obtener el incremento de pesos para la actualización: velocidad de aprendizaje por el delta por la entrada.

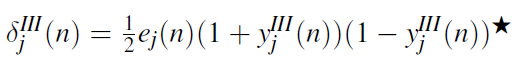
Ahí en Wji, el índice “j” seria la neurona “j” de esa capa, por eso es el gradiente δj y el índice “i” es para indicar que yi es la entrada a esa capa. Abajo se entiende más indicando las capas.

Así sería para ir calculando por neurona, después vemos para calcular de una vez para toda la capa.

Por ejemplo, en la fórmula de abajo tenemos que la variación de pesos de la capa 3 es igual a la tasa de aprendizaje por el delta de la capa 3 (δjIII) por la salida de la capa 2 (yIIi), donde esa salida de la capa 2 ahora será la entrada de la capa 3, por eso también podemos decir: *tasa de aprendizaje por el delta de la capa por la entrada a la capa*.



Otro detalle es que la fórmula de abajo para el delta de la capa 3, que en este caso sería la capa final o capa de salida, tiene una definición específica porque es la *capa final, entonces ahí aparece el error “e*” (yd - y).



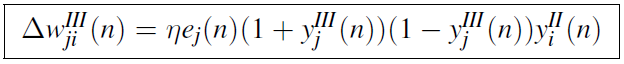
*Solo* *podemos* ***medir el error*** *para la capa de salida*, porque la salida deseada “yd” que conocemos es a nivel de salida de la red, es decir, no tenemos una salida deseada para la capa I o para la capa oculta por ejemplo, entonces *para las otras capas no tenemos un error específico para usar ahí.*

*Entonces,* ***¿Qué hace el algoritmo de retro propagación?*** *Podemos pensarlo como que de alguna manera trata de atribuirle a las capas anteriores, que tanto del error que obtuvimos a la salida de la red se debe a los pesos de esas capas anteriores, es decir, que tan responsables son de ese error que obtuvimos al final de la red*.

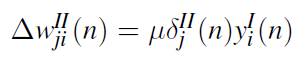
Esa sería la idea de la retro propagación: a la salida de la red tengo el error, y con ese error puedo calcular el delta “δ” explícito de la salida (como la fórmula de arriba), y ahora para calcular el delta de la capa anterior tengo que decirle de alguna manera a las neuronas de esa capa *qué tanto son responsables del error que obtuve al final, y la forma de asignarles esa responsabilidad es de acuerdo a qué tanto contribuyeron en ese error, y contribuyeron más si los pesos que unen a esas neuronas con la salida eran más grandes.*

Así sería dicho en términos “literales”, después lo que hacemos es ir calculando con la regla de la cadena.

Entonces, antes dijimos que la actualización de pesos era la *tasa de aprendizaje por el delta de la capa por la entrada a la capa* (o salida de la capa anterior), así que en la imagen de abajo podemos ver cómo sería la actualización ahora usando ese delta de la capa 3 (capa final) que calculamos arriba, que en la fórmula de abajo sería el error “ej” y los dos términos que le siguen (que son la derivada de la sigmoidea en esa capa), y al final tenemos la entrada a la capa 3 (salida de la capa 2 yiII).



-Para una capa anterior a la capa de salida (capa 2 en este caso), vamos haciendo la propagación hacia atrás, entonces ahora queremos calcular el delta de la capa 2 δII, y la fórmula para la actualización de los pesos será similar: tasa de aprendizaje por el delta por la entrada a esa capa.

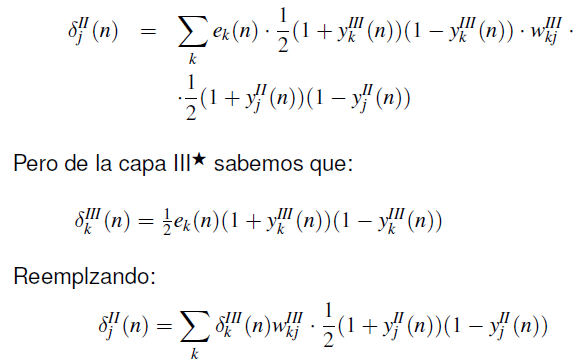


Ahora, como explicamos antes, no puedo tener la medida del error para la capa 2, así que ¿cómo se calcula el delta para esa capa 2 (δII)?

Haciendo las cuentas con la regla de la cadena se van obteniendo las ecuaciones de la página 58 a 64 del pdf, pero lo que importa es la pág. 64 donde nos termina quedando la ecuación de la imagen de abajo, donde podemos identificar que el término que tenemos en el primer renglón dentro de la sumatoria es el delta de la capa 3 (capa siguiente a la que estamos), y esa sumatoria sería una multiplicación con los pesos que unen la capa 3 con la capa 2, esa es la idea que hablamos antes.

El subíndice “k” ahí va a representar a las neuronas de la capa final, mientras que el “j” es para las neuronas de la capa actual.

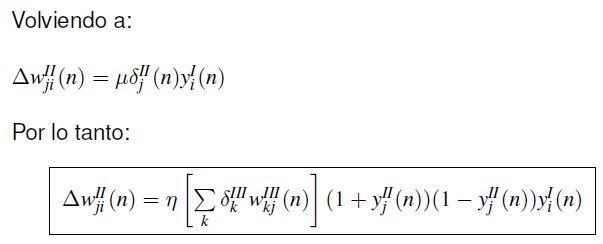
Reemplazando nos termina quedando esa ecuación del final, donde *la parte de la sumatoria con* ***δkIII(n)\*wkjIII*** *es lo que hace que se retro propaguen o que se imputen los errores a esa neurona* a través de los pesos que la unían con la capa siguiente, y al final está la derivada de la sigmoidea en la capa II.



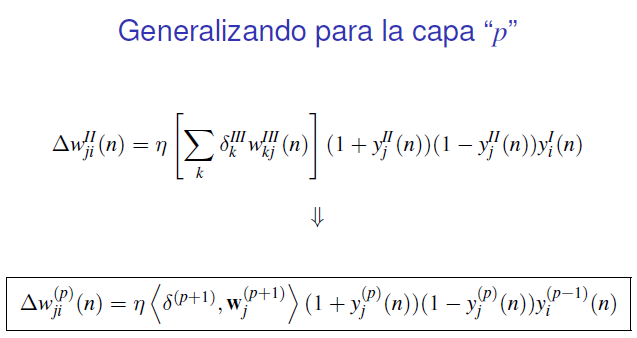
Si comparamos esa fórmula para el delta 2 δjII con la fórmula del δjIII, podemos ver que en la del δjIII teníamos el error por la *derivada de la función no lineal (de la función de activación sigmoidea)*, y ahora si vemos la del δjII, tenemos algo (el término δkIII(n)\*wkjIII) por la derivada (el término 1/2 (1+yjII)(1-yjII)).

Ese “algo” es la ***propagación del error hacia atrás***, por eso decimos que *ese término* ***δkIII(n)\*wkjIII*** *es lo que hace que se retro propague o que se impute el error a esa capa, es decir, pasamos el delta de la capa siguiente hacia atrás a través de los pesos.*

Una vez que tengo el delta de la capa 2, puedo calcular el incremento de los pesos como dijimos.



En ese caso teníamos solo una capa intermedia (la capa 2), pero se puede generalizar si tuviéramos más capas intermedias. Haciéndolo como en la imagen de abajo *de* ***forma general*** *para la capa “p”*, *el cálculo del incremento de los pesos para la capa “p” (wjip) sería como explicamos antes “μ” (usa “η” porque en esa constante ya incluye el ½ que aparecía) por la retro propagación con los pesos de la capa siguiente y deltas siguientes (p+1) hacia atrás (el producto interno <>, que sería la sumatoria que vimos antes), por la derivada de la función no lineal (lo que está entre paréntesis de (1+y)(1-y), que es la derivada de la sigmoidea en la capa “p”) y por la entrada a la capa “p” (salida de la capa anterior (p-1)).*



***Resumen del algoritmo***:

1. Inicialización aleatoria de los pesos, con valores chicos, por ejemplo entre -0.5 y 0.5 (no olvidar el peso para el sesgo), como hicimos en el perceptrón simple pero ahora para todas las matrices de pesos. Como dijimos, podemos ubicar los pesos de una capa como filas de una matriz (para después hacer **v** = **Wx**).
2. Propagación hacia adelante: puede ser con un *producto matricial* **v** = **Wx** para que quede más eficiente y obtener de una vez todas las salidas lineales “**v**” de una capa, así calculamos para cada capa y las pasamos por la función no lineal (sigmoidea), y obtenemos la salida no lineal “**y**” de toda la capa, la cual pasa a ser la entrada para la siguiente capa, propagamos con la matriz de pesos siguiente, obtenemos otra salida lineal, le aplicamos la no linealidad y así vamos avanzando por las capas hasta llegar a la salida final.
3. Propagación hacia atrás: calculamos el delta δ de la capa de salida usando el error “e” que obtuvimos al final de la red (yd - y) y aplicando la fórmula que vimos antes obtenemos el vector de deltas en la capa de salida (delta para cada neurona de la capa de salida). Luego usamos la fórmula para propagar hacia atrás el error e ir obteniendo los deltas en las capas anteriores.
4. Ajuste de los pesos: una vez que tengo todos los deltas calculados, actualizo los pesos calculando el incremento de pesos Δw necesario usando la fórmula que vimos de la tasa de aprendizaje por el delta por la entrada y sumando eso al peso actual.
5. Iteración: vuelve a 2 hasta convergencia o que se cumpla un criterio de finalización (que se llegue a un número máximo de épocas o una tasa de error aceptable por ejemplo).

En las páginas 69 en adelante del pdf se ve el ejemplo gráfico de todos los pasos.

-Cuando usamos por ejemplo wI10 el subíndice 10 quiere decir que ese peso va a la neurona “1” desde la entrada “0”, y el superíndice “I” significa que estamos en la capa I.

Agregar el sesgo significa agregar una columna más en la matriz de pesos, y también recordar agregar una entrada más de valor -1 para el X0. Recordar nuevamente que el sesgo va para todas las neuronas de todas las capas, entonces también lo debemos ir agregando en la primera columna de cada vector de salidas “**y**” porque son los vectores que se usan como entrada para la capa siguiente.

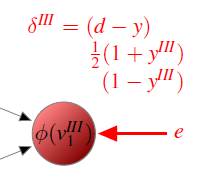
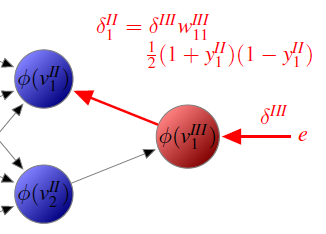
-Cuando se llega a la propagación hacia atrás, esa fórmula de abajo a la izquierda es importante, vemos que el delta en la capa final δIII es una medida del error (d - y) que sería el “e” que veíamos, por la derivada de la función no lineal en esa capa de salida.

Luego la medida del error cometido en la capa anterior (imagen de la derecha), va a tener que ver con propagar hacia atrás el error obtenido al final (propagar ese δIII) a través de los pesos, por eso después en la propagación hacia atrás tenemos cosas como en la imagen de la derecha, donde para calcular el δ1II (delta para la neurona 1 de la capa 2), usamos el δIII obtenido antes y lo multiplicado por el peso correspondiente (flecha marcada en rojo) y por la derivada de la función lineal como siempre.

Lo mismo se haría para el delta de la neurona de abajo (δ2II) usando el otro peso (w12III).

Ese multiplicar el delta por los pesos dijimos que era lo que nos permitía hacer la retro propagación del error.

Esa fecha marcada en rojo representa wII11 donde según lo que dijimos antes de la notación, el primer “1” dice que va a la neurona 1 (de la capa final) desde la entrada 1 (que sería la salida 1 de la capa anterior).

Pero luego, para el delta de la neurona verde que se marca en la imagen de abajo, vemos que esa neurona contribuyó al error con dos pesos, que son las dos conexiones marcadas en rojo hacia la capa siguiente, entonces el delta correspondiente a la misma (δ1I) escrito en rojo en la imagen, será el delta de las dos neuronas siguientes con las que tiene conexión δ1II y δ2II multiplicadas por los pesos correspondientes (w11II y w21II), y eso multiplicado por la derivada de la función no lineal como siempre.

Y lo mismo para las demás, obteniendo todos los deltas en esa propagación hacia atrás.

**IMPORTANTE**: otra cosa hablando de **vectorización** es que en esa ecuación roja, la sumatoria wII11 δII1 y wII21 δII2, se puede ver como un producto interno, donde tenemos un vector “**w**” y un vector **δ**, pero si lo analizamos en detalle, eso se puede expresar como un producto matriz por vector para hacerlo más eficiente.

La matriz sería la matriz de pesos de la capa, que iría ahí sacando la primera columna correspondiente al sesgo y **transponiendo** dicha matriz, esa sería la matriz que se debería multiplicar por el vector de los deltas, para que me dé la propagación en esa parte, que multiplicada por las salidas me va a dar el delta de esa capa.

Sería otra forma de hacerlo si lo queremos vectorizado.

Por **ejemplo**, al comienzo del Word deje este caso donde la estructura de una matriz de pesos **W** de una capa que coincide con el caso de la capa II de la imagen de abajo, será algo como:

[w10 w11 w12 w13

w20 w21 w22 w23]

Donde la primera fila son los pesos que van a la neurona 1 (primer sub índice) provenientes de las entradas 0, 1, 2, 3 (segundo sub índice). Y lo mismo para la segundo fila de la matriz que son los pesos que van hacia la neurona 2.

Y el vector de deltas para la capa II será: **δ**II = [δ1 δ2], entonces para hacer el producto matriz vector entre **W** y **δ** necesito quitar la columna correspondiente al sesgo (la primera) y transponer la matriz **W**, quedando:

[W11 W21

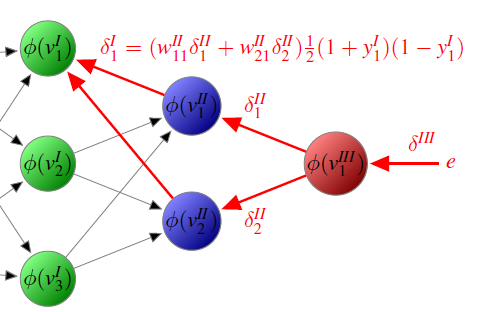
W12 W22

W13 W23]

Y ahora sí puedo hacer el producto matriz vector, ya que ahora tengo 3x2 y 2x1, obteniendo como resultado un vector de 3x1 que luego se multiplica por la derivada de la sigmoidea y obtenemos el vector de deltas para toda esa capa, siendo un vector 3x1 porque son tres neuronas.

Después como se menciona en la página siguiente, cuando vayamos a calcular el incremento de pesos, podemos hacer un producto entre vectores, entre el vector de deltas de la capa que será un vector columna y el vector de entradas de la capa que será un vector fila, y obtenemos una matriz de incremento de pesos:

Vector col. 3x1 (deltas capa I), producto con vector fila 1x4 (entradas capa I con el sesgo incluido), nos da la matriz 3x4 de incremento de pesos.



***Recordar*** *bien ese ejemplo anterior por si necesito explicar y hacer a manos esos productos, para saber los tamaños de los vectores, matrices, cuándo se transpone, cuándo se quita el sesgo y se agrega, etc.*

Una vez que tenemos todos los deltas calculados (de todas las capas), pasamos al **ajuste de pesos**, *porque un error común es pensar en calcular los incrementos de pesos de una determinada capa y ya ajustar los pesos correspondientes, pero eso está mal*, *porque para hacer la propagación hacia atrás completa e ir imputándole el error correspondiente a cada neurona*, *tengo que ir usando los pesos que hicieron que se cometa ese error en la salida final de la red*, así que si los voy actualizando voy a estar propagando el error con otros pesos que no fueron los que lo generaron.

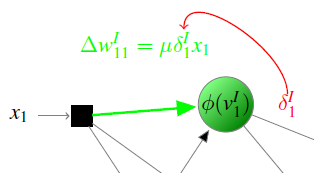
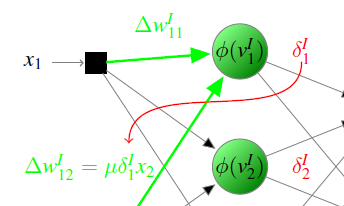
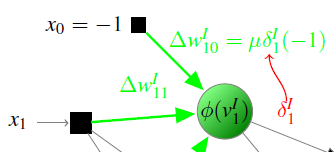
Así que *siempre primero calculamos todos los deltas y después actualizamos los pesos en cualquier orden*.

En la imagen de abajo a la izquierda vemos que por ejemplo para luego actualizar ese peso en verde, en el cálculo del incremento usamos el delta que calculamos antes y la entrada asociada a ese peso (x1). Lo que hacemos es obtener ese Δw y luego cuando vayamos a ajustar los pesos se lo tenemos que sumar al “w” actual, es decir, ese es el incremento que le debemos sumar al peso actual para ajustarlo.

Otra cosa interesante es que si los delta de la misma capa los tengo en un vector, y tengo el vector de entradas para esa capa en otro vector, podría hacer un *producto de un vector fila por un vector columna* y me va a dar una **matriz de incrementos de pesos** (o lo que le tengo que sumar a la matriz de pesos) y lo puedo trabajar **vectorizado** en vez de hacerlo elemento por elemento.

Y recordar no olvidarse del sesgo, porque ese peso también se aprende (si ya viene incluido en el vector de entradas ya se estaría).

*En la página anterior también expliqué ese producto entre vectores en el* ***ejemplo****, para que me quede claro.*

**Resumiendo**:

Hacemos el paso hacia adelante, una vez que tenemos la salida final comparamos con la salida deseada, eso nos da el *error* que nos sirve para calcular el delta en la salida, y luego comienzo a retro propagar para ir calculando los deltas anteriores como fuimos explicando. Una vez calculados los deltas de todas las capas, puedo actualizar los pesos capa por capa, sin importar el orden.

*Cuando hago el paso hacia adelante, no solo guardo la salida final en la última capa, sino que tengo que ir guardando las salidas de todas las capas, porque después en el paso hacia atrás las necesitamos para ir calculando los deltas.* Para eso podemos tener *un vector donde vayamos guardando todas las salidas.*

Y también es importante nunca olvidarnos del sesgo o bias, todas las capas lo tienen, así que esa entrada “X0” va a ir conectada a todas las neuronas de la red.

**Importante:**

En la clase hablamos (2022, usando Octave, en Python lo hacemos directo con Numpy) de usar “**Cell Arrays**”, que es un tipo de arreglo en Octave y Matlab que te permite añadir dentro cualquier cosa (de igual o distinto tipo). Por ejemplo, tener un vector de matrices, o un vector con datos de distinto tipo.

Para estos cell arrays se usa la función “cell” si los queremos crear dentro de un script, y sino desde la ventana de comandos para probar los podemos crear de la siguiente manera: c = { [1, 2], [3, 4], [5, 6] }.

Y para acceder a un elemento hacemos por ejemplo c{1} y nos devuelve [1, 2], o si hacemos c{1}(2) nos devolvería 2, ya que accede al elemento 1 del cell array (que es [1,2]) y busca la posición 2. Eso lo usamos por ejemplo para acceder a una capa y una neurona de dicha capa, haciendo cosas como c{i}(j).

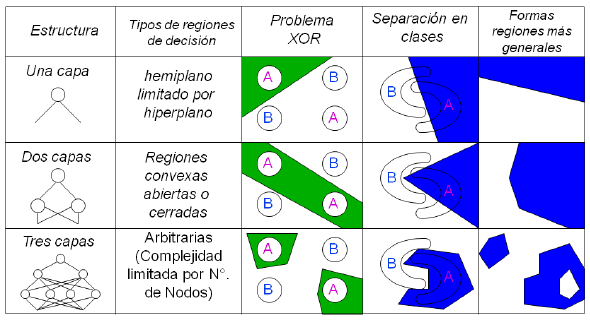
Es importante porque así podemos crear *estructuras para las matrices de pesos, para las salidas y para los deltas*: {WI, WII, WIII} {yI, yII, yIII} {δI, δII, δIII}

**Ejercicio 3 - Items**

**Item 1:** En el gráfico abajo del ejercicio 3 en el pdf, que muestra los datos de “concentlite.csv” se observan las dos clases (negros y rojos), debemos analizar que arquitectura de red debemos usar para que pueda resolver bien este problema. Para eso nos puede servir la imagen de abajo sobre las regiones de decisión, donde se muestran que tipo de regiones pueden generar distintas estructuras de redes.

Entonces el objetivo sería que, con una arquitectura lo más simple posible, logremos tener un buen desempeño en la clasificación de esos elementos, y la idea es que hagamos un gráfico parecido al que se muestra en el pdf del ejercicio, donde mostremos con un color una clase y con otro la otra clase.

Algo que se podría hacer es *usar más de dos colores en el* ***gráfico***, porque si hacemos por ejemplo una nube roja para los elementos del medio, puede que haya elementos erróneos alrededor y que no los veamos, en cambio sí usamos un código de colores como: azul para los clasificados como clase 1, rojo para los clasificados como clase 2, y después verde los clase 1 que se clasificaron como clase 2 y amarillos los clase 2 que se clasificaron como clase 1, así podemos distinguir los que están bien y los que fueron mal clasificados.



Como el enunciado pide que se pueda ***elegir libremente el número de capas de red y neuronas en cada capa***, una forma de hacer eso es *ingresar la arquitectura como un vector*. Por ejemplo, un vector [4, 2, 1] indicaría una red de 3 capas, donde la primera tiene 4 neuronas, la segunda 2 neuronas y la última 1 neurona.

También se debe poner la tasa de aprendizaje y criterio de parada igual que en el perceptrón simple. Acá también usamos el criterio de parada con el número máximo de épocas y lo combinamos con algún criterio de error máximo.

Para elegir la ***cantidad de capas y de neuronas en cada capa***, en el ***demo grafico online*** de redes neuronales (link más abajo) hay un ejemplo similar con puntos distribuidos de la misma manera que en este ejercicio (dos entradas y dos clases para clasificar), entonces fuimos probando que casos resolvían ese demo de forma gráfica, cambiando la cantidad de capas, neuronas y tasa de aprendizaje.

*Todo lo que explico abajo hasta donde dejo el link es todo usando el* ***demo online****. Tener en cuenta que no sabemos la cantidad de patrones que usan en ese demo, así que la cantidad de épocas puede ser muy distinto a la nuestra, y tampoco sabemos que tan distintos sean esos datos con los de nuestro archivo.*

*Opciones de estructuras:*

-Una opción de estructura que lo resuelve en el demo es utilizando dos capas, la capa de entrada con cuatro neuronas y la capa de salida con una o dos neuronas (dos salidas), es decir: estructura = [4 1] o [4 2].

-Otra opción que lo resuelve es con 5 neuronas en la capa de entrada y 1 neurona en la capa de salida.

-Después seguí investigando y encontré que con una sola capa de 3 neuronas también lo resuelve, y usando gamma = 0.1 hasta lo hace mejor que las otras, después con gamma = 0.01 en adelante le cuesta un poco más que las otras pero bien igual.

-Y con una capa de 4 neuronas sería aún mejor, porque con lo que explica DOC CSV en <https://www.youtube.com/watch?v=uwbHOpp9xkc&list=PL-Ogd76BhmcB9OjPucsnc2-piEE96jJDQ&index=2> a partir del minuto 8:25, lo que hacemos *es unir esas neuronas y logramos* ***combinar cuatro sigmoideas*** *dando una forma que tiene como un bulto en el centro y eso nos permite separar los dos grupos de datos al hacerse la intersección con el plano que se genera*.

*(Voy a anotar mejor todo eso del vídeo al final de este ítem 1, después de los links del demo y el tutorial).*

Y en las opciones de arriba del demo, en “Activation” se puede elegir la función de activación no lineal que queremos usar: la relu, la tangente hiperbólica, la lineal o la sigmoidea, donde esa última es la que queremos.

Usando arquitectura = [5 1], el mejor caso es con tasa de aprendizaje (learning rate) de 0.1 (recordar seleccionar la sigmoidea, porque las otras cambian), de ahí para abajo (0.01 o 0.03) ya le empieza a costar más porque seguro cae en algunos mínimos locales y le cuesta más para salir (consume más de 1000 épocas en esos casos). Con 0.1 anda más rápido porque le permite saltarse esos mínimos locales. Y después con 0.3 le cuesta mucho más.

Y el número de épocas hay que ir probando, porque en el demo (arriba dice “Epoch”) lo resuelve con un poco menos de 500 parece (usando gamma = 0.1), pero son distintos datos y un funcionamiento distinto a nuestro algoritmo.

Con arquitectura = [5 2] y por ejemplo gamma = 0.03 lo tiene que resolver más rápido, porque ahora tendría dos neuronas al final y puede clasificar las dos categorías más rápido.

Usando la arquitectura = [4 2], con gamma = 0.1 mejora bastante, lo resuelve más o menos en la mitad de épocas que con [5 1]. Y con gamma = 0.03 le cuesta más pero también es mejor que con la arquitectura [5 1].

Usando la arquitectura = [3], es decir, una capa de 3 neuronas, con gamma = 0.1 funciona mejor que las anteriores, lo resuelve en menor cantidad de épocas. Luego con gamma = 0.01 o más chicos ya le cuesta más.

**Importante**:

Cuando usamos una sola neurona en la capa final, podemos identificar una categoría y lo que queda fuera sería la otra, por ejemplo, el paciente está enfermo o no, es spam o no, etc.

Mientras que si usamos más salidas en la capa final podemos clasificar o identificar más categorías, por ejemplo, codificando en binario o usando winner takes all para identificar cuál es la salida que se activa. Entonces si tengo un caso con tres categorías para clasificar (como el ejercicio de Iris), yo puedo poner tres neuronas en la capa de salida y listo, una para cada salida.

En el demo, a la derecha en el “output” dice “test loss” y “training loss”, que sería perdida en el test y en el entrenamiento, entiendo que ese sería el error y va bajando al pasar las épocas, como se ve de forma gráfica en el dibujo.

Y a la izquierda tiene para cambiar la cantidad de datos para entrenamiento en comparación a datos de prueba (lo puse en 80%).

*Link del* ***demo****:* [https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.1&regularizationRate=0&noise=0&networkShape=4,2&seed=0.61796&showTestData=false&discretize=false&percTrainData=80&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false](https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.1&regularizationRate=0&noise=0&networkShape=4,2&seed=0.61796&showTestData=false&discretize=false&percTrainData=80&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cos)

*Tutorial de cómo usar el demo online y con muchas explicaciones importante*: <https://www.youtube.com/watch?v=FVozZVUNOOA>

*Anotaciones del vídeo del tutoriaL (****IMPORTANTE****)*:

(Minuto 12 a 16:30)

Se observa que el problema no se podía resolver con un perceptrón simple nada más, porque se hace una división con un plano que no divide correctamente las dos clases. *¿Qué debemos hacer? Añadir más neuronas y que dichas neuronas tengan una función de activación que vaya añadiendo no linealidades o distorsiones al plano que nos permitan encontrar una superficie con la cual efectivamente se puedan separar los puntos* (como si fuéramos doblando una dobla para darle la forma que queremos).

Como expliqué en la parte de debajo de anotaciones del otro vídeo, la figura que debemos formar tiene como una montaña en el centro, entonces la intersección de esa figura con el plano hace que se separen los puntos del centro con los de fuera.

Para conseguir una sola figura se fue probando ir añadiendo distintas capas y neuronas.

En primer lugar se añadió una capa oculta con una neurona, y al ejecutar vemos que no soluciona el problema, porque entre esa neurona y la salida no hacemos ninguna otra manipulación, así que es como trabajar con una sola capa. Entonces si añadimos dos neuronas, ahora cada neurona está añadiendo una no linealidad con su función de activación (nosotros usamos la sigmoidea como función de activación para ir generando las distorsiones al plano), y si ejecutamos vemos que comenzamos a obtener una región de separación más compleja, que separa un poco mejor las dos categorías pero que aún no lo resuelve.

Como con dos neuronas en la capa oculta no podemos generar un polígono cerrado, porque sería como generar un polígono con dos lados y no es posible, lo siguiente que probamos es añadir una tercera neurona a esa capa oculta, y al ejecutar vemos que ahora sí se genera una estructura que puede ser interceptada por un plano y generar nuestra frontera de separación.

Esa sería una solución al problema, usando [3, 1], luego si añadimos más neuronas o más capas (como expliqué arriba con estructuras [5 1], [4 1]) obtenemos otras soluciones del problema.

También después muestra que si añadimos las entradas x12 y x22, con una sola neurona en la primera capa y una neurona de salida la red llega a la solución del problema de forma perfecta, con 0 error. Lo mostró el profe también pero eso ya no lo vemos.

Es así porque sería resolver la ecuación de un círculo, y justo las categorías están en forma de círculo.

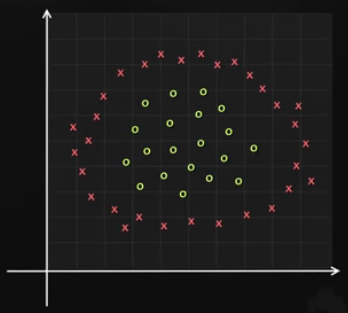
Entonces si tenemos un problema con algo circular, el hecho de ponerle las entradas al cuadrado algo lo tiene que ayudar.

Al usar la sigmoidea, como es una función suavizada, veremos que el contorno que se forma para separar los puntos es más suavizado que si usamos por ejemplo la Relu, que tenía como un doblez recto.

Otro **detalle importante** que tal vez se puede preguntar (y se puede probar en el demo), es que si añadimos muchas capas ocultas cada una con muchas neuronas y usamos como función de activación la lineal, eso es equivalente a usar una única neurona (a dejar solo las dos entradas en ese demo, que representa tener una neurona), porque la combinación lineal de todas las combinaciones lineales que se producen en cada neurona es equivalente a una sola neurona, así que el resultado simplemente es generar una línea que no nos permite separar las categorías.

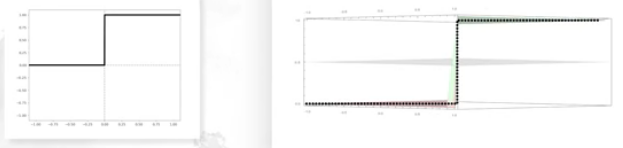
*Anotaciones del vídeo* <https://www.youtube.com/watch?v=uwbHOpp9xkc&list=PL-Ogd76BhmcB9OjPucsnc2-piEE96jJDQ&index=2> :

La imagen de abajo es el problema que se muestra en ese vídeo, similar al problema del ejercicio 3 (era de la guía 1 en 2022, pero es el ejercicio 1 guía 2 del 2023), donde como ejemplo dijo que un problema similar en la vida real puede ser el de separar que células son cancerígenas y cuáles no.

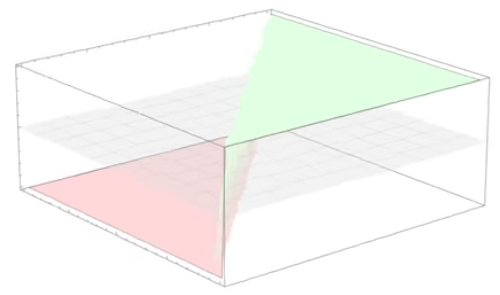


*¿Cómo hacemos para separar esas dos clases?*

Cuando veíamos el problema del AND, OR o XOR, usábamos la función de activación (función no lineal) escalonada (la signo en la guía 1), la cual si vemos geométricamente es la figura de abajo a la derecha, donde lo gris del medio sería el plano normal generado por la neurona y el efecto de esa función de activación es distorsionar el plano generado por la neurona, dando como resultado esa parte rosada y verde.

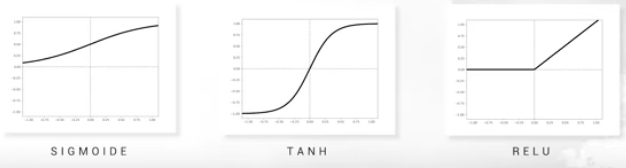


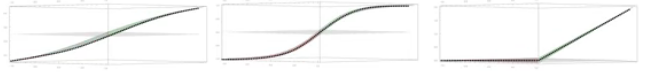
Así toda la geometría de ese plano distorsionado con la función escalonada (el plano gris de la imagen de abajo es el que distorcionamos con la función de activación no lineal) que este por debajo del plano pertenecerá a un grupo (grupo rojo) y lo que quede por encima del plano distorsionado pertenece al otro grupo (lo verde).



Ahora, ¿cómo sería esa figura si usamos otro tipo de función, como la sigmoidea, la tangente hiperbólica o la relu? En la imagen de abajo se muestran cada una.

(Usa la sigmoide que va de 0 a 1, nosotros la sigmoidea que va de -1 a 1).



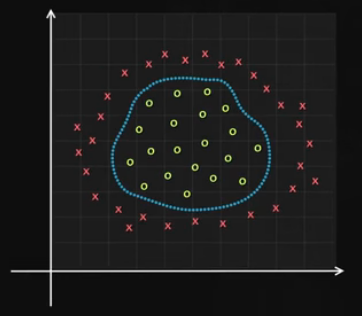
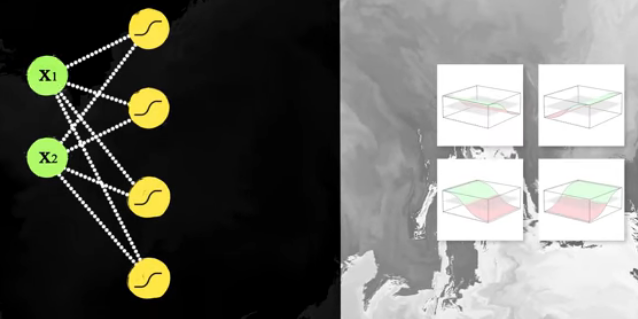
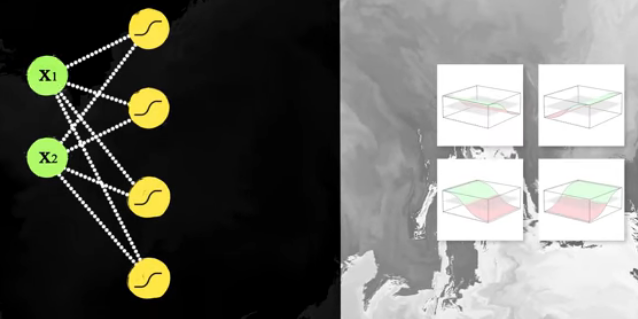


**IMPORTANTE**: (esto nos puede servir para explicar en la evaluación y entender mejor)

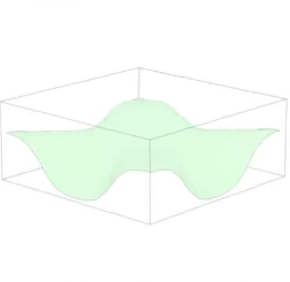
De momento, en los tres casos nuestra frontera sigue siendo una línea recta, debido a la intersección de la figura geométrica (lo verde y rojo) con el plano (gris).

Entonces, *¿cómo podemos conseguir una frontera curva como en la imagen de abajo (celeste) para solucionar el problema?* Esto lo logramos aprovechando que gracias a la función de activación ahora podemos encadenar varias neuronas.

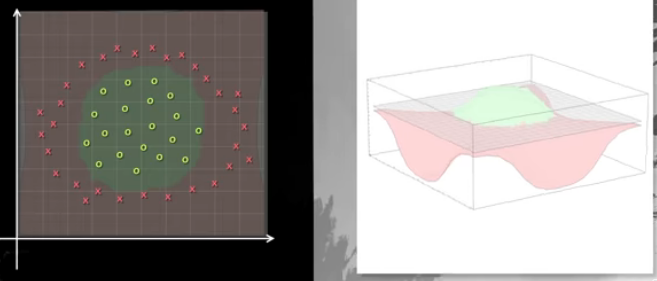
*Una posible solución* al problema sería la siguiente: en la primera capa colocamos una neurona con la función sigmoidea, donde la forma de esa función sigmoidea la vimos arriba pero en realidad la podemos ir adaptando según como ajustamos los parámetros de la red (los pesos), pudiendo ir cambiando la orientación de la figura por ejemplo, y si en vez de una neurona en esa capa ahora colocamos cuatro neuronas y cada una con su función de activación sigmoidea, *cada una con una orientación diferente*, obtendríamos algo como en la imagen de abajo a la derecha (lo podemos explicar doblando una hoja de papel para darle la forma de una sigmoidea, como hace Dot CSV en uno de los videos explicando).

Ahora, si añadimos una nueva capa con una neurona (la roja en la capa siguiente) podemos construir una combinación de esas cuatro figuras geométricas de arriba a la derecha generadas por las cuatro neuronas de la primera capa, obteniendo una figura como la que se muestra en la imagen de abajo, donde vemos una figura que tiene un bulto en el medio, como si fuera la cima de una montaña.

Esa figura es la **solución del problema**, porque como se puede ver en la imagen de abajo, la intersección del plano gris con la figura produce la frontera circular que estábamos buscando. Así nuestro problema de clasificación está resuelto gracias a la unión de varias neuronas.

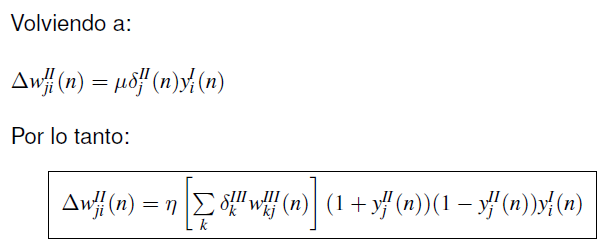


**Ejercicio 3 – Vídeo parte 2**

En la primera parte vimos el ítem 1 del ejercicio. Ahora veremos lo demás.

Con la fórmula de la imagen de abajo decíamos que la actualización de los pesos en una capa era la tasa de aprendizaje por el delta en esa capa por la entrada a esa capa.

Ahí está expresado para el peso “ji”, pero dijimos que lo podemos hacer para toda la matriz de pesos si el delta δ fuera el vector de deltas para la capa II en ese caso, y la entrada a la capa II o salida de la capa I (yIi) estaría puesta como un vector transpuesto (yIi) T, así *vector fila por vector columna nos daría una matriz*.



Entonces, ¿qué es esto del **término de momento**?

Va relacionado al momento físico, por ejemplo: pensemos que vamos corriendo en una bajada, si queremos frenar no lo podemos hacer de golpe, porque el cuerpo intenta seguir moviéndose en la dirección en la que íbamos. O si venimos corriendo y queremos doblar, no podemos doblar de golpe y cambiar de dirección bruscamente, sino que debemos frenar un poco. Eso se debe al momento físico.

Esa es la idea que tratamos de seguir, porque cuando hacemos estas actualizaciones usando el gradiente de error instantáneo, o sea para cada patrón actualizamos los pesos, tiende a pasar que vamos a ir cambiando de dirección muchas veces, porque una vez tengo que corregir un peso hacia un lado, el peso siguiente hacia otro lado y así, y eso se demostró que puede no ser muy bueno. (En un vídeo de DOT CSV muestra eso cuando habla del descenso del gradiente creo).

Una forma de mejorarlo sería achicar la tasa de aprendizaje, para que los pasitos que dé sean más chicos, pero *eso hace que el aprendizaje sea más lento, porque cuánto más chica es la tasa de aprendizaje, menos modifico los pesos.*

Entonces la gente se puso a pensar qué se podría hacer para *mantener la velocidad de convergencia y a su vez suavizar la trayectoria* para que no se vaya siempre para cualquier lado. Y la **idea** es similar a la que se usa con el momento físico: *a la variación de pesos que indica el gradiente (que es la que acabamos de calcular) le suma una componente que dice cómo venía la modificación anterior, entonces no solo hago la modificación con la nueva dirección que me dice el gradiente, sino que también le sumo un poco de cómo venía, de lo que hizo la última vez, así mantiene un poco del paso anterior y le agrega lo nuevo, y eso nos da una dirección intermedia que es la que vamos a usar*.

Otro detalle es que si en dos o más iteraciones vengo con la misma dirección, voy a aumentar la velocidad con la que se mueve, entonces también mejora en ese caso. Y puedo usar velocidad de aprendizaje más grande porque el término de momento me va a ayudar.

Si nos vamos a las fórmulas, antes para el incremento de pesos teníamos la ecuación Δwn = μδIIyIIT donde “y” esta transpuesto para hacer un producto de vector columna por vector fila y obtener una matriz como explicamos antes.

*Ahora el término de momento modifica eso*, diciendo que el incremento de los pesos en la iteración “n” será:

ΔwnII = μδIIyIIT + **α Δwn-1**

Donde *le suma una cierta cantidad “α” que sería la “constante del momento” por el delta de pesos en la iteración anterior*. Eso sería lo que explicamos arriba del efecto que provoca.

*Esa constante de momento hace que le sumemos un porcentaje de la dirección con la que venía.*

Entonces ese término **α Δwn-1** es lo que se llama “**término de momento**”, y su función, como vimos antes, es *suavizar la trayectoria para que no haya cambios tan bruscos* debidos a los cambios de dirección del gradiente.

*En general tiene un efecto estabilizador, que hace que, para una misma tasa de aprendizaje, lo estabiliza e incluso puedo subir un poquito la tasa de aprendizaje y mejorar la convergencia.*

*(En el ítem 2 del ejercicio debemos usar lo mismo del ítem 1 pero añadiendo el término de momento y calcular los tiempos de convergencia en los dos casos para comparar. Por eso que dije arriba entiendo que cuando apliquemos el término de momento debería obtener una convergencia un poquito más rápida, y podría probar subirle un poco la tasa de aprendizaje y ver si mejora más la convergencia. Para eso nosotros usamos los comandos tic y toc para calcular el tiempo y también contamos la cantidad de épocas que se usan).*

-Para casos particulares también puede haber *problemas*, porque puede que el gradiente nos indique una dirección y al sumarle ese momento nos movamos más hacía un lado que no es el adecuado y en el paso siguiente lo mismo y así quedar oscilando o algo similar.

*Pero en general al agregar momento suele mejorar el algoritmo y converger mejor.*

Y una vez que calcule el incremento de pesos para la nueva iteración (Δw), ahora sí puedo actualizar los pesos sumando ese incremento al peso actual, por ejemplo, para la capa II en ese caso sería:

wnII = wn-1II + ΔwnII

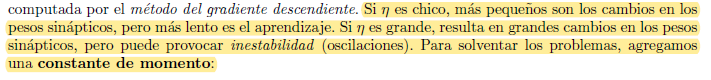
*Con esto actualizamos todos los pesos de la capa* II (porque usamos lo que expliqué de vector columna por vector fila igual a matriz de incrementos de pesos), mientras que antes obteníamos un solo incremento.

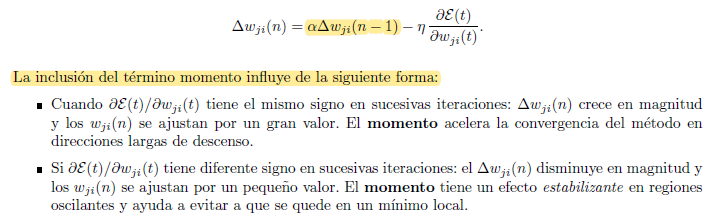
Y también ahora le estamos agregando esa cantidad “α” más el incremento para considerar el momento.

Para eso, el valor ΔwnII que voy calculando lo tengo que ir guardando para usar después en la iteración siguiente.

Para cada una de las capas voy a tener una matriz de incremento de pesos guardada Δw.

-En la imagen de abajo, “ƞ” es la tasa de aprendizaje.





***Repasando***:

Cada capa tiene asociada su matriz de pesos, al hacer la propagación hacia adelante calculo la salida de cada capa y las voy guardando porque luego las voy a necesitar para la retro propagación. Calculo el error al final y hago la propagación hacia atrás y a cada capa le voy guardando su vector de deltas “δ” que necesito para la actualización de los pesos. *Una vez calculados todos los δ*, aplico la ecuación **Δwncapa = μδcapa ycapaT + α Δwn-1** para cada una de las capas (el “T” es de transpuesto), donde sumamos ese término de momento α Δwn-1 que explicamos, por lo tanto, tengo que ir guardando también cuánto me había dado esa actualización de pesos anterior para sumarla en la siguiente, y al comienzo lo inicializo en 0 porque no hay iteración anterior.

-En el *ítem 1* dijimos que debíamos encontrar la arquitectura que mejor resuelva al “concentlite.csv”, viendo cuántas neuronas debía tener, probar la tasa de aprendizaje, ver cuántas épocas nos lleva según el criterio de error aceptable que le pongamos y demás.

Y ahora en el ***ítem 2*** *la idea de esta parte es que hagamos lo mismo pero ahora agregando el término de momento*, *y al agregar ese término deberíamos ver algún cambio en la velocidad de convergencia, es decir, en la cantidad de épocas que nos lleve.*

Puede que no lo veamos si lo hacemos una sola vez, porque si cada vez que corremos tenemos pesos iniciales distintos, puede ser que la primera vez que corrimos el algoritmo se haya partido de un punto inicial más fácil, entonces hay que probar correrlo varias veces sin término de momento y después varias veces con término de momento, y comparar la cantidad de iteraciones que nos da en promedio los dos casos, con la misma tolerancia de error por supuesto.

Sino otra forma de hacerlo podría ser fijar la semilla inicial para que inicien de la misma manera, desde el mismo punto inicial, pero es lo mismo.

**Item 3**

Esta última parte del ejercicio contempla otro aspecto importante que no hemos analizado todavía, y es *¿qué es lo que le pongo como entradas a una red?*

Si vemos la gráfica que hay debajo del tercer ítem del ejercicio, observamos datos de forma *bidimensional*, *dos entradas* (x, y) en ese caso, pero podría ser (x1, x2) para no confundirnos.

Lo que me dice este ítem del ejercicio es que, dados esos cuadraditos y cruces, saque su promedio en ambas coordenadas (la *media total*), y eso me va a dar un punto, y luego lo que me dice es que, cada uno de esos puntos (cruces y cuadrados) en vez de representarlos por la coordenada (x, y) que tienen, ahora los vamos a representar con el radio, la *distancia a ese punto de la media total que obtuvimos*, así que cada punto ya no va a ser un vector de dos coordenadas (x, y) sino que va a pasar a ser de *una sola coordenada* que será el radio que tiene hasta el punto de la media total.

El efecto que tiene eso es que ahora en la tabla de datos ya no voy a tener dos columnas para (x1, x2) sino que ahora voy a tener *una sola columna* “R” para el radio asociado a cada punto. Entonces, es un problema que en vez de ser bidimensional *pasa a ser* ***unidimensional***, y lo que se propone es que *resolvamos ese problema unidimensional con un perceptrón simple*, no con uno multicapa como veníamos usando, y que luego *comparemos los resultados obtenidos*.

**Importante**: lo que estamos haciendo es, en vez de usar las variables de entrada del problema como vinieron, le hicimos un procesamiento para ***extraer características***, en este caso, el radio, *y la idea es ver que aun usando un perceptrón simple vamos a obtener un resultado bastante bueno, y se hace eso para entender que es importante no solo que el clasificador que vamos a usar sea bueno (que tenga momento, que sepamos poner una buena tasa de aprendizaje, etc.), sino que también es importante que las variables de entrada que usamos reflejen realmente las características o tengan mucha información de las características del problema real, y si logramos tener variables de entrada que tengan buena información sobre el problema, por más que sean pocas podemos lograr buenos resultados.*

**EJERCICIO 4**

Este ejercicio tiene datos reales, no como los otros que venimos haciendo. Es un problema del mundo real.

Lo que tenemos es un género de plantas, llamado “Iris”, como se explica en el enunciado.

Tenemos tres clases en este problema: Iris setosa, versicolor y virginica, mientras que antes veníamos trabajando solo con dos clases.

Este dataset contiene información de las longitudes y anchos de los pétalos y sépalos de estas plantas, es decir, los datos vienen ordenados como:

Longitud de pétalo, ancho de pétalo, longitud de sépalo, ancho de sépalo, salida deseada (tres valores)

Teniendo esos datos de 50 plantas de cada una de las tres especies (en realidad el archivo de entrenamiento tiene 111 patrones y el de prueba 38).

*En el archivo se tiene esas mediciones (en cm) y un código binario que indica la especie (clase), ese código es:*

*[-1 -1 1] para la clase setosa, [-1 1 -1] para la clase versicolor y [1 -1 -1] para la clase virginica.*

Entonces tenemos todos esos patrones o datos, donde cada dato es un vector de cuatro valores, porque tiene longitud de pétalo, ancho de pétalo, longitud de sépalo, ancho de sépalo, es decir, los datos (entradas) están en R4. Y la diferencia con lo que venimos trabajando es que ahora, como *tenemos más de dos salidas*, vamos a usar una codificación porque tenemos *más de una neurona de salida*, ya que una sola neurona nos daría +1 o -1, así que necesitamos más y por eso usamos una codificación.

Una forma de hacerlo puede ser codificar en binario, entonces con dos neuronas podemos tener cuatro clases de salidas, pero es más complicado ya que la red tiene que aprender a clasificar y también a codificar en binario.

Entonces *lo más normal es usar un esquema donde se usa* ***una salida por clase***.

Entonces, *si tenemos tres clases, vamos a tener tres neuronas de salida*, y la salida deseada cuando el dato es de una clase, la neurona adecuada tiene que dar +1 y las otras dos -1.

*Cada neurona de salida está asociada a una clase*, por ejemplo, la neurona 1 a la virginica, la neurona 2 a la versicolor y la neurona 3 a la setosa, entonces cuando el dato sea de una virginica, tiene que dar +1 la salida 1 (asociada a la neurona 1) y las otras dos tienen que dar -1, entonces nos queda [1 -1 -1] como se indica en el enunciado, y así con las otras dos.

-Entonces sabemos que vamos a usar un *perceptrón multicapa* el cual debe tener *tres salidas* (capa final con tres neuronas, debido a la cantidad de clases que tenemos).

***¿Qué estructura usar?***

Viendo las gráficas que muestra en pdf, para pensar en la arquitectura de la red quizás la primera categoría que se encuentra más separada de las otras la podemos clasificar con una neurona y las otras dos categorías con otras dos, y tenemos tres salidas, así que podríamos probar con una arquitectura [3, 3] y sino con [4, 3].

Y sino, si se pregunta, la arquitectura más básica podría ser [3], que serían tres perceptrones simples, y se podría entrenar por ejemplo que cada neurona clasifique una de las tres categorías. Y usando dos capas, la estructura mínima sería [1, 3], ese “1” proyecta las cuatro dimensiones (los cuatro valores de ancho y largo que tenemos) en una dimensión

-*¿Qué* ***cambios*** *nos impone usar esta codificación?*

Por un lado, antes cuando teníamos una sola salida, ésta tenía que ser +1 o -1, y el error era un número. Ahora como la salida tiene tres valores, *tanto la salida obtenida como la salida deseada van a ser un* ***vector****, y el error va a ser un* ***vector de error***.

*Como segundo cambio está el cómo voy a contar los errores*, es decir, cómo sé si acertó o no.

Lo que hacíamos con una entrada era hacer que las salidas den +1 o -1 con la función signo (porque la salida lineal tiende a 1 o -1 pero nunca va a ser exactamente), y comparar eso con la salida deseada. Eso para contar la cantidad de errores en la etapa de prueba.

El problema acá es que podría ser que para varios valores nos de 1 si usamos eso. Por ejemplo: si tenemos como salida deseada yd = [1 -1 -1] y como salida obtenida y = [0.98, 0.01, -0.5], al aplicar la función signo sobre ese vector salida “y” nos daría un vector [1 1 -1], y si lo comparamos con la salida deseada “yd” vamos a decir que es un error, porque no coinciden, pero en realidad el vector “y” nos está diciendo que la red opto por la salida 1 y no por las otras, porque nos está dando 0.98 en la posición 1 (el valor más grande).

Entonces lo correcto acá es usar un enfoque que se llama ***winner takes all*** (“one hot” también se le llama), donde lo que se hace es que “la ganadora se lleve todo”, es decir, busco la salida con valor más grande y le pongo +1 y al resto -1. De esa manera, a la salida 0.98 le voy a poner un 1 y al resto -1, obteniendo así el vector [1 -1 -1] y ahora sí al comparar con yd estaría correcto.

***Importante****:*

*Recordar que esto es para el conteo de aciertos o errores en la etapa de pruebas, pero dentro del entrenamiento lo que se usa es el vector error, que es la resta entre la salida deseada y la salida obtenida*, y eso nos da un vector de errores que lo uso en la propagación hacia atrás.

Entonces *en realidad tengo* ***dos tipos de error*** (esta todo explicado más al comienzo del word): uno interno que usa la red durante el entrenamiento para actualizar los pesos (error cuadrático por ejemplo), y otro es el error interpretable por el humano para saber si nos dio o no la clase que queríamos.

-Otro detalle es que el bucle de entrenamiento de esto es similar al del perceptrón simple, hacemos una época actualizando los pesos (el primer for) y luego volvemos a pasar todos los patrones para estimar el desempeño que está teniendo en esa época (el segundo for para hacer la verificación de desempeño, que acá lo hacemos llamando a la función pruebas\_MLP casi al final de la función entrenamiento\_MLP, con los datos de entrenamiento, para ver si corta o sigue entrenando y corrigiendo los pesos) y determinar si el error ya es aceptable y paramos o si debemos seguir en el bucle para seguir ajustando los pesos.

Volviendo al enunciado

En este caso tenemos *pocos patrones*, entonces ¿cómo sabemos si el clasificador es bueno o no para compararlo con otros clasificadores? *Lo que se hace en los casos que tenemos pocos patrones es usar alguna variante de* ***validación cruzada***, y en este caso utilizamos“**leave\_k\_out**” y “**leave\_one\_out**”. Pero, ¿qué son esas dos?

Supongamos que tenemos cargados en una matriz todos los patrones del problema. Ahora, en el **leave\_k\_out** lo que hacemos es que voy a tomar los patrones de a “k” y se los deja afuera (por eso el nombre), los dejo afuera para el test. Entonces entreno con el resto de patrones y esos “k” que deje afuera los uso para probar.

Luego, tomo los siguientes “k” patrones, los dejo afuera para probar y entreno con el resto, y así sucesivamente voy tomando de a “k” patrones hasta que haya abarcado todos los patrones que tenga.

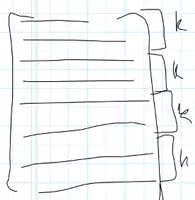
Cuántas veces lo voy a hacer depende del tamaño del conjunto y del valor de “k”.

Por ejemplo, si k=5 voy a tomar los 5 primeros patrones, los dejo afuera, entreno con el resto y pruebo con esos 5 patrones que deje afuera, luego tomo los 5 siguientes y así.

*Así al finalizar voy a haber usado todos los patrones para la etapa de pruebas alguna vez, y reportamos como siempre la media, varianza o desvío de todo*.

Mientras que **leave\_one\_out** sería llevar “k” al mínimo posible, es decir, tomar k=1, así que en el primer entrenamiento voy a dejar el primer patrón afuera, entreno con todos los demás y pruebo con ese solo que deje afuera, luego en el segundo entrenamiento dejo afuera al segundo patrón, entreno con los demás y pruebo con ese, y así sucesivamente hasta terminar y haber dejado afuera a cada uno de los patrones una vez para usarlos por separado en la prueba.

Al final saco la media de todos los resultados y su varianza o desvío.



-*Dijo el profe que para la evaluación es mejor si guardamos una tabla con los resultados obtenidos al usar k\_out y one\_out para mostrar eso porque va a demorar mucho en la ejecución.*

(En 2023 no lo usamos).

El ejercicio pide usar los dos métodos usando un perceptrón multicapa y *compararlos*.

**Extra**:

En el vídeo “a05 estimación de la capacidad de generalización: variantes de validación cruzada” (link abajo) habla de **leave\_k\_out** (que se explica en el vídeo a04) y de **k-fold**, donde la diferencia es que en k-fold el “k” es la cantidad de particiones que tengo, por ejemplo, si k=5 significa que voy a armar 5 particiones, en cambio en leave\_k\_out el “k” es la cantidad de patrones que dejo fuera para probar.

También se habla del problema del leave\_one\_out, que es que si tengo por ejemplo 100 patrones, tengo que entrenar a mi algoritmo 100 veces, y cada vez dejar uno solo fuera para probar, lo cual es bastante pesado. La ventaja del leave\_one\_out es que el sesgo de error es más pequeño que con otros.

<https://www.youtube.com/watch?v=eDT2-7I-V9g&list=PL7La0W0xzesW8hcRSXn54ZbM8VeloB3K8&index=5>

***Repasando el ejercicio 4:***

Vamos a usar un perceptrón multicapa, lo vamos a tener que modificar para contemplar el uso de tres clases (tres salidas), usando la codificación que se pide y que se explicó antes, y vamos a implementar el leave\_k\_out y el leave\_one\_out.

En todos estos métodos es recomendable usar **gráficas** para ver el entrenamiento, en cada época evaluar el desempeño (verificación) y reportar ese resultado en forma visual, con una gráfica donde podamos ver en el eje horizontal las épocas y en el eje vertical el error cuadrático medio por ejemplo, entonces iríamos viendo cómo va cambiando el error hasta llegar al final de las épocas. También podríamos hacer una gráfica de porcentaje de error en vez del error cuadrático medio, donde en ese caso podemos ver que la gráfica va bajando pero de forma más escalonada y quizás en algunas partes constante por lo que explicamos antes de que puede que los pesos estén cambiando pero que no se refleje en cambios en la clasificación de las clases, por eso no se modifica el promedio de error durante unas épocas.

También hacer esas gráficas nos permite saber cómo ir cambiando la tasa de aprendizaje, porque si vemos que pasa un periodo de muchas épocas donde no hay mejoras, ahí nos conviene subir la tasa de aprendizaje o cosas así.