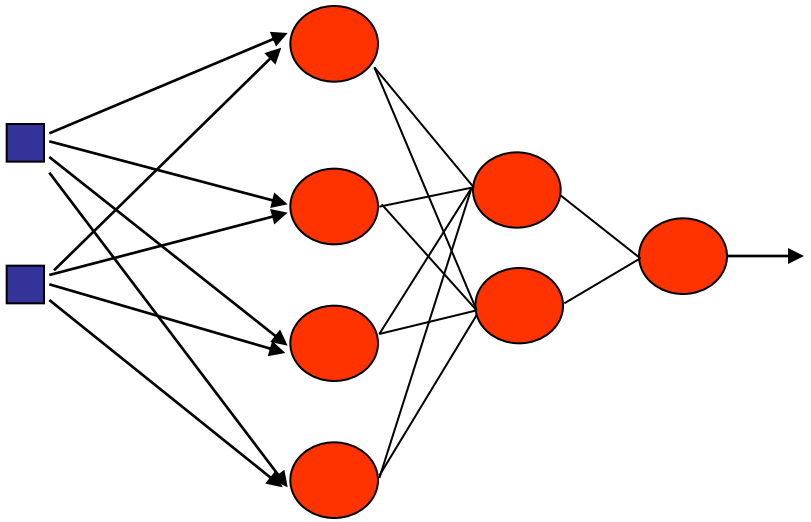
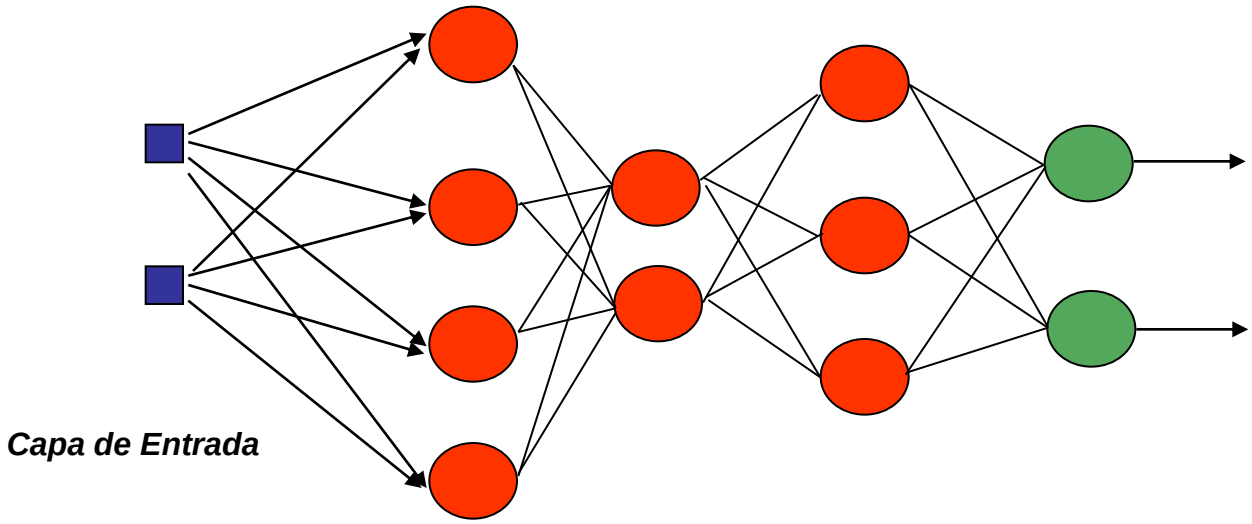
MLP
Multiple Layer Perceptron
(PERCEPTRONES MULTICAPAS)

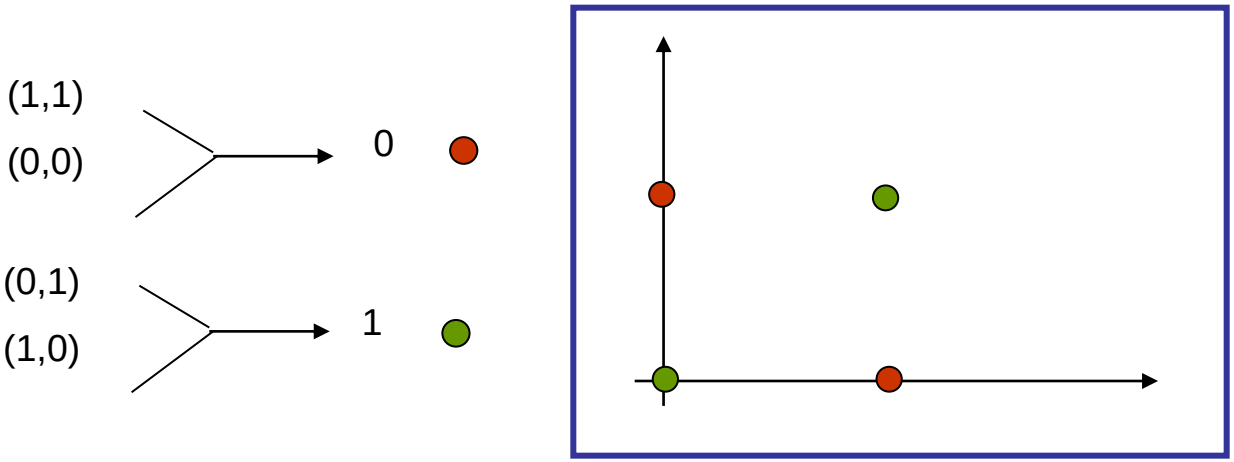


Un perceptrón multicapa es la extensión natural del perceptrón estudiado anteriormente.

Se obtiene considerando la capa de entrada, donde se encuentran las neuronas que reciben el estímulo, una o más capas ocultas de neuronas, donde se procesa el estímulo y finalmente la capa de salida, donde están las neuronas que generan la respuesta .



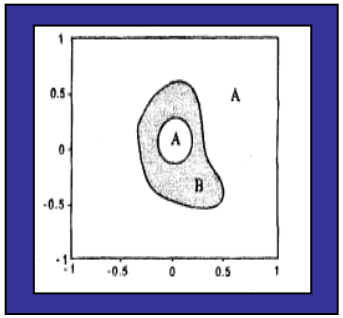
El trabajo desarrollado por *Minsky y Papert* en 1969 (*Minsky, M.L and Papert, S.A. Perceptrons. MIT Press)* mostró matemáticamente que el perceptrón es incapaz de realizar ciertas generalizaciones a partir de ejemplos. Por ejemplo, el problema del O-exclusivo.



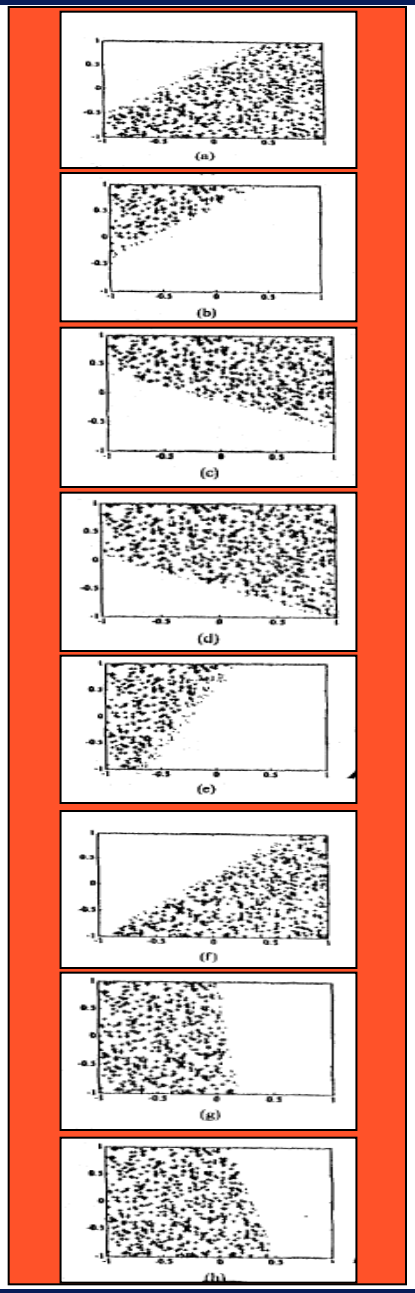
El trabajo de Minsky y Papert influyó notablemente para que surgieran dudas acerca de la capacidad de los perceptrones y sus variantes, como los perceptrones multicapas. Esto paralizó los desarrollos e investigaciones en esta área hasta aproximadamente la década de los 80.

Incluso al final del libro conjeturan que las limitaciones por ellos encontradas serían ciertas también para los perceptrones multicapas.

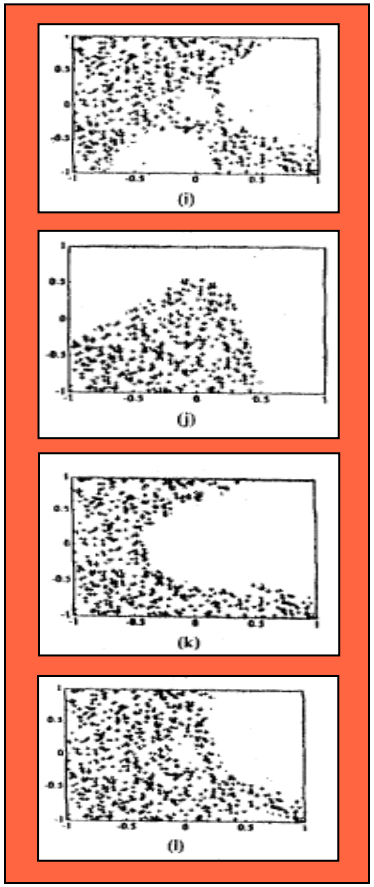
Es justamente el considerar más capas lo que hace que el perceptrón multicapa pueda lograr representaciones internas más complejas……..



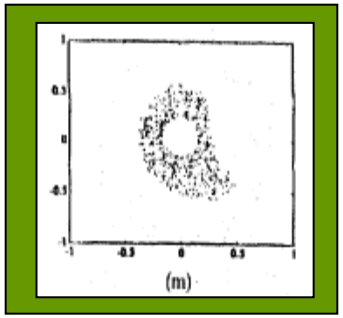
Entrada



1ra Capa oculta



2da Capa oculta



Salida

Los perceptrones multicapas tienen 3 características básicas:

1) El modelo para cada neurona en su estructura puede considerar funciones de transferencia distintas y no lineales.

Qué pasa si en un perceptrón multicapa sólo se utilizan funciones lineales ?

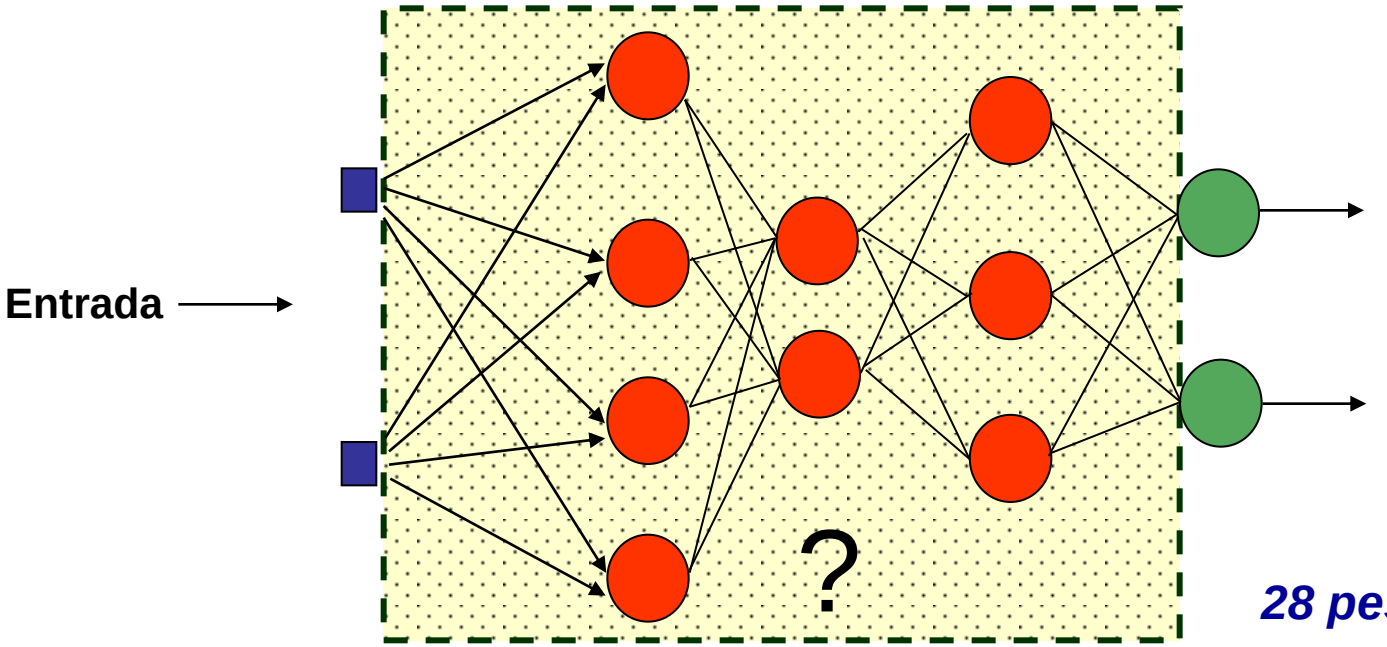
2) La presencia de una o más capas escondidas (*hidden layers)* que permiten que los perceptrones multicapas puedan resolver tareas complejas.

3) Alto grado de conectividad determinado por las sinapsis de la red neuronal.

Estas mismas características son a su vez limitantes, ya que:

1) El grado de no linealidad obtenido al considerar funciones de transferencia no lineales junto con la alta conectividad de la red hacen que desarrollos teóricos sean difíciles de obtener.

2) La presencia de capas ocultas hace que el proceso de entrenamiento sea difícil de visualizar y aún más entender el comportamiento de los pesos sinápticos.



28 pesos sinápticos !

ENTRENAMIENTO DE PERCEPTRONES MULTICAPAS

$$\Delta w_{ij}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ij}(n)}$$

El algoritmo de entrenamiento comúnmente utilizado para entrenar un perceptrón multicapa es el algoritmo de retropropagación (*error backpropagation algorithm o backprop algorithm).*

Muchos intentos al tratar de construir perceptrones que fueran capaces de realizar tareas más complejas fallaron precisamente por no contar con un algoritmo de entrenamiento adecuado como es el *backpropagation algortihm.*

La filosofía del algoritmo fue descubierta y reportada en distintos trabajos independientes.

Rumelhart, Hinton y Williams en su libro *Parallel Distriibuted Processing: Explorations in the Microstructures of Cognition (1986)*

Proponen el algoritmo de *backpropagation* tal como se conoce en la actualidad. Sin embargo, posterior a esta publicación se encontró que el algoritmo había sido descubierto independientemente por Parker (1985) y Lecun (1985).

Parker, D. Learning Logic:*Casting the cortex of the human brain in silicon*. Technical report TR-47, MIT press. (1985)

Lecun, Y. *Une procedure d'apprentissage pour reseau a seuil assymetrique*. Cognitiva, Vol.85, pp. 599-604, (1985)

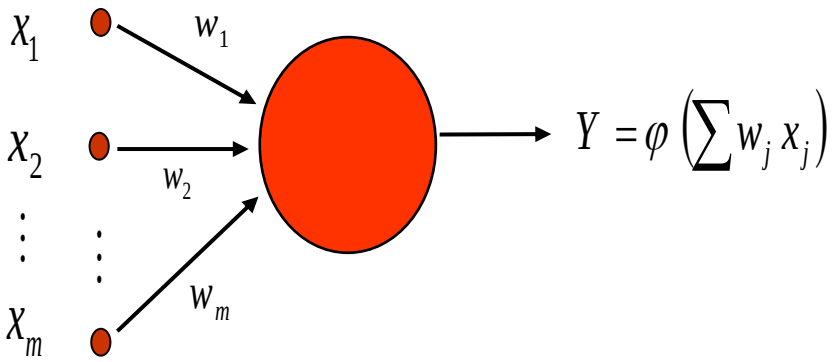
Posterior a esto se encontró que incluso el algoritmo de *backpropagation* había sido descrito y propuesto por Werbos, P (1974)

Werbos,P. Beyond Regression: New tools for prediction and analysis in behavioral sciences, Ph;D Thesis, Harvard University (1974)

Y antes de este trabajo el desarrollo del algoritmo de *backpropagation* fue desarrollado formalmente por Bryson, A y Ho, Y(1969)

Bryson, A y Ho, Y. Applied Optimal control (1969)

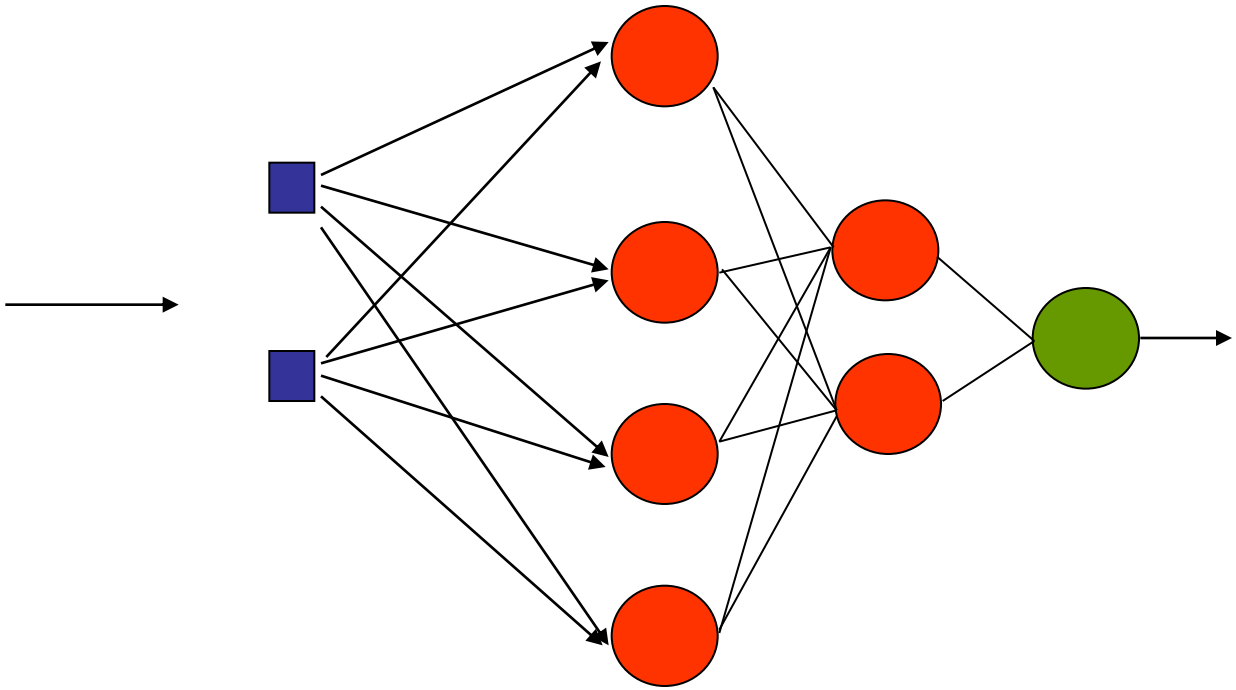
La regla delta es fácil de entender y visualizar



Es fácil determinar el error y utilizarlo para corregir los valores de los pesos sinápticos

$$w_j(n+1) = w_j(n) + \eta\, e(n) x_j(n)$$

La idea del algoritmo *error-backpropagation* es extender la fórmula de corrección de error de la regla delta (*delta rule*) al caso de perceptrones multicapas. Sin embargo el proceso se complica cuando hay muchas neuronas y capas ocultas.



Todas las neuronas son responsables por el error cometido por la red

El error sólo se puede medir en la capa de salida y no en las capas ocultas

¿Cómo utilizar este error para proceder a realizar los cambios en los pesos sinápticos de las neuronas en las capas escondidas ?

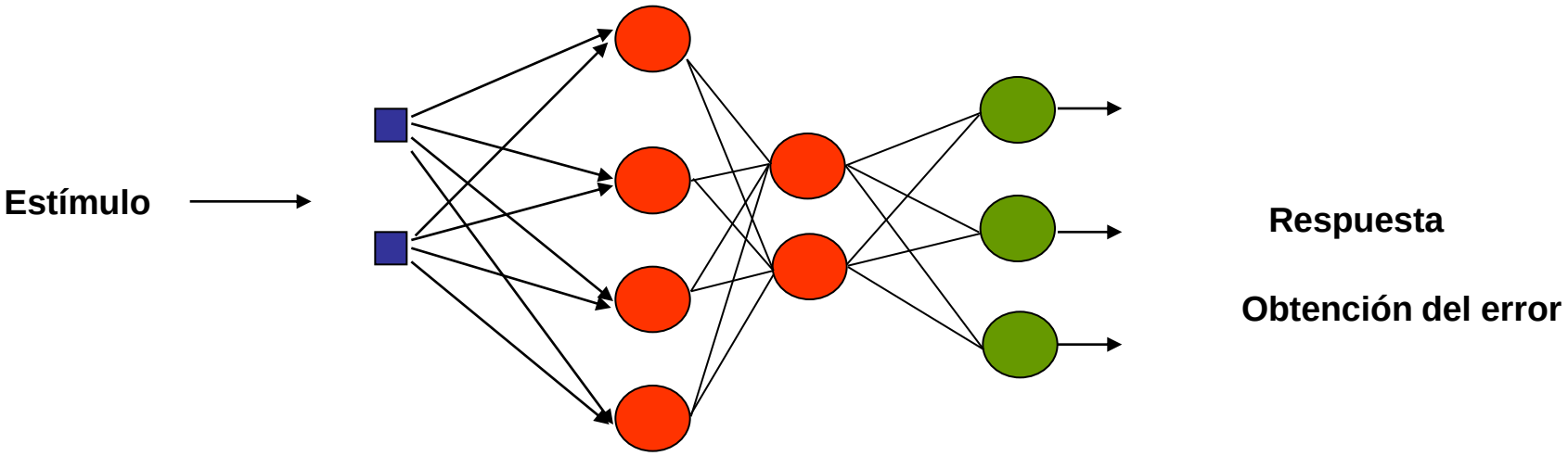
La idea es realizar dos pasos básicos:

1) Propagación hacia delante (*forward pass*)

Se presenta el estímulo a la red, cuyo efecto se propaga neurona por neurona y capa por capa.

Durante esta propagación los pesos sinápticos están fijos y no se modifican.

En la capa de salida se obtiene la respuesta al estímulo presentado y se calcula el error

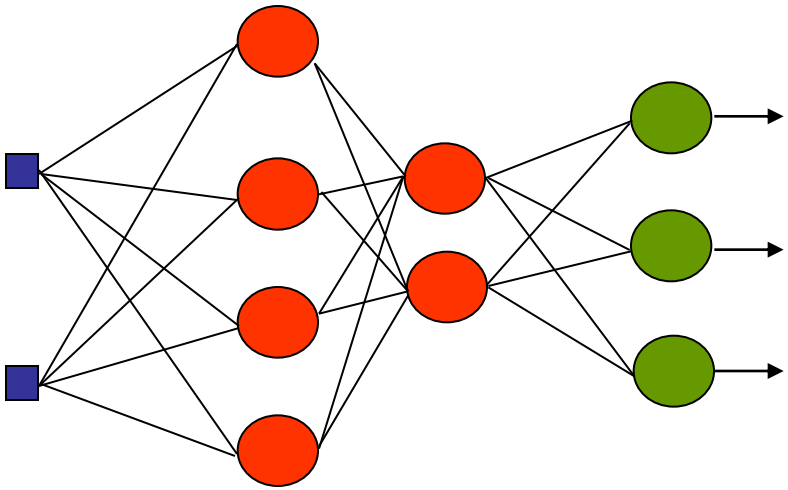


2) Propagación hacia atrás (*backward pass*)

El error obtenido en las capas de salida se propaga hacia atrás, neurona por neurona y capa por capa.

Esto se logra considerando una cantidad δ_j denominada *el gradiente local de la neurona j* que contiene la información del error. Este se calcula de manera recursiva en el sentido de la propagación.

Los pesos sinápticos de las neuronas se modifican siguiendo la regla delta con la información del gradiente local de cada neurona.



Obtención del error

DERIVACION DEL ALGORITMO DE RETROPROPAGACIÓN

(BACKPROPAGATION)

$$\delta_j(n) = y_j(n)\left(1 - y_j(n)\right)\sum_k \delta_k(n)\, w_{kj}(n)$$

Sea $e_j(n)$ el error cometido por la neurona j de la capa de salida en la n-ésima iteración

$$e_j(n) = d_j(n) - Y_j(n)$$

La idea es buscar una función de costo apropiada que tome en cuenta el error cometido por las neuronas en la capa de salida. Primero se considera:

$$\varepsilon(n) = \frac{1}{2}\sum_{j \in C} e_j^2(n)$$

C denota las neuronas en la capa de salida

Este es el error cometido por la red en la n-ésima iteración.

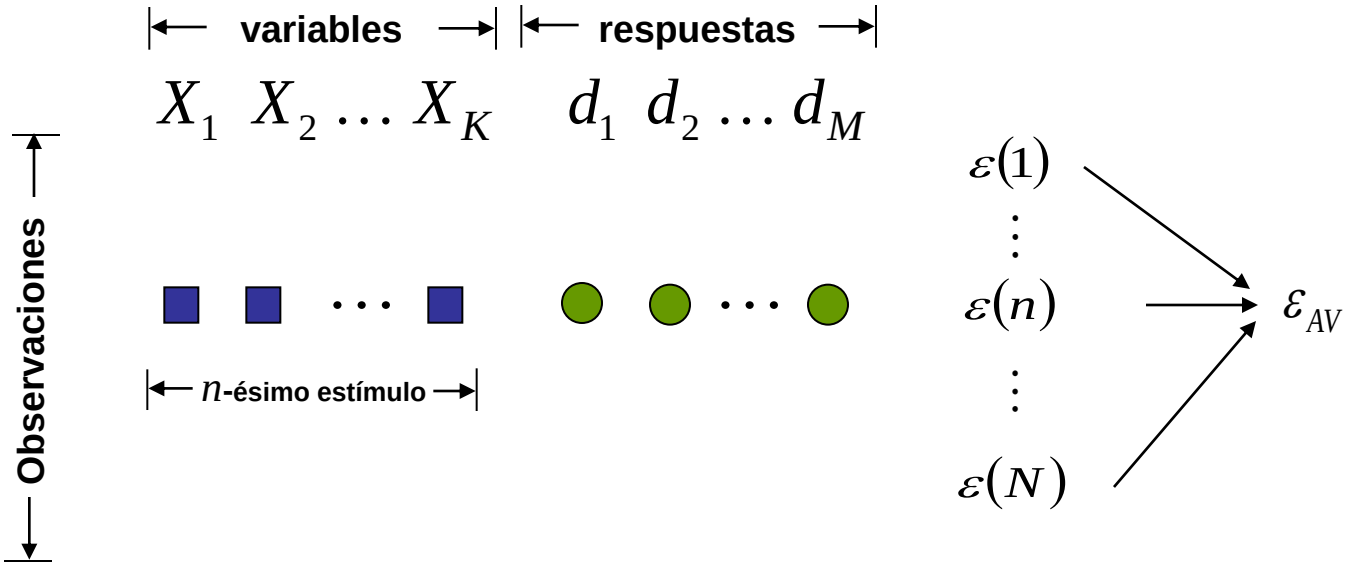
Luego, si se tienen N estímulos o valores de entrada entonces el error promedio cometido por la red después de presentarlos todos (después de una época) es

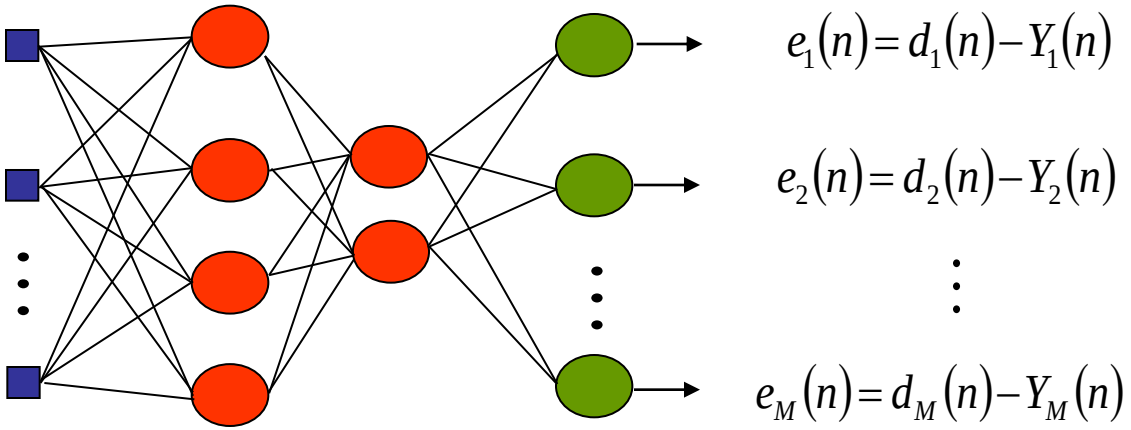
$$\varepsilon_{AV} = \frac{1}{N}\sum_{j=1}^{N}\varepsilon(n)$$

El objetivo del algoritmo de entrenamiento es ir ajustando los pesos sinápticos de manera tal de minimizar este error.

El algoritmo de backpropagation comunmente usado, hace este ajuste cada vez que se presenta un nuevo estímulo o valor de entrada a la red. Este es un estimado del cambio real del error promedio.

Es por esto que el valor mínimo del error no se alcanza en la primera época, si no que requiere de varias épocas para alcanzarlo y finalizar el entrenamiento.





Capa de Entrada (K neuronas)

Capa de Salida (M neuronas)

El error promedio es función únicamente los pesos sinápticos de la red.

$$\varepsilon_{AV} = \frac{1}{N}\sum_{n=1}^{N}\varepsilon(n)$$

$$\frac{1}{2}\sum_{j\in C} e_j^2(n)$$

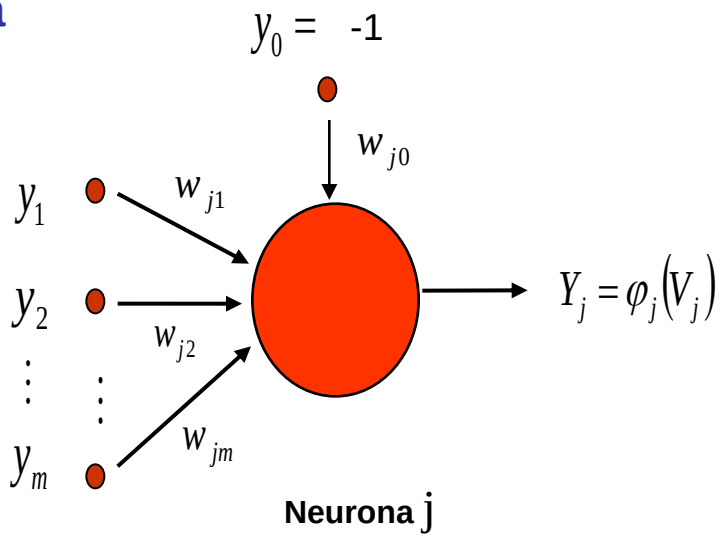
$$d_j(n) - Y_j(n) \qquad j \in C$$

Como los pesos se van modificando en cada iteración, el factor de corrección apropiado se obtiene calculando:

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)}$$

Este cálculo es distinto si la neurona está en la capa de salida o en alguna capa oculta (recordar que el error es una función de los pesos)

* Neurona j en la capa de salida



$w_{ji}(n)$ { **Valor en la n-ésima iteración del peso sináptico que conecta la j-ésima neurona de la capa de salida con la i-ésima neurona de la capa oculta anterior**

$V_j(n) = \sum_i w_{ji}(n) y_i(n)$ { **Estímulo recibido por la j-ésima neurona de la capa de salida en la n-ésima iteración**

$Y_j(n) = \varphi_j(V_j(n))$ { **Respuesta de la j-ésima neurona de la capa de salida en la n-ésima iteración**

Aplicando la regla de la cadena se obtiene que:

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = \frac{\partial \varepsilon(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial Y_j(n)} \frac{\partial Y_j(n)}{\partial V_j(n)} \frac{\partial V_j(n)}{\partial w_{ji}(n)}$$

$$\frac{\partial \varepsilon(n)}{\partial e_j(n)} = e_j(n)$$

$$\frac{\partial e_j(n)}{\partial Y_j(n)} = -1$$

$$\frac{\partial Y_j(n)}{\partial V_j(n)} = \varphi'_j(V_j(n))$$

$$\frac{\partial V_j(n)}{\partial w_{ji}(n)} = y_i(n)$$

En consecuencia:

$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = -e_j(n)\varphi'_j\left(V_j(n)\right)y_i(n)$$

Observaciones:

- **Cuando la función de activación es lineal, la fórmula anterior coincide con la regla delta. (demostrarlo !)**
- **El término correspondiente al error se puede calcular porque la neurona está en la capa de salida.**
- **La corrección necesaria es entonces:**

$$\Delta w_{ji}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = \eta\, e_j(n)\, \varphi'_j\left(V_j(n)\right)y_i(n)$$

El signo menos viene del algoritmo de descenso de gradiente!

El gradiente local de la neurona j en la capa de salida es la variación en el error debido al estímulo recibido por la neurona.

$$\delta_j(n) = -\frac{\partial \varepsilon(n)}{\partial V_j(n)}$$

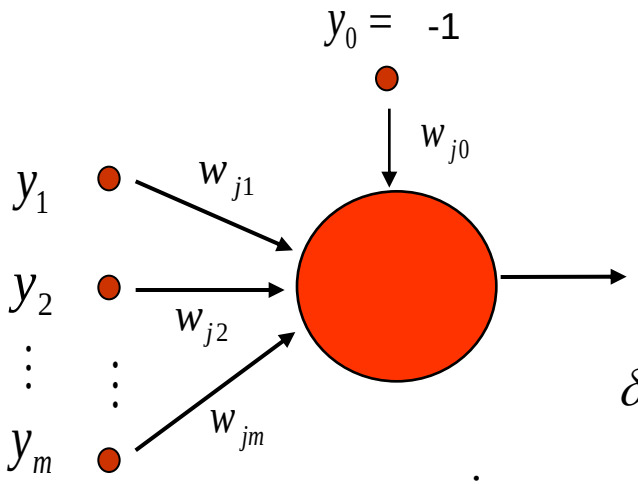
$$= -\frac{\partial \varepsilon(n)}{\partial e_j(n)} \frac{\partial e_j(n)}{\partial Y_j(n)} \frac{\partial Y_j(n)}{\partial V_j(n)}$$

$$= e_j(n)\varphi'_j(V_j(n))$$

y por lo tanto la corrección se puede escribir como:

$$\Delta w_{ji}(n) = \eta\, \delta_j(n) y_i(n)$$

Durante la actualización se calcula primero el gradiente local y luego sucesivamente se actualizan los pesos. Este proceso será igual para todas las neuronas de la red



$$\Delta w_{j0}(n) = -\eta\,\delta_j(n)$$
$$\Delta w_{j1}(n) = \eta\,\delta_j(n)\,y_1(n)$$
$$\Delta w_{j2}(n) = \eta\,\delta_j(n)\,y_2(n)$$
$$\vdots$$
$$\Delta w_{jm}(n) = \eta\,\delta_j(n)\,y_m(n)$$

* Neurona j en una capa oculta

Como no se conoce el valor del error $e_j(n)$ se redefine el gradiente local como:

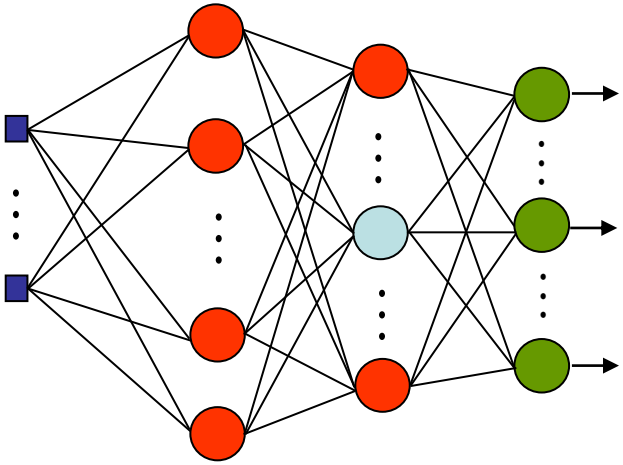
$$\delta_j(n) = -\frac{\partial \varepsilon(n)}{\partial Y_j(n)} \frac{\partial Y_j(n)}{\partial V_j(n)}$$

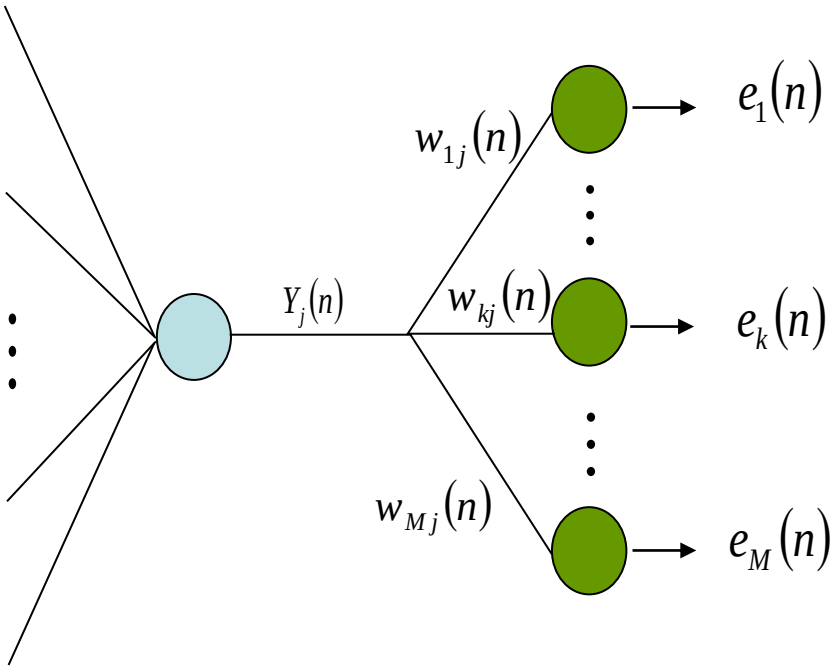
Ahora el término correspondiente al error no aparece, puesto que no hay dependencia explicita

$$= -\frac{\partial \varepsilon(n)}{\partial Y_j(n)} \varphi_j'(V_j(n))$$

Este término se calcula considerando dos casos

* Neurona j en una capa oculta que conecta con la capa de salida





Sumamos las contribuciones de todas las neuronas de la capa de salida

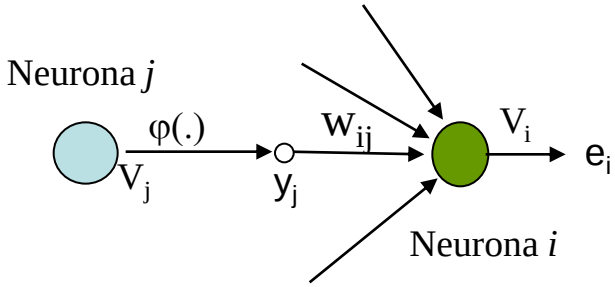
Utilizando la definición de $\varepsilon(n)$ se tiene que:

$$\frac{\partial \varepsilon(n)}{\partial Y_j(n)} = \sum_{i=1}^{M} e_i(n) \frac{\partial e_i(n)}{\partial Y_j(n)}$$

$$= \sum_{i=1}^{M} e_i(n) \frac{\partial [d_i(n) - \varphi_i(V_i(n))]}{\partial Y_j(n)}$$

$$= \sum_{i=1}^{M} -e_i(n) \varphi_i'(V_i(n)) \frac{\partial V_i(n)}{\partial Y_j(n)}$$

$$= \sum_{i=1}^{M} \boxed{-e_i(n) \varphi_i'(V_i(n))} w_{ij}(n)$$



gradiente local de la i-ésima neurona de la capa de salida

En consecuencia, el gradiente local de la neurona j de la capa oculta que conecta con la capa de salida es:

$$\delta_j(n) = -\frac{\partial \varepsilon(n)}{\partial Y_j(n)}\ \varphi'_j\left(V_j(n)\right)$$

$$= \varphi'_j\left(V_j(n)\right)\sum_{i=1}^{M}\delta_i(n)w_{ij}(n)$$

Hay que tener mucho cuidado con la notación !

Observaciones:

El gradiente local de las neuronas en la capa de salida contiene la información del error cometido por la red.

Este error se transmite a todas las neuronas de la capa oculta inmediata anterior mediante el gradiente local de estas, usando la ecuación anterior.

El gradiente local de cada neurona en la capa oculta se asocia entonces al error cometido por esta.

Como actualizar o corregir los pesos ?

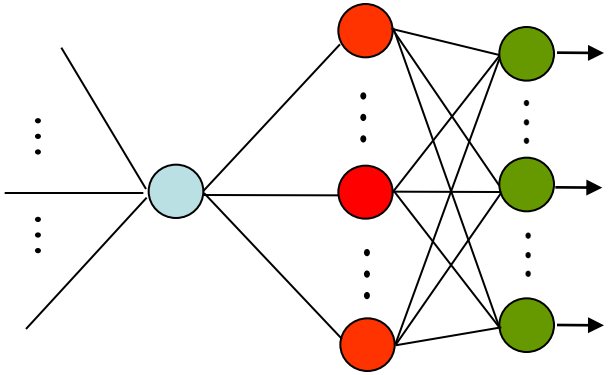
Los pesos se actualizan siguiendo la regla delta como si fuesen los pesos de la capa de salida, pero utilizando el gradiente local apropiado

$$\Delta w_{ji}(n) = \eta\, \delta_j(n)\, y_i(n)$$

$$\delta_j(n) = \varphi'_j\left(V_j(n)\right) \sum_{i=1}^{M} \delta_i(n)\, w_{ij}(n)$$

Respuesta de la i-ésima neurona en la capa anterior a la capa en la que se encuentra la neurona j

*** Neurona j en una capa oculta que conecta otras neuronas ubicadas en una capa oculta**



$$\delta_j(n) = -\frac{\partial \varepsilon(n)}{\partial Y_j(n)} \frac{\partial Y_j(n)}{\partial V_j(n)}$$

$$= \boxed{-\frac{\partial \varepsilon(n)}{\partial Y_j(n)}} \varphi_j'(V_j(n))$$

Si k es el número de neuronas en la capa oculta siguiente a la capa en la que se encuentra la neurona j entonces:

$$\frac{\partial \varepsilon(n)}{\partial Y_j(n)} = \sum_{\alpha=1}^{k} \frac{\partial \varepsilon(n)}{\partial Y_\alpha(n)} \frac{\partial Y_\alpha(n)}{\partial Y_j(n)}$$

Hay que tener mucho cuidado con la notación !

Ahora los valores Y_α corresponden a neuronas que conectan con la capa de salida y por lo tanto

$$\frac{\partial \varepsilon(n)}{\partial Y_\alpha(n)} = \sum_{i=1}^{M} \delta_i(n) w_{i\alpha}(n)$$

Luego, como

$$\frac{\partial Y_\alpha(n)}{\partial Y_j(n)} = \frac{\partial Y_\alpha(n)}{\partial V_\alpha(n)} \frac{\partial V_\alpha(n)}{\partial Y_j(n)} = \varphi'_\alpha(V_\alpha(n)) w_{\alpha j}$$

En consecuencia,

$$\frac{\partial \varepsilon(n)}{\partial Y_j(n)} = \sum_{\alpha=1}^{k} \varphi'_\alpha(V_\alpha(n)) \sum_{i=1}^{M} \delta_i(n) w_{i\alpha}(n) \, w_{\alpha j}(n)$$

gradiente local de la α-ésima neurona de la capa oculta inmediata posterior a la capa en la que se encuentra la neurona j

Con lo cual

$$\delta_j(n) = \varphi'_j(V_j(n)) \sum_{\alpha=1}^{K} \delta_\alpha(n) w_{\alpha j}(n)$$

La fórmula obtenida es exactamente igual a la anterior, sólo que relaciona el gradiente local de la neurona j con el gradiente local de las neuronas en la capa oculta inmediata posterior a la capa en la que se encuentra la neurona j

En conclusión:

- **Los pesos se actualizan utilizando siempre la misma fórmula:**

$$\Delta w_{ji}(n) = \eta\, \delta_j(n)\, y_i(n)$$

- **El gradiente local se calcula según la neurona esté en una capa oculta o en la capa de salida**

Capa de Salida

$$\delta_j(n) = [d_j(n) - Y_j(n)]\varphi'_j(V_j(n))$$

Capa Oculta

$$\delta_j(n) = \varphi'_j(V_j(n)) \sum_{\alpha} \delta_\alpha(n) w_{\alpha j}(n)$$

Algoritmo

- **Para cada patrón de entrada**
  - **Propagar la información hacia adelante para obtener todas las activaciones locales y el error cometido.**
  - **Propagar el error hacia atrás para obtener los gradientes locales en cada neurona.**
  - **Hacer la actualización de los pesos según fórmulas.**

- **Repetir hasta alcanzar condición de parada**
  - **gradiente sea pequeño en norma,**
  - **el cambio en el error promedio sea pequeño entre epocas sucesivas,**
  - **hasta alcanzar los resultados de generalización deseados,**
  - **número máximo de épocas.**

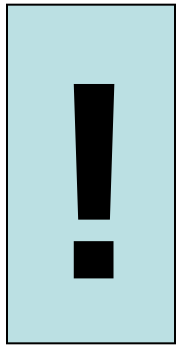
El algoritmo que vimos es con actualizaciones secuenciales (on-line, pattern, stochastic)

La contraparte con actualizaciones por lote (batch), los pesos se ajustan después de haber presentado todos los patrones, por lo que el error promedio por lotes considera la suma de los errores promedios secuenciales.

Si hicieramos la cuenta nuevamente, veriamos que los pesos se actualizan sumando los gradientes para cada patrón. (VERLO!!)

Secuencial	Lotes
• El entrenamiento secuencial requiere menos almacenamiento.	•Existen resultados de convergencia a mínimos locales.
•Por su naturaleza estocástica (presentación aleatoria de patrones), es menos probable detenerse en mínimo local.	•Fácil de paralelizar.
•Sencillo de implementar.	

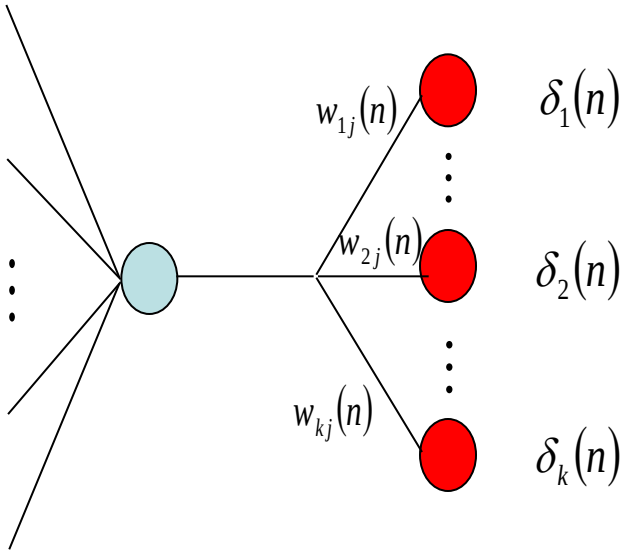
ALGUNAS CONSIDERACIONES SOBRE EL ALGORITMO DE BACKPROPAGATION



1. **El cálculo del gradiente local de la neurona j en una capa oculta se realiza utilizando la información de:**

   a) Los gradientes locales de las neuronas en la capa inmediata posterior a la capa en la que se encuentra la neurona j

   b) Los pesos que conectan a cada neurona en la capa inmediata posterior con la neurona j



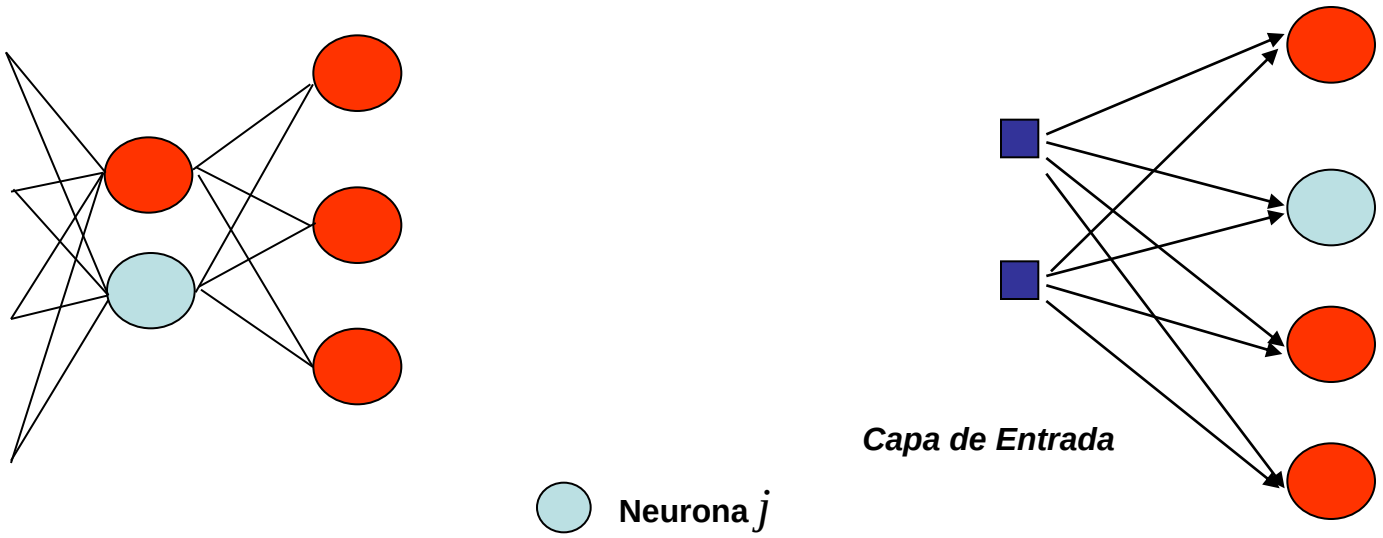
$$\delta_j(n) = \varphi'_j(V_j(n)) \sum_{\alpha=1}^{K} \delta_\alpha(n) w_{\alpha j}(n)$$

Neurona j

2. **La corrección de los pesos sinápticos de la neurona j se realiza utilizando el gradiente local δ_j y:**

   a) Los valores de entrada a esta, o equivalentemente, las respuestas de las neuronas en la capa inmediata anterior a la capa en la que se encuentra la neurona j

   b) Los estímulos de entrada si la neurona j se encuentra en la primera capa oculta



3. Actualización de los pesos y propagación del error

Se debe obtener primero los gradientes locales para cada una de las neuronas que forman la red y luego proceder a actualizar los pesos sinápticos

El entrenamiento ocurre en dos fases: Entrenamiento y Prueba

La data se particiona en dos subconjuntos:

- **Data de entrenamiento: se utiliza para escoger/calibrar el modelo**
  - **Data de estimación: entrenar la red (80%)**
  - **Data de validación: validar el modelo/red (20%)**
- **Data de prueba: probar generalización (más sobre este punto luego)**

Maximizar la cantidad y calidad de información en el conjunto de entrenamiento

Esto permite que el entrenamiento sea más rápido y que la red no invierta mucho tiempo en aprender algo que se sabe de antemano. Esto se puede lograr considerando:

- **Ejemplos para los cuales el error de entrenamiento es mayor**

  Revisar que no hay valores atípicos, cuya presencia puede ser catastrófica

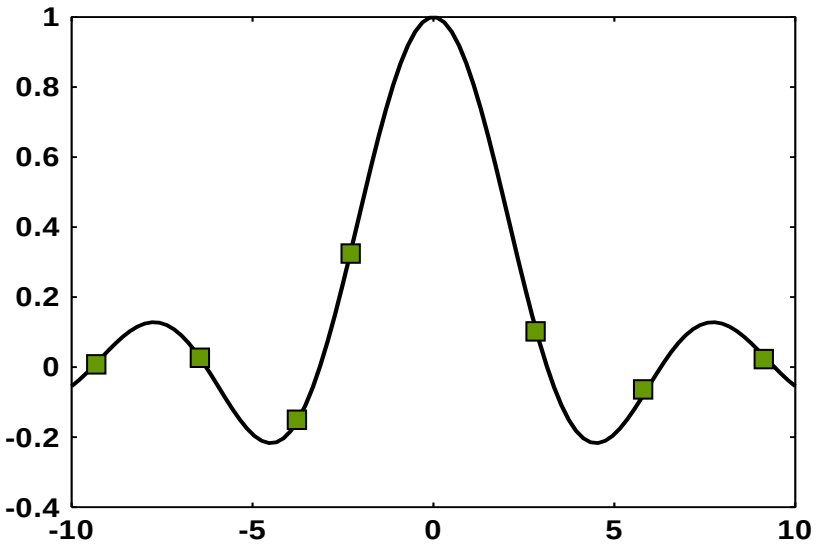
- **Ejemplos que son totalmente diferentes de los previamente usados (presentación aleatoria de los estímulos)**

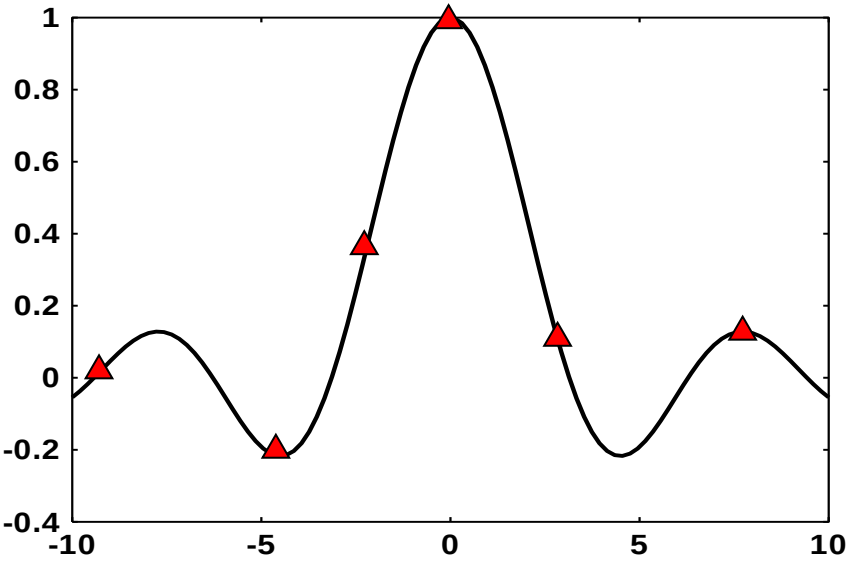
  Esto contribuye a evitar que estímulos consecutivos presentados a la red pertenezcan a la misma clase.

- **Ejemplos para los cuales la respuesta deseada es muy diferente al resto de los valores utilizados para la supervisión de la red**

- **Variables no correlacionadas**

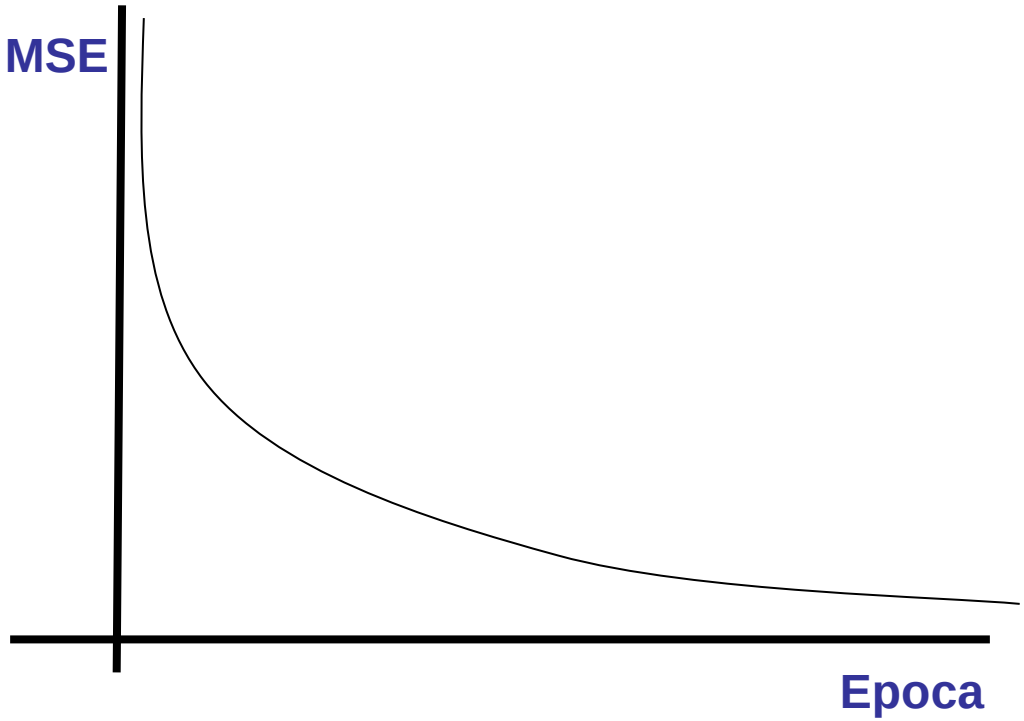
1er conjunto de entrenamiento $\left(x_j, f\left(x_j\right)\right)$





2do conjunto de entrenamiento $\left(x_j, f\left(x_j\right)\right)$

La gráfica a monitorear por excelencia es la de la progresión de los errores con respecto a las épocas.

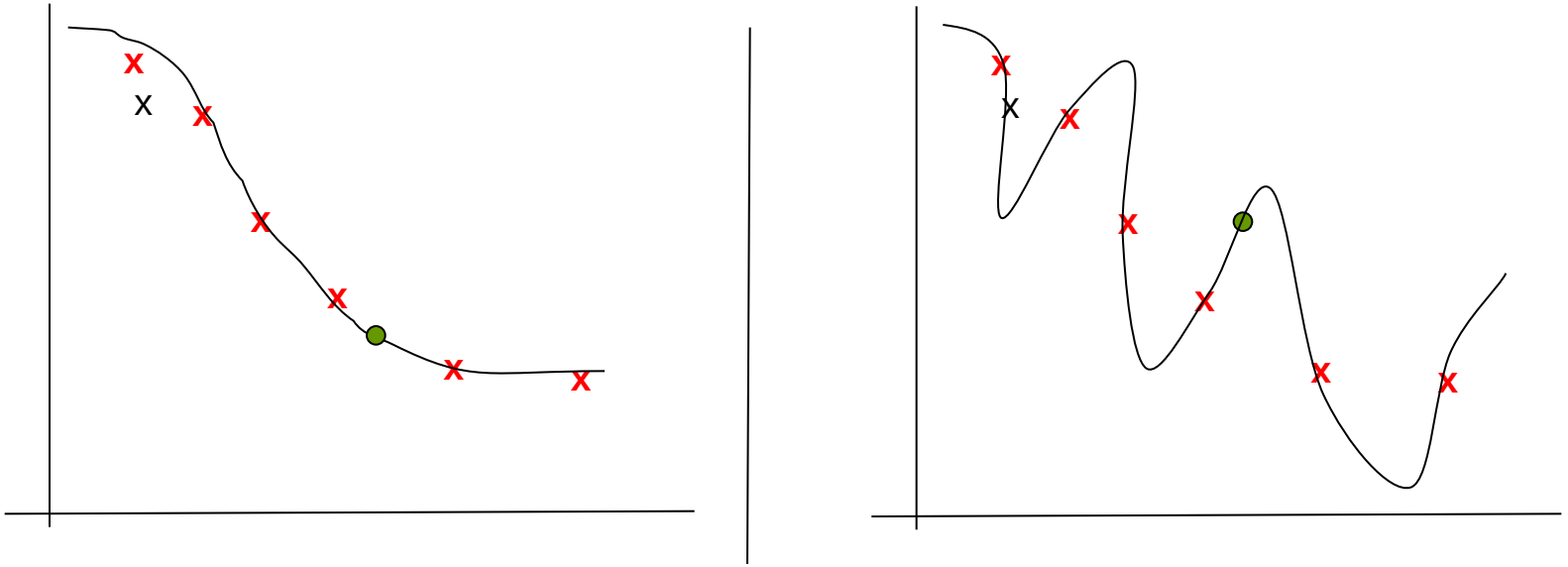


El error por si sólo no es una medida suficiente para garantizar el aprendizaje de la red !

- **Validación**
- **Cumplimiento de objetivos**
- **Generalización**

Generalización

Se dice que una red generaliza si logra clasificar/extrapolar/interpolar correctamente patrones de prueba (test data) que han sido generados usando los mismos mecanismo que el conjunto de datos usados para entrenar.



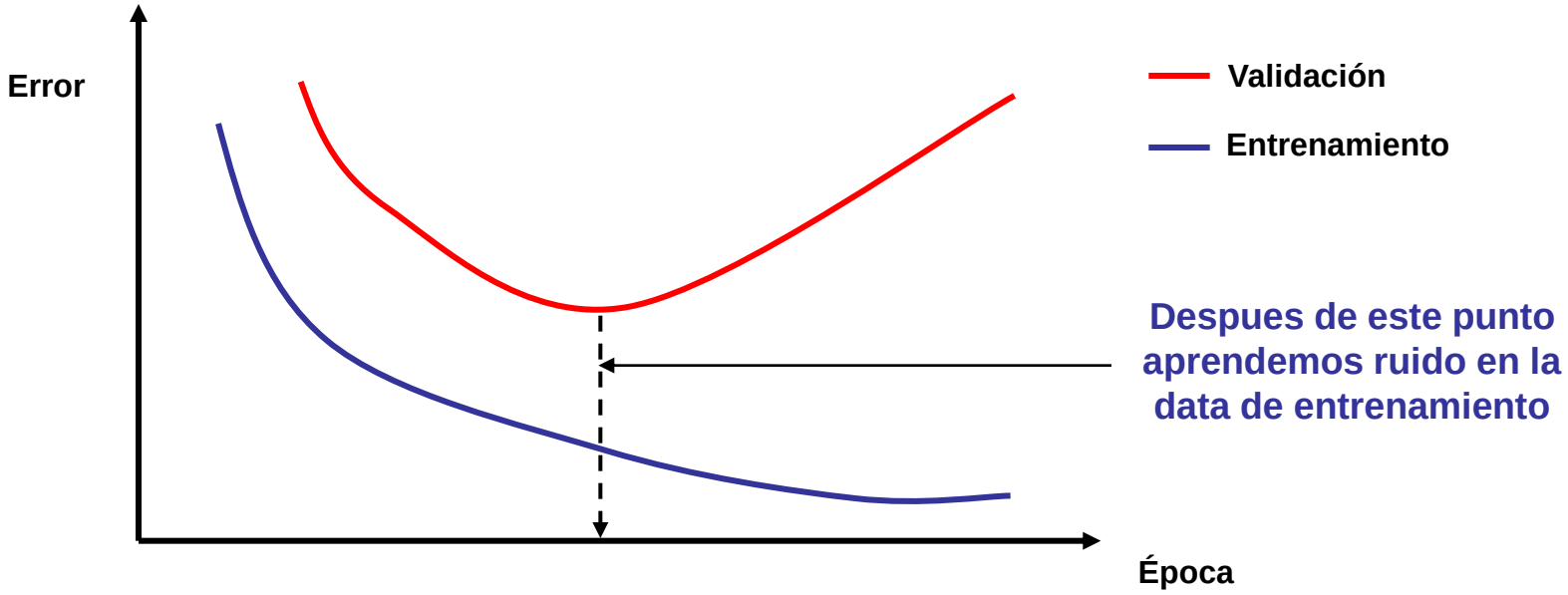
Un polinomio con pocos grados de libertad puede no interpolar bien la data, pero muchos grados de libertad pueden distorcionar la verdadera relación!

Sobre-entrenamiento (*Overtraining ,overfitting*)

Corresponde a la memorización del conjunto de entrenamiento y por lo tanto la pérdida de capacidad de generalización de la red neuronal.

Se puede obtener cuando se consideran muchas épocas o cuando la configuración de la red es muy compleja (muchas capas ocultas) para el problema que se quiere resolver.

Se puede detectar utilizando la curva de entrenamiento y validación.



Ciertamente el aprendizaje de la red depende no sólo de la calidad del conjunto de entrenamiento si no también de la cantidad de ejemplos con que cuenta.

Una regla empírica generalmente utilizada en la práctica es escoger el tamaño N del conjunto de entrenamiento utilizando la fórmula:

$$N \approx O\left(\frac{W}{\varepsilon}\right)$$

W **: # Total de parámetros de la red**

ε **: Tolerancia para el porcentaje de error en la validación**

Funciona siempre? Depende ....

N- **tamaño de conjunto de entrenamiento**

W- **número de parametros libres de la red (pesos sinápticos).**

Caso 1: N **< 30** W**, la practica indica que puede haber sobreentrenamiento y se justifica la parada prematura.**

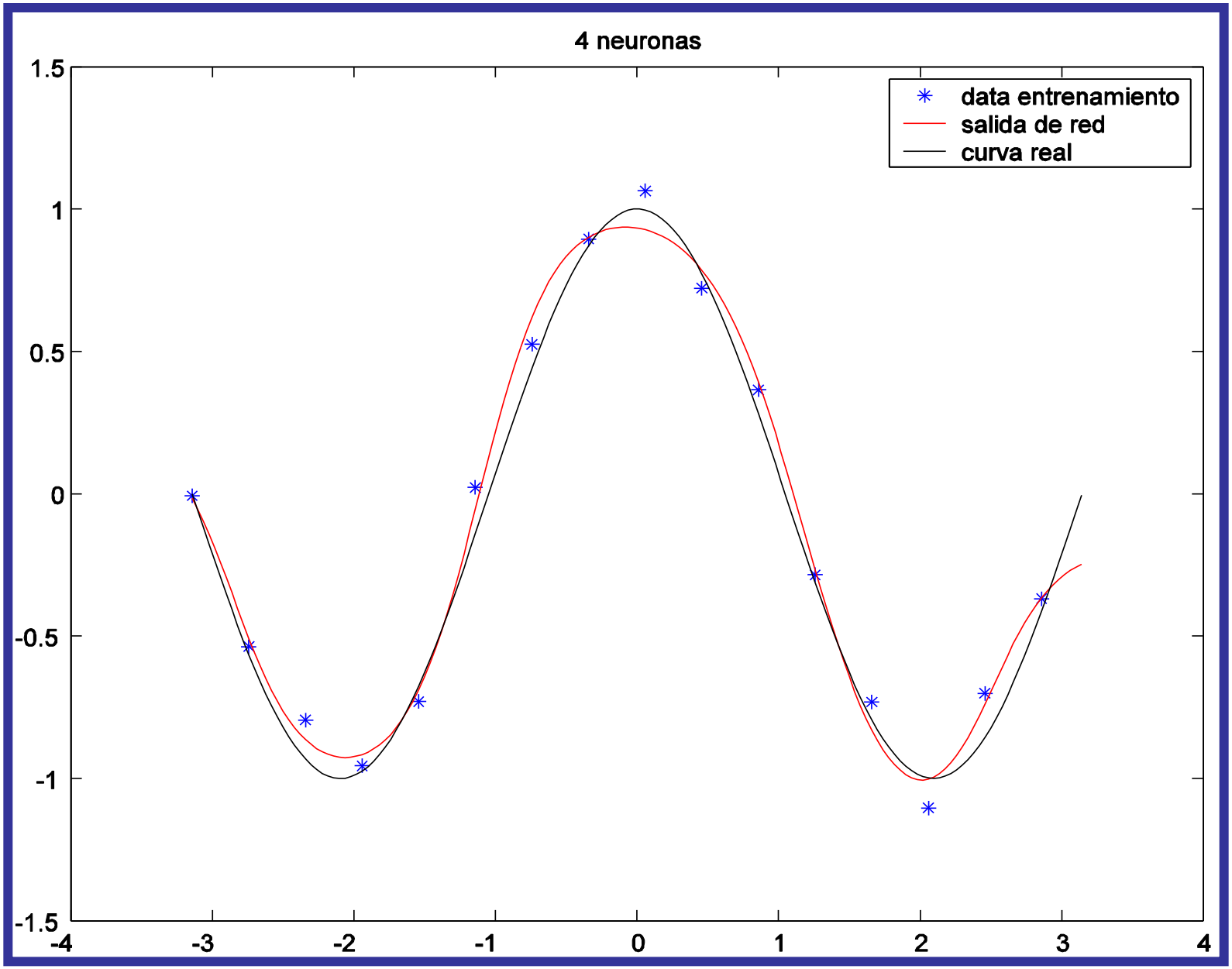
$$R = 1 - [(2W-1)^{1/2} - 1] / 2(w-1),$$ Amari (1996)

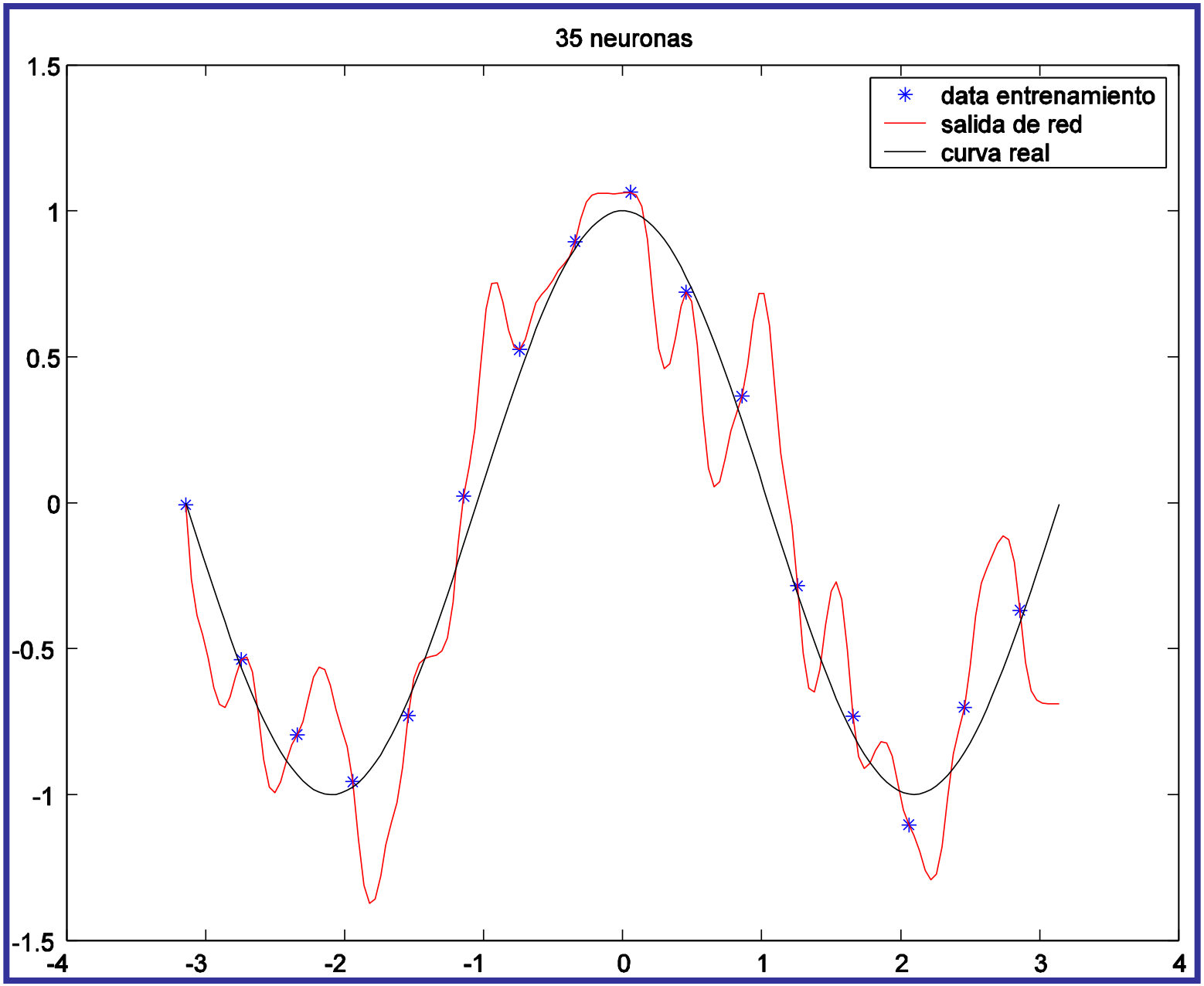
es el porcentaje de datos que se deben disponer para validar

Caso 2: N **> 30** W **la generalización no mejora con paradas prematuras.**

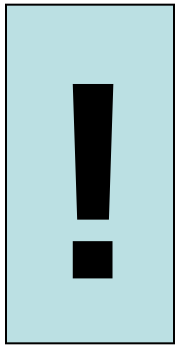
- **Las fallas en la generalización se pueden deber a:**
  - **Pobre calidad de los datos (entra basurita, sale basurita)**
  - **Sobre-entrenamiento (caletre)**
  - **Muchos parámetros libres (relación deficiente entre cantidad de datos y número de pesos)**

Un a red está bien entrenada si cuenta con buenas capacidades de generalización. Aqui aplica el viejo adagio chino "más no siempre es mejor"


4 neuronas
data entrenamiento
salida de red
curva real
1.5
1
0.5
0
-0.5
-1
-1.5
-4
-3
-2
-1
0
1
2
3
4


35 neuronas
data entrenamiento
salida de red
curva real
1.5
1
0.5
0
-0.5
-1
-1.5
-4
-3
-2
-1
0
1
2
3
4

Mejoras en el desempeño



Las mejoras en el desempeño pueden venir por muchas lados:

- **Algoritmo.**
  - **Mejoras en la convergencia con modificaciones en la iteración (momento) .**
  - **Entonación de los parámetros**
  - **Elección del algoritmo de optimización apropiado.**

- **Elección de la arquitectura:**
  - **Funciones de activación**
  - **Network prunning o network growing**
  - **Comite de expertos**
  - **boosting**

- **Los datos:**
  - **Transformación de datos.**
  - **Elección de *"features"* o patrones.**
  - **Presentación de los datos.**

El término de momento (momentum term)

Los pesos se actualizan utilizando la fórmula:

$$w_{ji}(n+1) = w_{ji}(n) + \Delta w_{ji}(n)$$
$$= w_{ji}(n) + \eta\, \delta_j(n) y_i(n)$$

η pequeño produce cambios pequeños y por lo tanto la convergencia del algoritmo es lenta. En consecuencia, el entrenamiento es lento

η grande produce cambios grandes y el algoritmo no converge. En consecuencia, no se puede entrenar a la red.

Un método simple para mejorar la convergencia del algoritmo (y por lo tanto la rapidez del entrenamiento) es modificar la fórmula anterior considerando:

$$\Delta w_{ji}(n) = \underbrace{\alpha\, \Delta w_{ji}(n-1)}_{\text{Término de momento}} + \eta\, \delta_j(n)\, y_i(n)$$

La idea es considerar la información del gradiente obtenida en la *(n-1)*-iteración. Esto permite:

- **Acelerar la convergencia del algoritmo**
- **Evitar cambios bruscos que produzcan la divergencia del algoritmo**
- **Evitar mínimos locales**

Es fácil observar que después de n-iteraciones:

$$\Delta w_{ji}(n) = \eta \sum_{t=0}^{n} \alpha^{n-t} \delta_j(t) y_i(t)$$

$$= -\eta \sum_{t=0}^{n} \alpha^{n-t} \frac{\partial \varepsilon(t)}{\partial w_{ji}(t)}$$

Esta expresión representa una serie de tiempo ponderada exponencialmente, con lo cual, para que la suma sea convergente se debe cumplir $|\alpha| < 1$

A continuación se estudian las consecuencias de esta importante fórmula

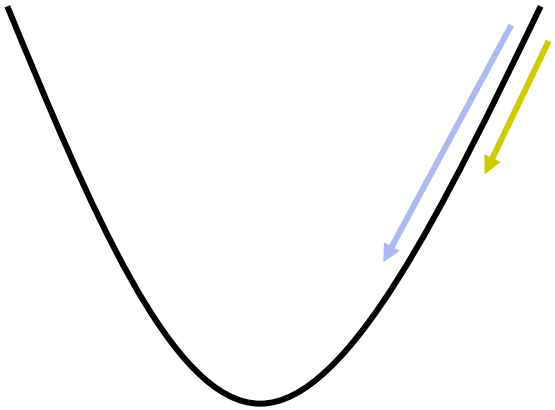
• **Si la derivada direccional tiene siempre el mismo signo entonces la corrección del peso sináptico es mayor. Esto permite acelerar el descenso de manera estable en la dirección del mínimo.**

$$\Delta w_{ji}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)}$$

Corrección original

$$\Delta w_{ji}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} - \eta \sum_{t=0}^{n-1} \alpha^{n-t} \frac{\partial \varepsilon(t)}{\partial w_{ji}(t)}$$

Corrección con término de momento

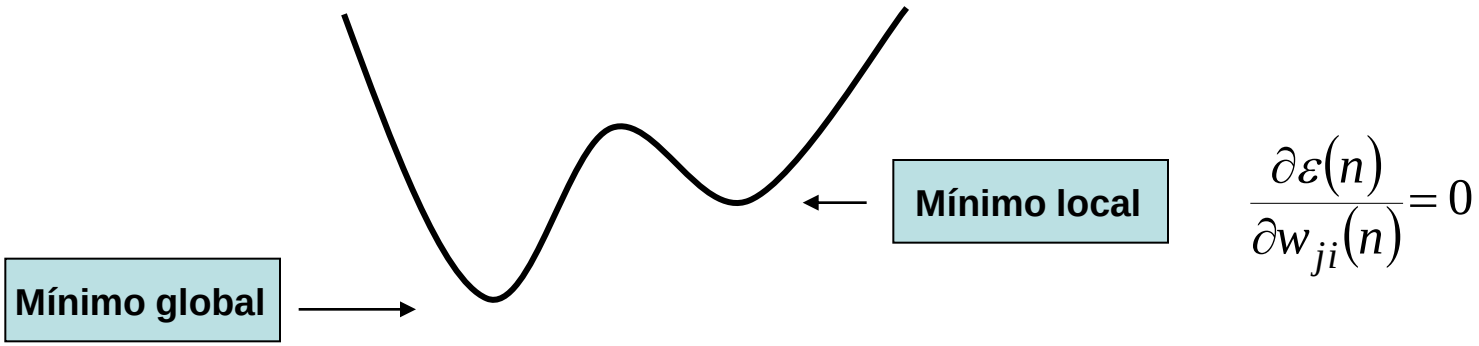


• **Si la derivada direccional cambia de signo en iteraciones consecutivas entonces el factor de corrección es pequeño.**

De esta forma, el uso del término de momento tiene un efecto estabilizador en la convergencia del algoritmo al evitar cambios bruscos no convenientes

• **El término de momento contribuye a evitar mínimos locales de la función de costo, ya que**

$$\Delta w_{ji}(n) = -\eta \frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} - \eta \sum_{t=0}^{n-1} \alpha^{n-t} \frac{\partial \varepsilon(t)}{\partial w_{ji}(t)}$$



$$\frac{\partial \varepsilon(n)}{\partial w_{ji}(n)} = 0$$

Acerca de la función de activación

Una de las funciones de activación más utilizadas en el algoritmo de *backpropagation* es la función logística:

$$\varphi(x) = \frac{1}{1+\exp(-\alpha x)} \qquad x \in R,\ \alpha > 0$$

ya que esta verifica $\varphi'(x) = \alpha\varphi(x)[1-\varphi(x)]$ **lo cual permite escribir las ecuaciones de propagación del error como:**

$$\delta_j(n) = \alpha Y_j(n)[1-Y_j(n)]\,[d_j(n)-Y_j(n)]$$ **Capa salida**

$$\delta_j(n) = \alpha Y_j(n)[1-Y_j(n)] \sum_{\alpha} \delta_\alpha(n) w_{\alpha j}(n)$$ **Capa oculta**

En general, al usar el algoritmo de *backpropagation* es posible lograr entrenamientos más rápidos considerando una función impar como función de activación. Esto es, que verifique:

$$\phi(-x) = -\phi(x)$$

La función tangente hiperbólica $\varphi(x) = a\tanh(bx)$ **verifica esta propiedad y las ecuaciones de propagación del error se pueden escribir como:**

$$\delta_j(n) = \frac{b}{a}[a - Y_j(n)][a + Y_j(n)][d_j(n) - Y_j(n)]$$ **Capa salida**

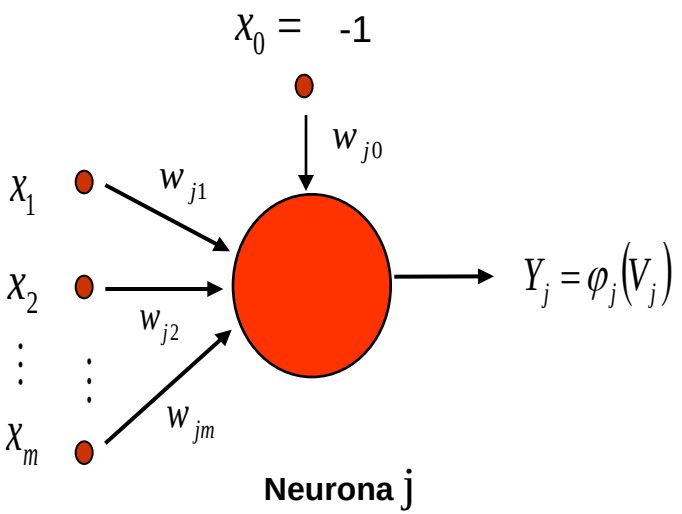
$$\delta_j(n) = \frac{b}{a}[a - Y_j(n)][a + Y_j(n)]\sum_{\alpha}\delta_{\alpha}(n)w_{\alpha j}(n)$$ **Capa oculta**

$a = 1.7159, b = 2/3$ **LeCun,Y. *Generalizations and network design strategies*. 1989**

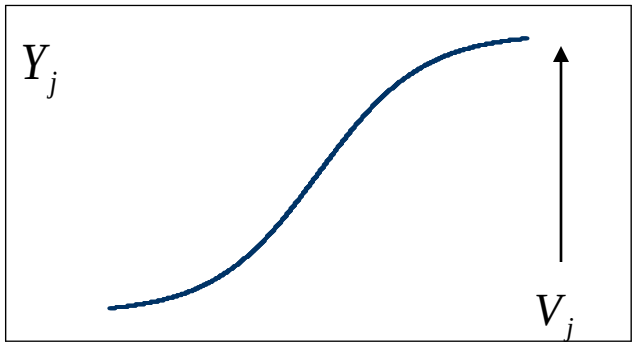
Normalización de los valores de entrada

En general las componentes del estímulo presentado a la red se encuentran en unidades distintas o son valores muy disímiles.

Esto puede lograr que las neuronas de la primera capa oculta se saturen y que el entrenamiento sea lento.



$$\Delta w_{ji}(n) = \eta_{ji}\ \delta_j(n) x_i(n)$$

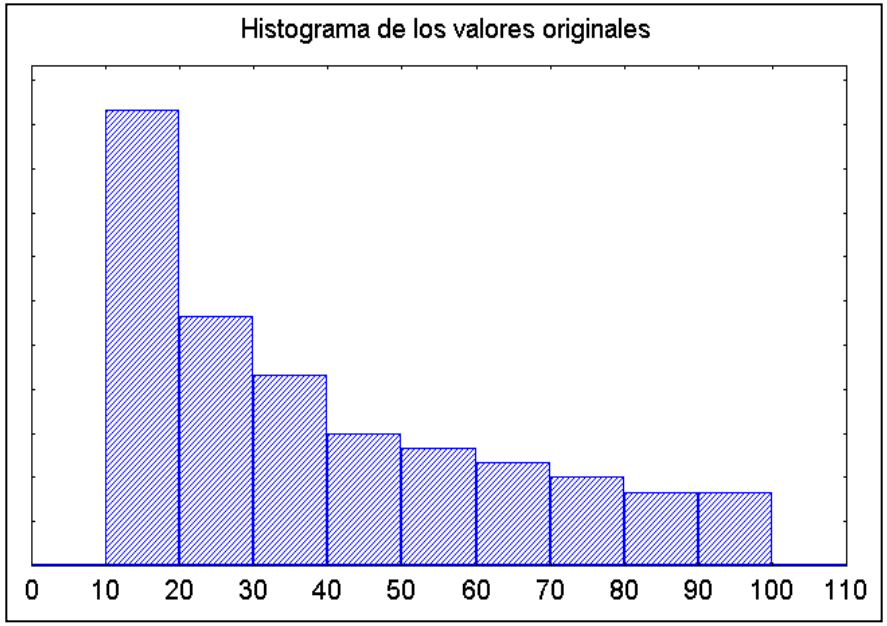


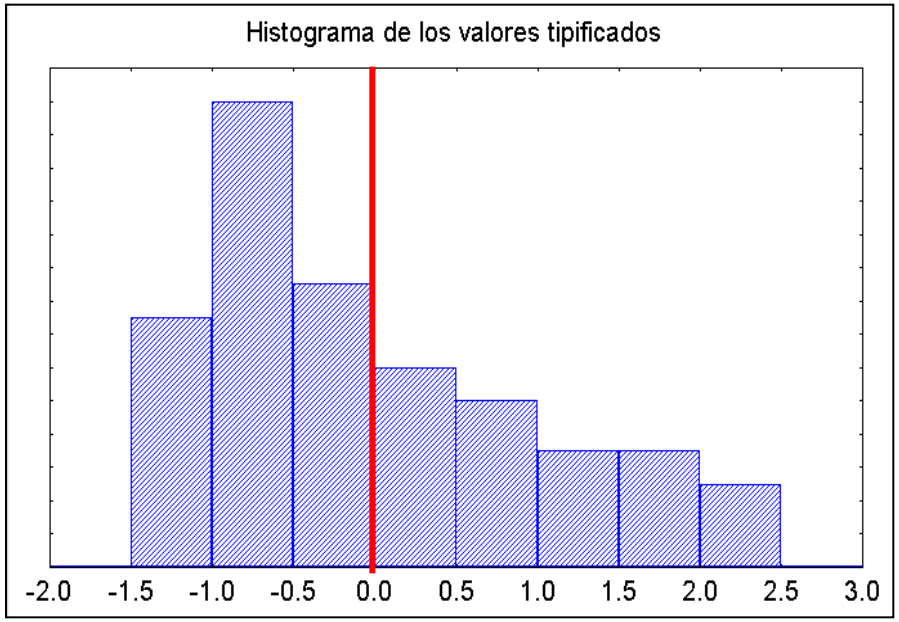
$$x_1, x_2, \ldots, x_{m-1} \in [0,1] \quad x_m \in [1E^6, 1E^{10}]$$

Un posible procedimiento es tipificar las variables de entrada para que sean de media cero y varianza 1

$$Z_j = \frac{X_j - \mu_j}{\sigma_j}$$

El nuevo conjunto de datos es adimensional y tiene distribución aproximadamente simétrica alrededor del valor cero.

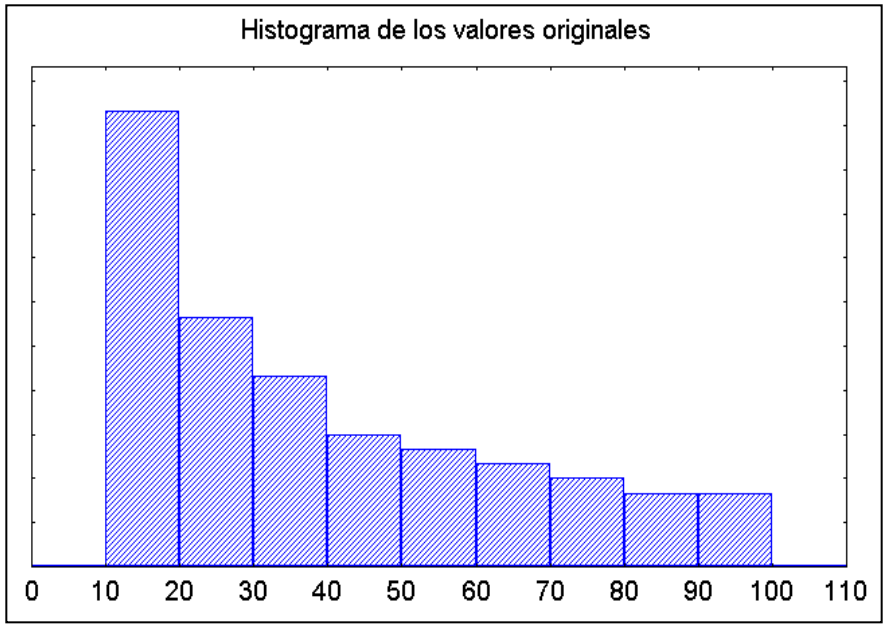


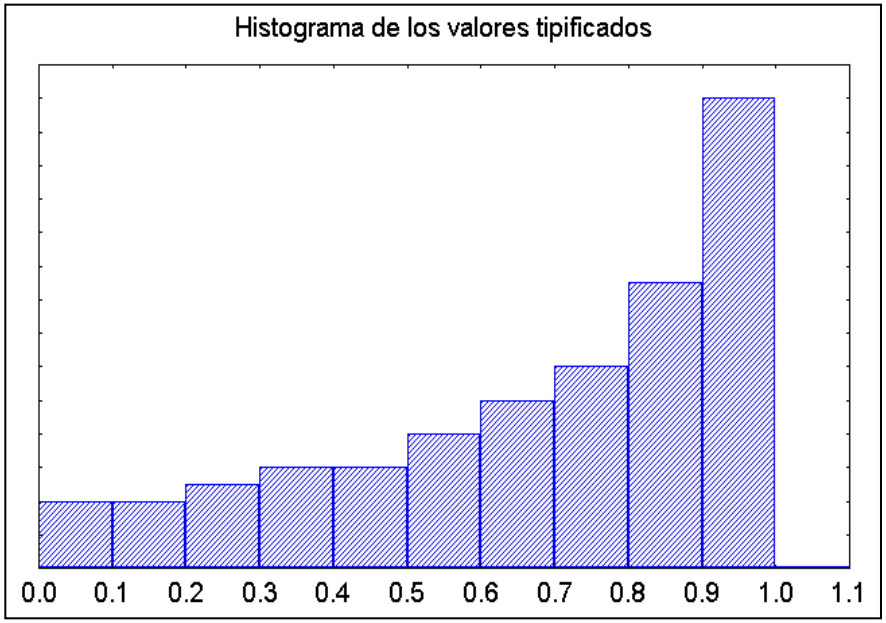


Otro posible procedimiento es considerar los valores máximos y mínimos de cada una de las variables y definir las variables tipificadas como:

$$Z_j = \frac{X_j^{max} - X_j}{X_j^{max} - X_j^{min}}$$

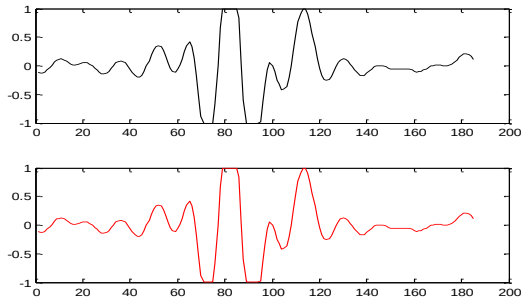
El nuevo conjunto de datos es adimensional y todos los los valores pertenecen al intervalo [0, 1].

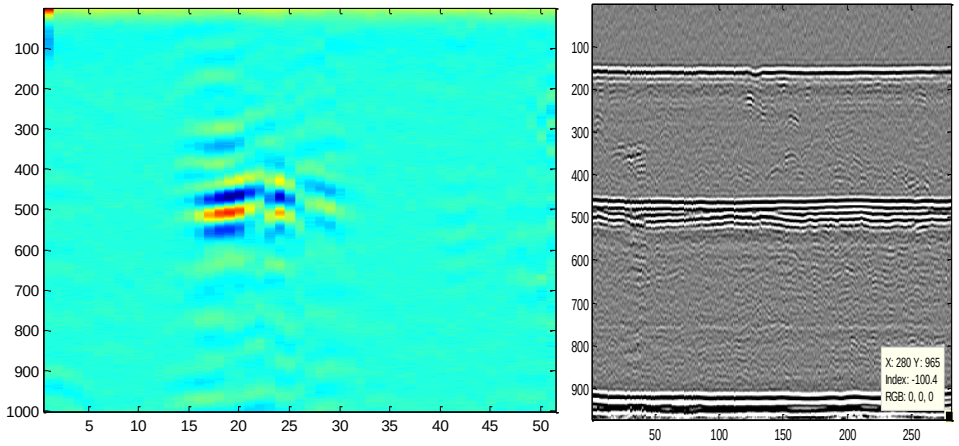




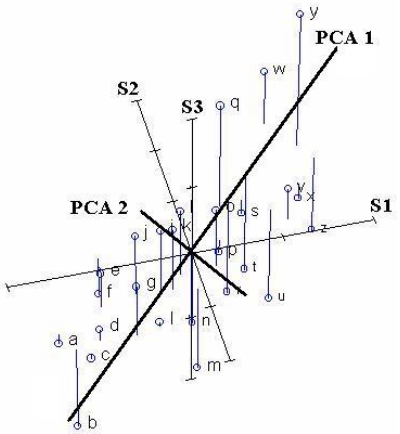
La extracción de data que sea relevante también puede requerir transformaciones de los datos:

- **Transformación de ondículas (Ejm: datos de reconocimiento de grietas, escorias, poros y falta fusión)**

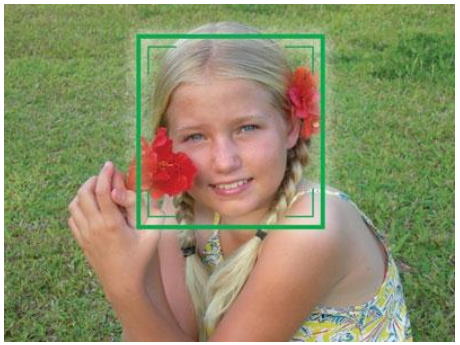




- **Componentes principales. Rotar la data para que la máxima variabilidad sea visible.**



- **Una matriz NxN de datos puede representarse como un vector** $x = (x_1, x_2, \cdots, x_{N^2})$



- **Suponga que se tienen 20 imágenes así y se quiere que cuando llegue una nueva imagen reconocer de quién es la foto.**
- **Se crea una matríz de imágenes** $Imagen = \begin{pmatrix} Im1 \\ \vdots \\ Im20 \end{pmatrix}$ **con ACP podemos mirar la distancia que hay entre una nueva y una almacenada. Proyectamos la data en un nuevo espacio y usamos esta data para el reconocimiento**

Rango de los valores de salida

Corresponde al rango de las funciones de activación de las neuronas en la capa de salida.

Cuando se utiliza la función logística o tangente hiperbólica se debe reestablecer el rango original de las variables devolviendo el cambio realizado mediante la tipificación.

También se pueden considerar funciones lineales en la capa de salida y se evita la tipificación y devolución del cambio de los valores utilizados para supervisar la red.

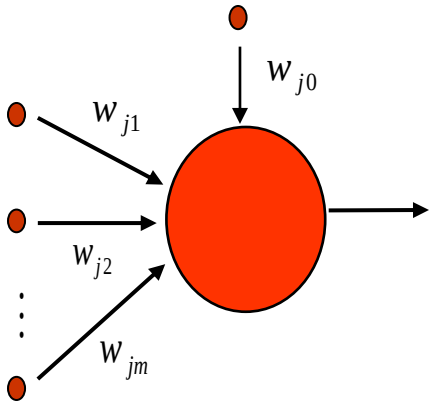
Se debe tener cuidado cuando los valores utilizados para supervisar a la red están dentro del rango de las funciones de activación pero son valores extremos, ya que las neuronas podrían saturarse.

Inicialización de los pesos sinápticos

Se debe evitar inicializar los pesos sinápticos con valores grandes ya que esto puede causar la saturación de las neuronas. Cuando esto ocurre los gradientes locales tiene valores pequeños y el proceso de entrenamiento es lento

Si la inicialización se realiza con valores pequeños, el algoritmo de *backpropagation* comienza a trabajar en una "región plana" de la función de costo o error, lo cual también produce retraso en el entrenamiento.

Una manera usual de inicializar los pesos asociados a una neurona es escogiéndolos como valores aleatorios de una distribución uniforme de media cero y varianza recíproca al número de conexiones de ésta.



$$w_{ji}(0) \approx U\left(0, \frac{1}{\sqrt{m}}\right)$$

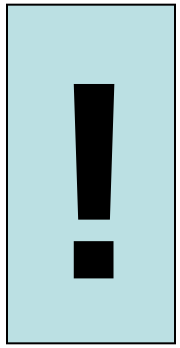
Actualización secuencial vs. por lotes

La actualización secuencial es computacionalmente más rápida, mas cuando la data es grande y redundante (El cálculo del Hesiano es computacionalmente complicado en este caso)

Reordenamiento de patrones

Esto permite que sea poco probable que patrones sucesivos pertenezcan a una misma clase.

OTRAS CONSIDERACIONES



Existen diversas estrategias para mejorar la generalización:

1) **Teoría de Regularización: Agregar una penalidad para lograr mejorar la forma de la función.**

2) **Método de parada prematura: Detener el algortimo de entrenamiento prematuramente (vimos algo de esto en la clase pasada) y algunas pautas sugeridas.**

3) **Network Prunning y Network Growing: Consiste en determinar las mejores arquitecturas de la red.**

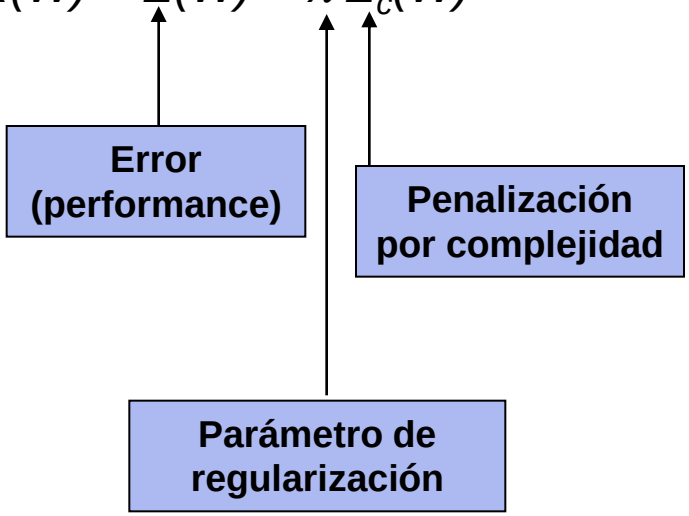
4) **Comité de redes: Formar un panel de expertos.**

5) **Validación cruzada: Escoger el mejor modelo usando la mejor data de entrenamiento posible.**

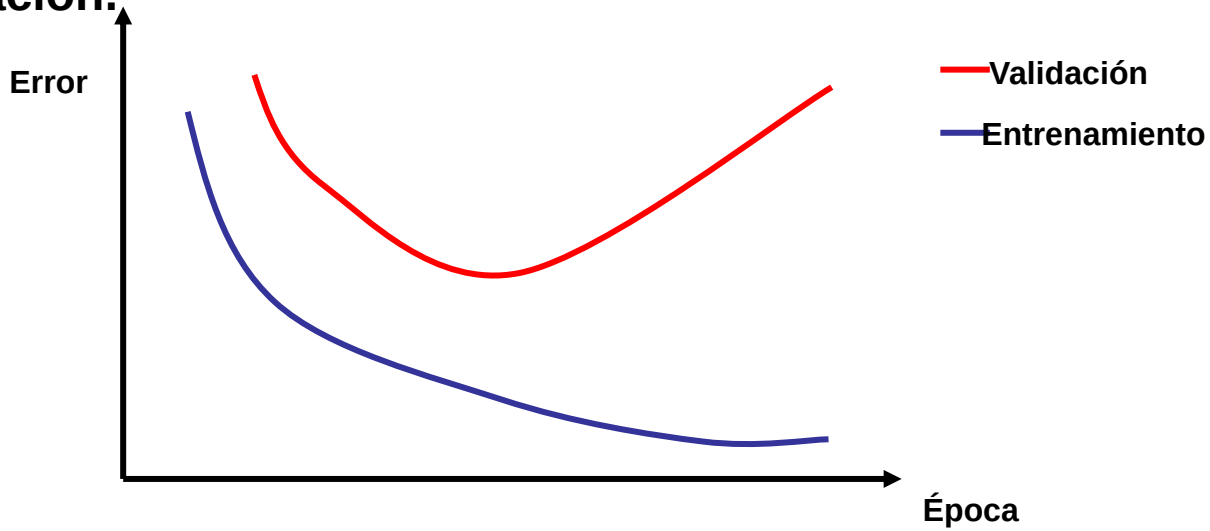
La idea es minimizar una función que no sea solamente función de los errores cometidos por la red, sino que tambien, penalice otras propiedades de la función.

En un problema de interpolación:

$$R(W) = E(W) + \lambda E_c(W)$$



Se puede detectar utilizando la curva de entrenamiento y validación.



Despues de este punto aprendemos ruido en la data de entrenamiento

N**- tamaño de conjunto de entrenamiento**

W**- número de parametros libres de la red (pesos sinápticos).**

Caso 1: N **< 30** W**, la practica indica que puede haber sobreentrenamiento y se justifica la parada prematura.**

Caso 2: N **> 30** W **la generalización no mejora con paradas prematuras.**

En general queremos la red mas simple que haga un buen trabajo.

Buen trabajo: **Minimice el error de validación/entrenamiento.**

Red simple: **Aquella con menor cantidad de conexiones**

Network Prunning

Consiste en empezar con una red compleja que capte la relación existente en la data y simplificarla sin perjudicar el "performance".

La mayoria de las técnicas se basan en la teoría de regularización, en donde penalizamos la complejidad del modelo, donde entendemos por complejidad un número grande de neuronas.

$$R(W) = E(W) + \lambda E_c(W)$$

- $E(W)$: **Error (performance)**
- $E_c(W)$: **Penalización por complejidad**
- λ: **Parámetro de regularización**

Eliminación de pesos:

1) $$E_c(w) = \frac{1}{2}\|w\|^2 = \frac{1}{2}\sum w_i^2$$

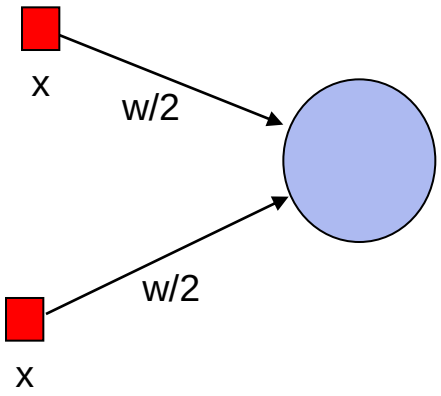
Esto obliga a que algunos pesos esten cercanos a cero ya que favorece pesos pequeños (aquellos con poca influencia sobre la red).

Estos son los pesos que mas probablemente tiendan a ajustar el ruido de la data y afectan la generalización.

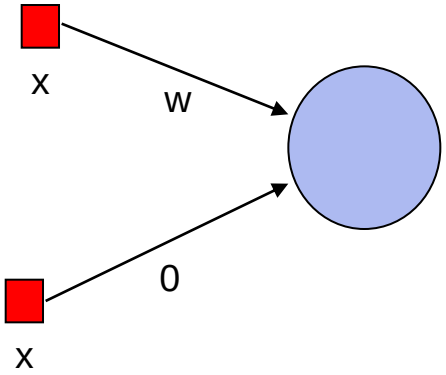
Entrenamos la red usando esta función de error, en este caso la actualización de los pesos cambia (cómo?)

Eliminamos aquellos pesos o conexiones sinápticas cuya importancia relativa sea pequeña.

La alternativa anterior tiende a favorecer muchos pesos pequeños sobre pocos grandes.



$E_c = w^2/2$

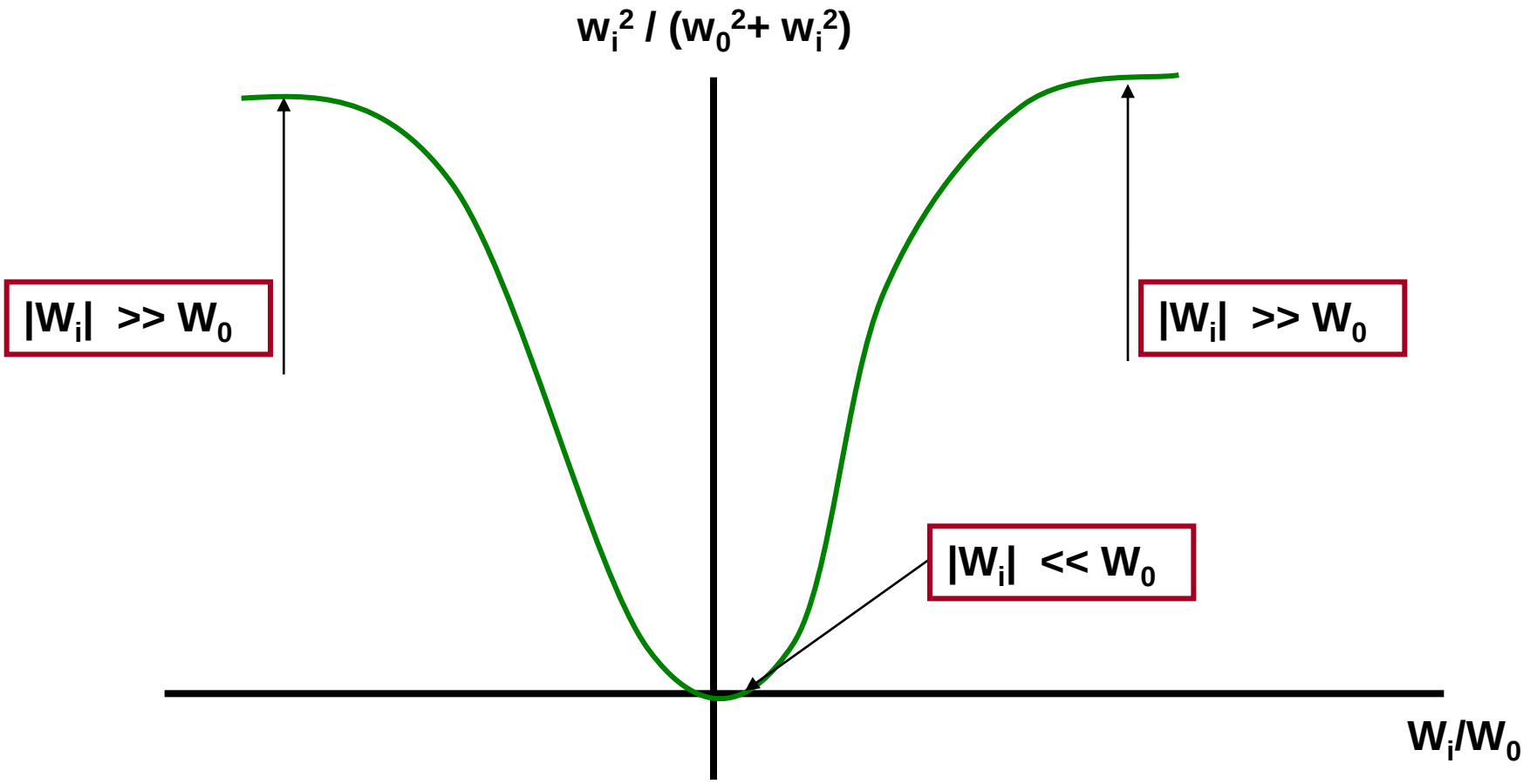


$E_c = w^2$

El error sin penalización es el mismo en ambos casos

Una alternativa es:

$$E_c = \sum \frac{w_i^2}{w_0^2 + w_i^2}$$



3) Weight saliency:

Aqui la idea es asignarle un índice a cada peso y determinar la salida del peso según este índice.

El indice se calcula basado en la minimización del error, en otras palabras buscamos un peso cuya salida no afecte el error cuadratico medio.

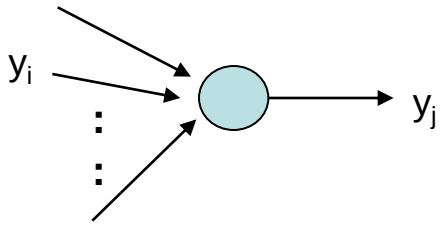
En la próxima tarea les pido que lean material sobre este método.

Node Prunning

En vez de eliminar conecciones sinápticas, eliminamos unidades de procesamiento completas (y sus respectivos pesos)

$$S_j = E(\text{sin nodo } j) - E(\text{con nodo } j)$$

Involucra pasar toda la data para determinar la conveniencia del nodo, (pero es mejor que hacerlo para cada peso!!). Una forma de aproximar si $\varphi_j(0)=0$



$$Y_j = \varphi_j(\alpha_j \Sigma w_{ji} y_i)$$

$$S_j = E(\alpha_j = 0) - E(\alpha_j = 1)$$

Validación Cruzada

Hold-out method: Particionar la data en K subconjuntos (K>1), entrenamos la red sobre todos los subconjuntos menos uno que usamos para validar. Calculamos el error de validación.

El error del "modelo" resulta de promediar los errores de validación cuando repetimos el procedimiento anterior a todos los subconjuntos.

Podemos quedarnos con el modelo que tenga menor error de validación.

Si K=1 (caso extremo), el método se conoce como el leave-one-out method

Otra forma de mejorar la generalización es no depender de una sola red, sino formar un comité.

Bajo este esquema, la respuesta es una promediación entre los miembros del comité (las redes). El error del comité es el error cuadrático medio de la respuesta del comité y la deseada, que es menor al error promedio de la redes actuando en solitario.

Sea y_i un modelo (experto) con error promedio E_i

$$E_{prom} = 1/L \sum E_i$$

$$Y_{com}(x) = 1/L \sum Y_i(x)$$

$$E_{com} \leq E_{prom}$$

El proceso anterior es el más sencillo de todos las formas de establecer un comité. Si hay expertos en los que confiamos más que en otros podemos condiderar una suma convexa de donde no todos los pesos sean iguales (**comité generalizado**).

Cómo elegir esos pesos?

Hay heurísticas que se pueden seguir (Bishop, cap 9.)

Boosting. **Entrenamos 3 expertos (E1, E2, E3) para que cada uno especialice su entrenamiento en data cada vez más dificil.**

1) Entrenamos a E1 usando N_1 muestras.

2) Entrenamos a E2 siguiendo los siguientes pasos:

lanzar una moneda.
Si el resultado es cara,

Pasar nuevos patrones por el primer experto y descartar los patrones correctamente clasificados, hasta que un patrón esté mal clasificado . Incorporar este patrón al conjunto de entrenamiento de E2 (N2).

sino,

Pasar nuevos patrones por el primer experto y descartar los patrones incorrectamente clasificados, hasta que un patrón esté bien clasificado. Incorporar este patrón al conjunto de entrenamiento de E2 (N2).

Entrenar a E2 con un total de N1 muestras

3) Entrenamos a E3 siguiendo los siguientes pasos para obtener el conjunto de entrenamiento:

Pasar un nuevo patrón por E1 y E2. Si ambos coinciden, desechar el patrón, sino incorporarlo al conjunto de entrenamiento de E3, hasta lograr N1 muestras.

Observación

Notar que el algoritmo de actualización de los pesos es en escencia un algortimo de optimización no-lineal aplicado al error cuadrático medio. Esta optimización puede hacerse con otros algoritmos distintos al algoritmo de descenso de gradiente (Newton, cuasi-Newton, gradiente conjugado, etc.). En Matlab estas variantes estan implementadas y muchas veces se obtienen mejores resultados.

Leer capitulo 7 (Bishop), sección 4.18 (Haykin), capítulo 5 (Manual Toolbox)

Resumen:

1) **Derivamos el backprop (versión secuencial).**
2) **Curvas de aprendizaje.**
3) **Heurísticas para mejorar el algoritmo (momemtum, funciones de activación, etc.)**
4) **Data de entrenamiento y preprocesamiento.**
5) **Generalización.**
    1) **Parada prematura**
    2) **Validacion cruzada**
    3) **Network prunning (regularización).**