Práctica 4 : Aprendizaje supervisado en redes multicapa.

Melisa Maidana Capitán

Junio 2013

Resumen

Se programaron diferentes arquitecturas para entrenar una red neuronal para aprender la regla XOR y para aprender el mapeo logístico. En el primer caso, se compararon las velocidades de aprendizaje para diferentes arquitecturas y cantidad de neuronas de entrada. El el segundo caso, se verificó la capacidad de generalización de la red, es decir, de resolver un problema que no fue aprendido.

Para programar estas redes se utilizó el algoritmo de retroprogación.

1. Aprendizaje supervisado

En los problemas de memorias asociativas, los pesos de las conexiones son medianamente determinados a partir de reglas simples como la de Hebb. Sin embargo, a veces resulta práctico adoptar formas iterativas, que permitan determinar los valores de los pesos mejorando los mismos paso a paso. Este es el proceso conocido como aprendizaje.

Existen dos tipos de aprendizaje para una red neuronal, supervisado y no supervisado. En el aprendizaje supervisado, la salida se compara con la respuesta esperada o çorrectaz es realimentada con el error cometido. Mientras que en el aprendizaje no supervisado no hay salidas correctas o incorrectas, si no que la red debe aprender sola a descubrir características interesantes en las entradas.

En general, en aprendizaje supervisado se entrena la red con un conjunto de pares de entrada-salida que concuerdan. Se dice que la red puede generalizar el aprendizaje cuando se le presenta un par entrada-salida que no se encuentra en su lista de entrenamiento, y la misma puede encontrar los pesos apropiados para obtener exitosamente la salida deseada.

Se denomina perceptrón a una red que tiene una capa de entrada (ξ) y una de salida (ζ), sin capas ocultas. Un algoritmo para lograr el aprendizaje en este tipo de redes es el "feed-forward". Un problema puede resolverse por medio de una red representada por un perceptrón simple, si es mismo es linearmente separable. Esta limitación se aplicación del perceptrón no se aplica a redes con capas ocultas. La importancia de las redes multicapa radica en su capacidad de aprendizaje utilizando algoritmos "back-propagation".

El algoritmo "back-propagation"da una regla para modificar los pesos de cualquier red "feed-forward"para aprender un conjunto de pares entrada-salida de entrenamiento.

1.1. Algoritmo de retropropagación en redes con una capa oculta

Dada una red de dos capas, se denomina O_i a la salida de la i-ésima neurona, V_j al valor de las neuronas de la capa oculta, ξ_k a los valores de las neuronas en la capa de entrada. Además se donomina

J_{jk} a los pesos que conectan la capa de entrada con la oculta, y W_{ij} a los pesos de las conexiones entre la capa oculta y la salida.

La entrada puede tomar valores ξ_k^μ con $k = 1...N$ y $\mu = 1...p$, donde p es el número de patrones y N el número de neuronas de la capa de entrada.

Dado un patrón μ, la unidad j de la capa oculta recibe una entrada dada por

$$h_j^\mu = \sum_k J_{jk}\xi_k^\mu \tag{1}$$

y produce una salida $V_j^\mu = g(h_j^\mu)$, donde g es generalmente una función no lineal.

La i-ésima unidad de salida recibe

$$h_i^\mu = \sum_j W_{ij}V_j^\mu = \sum_j W_{ij}g(\sum_k J_{ij}\xi_k^\mu) \tag{2}$$

y produce una salida $O_i^\mu = g(h_i^\mu)$.

Los valores de umbrales pueden considerarse agregando una neurona a la capa de entrada con valor seteado en -1 y conectada a todas las neuronas de la red.

El error o función de costo se define como

$$E = \frac{1}{2}\sum_{\mu i}(\xi_i^\mu - O_i^\mu)^2 \tag{3}$$

Y las reglas de adaptación de los pesos entra la capa de salida y la oculta están dadas por

$$\Delta Wij = \eta \sum_\mu \delta_i^\mu V_j^\mu \tag{4}$$

donde $\delta_i^\mu = g'(h_i^\mu)(\zeta_i^\mu - O_i^\mu)$ y η es la tasa de aprendizaje. Y entra la capa oculta y la de entrada

$$\Delta J_{jk} = \eta \sum_\mu \delta_j^\mu \xi_k^\mu \tag{5}$$

donde $\delta_j^\mu = g'(h_j^\mu)\sum_i W_{ij}\delta_i^\mu$.

En este informe se utilizó una función de transferencia no lineal $g(x) = tanh(x)$.

2. Arquitecturas y algorítmos para aprender XOR.

2.1. Algortimo de retropropagación para aprender XOR:Redes con dos neuronas de entrada

Dadas dos arquitecturas neuronales diferentes, se utilizó el algoritmo de retropropagación para aprender el XOR. En la Fig.(1) se muestran las arquitecturas utilizadas para aprender el XOR.

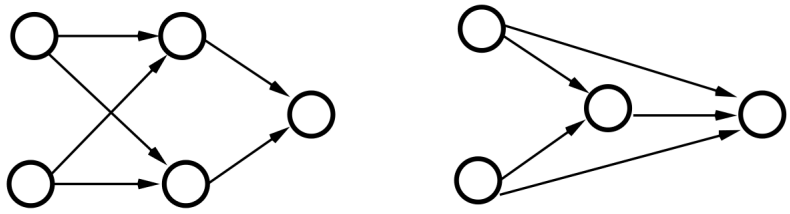

Figura 1: Arquitecturas utilizadas para aprender XOR. A las mismas se les agregó una neurona de entrada que representa los umbrales.

Para ambas arquitecturas se calculo el tiempo de convergencia, definido como el número de iteraciones necesarias para que la red aprenda a dar una salida correcta a los patrones de entrada.

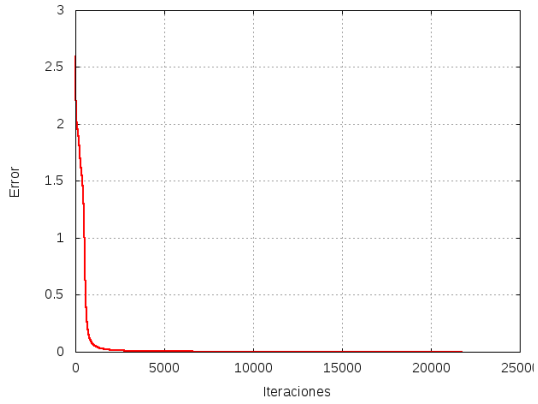

Figura 2: Error como función del número de iteraciones para la arquitectura de la Fig.(1).

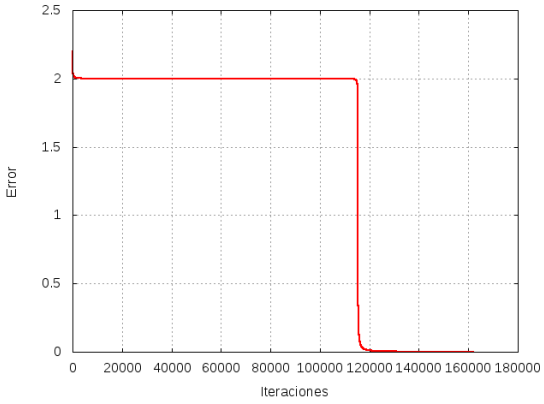

Figura 3: Error como función del número de iteraciones para la arquitectura de la Fig.(**??**).

Se observa que la red con mayor cantidad de neuronas en la capa oculta, aprende más rápido. Es decir, el error entre la salida deseada y la salida de la red disminute más rapidamente con el número de iteraciones.

La red entrenada consiste en que la misma tenga los pesos apreopiados para resolver todos los patrones aprendidos. Si llamamos i a las neuronas de entrada, j a las de oculta y k a las de la capa salida, y J_{ij} son los pesos de la capa de entrada hacia la oculta, W_{ij} a los pesos de la capa oculta hacia la de salida. Llamamos $U1_j$ a los umbrales de la capa de entrada a la oculta, y $U2_k$ a los umbrales de la campa de entrada a la de salida.

Entonces, los pesos que se obtuvieron para la red 1 son $J_11 = 1{,}47, J_{12} = 1{,}48, J_{21} = 1{,}77, J_{22} = 1{,}81$, $W_{11} = -2{,}61, W_{12} = 2{,}59$, $U1_1 = -1{,}28, U1_2 = 1{,}68$ y $U2 = -2{,}32$.

Entonces, los pesos que se obtuvieron para la red 1 son $J_11 = 2{,}19, J_{12} = 2{,}19$, $W_{11} = 4{,}67$,

$U1_1 = 2{,}19$ y $U2 = -2{,}27$.

2.2. Algortimo de retropropagación para aprender XOR: Redes con N neuronas de entrada y N' neuronas en la capa oculta

Se generalizó el problema de aprendizaje de XOR a una red con N neuronas en la capa de entrada, N' neuronas en la capa oculta y una neurona en la capa de salida. La generalización consiste en que la salida es el producto de las N entradas.

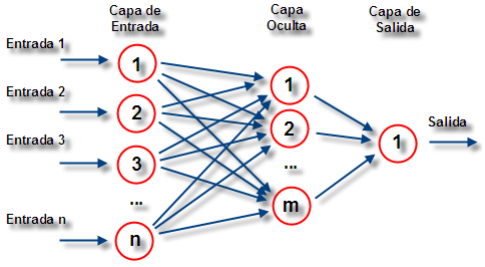

Figura 4:

Se utilizarón distintos valores de N y N' y se calculó el tiempo de convergencia para cada uno. La Fig.(5) muestra el error como función del número de iteraciones para diferentes valores de N y N'.

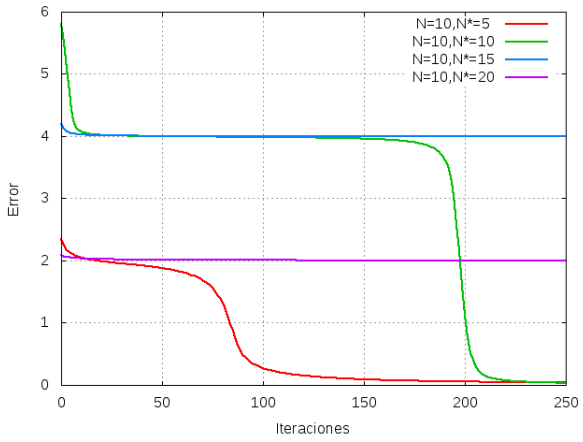

Figura 5: Error como función de la iteración para diferentes relaciones entre el número de neuronas de entrada y el número de neuronas de la capa oculta.

En la Fig.(5) se observa que para valores de $N' < N$ el error no disminuye, por lo que la red tarda más en aprender el problema planteado. Este resultado no tiene mucho sentido, porque en realidad en un problema de redes multicapa, se mapea el problema de la capa de entrada hacia las capas ocultas. Si la dimensión de las capas ocultas es menor que la dimensión de la capa de entrada, el mapeo reduce las dimensiones del problema, por lo que se dificulta el aprendizaje. No pudo encontrarse el error en este programa, pero si se resalta que el resultado no está teniendo mucho sentido.

3. Algoritmo de retropropagación para aprender mapeo logísto

Se programó la red que presenta la Fig.(6) para aprender el mapeo logístico: $x = 4x(1-x)$.

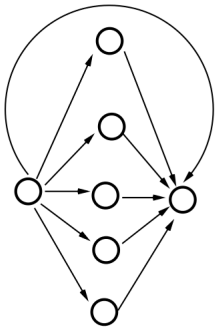

Figura 6: Arquitectura utilizada para aprender el mapeo logístico.

Una vez aprendido el mapeo logístico, se verificó que la red pueda generalizar. Es decir, la capacidad de dar una respuesta correcta ante patrones que no fueron aprendidos.

La Fig.(7) presenta los resultados para los patrones aprendidos y y la Fig.(8)para la generalización.

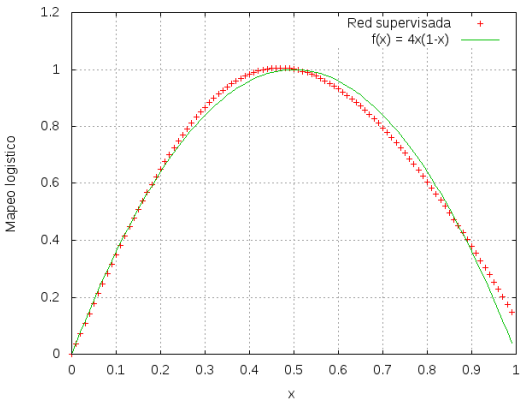

Figura 7: Aprendizaje de la red para 100 patrones equidistantes en el intervalo $[0, 1]$.

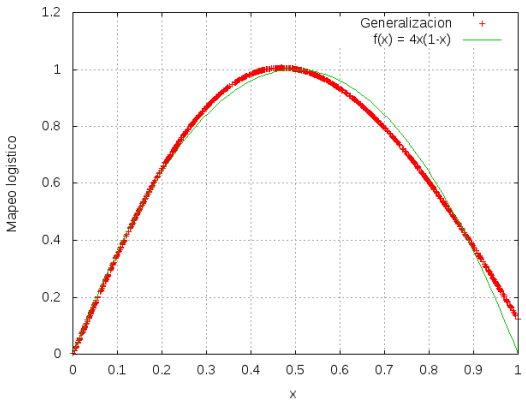

Figura 8: Presentación de 1000 valores aleatorios en el intervalo $[0, 1]$.

La Fig.(9) muestra la curva del error en el aprendizaje como función de las iteraciones cuando se le presentan 100 patrones.

Podemos calcular el error de aprendizaje como

$$E_{apren} = \frac{1}{2}\frac{1}{P}\sum_{i=1}^{P}[\zeta_i - o_i]^2 \tag{6}$$

donde P es el número de patrones presentados a la red para el aprendizaje. Y podemos calcular el error de generalización como

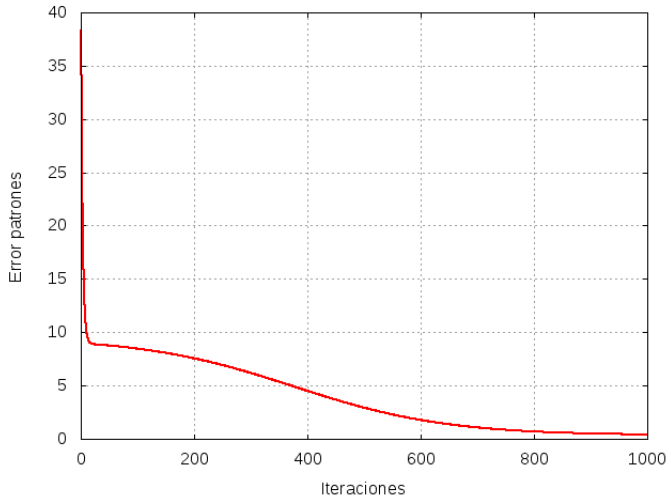

Figura 9: Curva de error en el aprendizaje del mapeo logístico.

$$E_{gral} = \frac{1}{2}\frac{1}{G}\sum_{i=1}^{G}[\zeta_i - o_i]^2 \tag{7}$$

donde G es el número de entradas que se le presentaron luego del aprendizaje.

Para un conjunto de 100 patrones equidistantes en el intervalo $[0, 1]$, se obtuvo un error de aprendizaje final $E_{apren} = 0{,}00049$, y para un conjunto de 1000 puntos elegidos al azar en el mismo intervalo (distintos a los patrones) se obtuvo $E_{gral} = 0{,}00050$.

4. Programas en C

4.1. Algoritmo para aprender XOR: Redes con 2 entradas

```
void InitWeigh(MatDoub &matrix);
void XOR_patrones(MatDoub &matrix);

#define N_IN 2
#define N_HIDE1 2
#define N_OUT 1
#define P 4
#define eta 0.01

int main(){

MatDoub patrones(P,N_IN);
MatDoub J(N_HIDE1,N_IN),W(N_OUT,N_HIDE1);
VecDoub umbral1(N_HIDE1);
double umbral2;
int estado_umbral=-1;
double salida_deseada,o;
VecDoub v(N_HIDE1);
```

```
double error,delta;
VecDoub delta2(N_HIDE1);

srand(time(0));

InitWeigh(J);
InitWeigh(W);
for(int i=0;i<N_HIDE1;i++)
        umbral1=rand()*1.0/RAND_MAX;
umbral2 = rand()*1.0/RAND_MAX;
XOR_patrones(patrones);

double error_global=1;
int it=0;
        while(error_global>0.001){
                error_global=0;
                for(int i=0;i<P;i++){
                        salida_deseada=patrones[i][0]*patrones[i][1];

                        for(int k=0;k<N_HIDE1;k++){
                                v[k]=0;
                                for(int j=0;j<N_IN;j++)
                                        v[k]+=J[k][j]*patrones[i][j];
                                v[k]+=umbral1[k]*estado_umbral;
                                v[k]=tanh(v[k]);
                                }
                        o=0;
                        for(int k=0;k<N_HIDE1;k++)
                                o+=W[0][k]*v[k];
                        o+=umbral2*estado_umbral;
                        o=tanh(o);
                        //cout << salida_deseada << " " << o << endl;
                        error = 0.5*(salida_deseada-o)*(salida_deseada-o)
                            ;
                        error_global+=error;
                        delta = (salida_deseada-o)*(1-o*o);

                        for(int k=0;k<N_HIDE1;k++)
                                delta2[k]=delta*W[0][k]*(1-v[k]*v[k]);

                        for(int k=0;k<N_HIDE1;k++)
                               W[0][k]+=eta*delta*v[k];
                        umbral2+=eta*delta*estado_umbral;

                        for(int k=0;k<N_HIDE1;k++){
                                for(int j=0;j<N_IN;j++)
                                        J[k][j]+=eta*delta2[k]*patrones[i
                                            ][j];
                                umbral1[k]+=eta*delta2[k]*estado_umbral;
                                }
                        }
                //cout << it << " " << error_global << endl;
                it++;
                }
return 0;
}
```

```
void InitWeigh(MatDoub &matrix){

int n = matrix.nrows();
int m = matrix.ncols();

for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
                matrix[i][j] = rand()*1.0/RAND_MAX;
                }
        }
}

void XOR_patrones(MatDoub &matrix){

int n = matrix.nrows();
int m = matrix.ncols();

matrix[0][0]=1;
matrix[0][1]=1;
matrix[1][0]=-1;
matrix[1][1]=-1;
matrix[2][0]=1;
matrix[2][1]=-1;
matrix[3][0]=-1;
matrix[3][1]=1;

}
```

Las modificaciones que suguiere el ejercicio 1.*b* se implementaron en el mismo algoritmo presentado anteriormente de la siguiente manera.

```
for(int k=0;k<N_HIDE1;k++)
        o+=W[0][k]*v[k];
for(int j=0;j<N_IN;j++)
        o+=W[0][N_HIDE1+j]*patrones[i][j];
o+=umbral2*estado_umbral;
o=tanh(o);
```

```
for(int k=0;k<N_HIDE1;k++)
        W[0][k]+=eta*delta*v[k];
for(int j=0;j<N_IN;j++)
        W[0][N_HIDE1+j]+=eta*delta*patrones[i][j];
umbral2+=eta*delta*estado_umbral;
```

4.2. Algoritmo para aprender XOR: Redes con N entradas

```
void InitWeigh(MatDoub &matrix);
void GeneraPatrones(MatDoub &matrix);

int main(){

        int N_IN,N_HIDE1,N_OUT,P;
```

```
        printf("Escriba el numero de neuronas de entrada.\n");
        scanf("%d",&N_IN);
        printf("Escriba el numero de neuronas de la capa oculta.\n");
        scanf("%d",&N_HIDE1);
        printf("Escriba el numero neuronas de salida\n");
        scanf("%d",&N_OUT);
        printf("Escriba el numero de patrones\n");
        scanf("%d",&P);

        FILE *archivo1,*archivo2;
        string output1=to_string("error_xor")+to_string("_")+to_string(
            N_IN)+to_string("_")+to_string(N_HIDE1)+to_string(".dat");
        if((archivo1 = fopen(output1.c_str(), "w")) == NULL){
            printf("No puedo abrir el archivo %s.\n", output1.c_str());
            exit(1);
        }

        string output2=to_string("salidas_xor")+to_string("_")+to_string(
            N_IN)+to_string("_")+to_string(N_HIDE1)+to_string(".dat");
        if((archivo2 = fopen(output2.c_str(), "w")) == NULL){
            printf("No puedo abrir el archivo %s.\n", output1.c_str());
            exit(1);
        }

    double eta = 0.01;
        MatDoub patrones(P,N_IN);
        MatDoub J(N_HIDE1,N_IN),W(N_OUT,N_HIDE1);
        VecDoub umbral1(N_HIDE1);
        double umbral2;
        int estado_umbral=-1;
        double o;
        VecDoub v(N_HIDE1);
        double error,delta;
        VecDoub delta2(N_HIDE1);

        srand(time(0));

        InitWeigh(J);
        InitWeigh(W);
        for(int i=0;i<N_HIDE1;i++)
                umbral1=rand()*1.0/RAND_MAX;
        umbral2 = rand()*1.0/RAND_MAX;
        GeneraPatrones(patrones);
        VecDoub salida_deseada(P);
        for(int i=0;i<P;i++){
                double prod=1;
                for(int j=0;j<N_IN;j++){
                        prod=prod*patrones[i][j];
                        }
                salida_deseada[i]=prod;
                }

        double error_global=1;
        int it=0;
        while(error_global>0.001){
                error_global=0;
```

```
                for(int i=0;i<P;i++){
                        for(int k=0;k<N_HIDE1;k++){
                                v[k]=0;
                                for(int j=0;j<N_IN;j++)
                                        v[k]+=J[k][j]*patrones[i][j];
                                v[k]+=umbral1[k]*estado_umbral;
                                v[k]=tanh(v[k]);
                                }
                        o=0;
                        for(int k=0;k<N_HIDE1;k++)
                                o+=W[0][k]*v[k];
                        o+=umbral2*estado_umbral;
                        o=tanh(o);
                        cout << salida_deseada[i] << " " << o << endl;
                        error = 0.5*(salida_deseada[i]-o)*(salida_deseada
                            [i]-o);
                        error_global+=error;
                        delta = (salida_deseada[i]-o)*(1-o*o);

                        for(int k=0;k<N_HIDE1;k++)
                                delta2[k]=delta*W[0][k]*(1-v[k]*v[k]);

                        for(int k=0;k<N_HIDE1;k++)
                                W[0][k]+=eta*delta*v[k];
                        umbral2+=eta*delta*estado_umbral;

                        for(int k=0;k<N_HIDE1;k++){
                                for(int j=0;j<N_IN;j++)
                                        J[k][j]+=eta*delta2[k]*patrones[i
                                            ][j];
                                umbral1[k]+=eta*delta2[k]*estado_umbral;
                                }
                        }
                fprintf(archivo1,"%i\t%lf\n",it,error_global);
                it++;
                }

        for(int i=0;i<N_HIDE1;i++){
                for(int j=0;j<N_IN;j++)
                        fprintf(archivo2,"%lf\n",J[i][j]);
                fprintf(archivo2,"%lf\n",umbral1[i]);
                }

        for(int i=0;i<N_OUT;i++){
                for(int j=0;j<N_HIDE1;j++)
                        fprintf(archivo2,"%lf\n",W[i][j]);
                fprintf(archivo2,"%lf\n",umbral2);
                }

        return 0;
        }

void InitWeigh(MatDoub &matrix){

        int n = matrix.nrows();
        int m = matrix.ncols();
```

```
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        matrix[i][j] = rand()*1.0/RAND_MAX;
                        }
                }
        }


void GeneraPatrones(MatDoub &matrix){

        int n = matrix.nrows();
        int m = matrix.ncols();

        double aleatorio;
        for(int i=0;i<n;i++){
                for(int j=0;j<m;j++){
                        matrix[i][j]=1;
                        aleatorio = rand()*1./RAND_MAX;
                        if(aleatorio < 0.5)
                                matrix[i][j] = -1;
                        }
                }
        }
```

4.3. Algoritmo para aprender mapeo logístico

```
int main(){

        int N_IN=1,N_HIDE1=5,N_OUT=1,P=100;
        /*printf("Escriba el numero de neuronas de entrada.\n");
        scanf("%d",&N_IN);
        printf("Escriba el numero de neuronas de la capa oculta.\n");
        scanf("%d",&N_HIDE1);
        printf("Escriba el numero neuronas de salida\n");
        scanf("%d",&N_OUT);
        printf("Escriba el numero de patrones\n");
        scanf("%d",&P);*/

        double eta = 0.001;

        VecDoub J(N_HIDE1),W(N_HIDE1);
        double umbral,deltaumbral;
        VecDoub salida_deseada(P);
        VecDoub z(P),o(P);
        VecDoub v(N_HIDE1),delta2(N_HIDE1);
        double delta1;

        srand(time(0));

        for(int i=0;i<N_HIDE1;i++){
                J[i]=rand()*1.0/RAND_MAX;
                W[i]=rand()*1.0/RAND_MAX;
                }
        umbral = rand()*1.0/RAND_MAX;
```

```
for(int k=0;k<P;k++)
        z[k]=k*1.0/P;

double error_patrones=1;
int it=0;
while(error_patrones > 0.05){
        error_patrones=0;
        for(int k=0;k<P;k++){
                salida_deseada[k] = 4*z[k]*(1-z[k]);
                for(int i=0;i<N_HIDE1;i++)
                        v[i]=tanh(J[i]*z[k]);

                o[k]=umbral*z[k];
                for(int i=0;i<N_HIDE1;i++)
                        o[k]+=W[i]*v[i];

                delta1 = (salida_deseada[k]-o[k]);
                for(int i=0;i<N_HIDE1;i++)
                        delta2[i]=W[i]*delta1*(1-v[i]*v[i]);

                for(int i=0;i<N_HIDE1;i++){
                        W[i]+=eta*delta1*v[i];
                        J[i]+=eta*delta2[i]*z[k];
                        }
                umbral+=eta*delta1*z[k];
                error_patrones+=delta1*delta1*0.5;
        }
        //cout << it << " " << error_patrones << endl;
        it++;
        }

double error=0;
for(int i=0;i<P;i++)
        error+=0.5*(salida_deseada[i]-o[i])*(salida_deseada[i]-o[
            i]);
error/=P;
        cout << "Error de aprendizaje = " << error << endl;
        //cout << z[i] << " " << salida_deseada[i] << " " << o[i]
             << endl;

double s,salida;
it=0;
error=0;
double out;
while(it<1000){
        s=rand()*1.0/RAND_MAX;
        out=4*s*(1-s);
        VecDoub v(N_HIDE1);
        for(int i=0;i<N_HIDE1;i++)
                v[i]=tanh(s*J[i]);
        salida=umbral*s;
        for(int i=0;i<N_HIDE1;i++)
                salida+=v[i]*W[i];
        //cout << s << " " << salida << endl;;
        error+=0.5*(out-salida)*(out-salida);
        it++;
```

```
            }
        error/=1000;
        cout << "Error de generalizacion = " << error << endl;

        return 0;
        }
```

Referencias

[1] Introduction to the theory of neural computation. John Hertz,Anders Krogh,Richard G.Palmer. Westview Press (1991).