BACK-PROPAGATION LEARNING

1. Asigne pesos iniciales $w_{kj}^{l} = w_{kj}^{l}(0)$ para todos los *k,j,l*.
2. Seleccione la constante de aprendizaje $\eta(0)$ -puede decrementarse con el tiempo-.
3. Para p=1, 2, … (los patrones, cíclicamente)
   Para l = L, L-1, … (las capas)
   Para todos los *k,j* en la capa l (los pesos)
   Calcule el error $\varepsilon_{k}^{l}\left[p, w_{kj}^{l}(p-1)\right]$ así:

$$\varepsilon_{k}^{l}\left[p, w_{kj}^{l}(p-1)\right] = \begin{cases} \tau_{k}(p) - y_{k}^{L}(p, w_{kj}^{l}) & \text{si } l=L \\ \sum_{v=1}^{N_{l+1}} \varepsilon_{v}^{l+1}\left(p, w_{kj}^{l}(p-1)\right) f'\left[u_{v}^{l+1}\left(p, w_{kj}^{l}(p-1)\right)\right] w_{vk}^{l+1}(p-1) & \text{otro } l \end{cases}$$

   Calcule el gradiente de $E\left(p, \vec{W}\right)$ en $w_{kj}^{l}(p-1)$ así:

$$\frac{\partial E\left(p, \vec{W}\right)}{\partial w_{kj}^{l}} = -\varepsilon_{k}^{l}(p, w_{kj}^{l}(p-1)) f'\left[u_{k}^{l}(p, w_{kj}^{l}(p-1))\right] y_{j}^{l-1}(p)$$

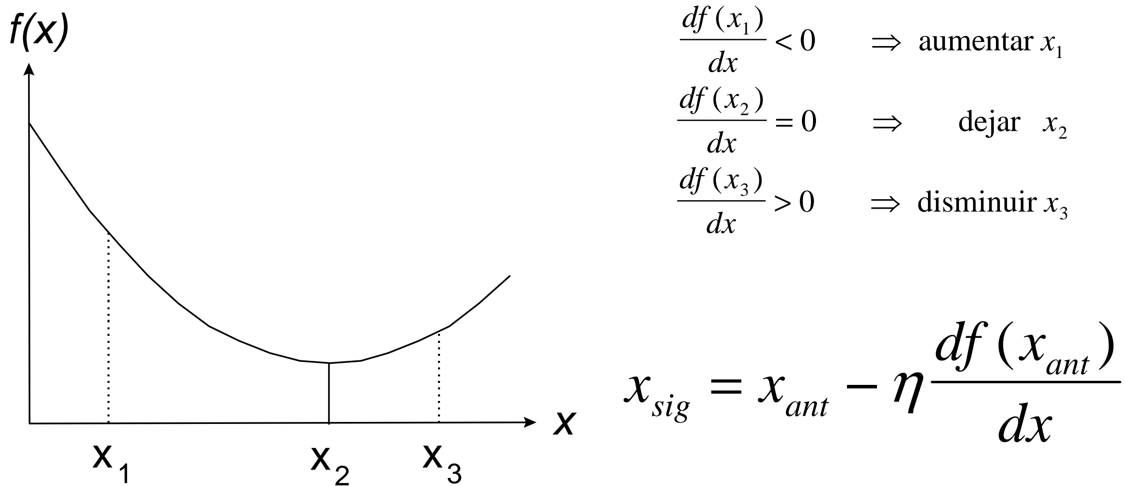
   Actualice el valor de este peso mediante

$$w_{kj}^{l}(p) = w_{kj}^{l}(p-1) - \eta(p) \frac{\partial E\left(p, \vec{W}\right)}{\partial w_{kj}^{l}}$$

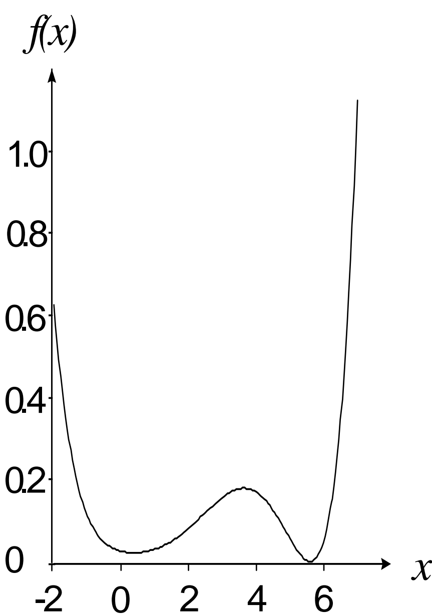
   Siguiente peso
   Siguiente capa
   Siguiente patrón (cíclicamente, hasta que el cambio en los pesos sea insignificante)

El anterior es el algoritmo de aprendizaje BPL, como aparece en la mayoría de libros sobre redes neuronales. Resulta muy difícil de leer y, con mayor razón, de entender. Sin embargo, entre toda esa maraña de subíndices y superíndices, podemos descubrir en la última ecuación que unas variables w se están actualizando con un incremento negativamente proporcional a la derivada de una función $E(w)$. Parece ser que el algoritmo BPL quiere minimizar $E(w)$, de acuerdo con el "método del gradiente"….

Método del Gradiente



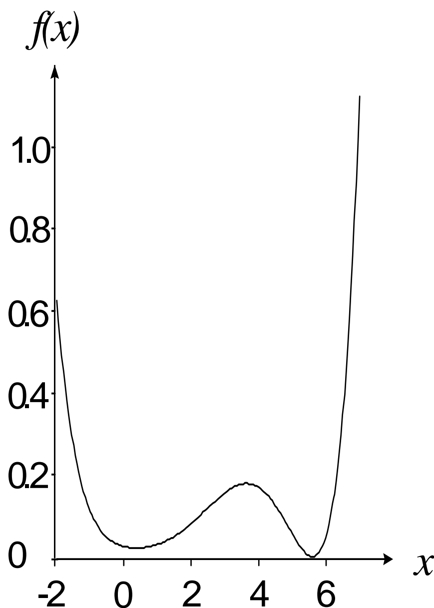
Efectivamente, si queremos encontrar el mínimo de *f(x)* y estamos en un punto donde *f'(x)* es negativa, será necesario incrementar *x* en busca del mínimo. Pero si *f'(x)* es positiva, debemos decrementar *x*. Esto es, debemos movernos en el sentido contrario al de la derivada, como parecía sugerir el algoritmo BPL. Una constante de proporcionalidad η determinará la magnitud de ese movimiento.



$$f(x) = \frac{1}{1000}\left(0.5x^3 + 3x^2 - 2x + 5\right)^2$$

```
double derivada(double x)
{
  return (x3+6x2-4x+10)(1.5x2+6x-2)/1000;
}
void main(void)
{
  double x, xant, Eta;
  cout << "      Eta ? "; cin >> Eta;
  cout << "x inicial ? "; cin >> x;
  do
  {
    xant = x;
    x = xant - Eta * derivada(xant);
  } while(fabs(x - xant) > 0.000001);
}
```

El anterior programa en C muestra el procedimiento para buscar el mínimo de una *f(x)* particular, dados un punto inicial y una constante de proporcionalidad η. El algoritmo termina cuando el incremento sea insignificante.



$$x_{n+1} = x_n - \eta \frac{d}{dx} f(x_n)$$

```
Eta ? 1
x inicial ? -2
0: f(-2) = 0.625                       f '() = -1
1: f(-1) = 0.11025                     f '() = -0.1995
2: f(-0.8005) = 0.0770863      f '() = -0.136338
3: f(-0.664162) = 0.0608111 f '() = -0.103663
4: f(-0.5605) = 0.0511443      f '() = -0.0834473
5: f(-0.477052) = 0.0447711 f '() = -0.069637
…
222: f(0.366984) = 0.0215793           f '() = -1.04596e-006
223: f(0.366985) = 0.0215793           f '() = -9.98351e-007
```

NO LLEGO AL VERDADERO MINIMO!

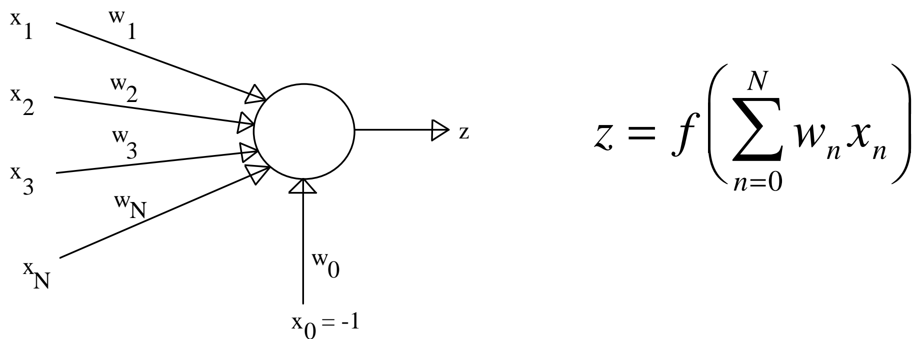
```
Eta ? 5
x inicial ? 5
0: f(5) = 0.05625                      f '() = -0.1425
1: f(5.7125) = 0.00300695    f '() = 0.0578272
2: f(5.42336) = 0.00693551   f '() = -0.0715223
3: f(5.78098) = 0.00842207   f '() = 0.101246
….
2147: f(5.81376) = 0.0121104          f '() = 0.124007
2148: f(5.19372) = 0.0301088          f '() = -0.124007
2149: f(5.81376) = 0.0121104          f '() = 0.124007
2150: f(5.19372) = 0.0301088          f '() = -0.124007
```

INESTABLE!

```
Eta ? 2
x inicial ? 5
0: f(5) = 0.05625                      f '() = -0.1425
1: f(5.285) = 0.0194959                f '() = -0.10762
2: f(5.50024) = 0.00243065   f '() = -0.0448307
3: f(5.5899) = 5.17009e-005  f '() = -0.00697191
4: f(5.60384) = 1.57654e-007           f '() = -0.000388774
5: f(5.60462) = 2.65361e-010           f '() = -1.59587e-005
6: f(5.60465) = 4.29573e-013           f '() = -6.42109e-007
7: f(5.60466) = 6.94282e-016           f '() = -2.58142e-008
```

A VECES FUNCIONA!

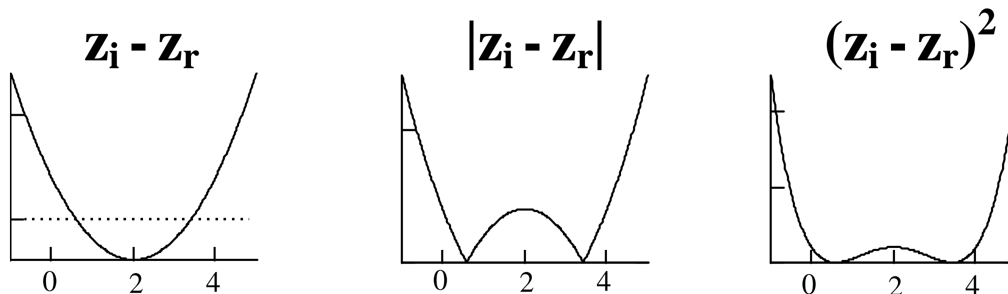
Las anteriores ejecuciones muestran que, con distintos valores de $x_{inicial}$ y η, el algoritmo anterior puede tener resultados insatisfactorios : Convergir a un mínimo local o, sencillamente, no convergir. Sin embargo, !A veces funciona!



Modelo de un elemento computacinal

Para un $\vec{\mathbf{X}} = \{ x_1, x_2, \ldots, x_N \}$, debería producirse $\mathbf{z_i}$ pero, dados unos pesos $\vec{\mathbf{W}} = \{w_0, w_1, \ldots, w_N\}$, la neurona produce $\mathbf{z_r}$.

Pues bien, en un elemento computacional de una red neuronal que recibe como entrada un patrón $\{x_1, x_2, \ldots, x_N\}$ se produce un error si la salida real z_r difiere de la salida ideal z_i. ¿Si z_r es función de los pesos w_k, porqué no seleccionar los pesos para minimizar el error $z_i - z_r$?



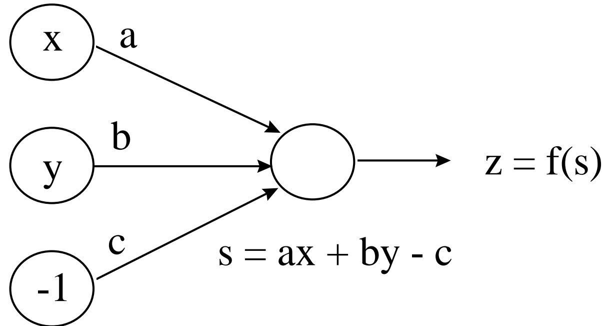
Bueno, en principio minimizar el error podría conducirnos a un gran error negativo y lo que queremos es que el error sea cero. La función valor absoluto incluye mínimos en los puntos en que el error sea cero, pero como el método del gradiente exige derivar, preferimos trabajar con el error cuadrado:

APRENDER : Minimizar el error cuadrado

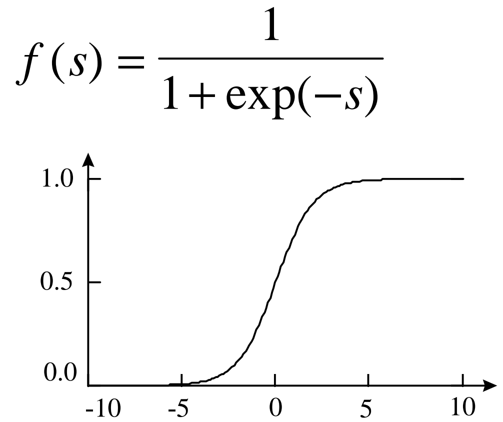
$$\vec{W}_{n+1} = \vec{W}_n - \eta \nabla \left[\left(z_i - z_r\left(\vec{W}\right) \right)^2 \right]$$

Como cada peso se va a actualizar de acuerdo con la respectiva derivada parcial del error cuadrado, el vector de pesos completo se actualiza de acuerdo con el vector gradiente.

Por ejemplo, queremos enseñar a la siguiente red neuronal a que responda con z_i cuando a la entrada tenga un punto particular *(x,y)*, y para eso buscaremos los pesos *(a,b,c)* que minimicen el error. La función de activación utilizada es la sigmoide.



$$f(s) = \frac{1}{1+\exp(-s)}$$



$$E = (z_i - z)^2 = e^2$$ (Error cuadrado)

$$a_{n+1} = a_n - \eta\frac{\partial E}{\partial a_n} \quad b_{n+1} = b_n - \eta\frac{\partial E}{\partial b_n} \quad c_{n+1} = c_n - \eta\frac{\partial E}{\partial c_n}$$ (método del gradiente)

Las derivadas parciales del error cuadrado están en términos de la derivada de la función de activación:

$$\frac{\partial E}{\partial \alpha} = 2(z_i - z)\frac{\partial(z_i - z)}{\partial \alpha} = -2(z_i - z)\frac{\partial z}{\partial \alpha} = -2(z_i - z)\frac{df(s)}{ds}\frac{\partial s}{\partial \alpha} = -2ef'(s)\frac{\partial s}{\partial \alpha}$$

$$a_{n+1} = a_n + \eta e f'(s)x \qquad b_{n+1} = b_n + \eta e f'(s)y \qquad c_{n+1} = c_n - \eta e f'(s)$$

En este caso particular, la derivada de $z=f(s)$ se expresa fácilmente en términos de z:

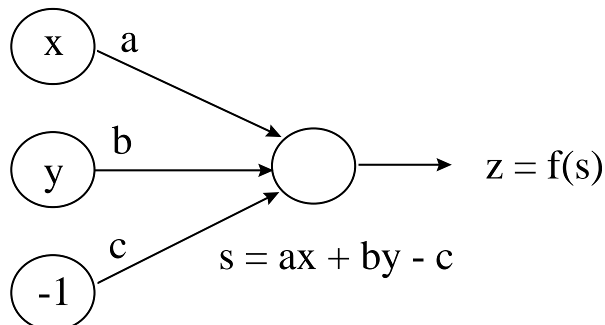
$$z = f(s) = \frac{1}{1+e^{-s}} = \left(1+e^{-s}\right)^{-1}$$

$$f'(s) = (-1)\left(1+e^{-s}\right)^{-2}(-1)e^{-s} = \frac{e^{-s}}{\left(1+e^{-s}\right)^2} = \frac{1}{1+e^{-s}}\frac{1+e^{-s}-1}{1+e^{-s}} = \frac{1}{1+e^{-s}}\left(1-\frac{1}{1+e^{-s}}\right)$$

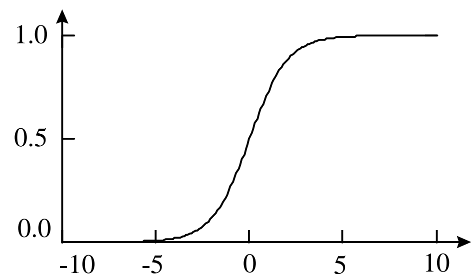
$$f'(s) = f(s)\left(1-f(s)\right) = z(1-z)$$

$$\boxed{a_{n+1} = a_n + \eta e z(1-z)x \qquad b_{n+1} = b_n + \eta e z(1-z)y \qquad c_{n+1} = c_n - \eta e z(1-z)}$$

Las anteriores tres ecuaciones determinan el algoritmo de entrenamiento para la red mostrada arriba



$$f(s) = \frac{1}{1 + \exp(-s)}$$

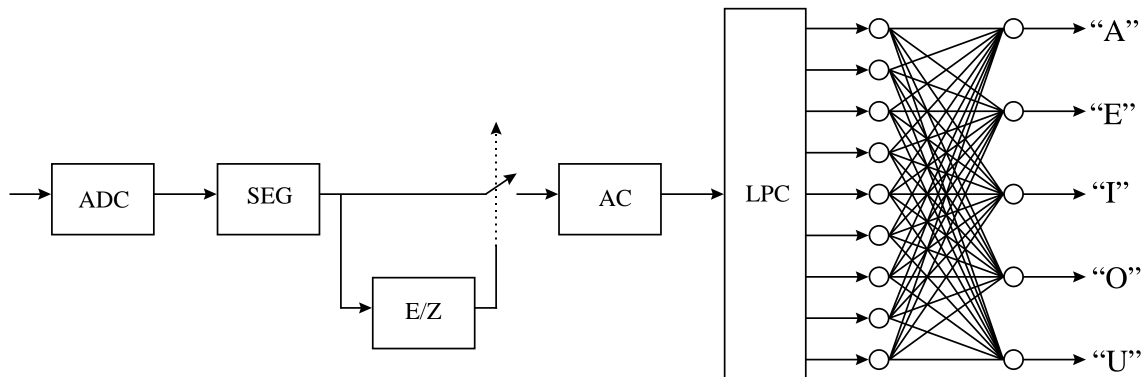



```
double f(double s)
{
  return 1.0/(1.0 + exp(-s));
}
void main(void)
{
  double x = -1, y = 2, zi = 1;
  double Eta=2, a = 1, b = 1, c = 1;
  double z, e, Delta;
  do
  {
    z = f(a*x + b*y - c);
    e = zi - z;
    Delta = Eta*e*z*(1-z);
    cout << a << b << c << z << e << Delta << endl;
    a += Delta*x;
    b += Delta*y;
    c -= Delta;
  } while(fabs(e) > 0.1);
}
```

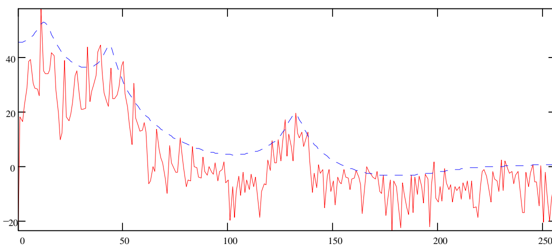
Por ejemplo, el anterior programa busca los pesos *a,b,c* que minimicen el error cuadrado si se quiere que la red produzca 1.0 cuando las entradas sean (-1,2). Como se dijo, la función sigmoide utilizada tiene la particularidad de que $f'(s) = f(s)*(1-f(s))$, y el programa usa esta propiedad al calcular Delta en el tercer renglón de la iteración. La siguiente tabla muestra la salida del programa anterior, que después de 5 iteraciones le enseñó a la red a clasificar el punto (-1,2). Por supuesto, una red neuronal que sepa clasificar un punto y tenga una salida incierta para los demás puntos es absolutamente inútil!

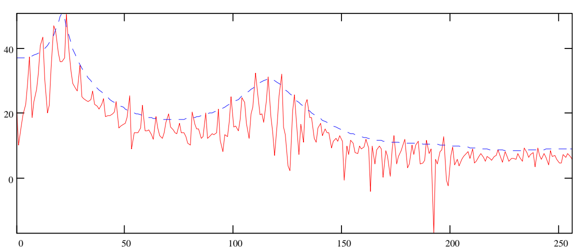
a	**b**	**c**	**z**	**e**	**d**
1	1	1	0.5	0.5	0.25
0.75	1.5	0.75	0.817574	0.182426	0.0544162
0.695584	1.60883	0.695584	0.861344	0.138656	0.0331195
0.662464	1.67507	0.662464	0.883419	0.116581	0.0240133
0.638451	1.7231	0.638451	0.897458	0.102542	0.0188733

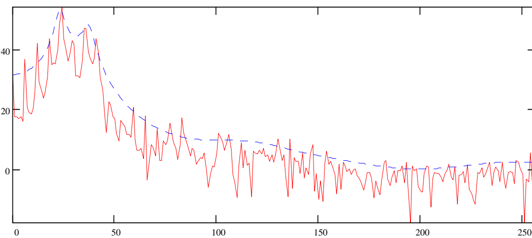
0.619578	1.76084	0.619578	0.90742	0.0925799	0.0155551

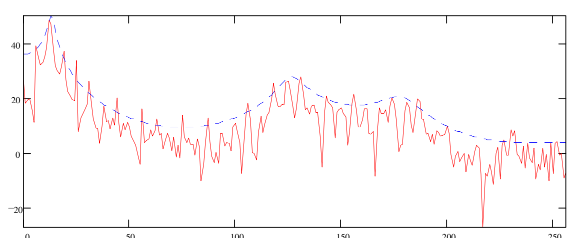


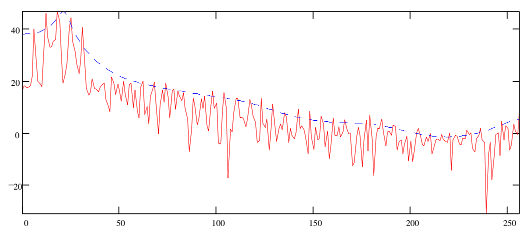
Identificación de Vocales mediante una RN **muy** sencilla









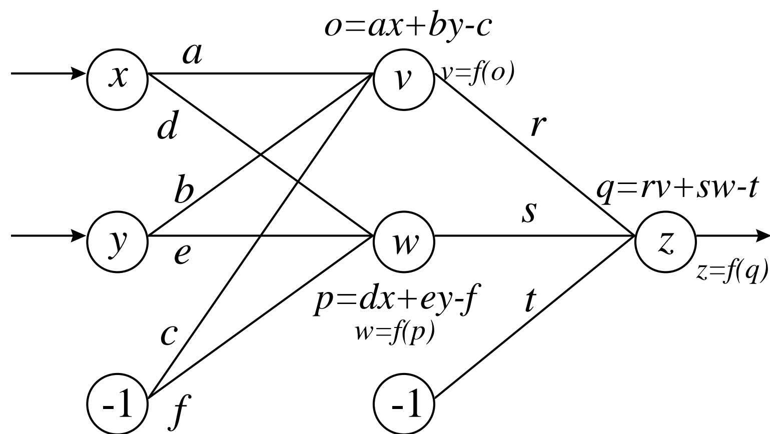


Espectro de un ejemplo de cada una de las cinco vocales con su respectiva envolvente espectral

Por ejemplo, al entrenar un elemento computacional para reconocer una vocal, deben dársele muchas muestras tanto de la vocal correcta como de otras vocales. Una función de discriminación adecuada para clasificar vocales puede ser la envolvente espectral, como muestra esta transparencia. Un sistema de reconocimiento de vocales extraerá de cada vocal capturada los parámetros de la envolvente espectral (parámetros LPC) y los someterá a una red neuronal debidamente entrenada. En este caso se usaron cinco neuronas de 12 entradas y, para entrenarlas, se usó el algoritmo anterior con la única modificación de que, después de cada iteración, se cambiaba el patrón de entrenamiento.

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
void main(void)
{
  FILE   *fpt;
  double x[50][10], w[11][5], Eta=8.0,
         z[5], z_[5];
  int    e,i,j,k,l;
  fpt = fopen("polinomi.dat","r");
  for(i=0;i<50;i++) for(j=0;j<10;j++)       // Carga los coeficientes LPC
     fscanf(fpt,"%lf",&x[i][j]);
  fclose(fpt);
  for(i=0;i<11;i++) for(j=0;j<5;j++)        // Inicializa las conexiones
   w[i][j]=0.5-((double)rand())/((double)RAND_MAX);
  for(i=0;i<5;i++) z_[i] = 0.0;
  do {
    e = 0; l=0;
    for(i=0;i<50;i++) {
      z_[l] = 1.0;                                // Salida ideal
      for(j=0;j<5;j++) {
        z[j] = -w[10][j];                   // Calcula la salida real
        for(k=0;k<10;k++) z[j] += w[k][j]*x[i][k];
        z[j] = 1.0/(1.0 + exp(-z[j]));
      }
      if(z[l] < 0.5) e += 1;                // Clasificación incorrecta
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
      for(k=0;k<11;k++) for(j=0;j<5;j++)    // APRENDIZAJE!!
        if(k<10)
           w[k][j] += (z_[j]-z[j])*x[i][k]*z[j]*(1.0-z[j])*Eta;
        else
           w[k][j] -= (z_[j] - z[j])*z[j]*(1.0 - z[j])*Eta;
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
      z_[l] = 0.0; l++; if(l==5) l=0;       // Siguiente vocal
    }
  } while(e > 0);                           // Hasta que no haya errores
}
```

Este programa entrena a la red neuronal de la transparencia anterior para reconocer cada una de las cinco vocales. En el archivo "polinomi.dat" se encuentran diez coeficientes LPC para 10 grupos de cinco vocales, que serán los patrones de entrenamiento. En el lazo *do-while* se codifica el algoritmo desarrollado hasta ahora: El primer *for(i=...)* de ese lazo recorre cada uno de los 50 patrones; en el siguiente *for(j=...)* se calcula la salida de cada una de las cinco neuronas, en el lazo *for(k=...) for(j=...)* se lleva a cabo el aprendizaje actualizando cada uno de los pesos sinápticos de acuerdo con el gradiente, como se dedujo de las transparencias anteriores. El ciclo se repite hasta que las 50 vocales patrón se hayan reconocido satisfactoriamente.



Dados ***x*** y ***y***, se desea que la salida sea z_i. ¿Cómo debe ser el proceso iterativo sobre los pesos ***a,b,c,d,e,f,r,s*** y ***t***?

$$E = (z_i - z)^2 = e_z^{\ 2}$$

Ya vimos que

$$\frac{\partial E}{\partial r} = -2 \cdot e_z \cdot f'(q) \cdot v, \quad \frac{\partial E}{\partial s} = -2 \cdot e_z \cdot f'(q) \cdot w, \quad \frac{\partial E}{\partial t} = 2 \cdot e_z \cdot f'(q)$$

Pero ¿qué decir de la derivada respecto a ***a***?

$$\frac{\partial E}{\partial a} = -2 \cdot e_z \cdot \frac{\partial z}{\partial a} = -2 \cdot e_z \cdot f'(q) \frac{\partial q}{\partial a} = -2 \cdot e_z \cdot f'(q) \frac{\partial q}{\partial v} \frac{\partial v}{\partial a}$$

$$\frac{\partial E}{\partial a} = -2\left[e_z \cdot f'(q) \cdot r\right] \frac{\partial v}{\partial o} \frac{\partial o}{\partial a} = -2 \cdot \left[e_z \cdot f'(q) \cdot r\right] \cdot f'(o) \cdot x$$

Cuando hay múltiples capas el procedimiento descrito hasta ahora se puede seguir aplicando para actualizar los pesos sinápticos de la capa de salida. Pero, como no sabemos cuál es el error de las capas escondidas, no podemos aplicarlo directamente en la actualización de los pesos de estas capas. Sin embargo, al derivar el cuadrado del error de salida respecto al peso sináptico ***a*** de la capa anterior, vemos que se obtiene una expresión semejante : Aparece la derivada de la función de activación de la neurona en cuestión (***v***), el valor de la entrada de la conexión sináptica en cuestión (***x***) y otro término que, al comparar las expresiones, haría las veces de un "error" en esta neurona. (para que las cosas sean más claras, hemos evitado introducir subíndices; pero obsérvese cómo el alfabeto se agota rápidamente!)

Igualmente,

$$\frac{\partial E}{\partial b} = -2 \cdot \left[e_z \cdot f'(q) \cdot r \right] \cdot f'(o) \cdot y \qquad \frac{\partial E}{\partial c} = 2 \cdot \left[e_z \cdot f'(q) \cdot r \right] \cdot f'(o)$$

Por otro lado,

$$\frac{\partial E}{\partial d} = -2 \cdot e_z \cdot \frac{\partial z}{\partial d} = -2 \cdot e_z \cdot f'(q) \frac{\partial q}{\partial d} = -2 \cdot e_z \cdot f'(q) \frac{\partial q}{\partial w} \frac{\partial w}{\partial d}$$

$$\frac{\partial E}{\partial d} = -2 \left[e_z \cdot f'(q) \cdot s \right] \frac{\partial w}{\partial p} \frac{\partial p}{\partial d} = -2 \cdot \left[e_z \cdot f'(q) \cdot s \right] \cdot f'(p) \cdot x$$

De la misma manera,

$$\frac{\partial E}{\partial e} = -2 \cdot \left[e_z \cdot f'(q) \cdot s \right] \cdot f'(p) \cdot y \qquad \frac{\partial E}{\partial f} = 2 \cdot \left[e_z \cdot f'(q) \cdot s \right] \cdot f'(p)$$

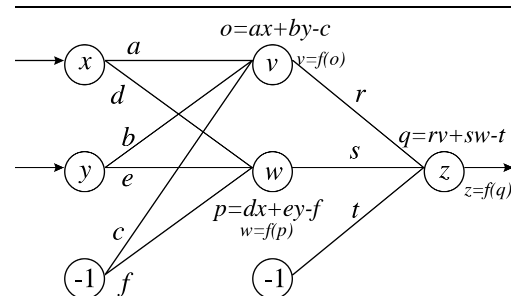
Podemos encontrar la misma relación si derivamos el error cuadrado respecto a cada uno de los pesos de la capa escondida, como muestran las expresiones anteriores.

Aplicando las anteriores derivadas en el método del gradiente,

$$\begin{aligned}
r_{n+1} &= r_n + \eta \cdot e_z \cdot f'(q) \cdot v \\
s_{n+1} &= s_n + \eta \cdot e_z \cdot f'(q) \cdot w \\
t_{n+1} &= t_n + \eta \cdot e_z \cdot f'(q) \cdot (-1) \\
a_{n+1} &= a_n + \eta \cdot e_v \cdot f'(o) \cdot x \\
b_{n+1} &= b_n + \eta \cdot e_v \cdot f'(o) \cdot y \\
c_{n+1} &= c_n + \eta \cdot e_v \cdot f'(o) \cdot (-1) \\
d_{n+1} &= d_n + \eta \cdot e_w \cdot f'(p) \cdot x \\
e_{n+1} &= e_n + \eta \cdot e_w \cdot f'(p) \cdot y \\
f_{n+1} &= f_n + \eta \cdot e_w \cdot f'(p) \cdot (-1)
\end{aligned}$$

donde

$$\begin{aligned}
e_z &= z_i - z \\
e_v &= e_z \cdot f'(q) \cdot r \\
e_w &= e_z \cdot f'(q) \cdot s
\end{aligned}$$



Así, definiendo adecuadamente un "error" para cada una de las neuronas de la capa escondida, podemos encontrar un algoritmo uniforme para la actualización de los pesos: Hemos "propagado hacia atrás" el error a la salida (back propagation).

Si la función de activación es la sigmoide,

$$f'(q) = z \cdot (1 - z) \quad f'(o) = v \cdot (1 - v) \quad f'(p) = w \cdot (1 - w)$$

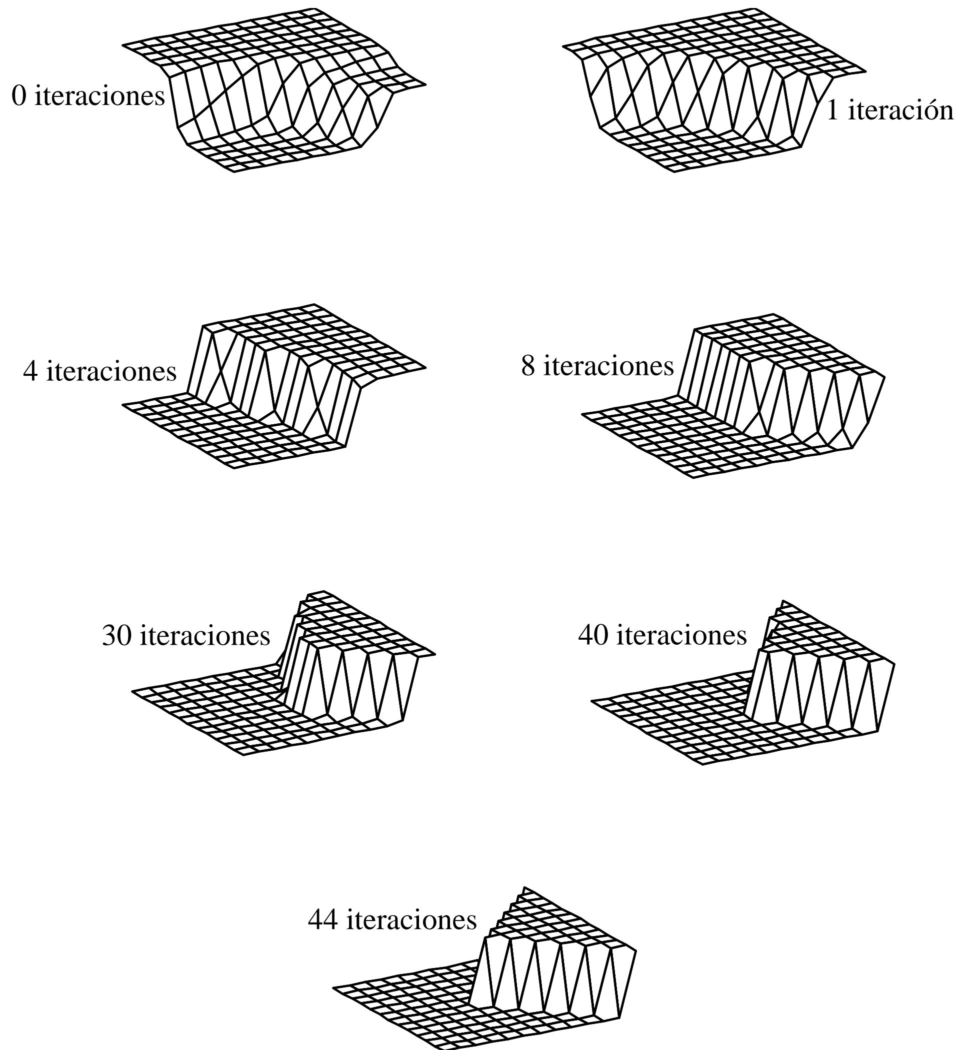
Utilizando la función de activación sigmoide, las derivadas se pueden encontrar fácilmente a partir de las salidas de cada neurona y, así, se podría utilizar el siguiente programa para entrenar la red neuronal de la transparencia anterior:

```
#include <stdlib.h>
#include <math.h>
#define rnd ((double)rand()/(double)RAND_MAX)
void main(void) {
   double a,b,c,d,e,f,r,s,t,   o,p,q,
          v,w,x,y,z,zi,        ez, ev, ew;
   int    i,j,n=0;
   double patron[169][2], objetivo[169], Eta=1.5;
   for(i=0; i<=12; i++) {
     x = (((double) i)/3.0) - 2.0;
     for(j=0; j<=12; j++) {
        y = (((double) j)/3.0) - 2.0;
        patron[n][0] = x;  patron[n][1] = y;
        if((y >= -x) && (y <= x))
           objetivo[n]=1.0; else objetivo[n++] = 0.0;
     }
   }
   a = 0.5 - rnd;     b = 0.5 - rnd;     c = 0.5 - rnd;
   d = 0.5 - rnd;     e = 0.5 - rnd;     f = 0.5 - rnd;
   r = 0.5 - rnd;     s = 0.5 - rnd;     t = 0.5 - rnd;
   do {
     j=0;
     for(n=0; n<169; n++) {
        x   = patron[n][0]; y  = patron[n][1]; zi = objetivo[n];
        o  = a*x + b*y - c; v  = 1.0/(1.0 + exp(-20.0*o));
        p  = d*x + e*y - f; w  = 1.0/(1.0 + exp(-20.0*p));
        q  = r*v + s*w - t; z  = 1.0/(1.0 + exp(-20.0*q));
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
ez = zi - z;           ev = ez*z*(1-z)*r;     ew = ez*z*(1-z)*s;
r += Eta*ez*z*(1-z)*v; s += Eta*ez*z*(1-z)*w; t -= Eta*ez*z*(1-z);
a += Eta*ev*v*(1-v)*x; b += Eta*ev*v*(1-v)*y; c -= Eta*ev*v*(1-v);
d += Eta*ew*w*(1-w)*x; e += Eta*ew*w*(1-w)*y; f -= Eta*ew*w*(1-w);
//////////////////////////////////////////////////////////////////////////////////////////////////////////////////
///
        if (fabs(ez) >= 0.5) j++;
     }
     Eta *= 0.9995;
   } while (j > 0);
}
```

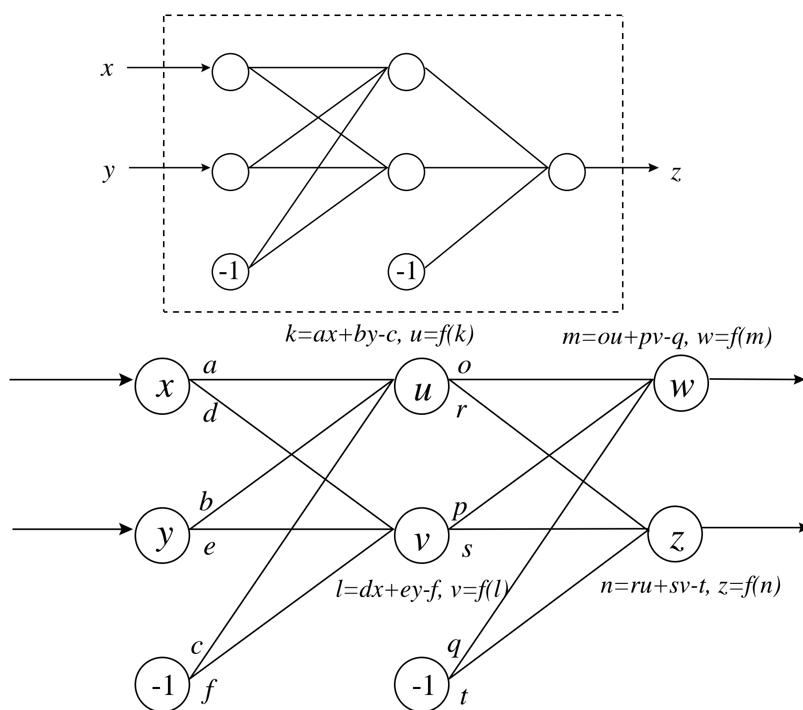
Primero se construyen los arreglos de patrones y función objetivo para cada patrón, luego se inicializan los pesos aleatoriamente y después se entra al ciclo *do-while* de entrenamiento: Para cada patrón se evalúa la red, se calcula el error de salida, se

propaga hacia atrás para calcular los "errores" en la capa escondida y se actualiza cada uno de los pesos correspondientemente.

Esta transparencia muestra cómo la red va aprendiendo a clasificar los puntos del arreglo de patrones a medida que se va iterando el procedimiento de entrenamiento del programa anterior. Después de 44 iteraciones, la red ya sabe identificar cada uno de los patrones.



Convergencia de $\boldsymbol{z(x,y)}$ con el entrenamiento



¿Y cuando hay más de una salida?
El nuevo error cuadrado es la suma de los cuadrados de e_w y e_z:

$$E = (w_i - w)^2 + (z_i - z)^2 = e_w{}^2 + e_z{}^2$$

Al derivar respecto a los pesos de las neuronas de la capa de salida se obtienen los mismos resultados anteriores:

$$\frac{\partial E}{\partial o} = -2 \cdot e_w \cdot f'(m) \cdot u \qquad \frac{\partial E}{\partial p} = -2 \cdot e_w \cdot f'(m) \cdot v \qquad \frac{\partial E}{\partial q} = 2 \cdot e_w \cdot f'(m)$$

$$\frac{\partial E}{\partial r} = -2 \cdot e_z \cdot f'(n) \cdot u \qquad \frac{\partial E}{\partial s} = -2 \cdot e_z \cdot f'(n) \cdot v \qquad \frac{\partial E}{\partial t} = 2 \cdot e_z \cdot f'(n)$$

Sin embargo, al derivar respecto a los pesos de las neuronas de la capa escondida, participan tanto e_w como e_z:

$$\frac{\partial E}{\partial a} = -2 \cdot e_w \frac{\partial w}{\partial a} - 2 \cdot e_z \frac{\partial z}{\partial a} = -2 \cdot e_w \cdot f'(m) \frac{\partial m}{\partial a} - 2 \cdot e_z \cdot f'(n) \frac{\partial n}{\partial a}$$

$$\frac{\partial E}{\partial a} = -2 \cdot e_w \cdot f'(m) \cdot o \cdot \frac{\partial u}{\partial a} - 2 \cdot e_z \cdot f'(n) \cdot r \cdot \frac{\partial u}{\partial a} = -2 \cdot e_u \cdot \frac{\partial u}{\partial a}$$

donde $e_u = e_w f'(m) \cdot o + e_z f'(n) \cdot r$, de manera que

$$\frac{\partial E}{\partial a} = -2 \cdot e_u \cdot f'(k) \cdot x$$

Con más de una neurona en la capa de salida se busca minimizar la suma de los cuadrados de los errores en cada neurona. En este caso, cada error de salida se propaga hacia atrás al definir los errores de las neuronas de las capas escondidas.

Entonces,

$$\frac{\partial E}{\partial o} = -2 \cdot e_w \cdot f'(m) \cdot u \qquad \frac{\partial E}{\partial p} = -2 \cdot e_w \cdot f'(m) \cdot v \qquad \frac{\partial E}{\partial q} = 2 \cdot e_w \cdot f'(m)$$

$$\frac{\partial E}{\partial r} = -2 \cdot e_z \cdot f'(n) \cdot u \qquad \frac{\partial E}{\partial s} = -2 \cdot e_z \cdot f'(n) \cdot v \qquad \frac{\partial E}{\partial t} = 2 \cdot e_z \cdot f'(n)$$

$$\frac{\partial E}{\partial a} = -2 \cdot e_u \cdot f'(k) \cdot x \qquad \frac{\partial E}{\partial b} = -2 \cdot e_u \cdot f'(k) \cdot y \qquad \frac{\partial E}{\partial c} = 2 \cdot e_u \cdot f'(k)$$

$$\frac{\partial E}{\partial d} = -2 \cdot e_v \cdot f'(l) \cdot x \qquad \frac{\partial E}{\partial e} = -2 \cdot e_v \cdot f'(l) \cdot y \qquad \frac{\partial E}{\partial f} = 2 \cdot e_v \cdot f'(l)$$

donde

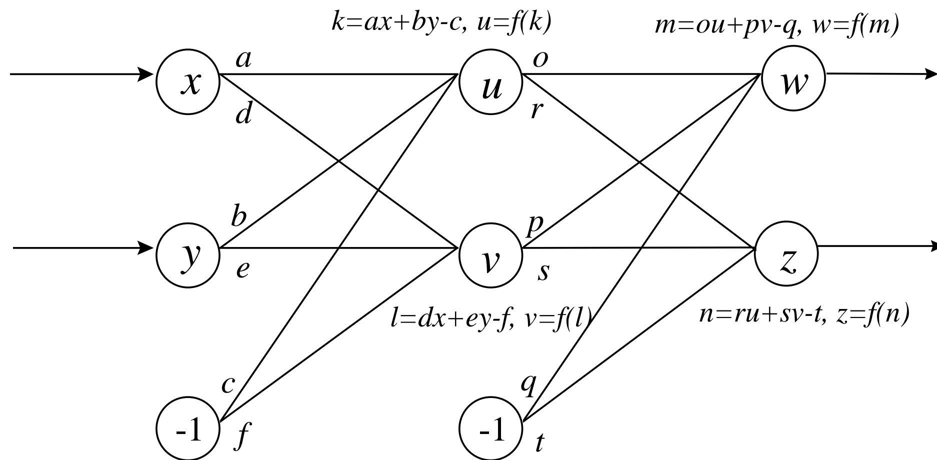
$$e_w = w_i - w \qquad e_z = z_i - z$$

$$e_u = e_w f'(m) \cdot o + e_z f'(n) \cdot r \qquad e_v = e_w f'(m) \cdot p + e_z f'(n) \cdot s$$

Si *f* es la función sigmoide,

$$f'(k) = u \cdot (1 - u) \qquad f'(l) = v \cdot (1 - v)$$

$$f'(m) = w \cdot (1 - w) \qquad f'(n) = z \cdot (1 - z)$$



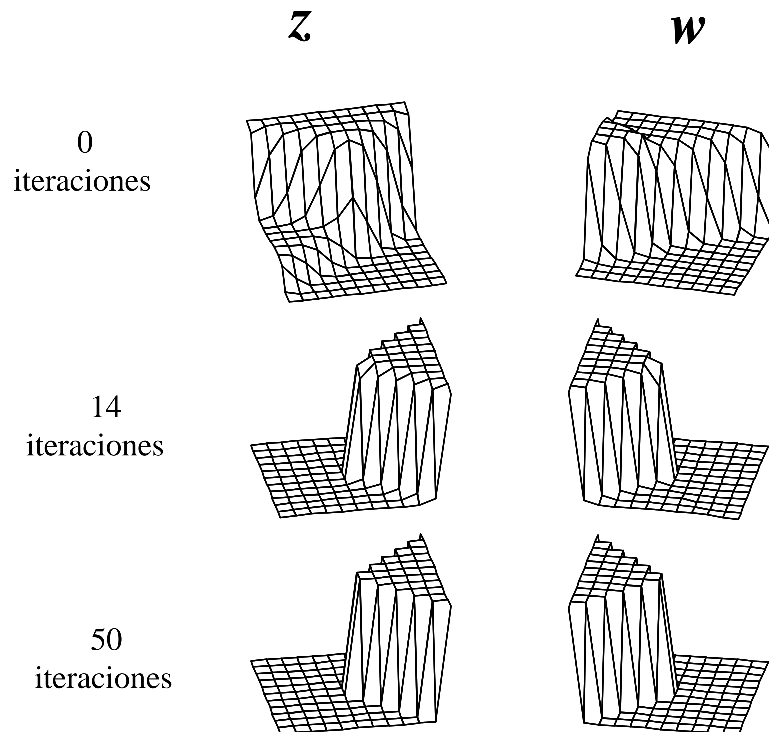

```
do
{
   j=0;
   for(h=0; h<144; h++)
   {
      x  = patron[h][0];     y = patron[h][1];
      wi = objetivo[h][0]; zi = objetivo[h][1];

      k  = a*x + b*y - c; u = 1/(1+exp(-20*k));
      l  = d*x + e*y - f; v = 1/(1+exp(-20*l));
      m  = o*u + p*v - q; w = 1/(1+exp(-20*m));
      n  = r*u + s*v - t; z = 1/(1+exp(-20*n));

      ez = zi - z;                  ew = wi - w;
      eu = ew * w*(1-w) * o  +  ez * z*(1-z) * r;
      ev = ew * w*(1-w) * p  +  ez * z*(1-z) * s;

      if((fabs(ez)>=0.5)||(fabs(ew)>=0.5)) j++;

      o += Eta*ew*w*(1-w)*u;
      p += Eta*ew*w*(1-w)*v;
      q -= Eta*ew*w*(1-w);
      r += Eta*ez*z*(1-z)*u;
      s += Eta*ez*z*(1-z)*v;
      t -= Eta*ez*z*(1-z);
      a += Eta*eu*u*(1-u)*x;
      b += Eta*eu*u*(1-u)*y;
      c -= Eta*eu*u*(1-u);
      d += Eta*ev*v*(1-v)*x;
```

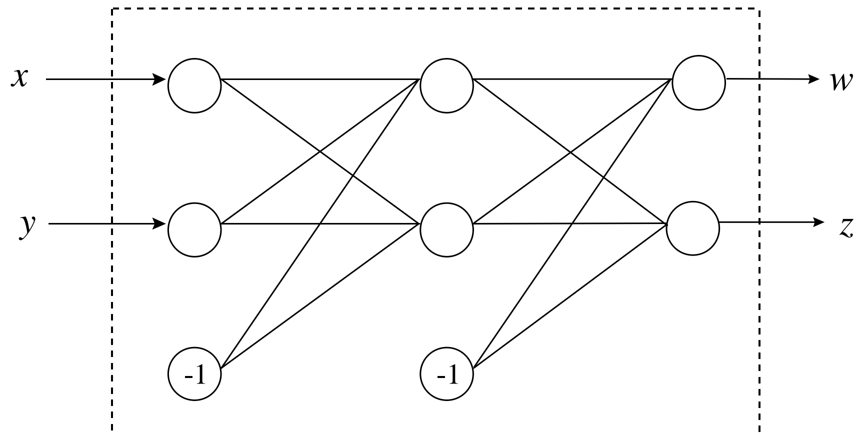
```
        e += Eta*ev*v*(1-v)*y;
        f -= Eta*ev*v*(1-v);
    }
    Eta *= 0.995;
} while (j > 0);
```

Ciclo de aprendizaje de la red

De acuerdo con las expresiones de la transparencia anterior, el ciclo de aprendizaje de la red puede codificarse en C como muestra este listado. Obsérvese el procedimiento de siempre: se introduce cada uno de los patrones, se evalúa la red, se calculan los errores a la salida, se propagan estos errores hacia las capas escondidas y, con ellos, se actualizan los pesos de las conexiones sinápticas, repitiéndose el proceso hasta obtener un desempeño satisfactorio.

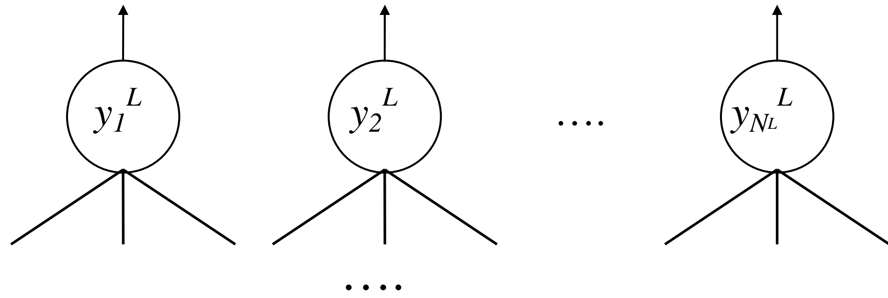
Esta transparencia muestra la evolución de la respuesta de la red neuronal a medida que se realizan iteraciones del algoritmo de entrenamiento de la transparencia anterior. Obsérvese que a las 50 iteraciones la respuesta es la deseada.





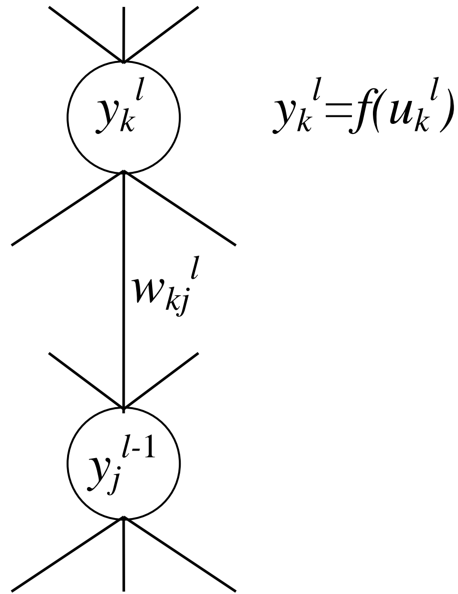
La siguiente transparencia representa un perceptrón multicapa generalizado: Hay L+1 capas, desde la capa *0*, que es la de entrada, hasta la capa de salida L. En la capa l ($0 \leq l \leq L$) hay N_l neuronas, entre las cuales la k-ésima neurona se conecta con la neurona j de la capa *l-1* mediante el peso sináptico w_{kj}^{l} ($1 \leq j \leq N_{l-1}$).

Capa L
N_L nodos

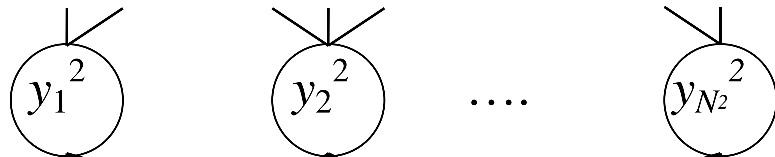


Capa l
N_l nodos

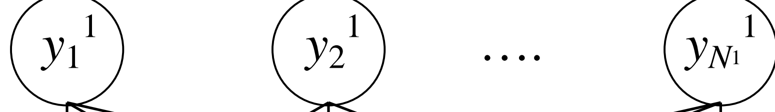
Capa l-1
N_{l-1} nodos



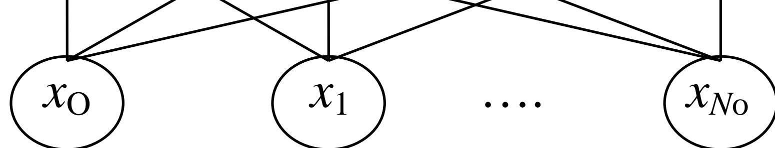
Capa 2
N_2 nodos



Capa 1
N_1 nodos



Capa 0
N_0 entradas



BACK-PROPAGATION LEARNING

1. Asigne pesos iniciales $w_{kj}^{l} = w_{kj}^{l}(0)$ para todos los *k,j,l*.
2. Seleccione la constante de aprendizaje $\eta(0)$ -puede decrementarse con el tiempo-.
3. Para p=1, 2, … (los patrones, cíclicamente)

   Para l = L, L-1, … (las capas)

   Para todos los *k,j* en la capa l (los pesos)

   Calcule el error $\varepsilon_k^l\left[p, w_{kj}^l(p-1)\right]$ así:

$$\varepsilon_k^l\left[p, w_{kj}^l(p-1)\right] = \begin{cases} \tau_k(p) - y_k^L(p, w_{kj}^l) & \text{si } l=L \\ \sum_{v=1}^{N_{l+1}} \varepsilon_v^{l+1}\left(p, w_{kj}^l(p-1)\right) f'\left[u_v^{l+1}\left(p, w_{kj}^l(p-1)\right)\right] w_{vk}^{l+1}(p-1) & \text{otro } l \end{cases}$$

   Calcule el gradiente de $E\left(p, \vec{W}\right)$ en $w_{kj}^l(p-1)$ así:

$$\frac{\partial E\left(p, \vec{W}\right)}{\partial w_{kj}^l} = -\varepsilon_k^l(p, w_{kj}^l(p-1)) f'\left[u_k^l(p, w_{kj}^l(p-1))\right] y_j^{l-1}(p)$$

   Actualice el valor de este peso mediante

$$w_{kj}^l(p) = w_{kj}^l(p-1) - \eta(p) \frac{\partial E\left(p, \vec{W}\right)}{\partial w_{kj}^l}$$

   Siguiente peso

   Siguiente capa

   Siguiente patrón (cíclicamente, hasta que el cambio en los pesos sea insignificante)

Aquí hemos repetido el algoritmo de aprendizaje BPL de la primera transparencia, con la esperanza de que ahora parezca más fácil de leer y, por supuesto, de entender. Después de inicializar los pesos y la constante de aprendizaje se entra al ciclo de entrenamiento en el que se recorren todos los patrones y para cada uno de ellos se evalúa la red y se calculan los errores de cada neurona (directamente en la capa de salida, propagando el error de salida en las capas escondidas), despues de lo cual se puede calcular el gradiente para actualizar los pesos correspondientemente, repitiendo el proceso hasta que se satisfaga algún criterio adecuado de finalización.

Librería de Clases en C++ para ANN

```
class RedNeuronal
{
  private:
    int     NumEntrada;
    int     NumEscondida;
    int     NumSalida;

    Neurona *CapaEntrada;
    Neurona *CapaEscondida;
    Neurona *CapaSalida;

    Pesos   *EntradaEscondida;
    Pesos   *EscondidaSalida;

    double  Activacion(double x);
    double  SumaPonderada(int capa, int neurona);

  public:
    RedNeuronal(int NEntrada, int NEscondida, int NSalida);
    RedNeuronal(char *NombreArchivo);
    ~RedNeuronal();

    void    Evaluacion(double *x);
    void    AsignacionPeso(int capa, int neurona,
                           int origen, double valor);
    double  LecturaPeso(int capa, int neurona, int origen);
    double  LecturaNeurona(int capa, int neurona);
    void    Aprendizaje(int NumPatrones, double *patron,
                        double *objetivo, double velocidad);
    void    Almacena(char *NombreArchivo);
};
```

Para ver el algoritmo BPL en una aplicación real, esta transparencia y la siguiente muestran dos fracciones de una librería de clases en C++, desarrollada por el autor, para experimentar con una perceptrón de tres capas. En esta primera parte se muestra la declaración de la clase RedNeuronal que incluye como variables privadas el número de neuronas en cada capa y los arreglos de pesos entre las capas. Dentro de los métodos definidos para esta clase de objeto se encuentra uno llamado “Aprendizaje” que, como muestra la siguiente transparencia, implementa el algoritmo BPL.

SUBRUTINA BPL

```
void RedNeuronal::Aprendizaje(int NumPatrones, double *patron, double *objetivo, double velocidad)
{
 int i, p,                              // indice general e indice de patrones
      neurona, neurona_ant;             // indice de neuronas en la capa actual y en la anterior
 double *eh, *es, *x;                   // errores en las capas escondida y de salida, vector de entrada

 x  = new double[NumEntrada];           // Reserva memoria para el vector de entrada y los errores
 eh = new double[NumEscondida];         // en cada capa
 es = new double[NumSalida];

 for(p=0; p<NumPatrones; p++)           // Recorre todos los patrones
 {                                      // Construye un vector de entrada para este patron
  for(i=0; i<NumEntrada; i++) x[i] = patron[p*NumEntrada + i];
  Evaluacion(x);                        // Calcula las salidas de las neuronas para este patron

  for(neurona = 0; neurona<NumSalida; neurona++)          // errores de salida
     es[neurona] = objetivo[p*NumSalida + neurona] - CapaSalida[neurona].valor;
  for(neurona = 0; neurona<NumEscondida; neurona++)
  {                                     // Propaga el error a las neuronas escondidas
    eh[neurona]=0;
    for(i=0; i<NumSalida; i++)
      eh[neurona] += ( es[i] *
                       (CapaSalida[i].valor * (1 - CapaSalida[i].valor)) *
                       EscondidaSalida[neurona*NumSalida+i].peso          );
  }

  for(neurona = 0; neurona<NumSalida; neurona++) // Actualiza los pesos sinapticos de las
    for(neurona_ant = 0; neurona_ant<=NumEscondida; neurona_ant++) // neuronas de salida
      EscondidaSalida[neurona_ant*NumSalida + neurona].peso +=
           (  velocidad * es[neurona] *
              CapaEscondida[neurona_ant].valor *
              (CapaSalida[neurona].valor * (1 - CapaSalida[neurona].valor))         );

  for(neurona = 0; neurona<NumEscondida; neurona++) // Actualiza los pesos sinapticos de las
    for(neurona_ant = 0; neurona_ant<=NumEntrada; neurona_ant++) // neuronas escondidas
      EntradaEscondida[neurona_ant*NumEscondida + neurona].peso +=
          (  velocidad * eh[neurona] *
             CapaEntrada[neurona_ant].valor *
             (CapaEscondida[neurona].valor  * (1 - CapaEscondida[neurona].valor))     );
 }
 delete eh; delete es; delete x;                          // Libera la memoria reservada
}
```