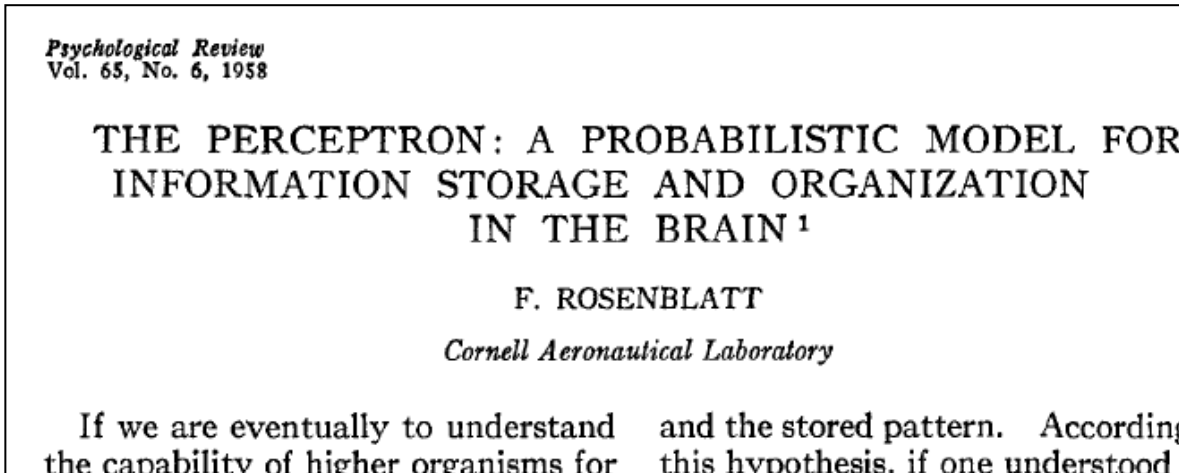
Redes Neuronales Artificiales

El perceptrón ocupa un lugar especial en el desarrollo histórico de las redes neuronales: fue la primera red neuronal descrita algorítmicamente. Su invento por Rosenblatt, un **psicólogo**, inspiró a ingenieros, físicos y matemáticos a dedicar sus esfuerzos de investigación a diferentes aspectos de las redes neuronales en los años sesenta y setenta. Además, es verdaderamente notable descubrir que el perceptrón es tan válido hoy como lo fue en 1958 cuando se publicó por primera vez el paper de Rosenblatt que aquí acompaño



Psychological Review
Vol. 65, No. 6, 1958

THE PERCEPTRON: A PROBABILISTIC MODEL FOR INFORMATION STORAGE AND ORGANIZATION IN THE BRAIN[1]

F. ROSENBLATT

Cornell Aeronautical Laboratory

If we are eventually to understand the capability of higher organisms for perceptual recognition, generalization, recall, and thinking, we must first have answers to three fundamental questions:

1. How is information about the physical world sensed, or detected, by the biological system?

and the stored pattern. Accordin
this hypothesis, if one understood
code or "wiring diagram" of the ne
ous system, one should, in princi
be able to discover exactly what
organism remembers by reconstru
ing the original sensory patterns fr
the "memory traces" which they h
left, much as we might develop
photographic negative, or transl

El **perceptrón** es la forma más simple de una **red neuronal** utilizada para la **clasificación** de **observaciones** que se dice que son linealmente separables (es decir, que se encuentran en lados opuestos de un hiperplano). El percetprón simple consiste en **una sola neurona** con **pesos** y **sesgos** sinápticos **ajustables**.

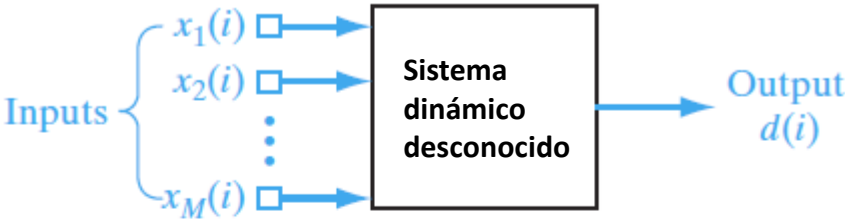
Algoritmo de Cuadrado Medio Mínimo - LMS (*Least Mean Square*)

Tenemos el datset *T:*

$$\mathcal{T}: \{\mathbf{x}(i), d(i); i = 1, 2, ..., n\}$$

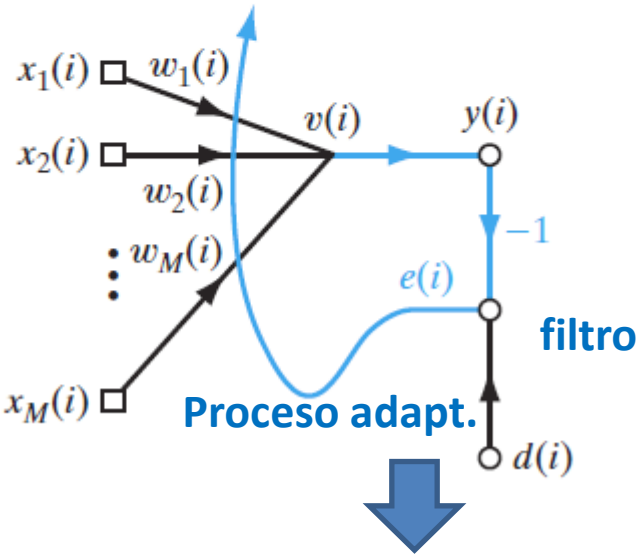
Donde la íesima observación tiene las siguientes características:

$$\mathbf{x}(i) = [x_1(i), x_2(i), ..., x_M(i)]^T$$

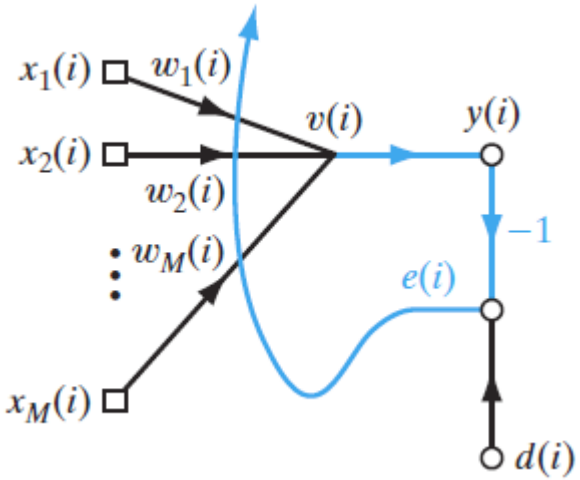


El modelo de RNA opera bajo un **algoritmo** que controla sistemáticamente los ajustes a los **pesos sinápticos** de la neurona:

- Determinación arbitraria de los pesos sinápticos de la neurona.
- Ajustes de los pesos sinápticos en respuesta a las variaciones estadísticas en el comportamiento del sistema realizadas de forma continua
- Cálculo de ajustes a los pesos sinápticos luego de la carga y procesamiento de las características de una observación *x(i)*



A este modelo se lo denomina **filtro adaptativo**



Como la neurona es **lineal**, la salida *y(i)* es:

$$y(i) = \sum_{k=1}^{M} w_k(i)x_k(i)$$

La salida de la neurona *y(i)* se compara con la salida correspondiente *d(i)* recibida del sistema desconocido en el momento *i* **siendo el error:** $e(i) = d(i) - y(i)$

La manera en que se usa la señal de error e (i) para controlar los ajustes a los pesos sinápticos de la neurona está determinada por la función de costo utilizada para derivar el algoritmo de filtrado adaptativo de interés. Este problema está estrechamente relacionado con el de la optimización.

Optimización No Restringida:: Gradiente Descendente

Consideremos una $\mathscr{E}(\mathbf{w})$ que sea una función continua y diferenciable de algún peso desconocido (parámetro) vector **w.** La función $\mathscr{E}(\mathbf{w})$ mapea los elementos de **w** a números reales (el costo). Queremos encontrar una solución óptima **w*** que minimice la funcion costo respecto a **w.** Por ende la función en ese valor de **w*** deberá cumplir: $\nabla\mathscr{E}(\mathbf{w}^*) = \mathbf{0}$

$$\nabla\mathscr{E}(\mathbf{w}) = \left[\frac{\partial\mathscr{E}}{\partial w_1}, \frac{\partial\mathscr{E}}{\partial w_2}, \ldots, \frac{\partial\mathscr{E}}{\partial w_M}\right]^T$$

Una clase de algoritmos de optimización no restringida que es particularmente adecuada para el diseño de filtros adaptativos se basa en la idea del descenso iterativo local, cuyos pasos son :
Determinamos los *w (0) y l*uego generamos una secuencia de vectores de peso w (1), w (2),. . . (siendo el número la iteración) de modo que la función de costo $\mathscr{E}(\mathbf{w})$ se reduzca en cada iteración del algoritmo, donde *w(n)* es el valor antiguo del vector de ponderación y *w(n+1)* es su valor actualizado.

$$\mathscr{E}(\mathbf{w}(n+1)) < \mathscr{E}(\mathbf{w}(n)) \qquad \textbf{(1)}$$

En el método del **Gradiente Descendente** , los ajustes sucesivos aplicados al vector de peso **w** son en la dirección del descenso de la función, es decir, en la dirección **opuesta** al vector de **gradiente**. Por ende la fórmula para actualizar los valores de **w** es: $\mathbf{w}(n+1) = \mathbf{w}(n) - \eta\nabla\mathscr{E}(\mathbf{w})$

Por ende la variación $\Delta\mathbf{w}(n) = \mathbf{w}(n+1) - \mathbf{w}(n) = -\eta\nabla\mathscr{E}(\mathbf{w})$ **(2)**

Pero esta ecuación ¿cumple la condición (1) $\mathscr{E}(\mathbf{w}(n+1)) < \mathscr{E}(\mathbf{w}(n))$? Para responderlo hagamos una expansión de Taylor alrededor de $\mathbf{w}(n)$ para aproximar $\mathscr{E}(\mathbf{w}(n+1))$

Llamemos al gradiente: $\mathbf{g} = \nabla\mathscr{E}(\mathbf{w})$

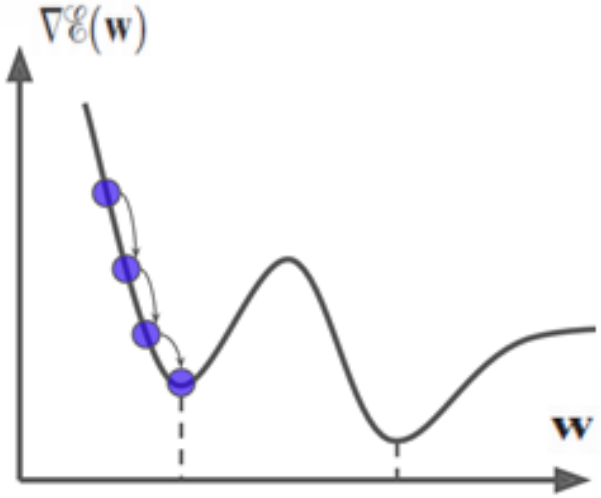
Hagamos la aproximación aplicando series de Taylor:

$$\mathscr{E}(\mathbf{w}(n+1)) \approx \mathscr{E}(\mathbf{w}(n)) + \mathbf{g}^T(n)\Delta\mathbf{w}(n) \quad \textbf{(3)}$$

Sustituyendo (2) en (3) tengo:

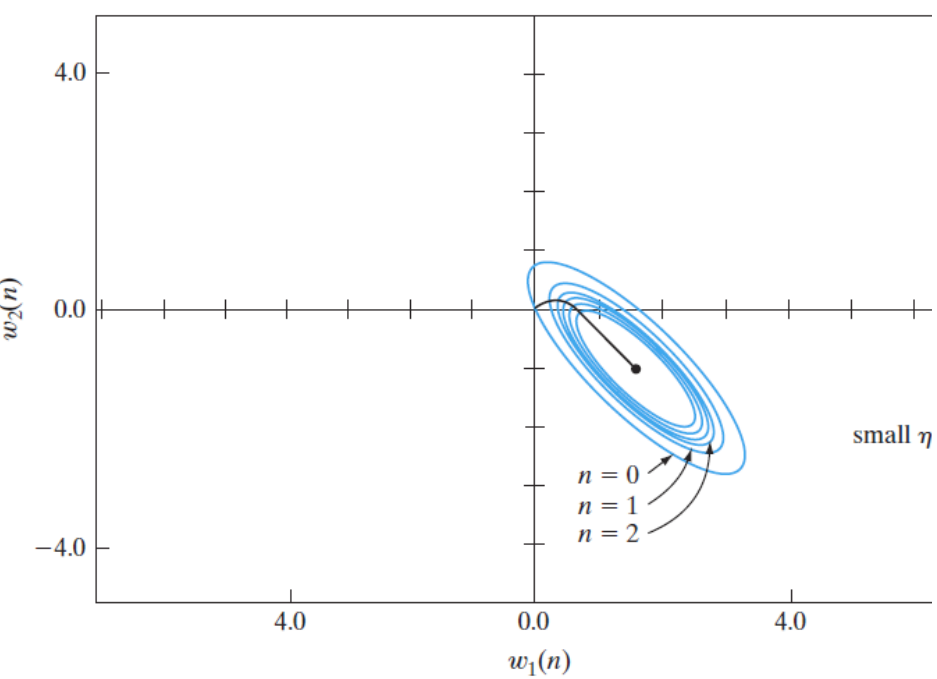
$$\begin{aligned}\mathscr{E}(\mathbf{w}(n+1)) &\approx \mathscr{E}(\mathbf{w}(n)) - \eta\mathbf{g}^T(n)\mathbf{g}(n) \\ &= \mathscr{E}(\mathbf{w}(n)) - \eta\|\mathbf{g}(n)\|^2\end{aligned}$$

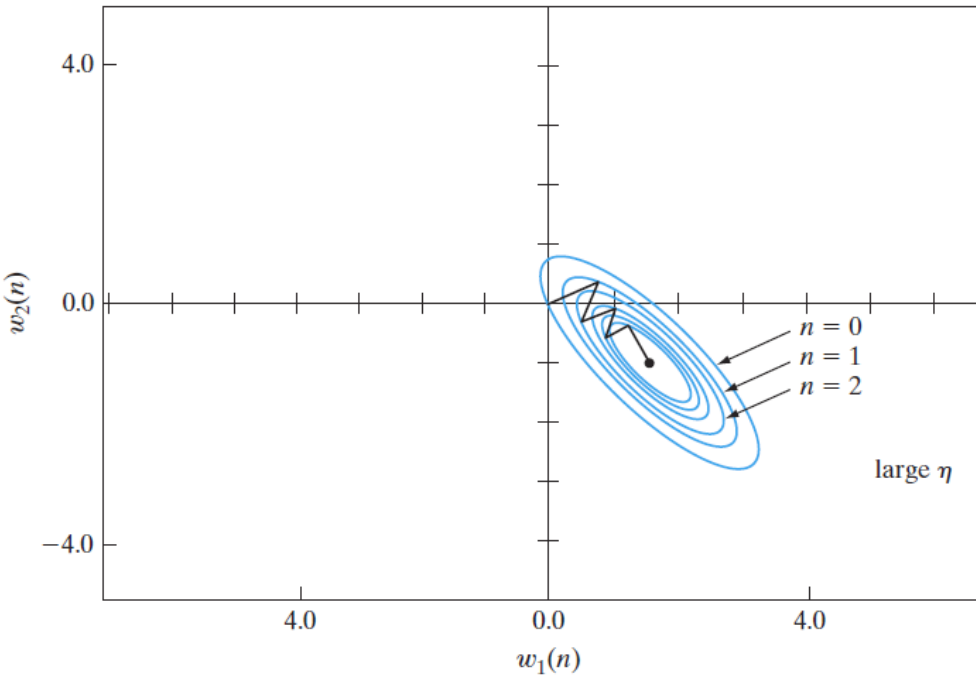
Que nos indica que para una tasa de aprendizaje positiva η la función de costo baja de valor de iteración en iteración.



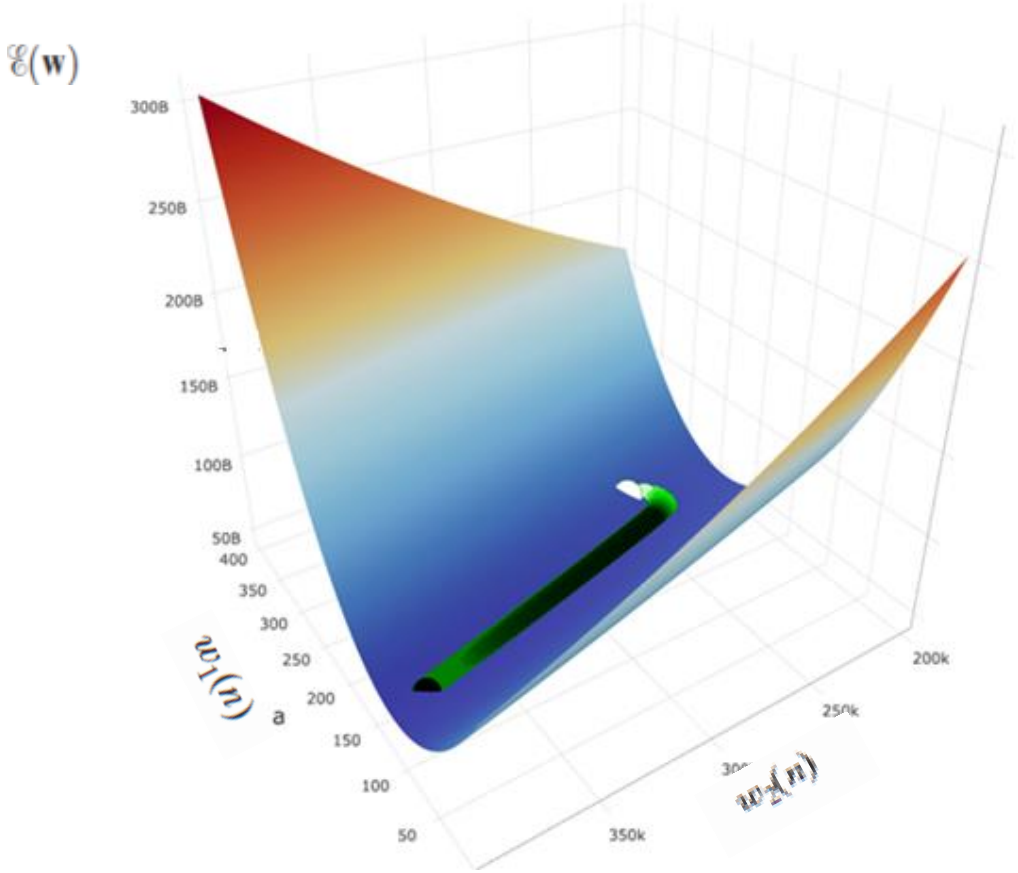
El método del Gradiente Descendente converge lentamente a la solución óptima ***w****. Además, la tasa de aprendizaje η tiene una profunda influencia en su comportamiento de convergencia:

- Cuando η es pequeño, la respuesta transitoria del algoritmo está muy controlada, ya que la trayectoria trazada por ***w(n)*** sigue una trayectoria suave en el plano w, como se ilustra en la figura de la izquierda
- Cuando η es grande, la respuesta transitoria del algoritmo no está controlada, ya que la la trayectoria de ***w(n)*** sigue una trayectoria zigzagueante (oscilatoria), como se ilustra en la figura de la derecha.

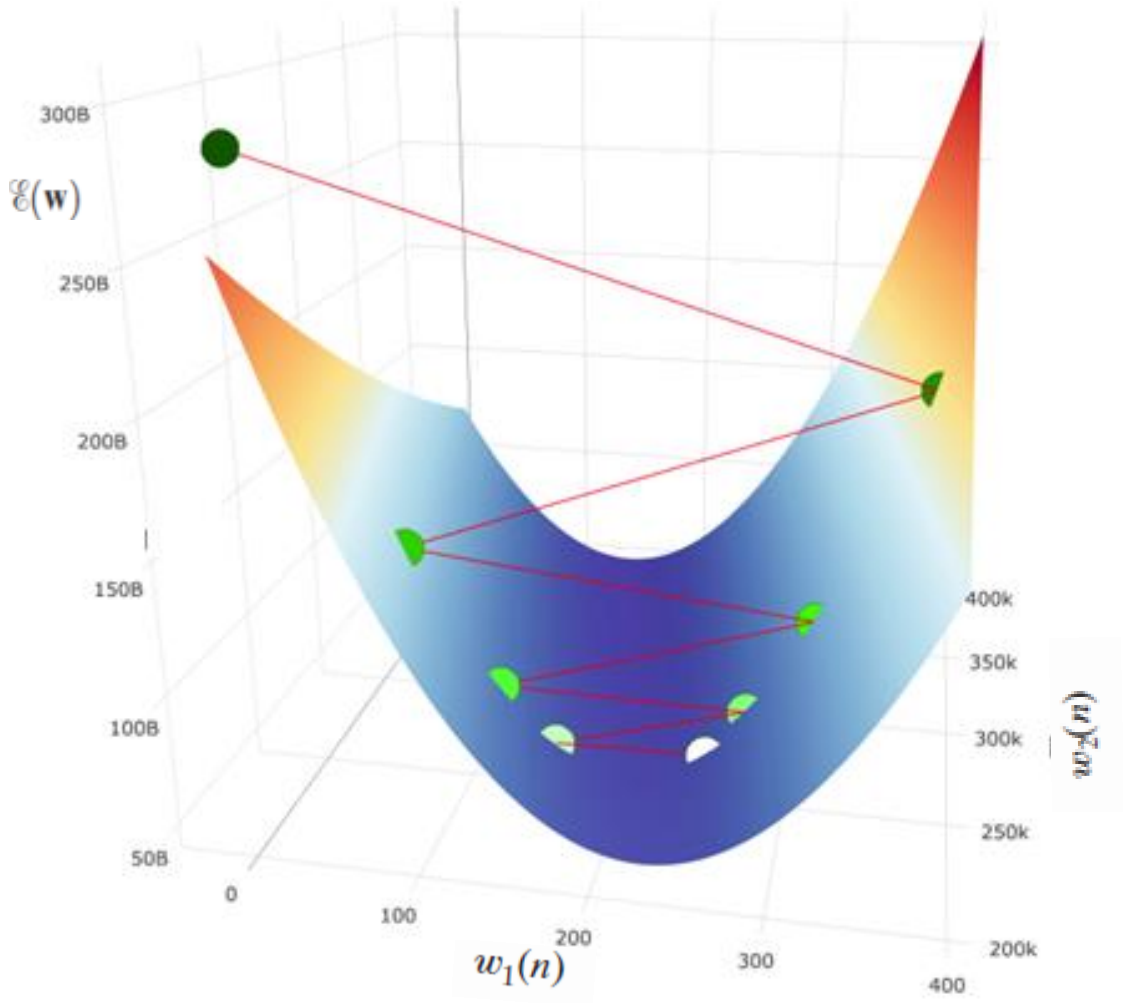


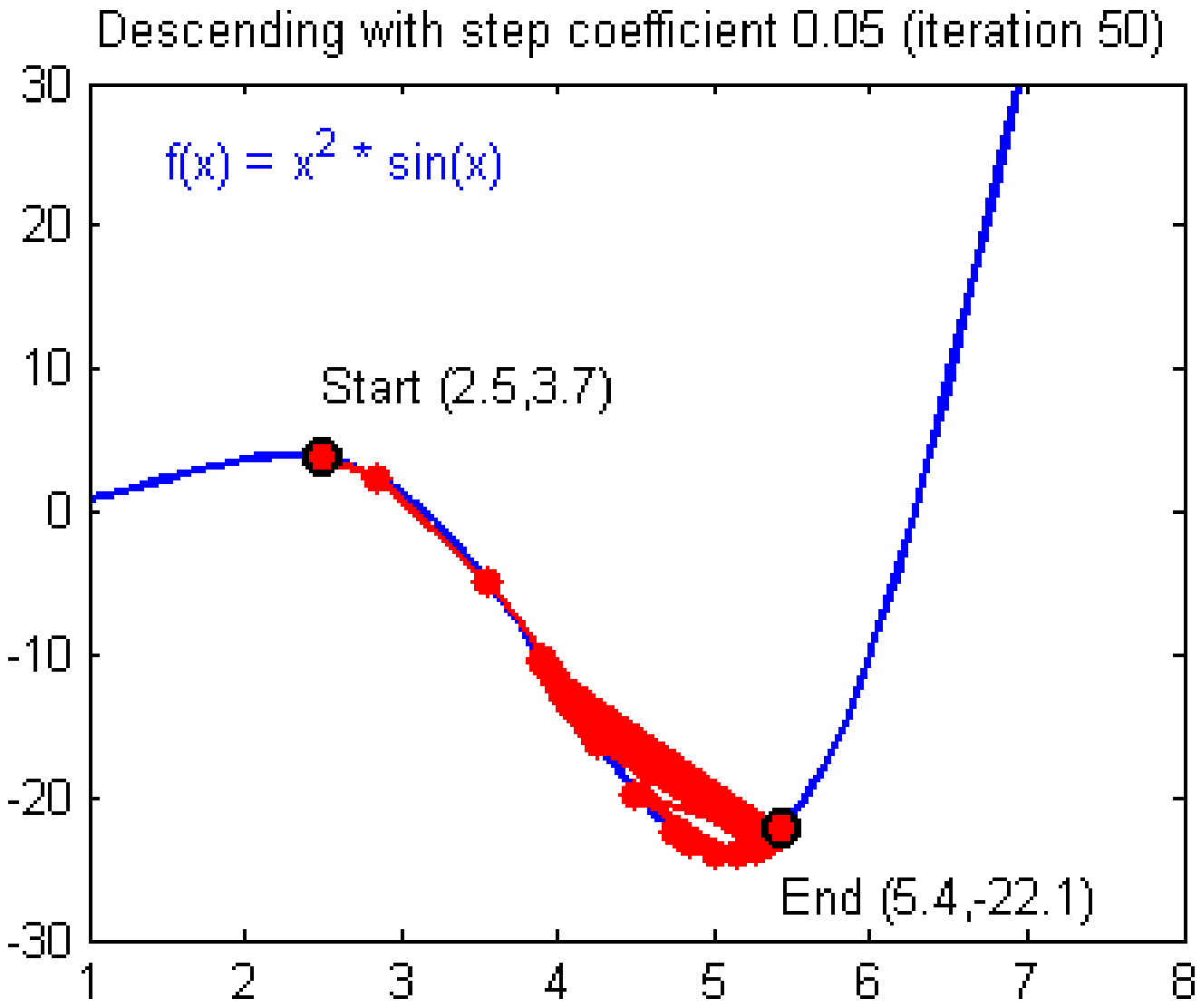


A su vez El método del Gradiente Descendente depende d e los valores iniciales w(o). Ellos representan el punto de partida a partir del cuál se van a ir actualizando sucesivamente los pesos w(n+1) en cada iteración, y dependiendo de la forma de la superficie de la función $\mathscr{E}(\mathbf{w})$ es que cambiará la velocidad y la suavidad de la trayectoria de convergencia
Este primer ejemplo es con w(o) en una zona cercana al mínimo (local)



Este primer ejemplo es con w(o) en una zona lejana al mínimo (local)

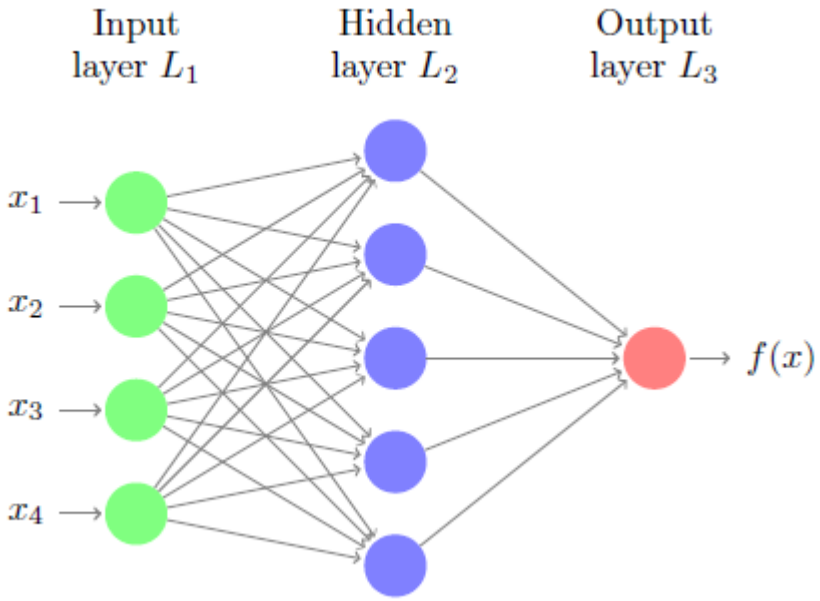



Descending with step coefficient 0.05 (iteration 50)
f(x) = x^2 * sin(x)
Start (2.5,3.7)
End (5.4,-22.1)
30
20
10
0
-10
-20
-30
1
2
3
4
5
6
7
8

Teoría de Redes Neuronales Artificiales

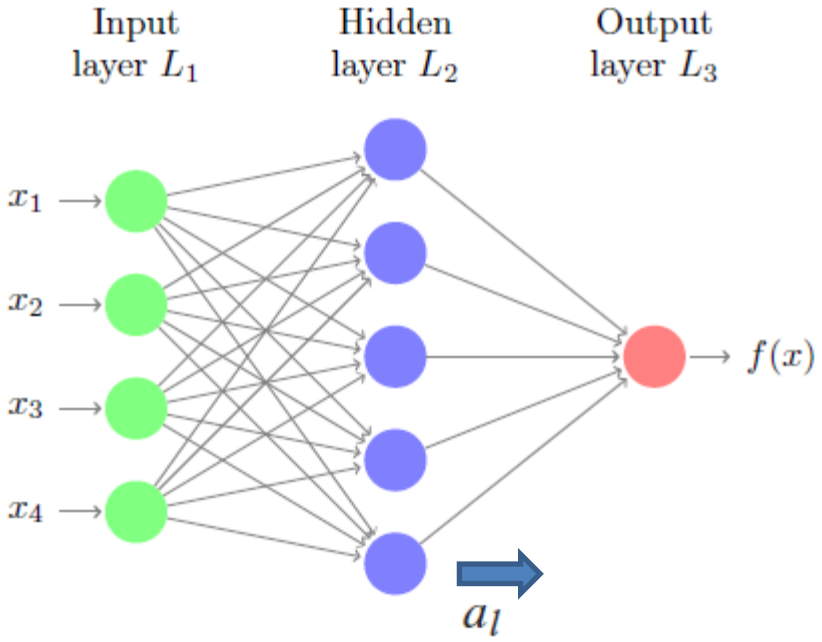
Una red neuronal es un modelo altamente parametrizado, inspirado en la arquitectura. del cerebro humano,

.



Aquí tenemos 4 predictores o entradas x_j 5 unidades, o neuronas, ocultas cuya salida es $a_\ell = g(w_{\ell 0}^{(1)} + \sum_{j=1}^{4} w_{\ell j}^{(1)} x_j)$ una unidad, o neurona, de salida , cuyo output es:

$$o = h(w_0^{(2)} + \sum_{\ell=1}^{5} w_\ell^{(2)} a_\ell)$$



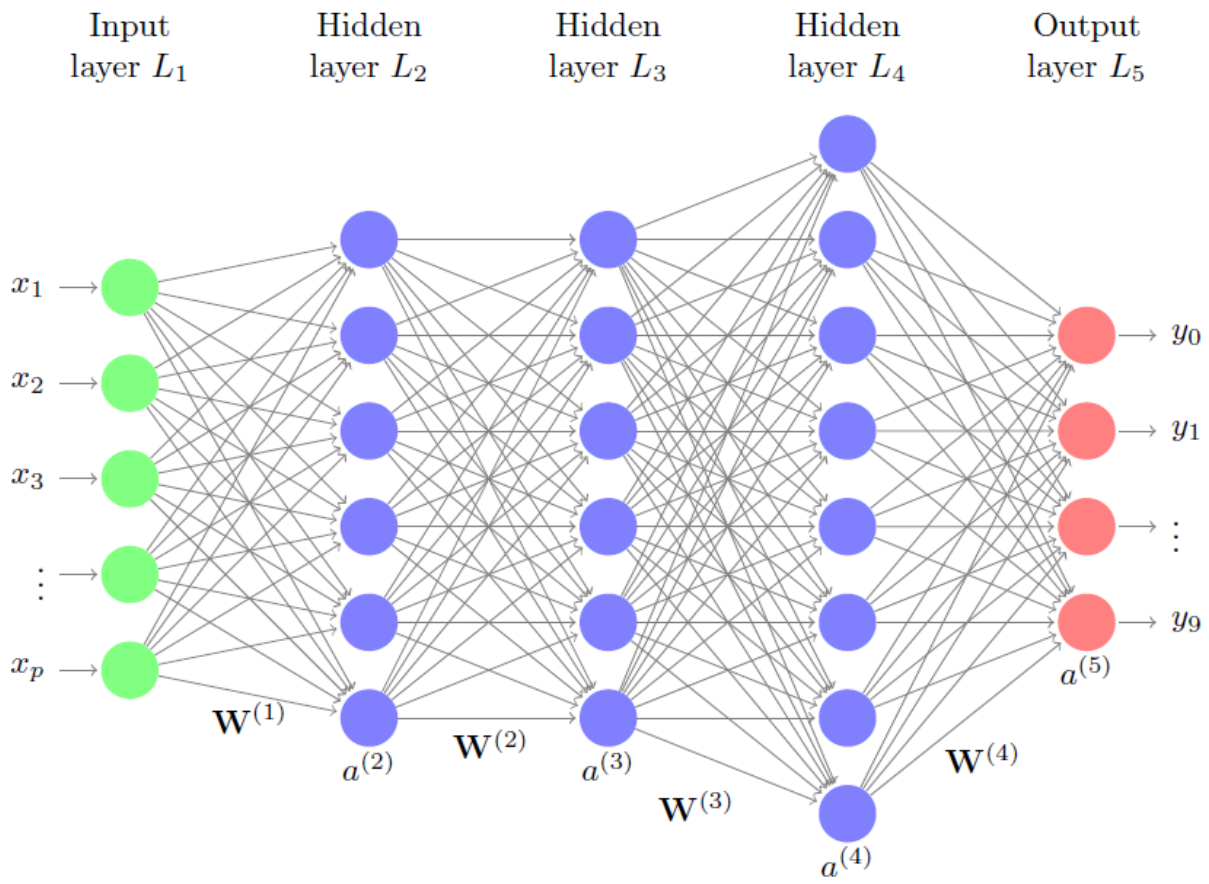
Cada neurona de la capa intermedia está conectada a la capa de entrada a través de de pesos $w_{\ell j}^{(1)}$ donde:

- (1) índice de la capa de origen
- ℓ : índice de neurona de la capa destino
- j : índice de neurona de la capa origen

El intercepto (u ordenada al origen) es $w_{\ell 0}^{(1)}$ llamado *bias* y la función *g* es la encargada de imprimirle "no-linealidad" al modelo. Un ejemplo clásico de función *g* es la sigmoidea (función logística) $g(t) = 1/(1 + e^{-t})$

La capa final o de salida también tiene pesos, y una función de salida *h*. Para regresión cuantitativa h es la combinación lineal de las salidas de la última capa oculta y para una respuesta categórica sería ,una vez más, la función sigmoide (en su formato "softmax" si la respuesta es multinomial)

Redes con más de una capa oculta



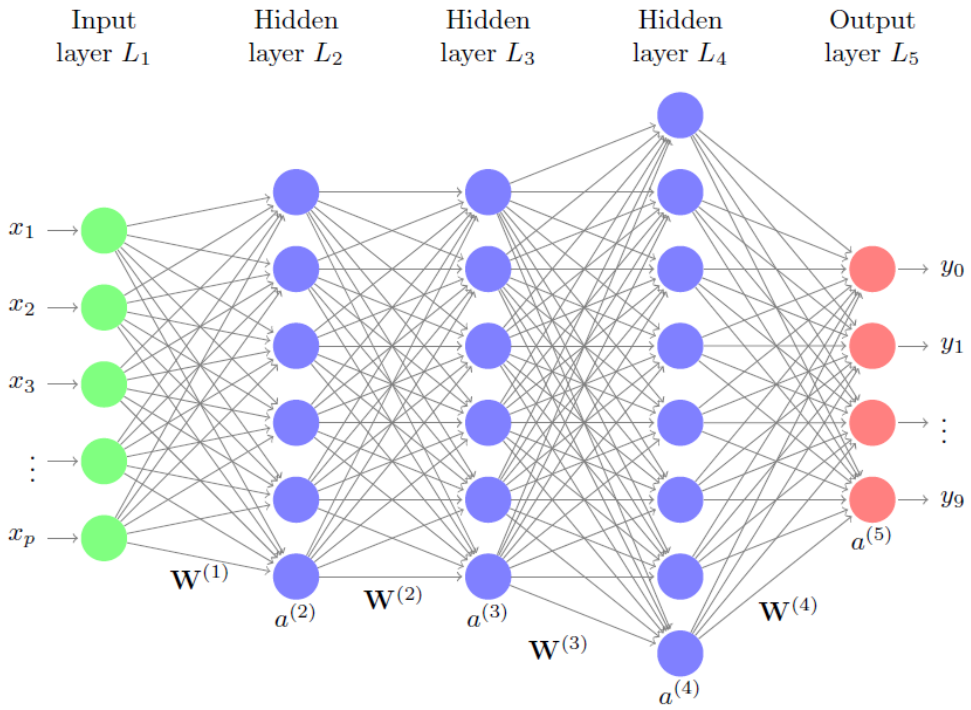
De la primer a la segunda capa tendremos:

$$z_\ell^{(2)} = w_{\ell 0}^{(1)} + \sum_{j=1}^{p} w_{\ell j}^{(1)} x_j$$

z es la CL de los ouputs ponderados + el bias de la capa de la capa de origen (1) que llegan a la neurona *l* de la capa destino (2)

$$a_\ell^{(2)} = g^{(2)}(z_\ell^{(2)})$$

a es la salida de la neurona de la capa (2) fruto de aplicar la función no-lineal g a z



Generalizando para las capas destino k y origen k-1 tendremos:

$$z_\ell^{(k)} = w_{\ell 0}^{(k-1)} + \sum_{j=1}^{p_{k-1}} w_{\ell j}^{(k-1)} a_j^{(k-1)}$$

$$a_\ell^{(k)} = g^{(k)}(z_\ell^{(k)}).$$

Que en notación vectorial sería:

$$z^{(k)} = \boldsymbol{W}^{(k-1)} a^{(k-1)}$$
$$a^{(k)} = g^{(k)}(z^{(k)})$$

Si la variable respuesta del modelo es multinomial (o sea hay más de dos clases o catagorías objetivo) entones la función no lineal de salida es la *softmax:*

$$g^{(K)}(z_m^{(K)}; z^{(K)}) = \frac{e^{z_m^{(K)}}}{\sum_{\ell=1}^{M} e^{z_\ell^{(K)}}}$$

Función de pérdida

La función de rendimiento (también llamada función de pérdida) estándar para evaluar qué tan bien está haciendo una red neuronal se da a través de lo siguiente:

$$P = \frac{1}{2}(t - o)^2$$

La derivada de P con respecto a o es simple:

$$\begin{aligned} \frac{dP}{do} &= \frac{d}{do}[\ \frac{1}{2}(t - o)^2] \\ &= \frac{2}{2} \times (t - o)^1 \times -1 \\ &= -(t - o) \end{aligned}$$

La función sigmoide, $y = 1 / (1 + e^{-x})$, se usa en lugar de una función escalonada en redes neuronales artificiales porque el sigmoide es continuo, y necesita continuidad siempre que quiera usar el ascenso de gradiente. Además, la función sigmoide tiene varias cualidades deseables. Por ejemplo, el valor de la función sigmoidea, ***y***, se acerca a 1 cuando x se vuelve altamente positivo; 0 cuando ***x*** se vuelve altamente negativo; y es igual a 1/2 cuando x = 0. Mejor aún, la función sigmoide presenta una derivada muy simple de la salida, ***y***, con respecto a la entrada, ***x***:

$$
\begin{aligned}
\frac{dy}{dx} &= \frac{d}{dx}\left(\frac{1}{1+e^{-x}}\right) \\
&= \frac{d}{dx}(1+e^{-x})^{-1} \\
&= -1 \times (1+e^{-x})^{-2} \times e^{-x} \times -1 \\
&= \frac{1}{1+e^{-x}} \times \frac{e^{-x}}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \times \frac{1+e^{-x}-1}{1+e^{-x}} \\
&= \frac{1}{1+e^{-x}} \times \left(\frac{1+e^{-x}}{1+e^{-x}} - \frac{1}{1+e^{-x}}\right) \\
&= y(1-y)
\end{aligned}
$$

La derivada de la salida ***y*** con respecto a la entrada ***x*** se expresa como una simple función de la salida ***y***!!!

Propagación de errores hacia atrás *(BackPropagation)*

Backpropagation es un técnica originada a partir del concepto de **gradiente descendente**. Intentamos el mínimo de una función de pérdida ***P***, cambiando los pesos asociados con las neuronas, movéndonos en la dirección del gradiente descendente en un espacio vectorial en el que ***P*** es función de los pesos ***w.*** Es decir, se mueve en la dirección del descenso más rápido si damos un paso en la dirección con los componentes regidos por la siguiente fórmula, que muestra cuánto cambiar un peso, w, en términos de una derivada parcial:

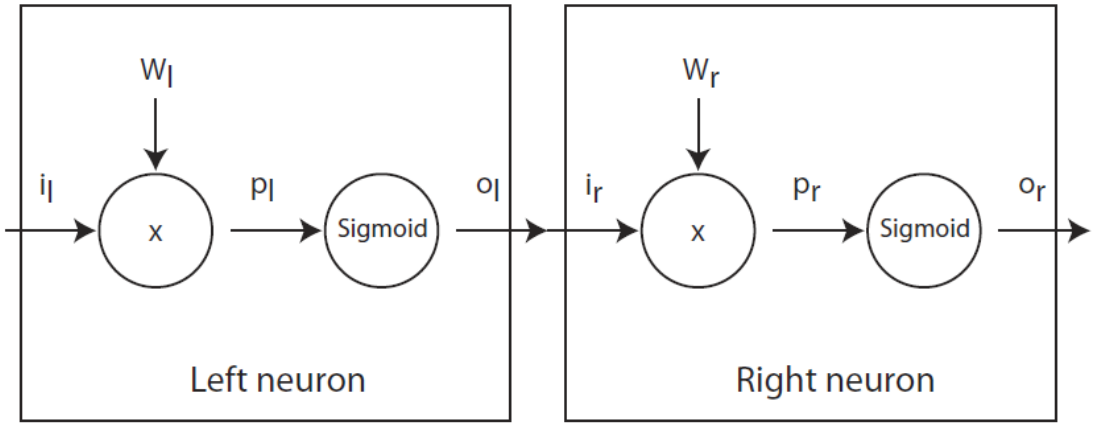
$$\Delta w \propto \frac{\partial P}{\partial w}$$

La variación que aplicaremos sobre cada peso será proporcional a una constante de α *llamada tasa de aprendizaje.* En consecuencia, el nuevo peso **w** viene dado po:

$$w' = w - \alpha \times \frac{\partial P}{\partial w}$$

Backpropagation en Red Neuronal Simple

Considere la red neuronal más simple posible: una entrada, una salida y dos neuronas, la neurona izquierda y la neurona derecha. Una red con dos neuronas es la más pequeña que ilustra cómo se pueden calcular los derivados capa por capa



Tenga en cuenta que los subíndices indican capa. Por lo tanto, i_l, w_l, p_l y o_l son la entrada, el peso, el producto y la salida asociados con la neurona de la izquierda (l), mientras *que* i_r, w_r, p_r y o_r son la entrada, el peso, el producto y la salida asociados con la neurona a la derecha. Por supuesto, $o_l = i_r$.
Supongamos que la salida de la neurona derecha, o, es el valor que determina el rendimiento ***P***. Para calcular la derivada parcial de **P** con respecto al peso en la neurona correcta, w_r, necesita la regla de la cadena, que le permite calcular derivadas parciales de una variable con respecto a otra en términos de una variable intermedia. En particular, para w_r, ti ene lo siguiente, tomando o para ser la variable intermedia:

$$\frac{\partial P}{\partial w_r} = \frac{\partial P}{\partial o_r} \times \frac{\partial o_r}{\partial w_r}$$

Ahora, puedes repetir, usando la regla de la cadena para convertir $\frac{\partial o_r}{\partial w_r}$ en $\frac{\partial o_r}{\partial p_r} \times \frac{\partial p_r}{\partial w_r}$

$$\frac{\partial P}{\partial w_r} = \frac{\partial P}{\partial o_r} \times \frac{\partial o_r}{\partial p_r} \times \frac{\partial p_r}{\partial w_r}$$

Ya hemos visto dos de las derivadas. Siendo la tercera, $\frac{\partial p_r}{\partial w_r} = \frac{\partial (w_r \times o_l)}{\partial w_r}$ es fácil de calcular:

$$\frac{\partial P}{\partial w_r} = [(t - o_r)] \times [o_r(1 - o_r)] \times [i_r]$$

Repitiendo el análisis para w_l se obtiene lo siguiente:

$$\begin{aligned}
\frac{\partial P}{\partial w_l} &= \frac{\partial P}{\partial o_r} \times \frac{\partial o_r}{\partial w_l} \\
&= \frac{\partial P}{\partial o_r} \times \frac{\partial o_r}{\partial p_r} \times \frac{\partial p_r}{\partial w_l} \\
&= \frac{\partial P}{\partial o_r} \times \frac{\partial o_r}{\partial p_r} \times \frac{\partial p_r}{\partial o_l} \times \frac{\partial o_l}{\partial w_l} \\
&= \frac{\partial P}{\partial o_r} \times \frac{\partial o_r}{\partial p_r} \times \frac{\partial p_r}{\partial o_l} \times \frac{\partial o_l}{\partial p_l} \times \frac{\partial p_l}{\partial w_l} \\
&= [(t - o_r)] \times [o_r(1 - o_r)] \times [w_r] \times [o_l(1 - o_l)] \times [i_l]
\end{aligned}$$

Por lo tanto, la derivada consiste en productos de términos que ya han sido computados y términos cercanos a w_l. Esto es más claro si escribe las dos derivadas una al lado de la otra:

$$\frac{\partial P}{\partial w_r} = (t - o_r) \times o_r(1 - o_r) \times i_r$$

$$\frac{\partial P}{\partial w_l} = (t - o_r) \times o_r(1 - o_r) \times w_r \times o_l(1 - o_l) \times i_l$$

Puede simplificar las ecuaciones definiendo *δs* de la siguiente manera, donde cada delta se asocia con la neurona izquierda o derecha:

$$\delta_r = o_r(1 - o_r) \times (d - o_r)$$

$$\delta_l = o_l(1 - o_l) \times w_r \times \delta_r$$

Luego, puedes escribir las derivadas parciales con la δs:

$$\frac{\partial P}{\partial w_r} = i_r \times \delta_r$$

$$\frac{\partial P}{\partial w_l} = i_l \times \delta_l$$

Si agrega más capas al frente de la red, cada peso tiene derivadas parciales que se calculan como la derivada parcial del peso de la neurona izquierda. Es decir, cada uno tiene una derivada parcial determinada por su entrada y su delta, donde su delta a su vez está determinada por su salida, el peso a su derecha y el delta a su derecha. Por lo tanto, para los pesos en la capa final, calcula el cambio de la siguiente manera, donde uso f como el subíndice en lugar de r para enfatizar que el cálculo es para la neurona en la capa final:

$$\Delta w_f = \alpha \times i_f \times \delta_f$$

donde

$$\delta_f = o_f(1 - o_f) \times (d - o_f)$$

Para todas las demás capas, calcula el cambio de la siguiente manera:

$$\Delta w_l = \alpha \times i_l \times \delta_l$$

donde

$$\delta_l = o_l(1 - o_l) \times w_r \times \delta_r$$

Backpropagation en redes multi-capa.

Por supuesto, realmente desea volver a las fórmulas de propagación no solo para cualquier número de capas sino también para cualquier número de neuronas por capa, cada una de las cuales puede tener múltiples entradas, cada una con su propio peso. En consecuencia, debe generalizar en otra dirección, permitiendo múltiples neuronas en cada capa y múltiples ponderaciones unidas a cada neurona.
La generalización es una aventura en sumas, con muchos subíndices para ser directos, pero al final, el resultado coincide con la intuición. Para la capa final, puede haber muchas neuronas, por lo que la fórmula necesita un índice, k, que indique qué neurona nodo final está en juego. Para cualquier peso contenido en la neurona de la capa final, fk, se calcula el cambio de la entrada correspondiente al peso y del δ asociado con la neurona de la siguiente manera:

$$\Delta w = \alpha \times i \times \delta_{f_k}$$

$$\delta_{f_k} = o_{f_k}(1 - o_{f_k}) \times (d_k - o_{f_k})$$

Tenga en cuenta que la salida de cada salida de neurona de capa final se resta de la salida deseada para esa neurona.

Para otras capas, también puede haber muchas neuronas, y la salida de cada una puede influir en todas las neuronas en la siguiente capa a la derecha. El cambio en el peso tiene que explicar lo que sucede con todas las neuronas de la derecha, por lo que aparece una suma, pero de lo contrario, calcula el cambio, como antes, a partir de la entrada correspondiente al peso y de la δ asociada con la neurona:

$$\Delta w = \alpha \times i \times \delta_{l_i}$$

$$\delta_{l_i} = o_{l_i}(1 - o_{l_i}) \times \sum_j w_{l_i \to r_j} \times \delta_{r_j}$$

Tenga en cuenta que $w_{li} \rightarrow r_j$ es el peso que conecta la neurona lateral derecha $j^{ésima}$ con la salida de la neurona lateral izquierda *i*.

Algoritmo de Red Neuronal multi-capa

Como hemos visto, un modelo de red neuronal es una función recursiva compleja $f(x; W)$ del vector de variables predictoras **x**, y de la colección de pesos *W.*
Las funciones $g^{(k)}$ más utilizadas son fácilmente diferenciables

Función de pérdida : Dado un set de entrenamiento $\{x_i, y_i\}_1^n$ y una función de pérdida (o error) $L[y, f(x)]$, trataremos de resolver

$$\underset{W}{\text{minimize}} \left\{ \frac{1}{n} \sum_{i=1}^{n} L[y_i, f(x_i; W)] + \lambda J(W) \right\} \qquad \textbf{(1)}$$

donde $J(W)$ es un término de regularización no negativo sobre los elementos de *W*, y $\lambda \geq 0$ es un parámetro de ajuste. Un ejemplo clásico de regularización es el “ridge” (que hemos visto en regresión) que en ANN se define como:

$$J(W) = \frac{1}{2} \sum_{k=1}^{K-1} \sum_{j=1}^{p_k} \sum_{\ell=1}^{p_{k+1}} \left\{ w_{\ell j}^{(k)} \right\}^2$$

1. Dado un par *x,y* de datos observado corre un proceso "hacia adelante" *(feedforward)* computando las funciones de activación $a_\ell^{(k)}$ en cada una de las capas

2. Para cada neurona de salida ***l*** de la capa $\boldsymbol{L_k}$ calcular:

$$\delta_\ell^{(K)} = \frac{\partial L[y, f(x, \mathcal{W})]}{\partial z_\ell^{(K)}}$$
$$= \frac{\partial L[y, f(x; \mathcal{W})]}{\partial a_\ell^{(K)}} \dot{g}^{(K)}(z_\ell^{(K)})$$

3. Para las capas k= K-1, K-2,….., 2 y para cada neurona *l* de cada capa *k* calcular:

$$\delta_\ell^{(k)} = \left(\sum_{j=1}^{p_{k+1}} w_{j\ell}^{(k)} \delta_j^{(k+1)} \right) \dot{g}^{(k)}(z_\ell^{(k)})$$

4. Las derivadas parciales viene dadas por:

$$\frac{\partial L[y, f(x; \mathcal{W})]}{\partial w_{\ell j}^{(k)}} = a_j^{(k)} \delta_\ell^{(k+1)}$$

Modelo de redes Neuronales *neuralnet*

- **Presentación**: neuralnet es una función utilizada para entrenar modelos de redes neuronales utilziadno las tñecncias de backpropagation , resilient backpropagation, con o sin weight backtracking o la convergebncia global mopdificada de Anastasiadis. La función opermite configuraciones muy flexibles al permitie elegir entre diferentes tipos de error y de funciones de actiovación. A su vez implementa el cálculo de pesos generalizados de Intrator

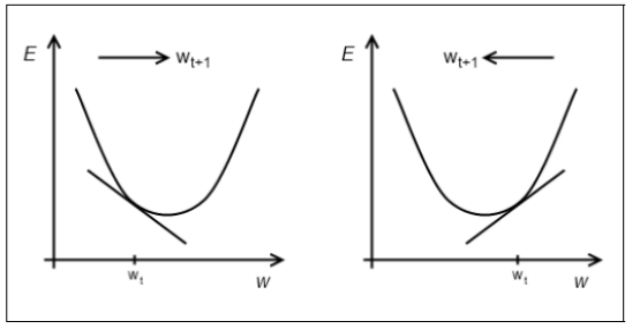
- El modelo se entrena de la siguiente manera:

```
neuralnet(formula, data, hidden = 1, threshold = 0.01,
        stepmax = 1e+05, rep = 1, startweights = NULL,
        learningrate.limit = NULL,
        learningrate.factor = list(minus = 0.5, plus = 1.2),
        learningrate=NULL, algorithm = "rprop+",
        err.fct = "sse", act.fct = "logistic",
        linear.output = TRUE, exclude = NULL,
        constant.weights = NULL, likelihood = FALSE)
```

- **Argumentos:**
  - **formula:** la fórmula escrita que describe la relación entre la variable respuesta y las covariantes (variables predictoras)
  - **data:** el dataset
  - **hidden:** un vector cuya dimensión es la cantidad de capas y sus elementos la cantidad de neurona de cada capa
  - **threshold**: Si las derivadas parciales superan todas este límite se corta el entrenamiento del modelo. Valor por defecto=0.01
  - **stepmax:** la cantidad máxima de pasadas de entrenamiento, también llamadas épocas (barrido de todas as observaciones y ajuste de pesos de neuronas)
  - **rep**: número de veces que entrena el modelo, Cada vez modifica los pesos de entrada. Default=1
  - **startweights**: es un vector que contiene los valores iniciales de los pesos. Por defecto este vector se conforma de valores al azar obtenidos a partir de una distribución normal
  - **learningrate.limit**: vector o lista conteniendo el límite inferior y el superior para el factor de multiplicación para los límites superiores o inferiores de la tasa de aprendizaje (sólo se utiliza en RPROP and GRPROP)
  - **algorithm**: indica el tipo de algoritmo a utilizar. Los tipos son

- "backprop": es el algoritmo clásico de backpropagation
- "rprop+" y "rprop-": significant *resilient backpropagation* con o sin *weight backtracking*

Resilient backprop



$$w_k^{(t+1)} = w_k^{(t)} - \eta \cdot \frac{\partial E^{(t)}}{\partial w_k^{(t)}}$$

$$w_k^{(t+1)} = w_k^{(t)} - \eta_k^{(t)} \cdot \operatorname{sign}\left(\frac{\partial E^{(t)}}{\partial w_k^{(t)}}\right)$$

weight backtracking

Hay un factor de amplificación (eta.plus) para el caso en el que se mantiene el signo de la derivada en iteraciones sucesivas. Hay un factor de reducción para cuando las derivadas son diferentes (eta.minus). Sino son iguales, se incrementa la tasa de aprendizaje en un proporcionalmente al valor de eta.plus y se actualizan los pesos de la forma usual. En caso contrario, cuando las derivadas tienen distinto signo, primero se lleva el peso al valor de la iteración anterior, y se penaliza a la tasa de aprendizaje reduciendo su valor en forma proporcional al eta.minus

```
for all weights{
    if (grad.old*grad>0){
        delta := min(delta*eta.plus, delta.max)
        weights := weights - sign(grad)*delta
        grad.old := grad
    }
    else if (grad.old*grad<0){
        weights := weights + sign(grad.old)*delta
        delta := max(delta*eta.minus, delta.min)
        grad.old := 0
    }
    else if (grad.old*grad=0){
        weights := weights - sign(grad)*delta
        grad.old := grad
    }
}
```

- "sag" and "slr": indican la forma de modificar globalmente la convergencia el algoritmo. Definen como cambia la tasa de aprendizaje para todo el algoritmo en función e evaluar todas las derivadas parciales. En “sag" se va a tomar el menor de los gradientes y en “slr" la menor de las tasa de aprendizaje

- **“err.fct”:** la función diferenciable de error o de pérdida. Los valores que puede tomar son:
  - “sse” por suma de los cuadrados del error (por defecto)

$$E_{sse} = \frac{1}{2} \sum_{l=1}^{L} \sum_{h=1}^{H} (o_{lh} - y_{lh})^2$$

  - “ce” por entropía cruzada. (utilizada para respuestas categóricas)

$$E_{ce} = \frac{1}{2} \sum_{l=1}^{L} \sum_{h=1}^{H} [y_{lh} \log(o_{lh}) + (1 - y_{lh}) \log(1 - o_{lh})]$$

- **“act.fct”:** la función de activación. Pude tomar los valores "logistic", para función logística (valor por defecto) o "tanh", por tangente hiperbólica.

- **"linear.output":** Valor lógico que indica si se aplica la función de activación a las neuronas de salida. En caso afirmativo su valor es FALSE (utilizada para respuestas categóricas). El default es TRUE

- **"exclude"**: Un matriz n*3 donde se explicita qué peso no se va a entrenar. La prime columna indica que capa, la segunda la neurona de entrada y la tercera la neurona de salida.

- **"constant.weights"**: vector especificando los valores de los pesos que estñan exlcuidos del entrenamiento y se toman como fijos

- **"likelihood":** si la función de error es de logversomilitud (utilizada para respuestas categóricas) toma el valor TRUE. Default es FALSE.

Información del modelo obtenido:

La librereñia luego de corre la funciñón neuralnet entrega el resultado en un objeto de calse nn. Los siguientes son los elementos que componens del objeto, que se se acceden con $ o indicando entre corchetes su índice:

- **call:** sintaxis de la función neuralnet
- **response:** variable respuesta, u objetivo observada (extraída del argumento "data")
- **covariate:** las variables predictoras (extraída del argumento "data")
- **model.list:** las variables que se usaron para entrenar el modelo (extraída del argumento "fórmula!)
- **err.fct**: la función de activación utilizada para entrenar el modelo
- **act.fct:** la función e activación utilizada para entrenar el modelo
- **data:** el dataset utilizado (extraída del argumento "data")
- **net.result:**. Lista que contiene el resultado general, la salida de la red neuronal para cada r
- **weights**, Una lista contenido los valores de los pesos para cada neurona para cada replicación
- **generalized.weights**, dan una idea de la influencia de cada variable en el resultado arrojado por el modelo- La formula de los pesos generalizados es:

$$\tilde{w}_i = \frac{\partial \log\left[\frac{o(x)}{1-o(x)}\right]}{\partial x_i}$$