Curso
2010 | 2011

Desde las RNAs Monocapa hasta la Estructura Multicapa

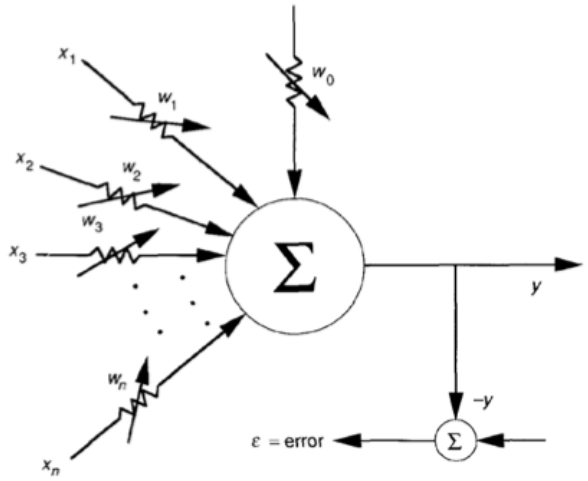
Práctica 2

David Morales Sáez
Alberto Manuel Mireles Suárez

ÍNDICE

- Planteamiento y descripción del problema a resolver

El primer problema a resolver es la resolución de un OR-Exclusivo (XOR) mediante ACLs, primero con una estructura monocapa y luego con una multicapa. Para la creación del entorno en el que estudiaremos el comportamiento de esta operación utilizamos el entorno de desarrollo Qt, basado en el lenguaje C++. El siguiente esquema representa el sistema de aprendizaje y resolución de la RNA.



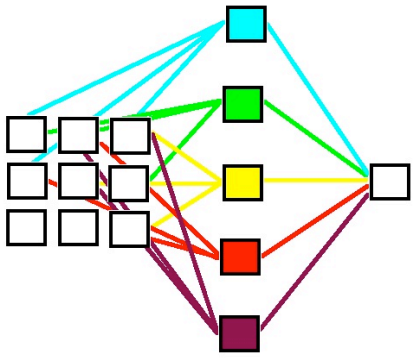
El XOR que hemos implementado sólo tendrá dos entradas binarias y una única salida, como muestra la siguiente tabla de verdad:

Entradas		Salidas
0	0	0
0	1	1
1	0	1
1	1	0

Como se demostrará más adelante, esta función lógica, no es computable en una RNA con una única neurona ACL, sino que debemos utilizar otra más, estando en la capa oculta. Esta neurona servirá de control para anular el sumatorio final en el caso en el que las dos entradas sean positivas.

Entradas			Salidas
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

En el segundo ejercicio, debemos implementar un perceptrón monocapa que distinga entre líneas verticales y horizontales de un patrón cuadrado de rango 3.



La capa de entrada se indica mediante un patrón de rango 3 (como se ha indicado anteriormente) y la capa de salida tendrá una única neurona, que nos indicará si el patrón es horizontal o vertical.

Para el aprendizaje de este perceptrón, utilizaremos los siguientes patrones de entrada dados por el guión de la práctica:

1	1	1
0	0	0
0	0	0

0	0	0
1	1	1
0	0	0

0	0	0
0	0	0
1	1	1

1	0	0
1	0	0
1	0	0

0	1	0
0	1	0
0	1	0

0	0	1
0	0	1
0	0	1

En la capa oculta tendremos cinco neuronas que conectarán la capa de entrada con la de salida de una manera bien definida, es decir, todas las neuronas de la capa oculta no estarán conectadas con todas las neuronas de la capa de entrada, sino de la siguiente manera:

1	0	1
1	0	0
0	0	0

1	0	1
1	0	0
0	0	0

0	0	1
0	1	0
0	0	1

0	1	0
1	0	0
0	0	1

0	1	1
0	1	0
0	0	0

En la neurona de salida, obtendremos dos posibles valores, 1 ó -1, que indicarán si el patrón de entrada es horizontal o vertical, respectivamente.

- Estudio Teórico

- Red monocapa: ACL

Para el primer ejercicio de la práctica hemos implementado un ACL, con su función de activación identidad:

$$\rho = \sum \omega_{ij} * \alpha_i$$

Para ello, hemos utilizado el aprendizaje conocido como Regla Delta, cuya modificación de pesos se basa en la siguiente fórmula:

$$\omega_{ij}^{n} = \omega_{ij}^{v} + \alpha * \delta * x_i$$

siendo ω_{ij}^{n} el peso a calcular, ω_{ij}^{v} el peso actual, α el ratio de aprendizaje, δ la diferencia entre la salida esperada y la salida actual y x_i la entrada a la neurona en ese instante.

Para conseguir que la RNA aprenda y se ejecute de la manera deseada, utilizaremos unos patrones de entrada bien definidos (la tabla de verdad), los cuales se presentarán a la red.

La regla delta se basa en calcular el error cometido con respecto a la salida esperada (δ) y calcular los nuevos pesos en base a éste. Una vez modificados los pesos, volveremos a iterar, rehaciendo el cálculo. Esto se hará tantas veces como se desee. Una vez que el error cometido sea 0, la red habrá obtenido los pesos óptimos, pero como comprobaremos, esto no se dará nunca, ya que esta función no es resoluble con un único sumatorio pesado.

- Red multicapa

En este caso, añadiremos a la red una neurona que imite la función lógica AND, facilitando la resolución del problema. Utilizaremos el mismo aprendizaje que en el caso anterior y podremos ver que esta vez el error tiende a 0.

- Perceptrón

El perceptrón es una RNA capaz de distinguir distintos tipos de patrones, en nuestro caso distingue entre líneas horizontales y verticales. El perceptrón consta de tres capas: una de entrada, una oculta y una de salida. La capa de entrada, también conocida como retina, es la capa por la cual introducimos el patrón a estudiar, y consta de 9 neuronas. La capa oculta es la capa que conecta tanto la entrada como la salida, haciendo los cálculos pertinentes para la resolución del problema. En nuestro caso tendrá 5 neuronas, ya que es el número de distintas conexiones definidas por el guión de la práctica. La capa de salida es la que nos muestra el resultado de la red, por lo que está formada por una única neurona.

El estímulo de entrada (patrón de entrada) incide sobre las unidades de la retina de la red, provocando así la activación de estas unidades sensitivas. Estos impulsos son enviados a la capa oculta. Si la suma de señales que llegan a cada unidad supera su umbral se activará la unidad, en caso contrario, se inhibirá. Una vez activadas las correspondientes unidades de la capa oculta, se computará la salida mediante un sumatorio pesado. En el caso en el que este sumatorio sea superior a un umbral, indicará que el patrón de entrada será horizontal, en caso contrario, será vertical.

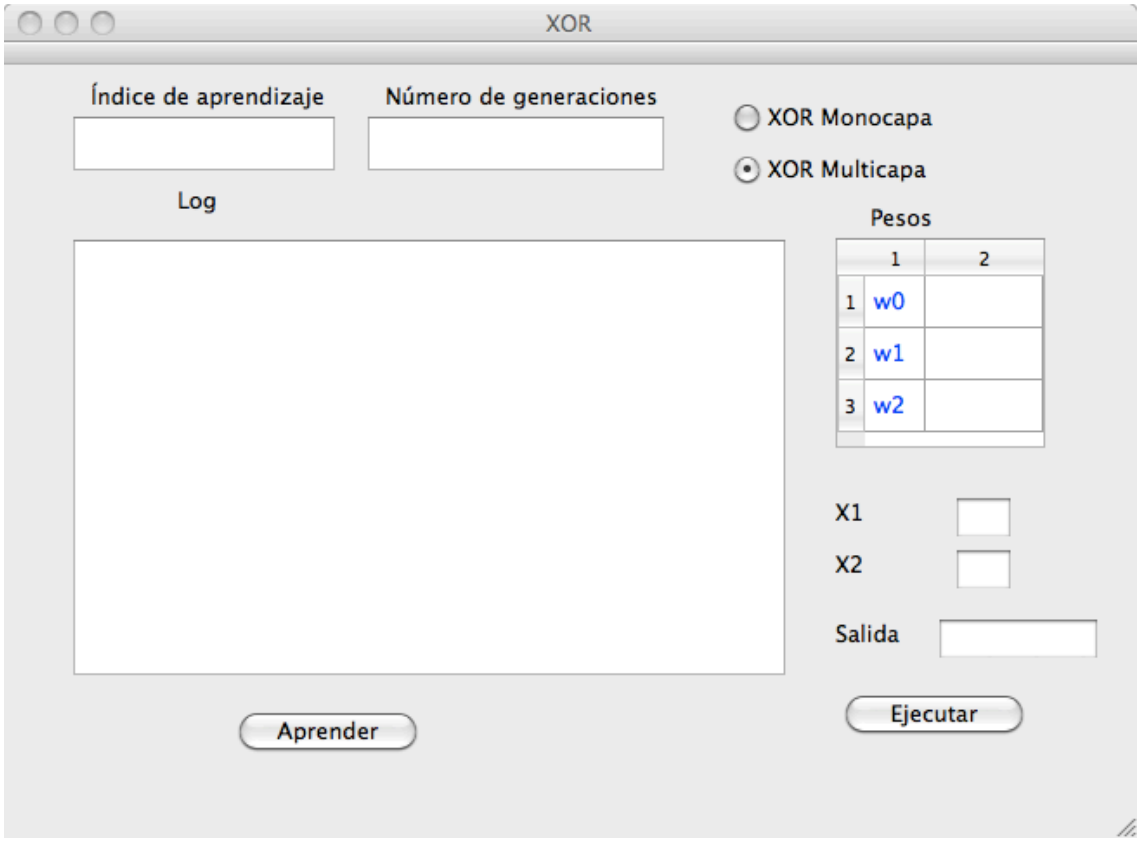
Al igual que en los anteriores ejercicios, utilizaremos la regla delta (explicado anteriormente) como proceso de aprendizaje.

- Descripción del desarrollo de la práctica y su implementación

Debido a la amplitud del código, se ha incluido el mismo en un anexo.

- XOR Monocapa/Multicapa

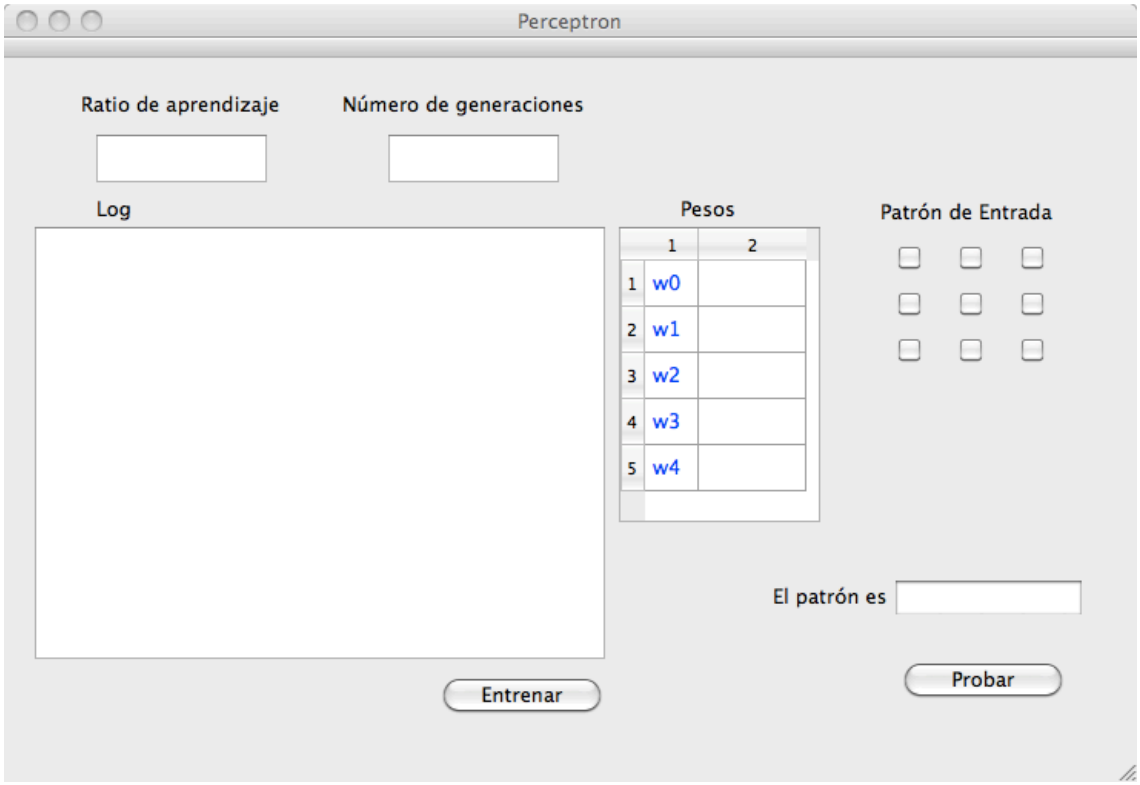
Para el estudio de este problema hemos creado una aplicación que permite realizar esta operación tanto con una red monocapa como multicapa. Como podremos ver en la siguiente imagen, la interfaz permite manejar gran cantidad de parámetros, desde el índice de aprendizaje hasta el número de generaciones en la que aprenderá la red. Además, permitimos ejecutar la red con una entrada ya definida para su uso:



En el campo índice de aprendizaje, debemos definir el ratio deseado; en número de generaciones indicamos la cantidad de iteraciones que hará; el Log mostrará la evolución del error a lo largo del tiempo. En la parte superior derecha podemos escoger entre las dos distintas redes, monocapa o multicapa. La tabla de pesos mostrada indicará los pesos que hay en el instante (w2 solo variará en el caso multicapa). Finalmente, X1 y X2 serán las entradas de la red para someterla a una ejecución, y salida será el resultado que produzca la misma. Hemos de indicar que la salida muy rara vez será un entero, sino un valor aproximado al mismo, siempre y cuando la red halla aprendido adecuadamente.

- Perceptrón

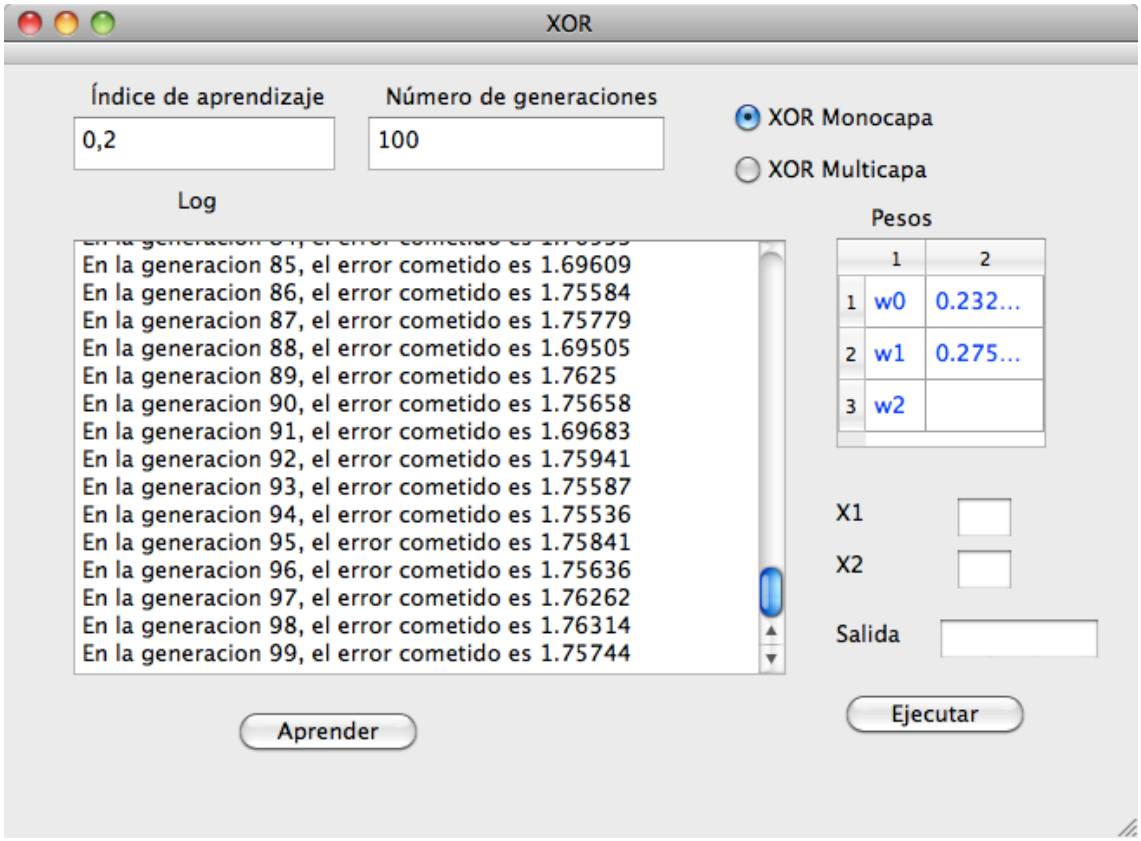
Para este problema hemos creado una aplicación distinta, donde podremos tanto ver el aprendizaje de la red, como utilizarla.

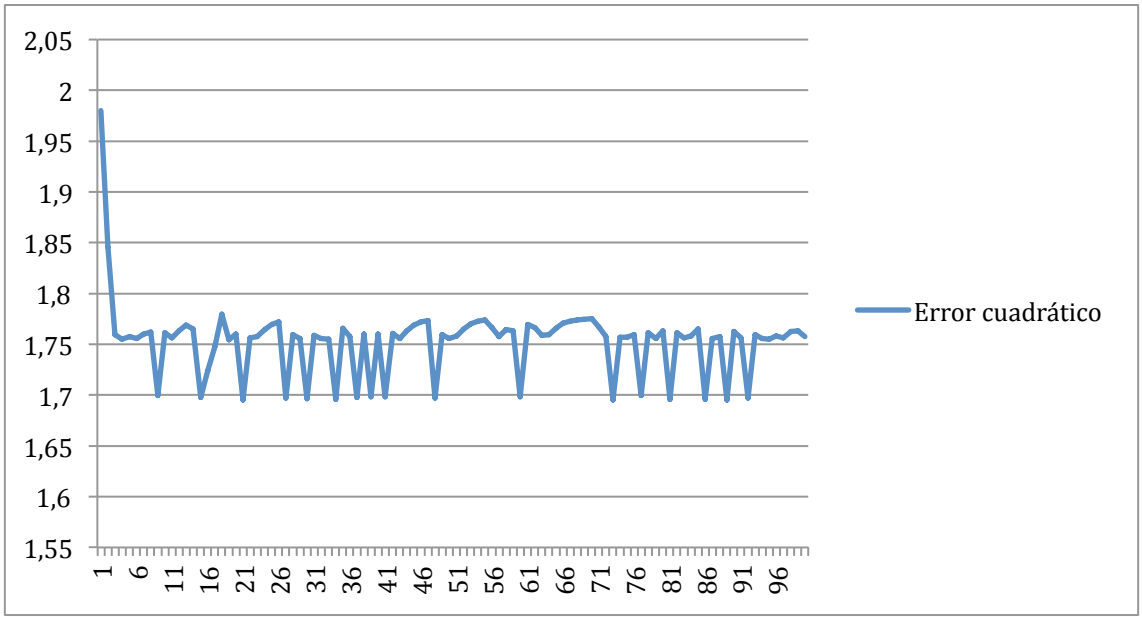


- Resultados y conclusiones

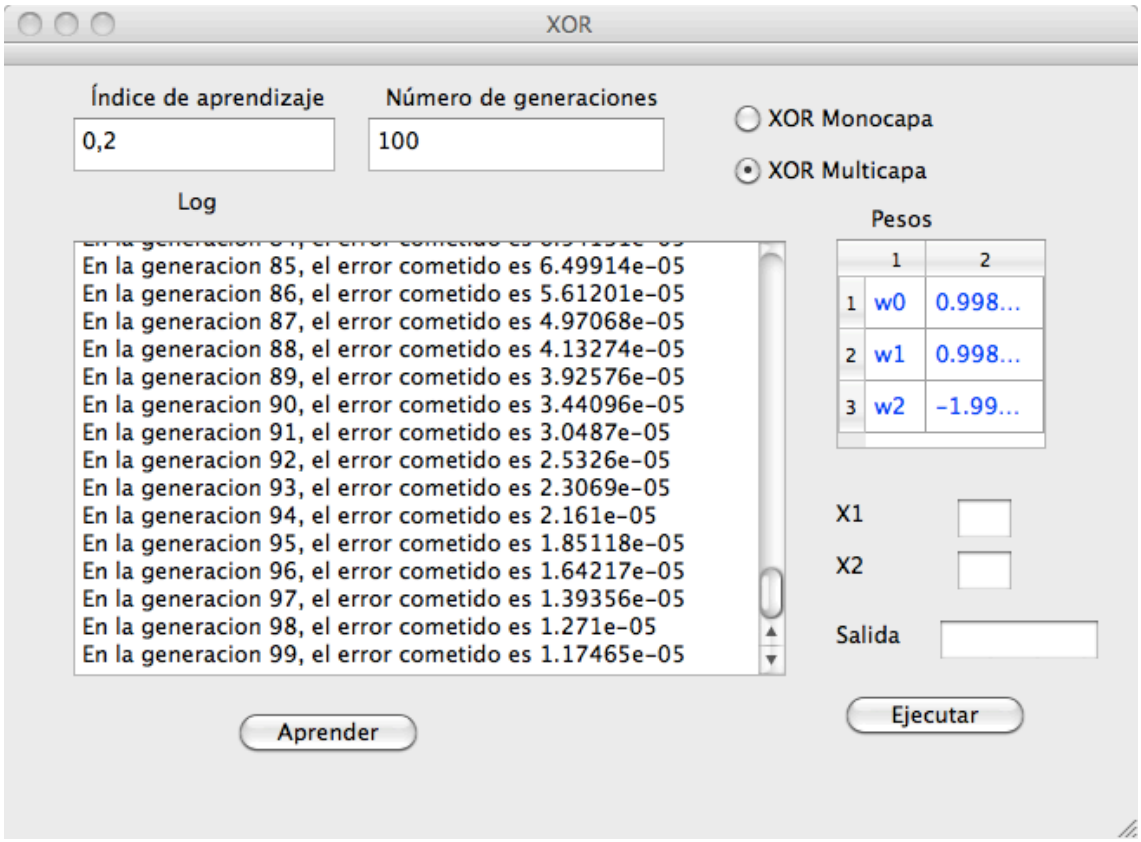
- Primer ejercicio

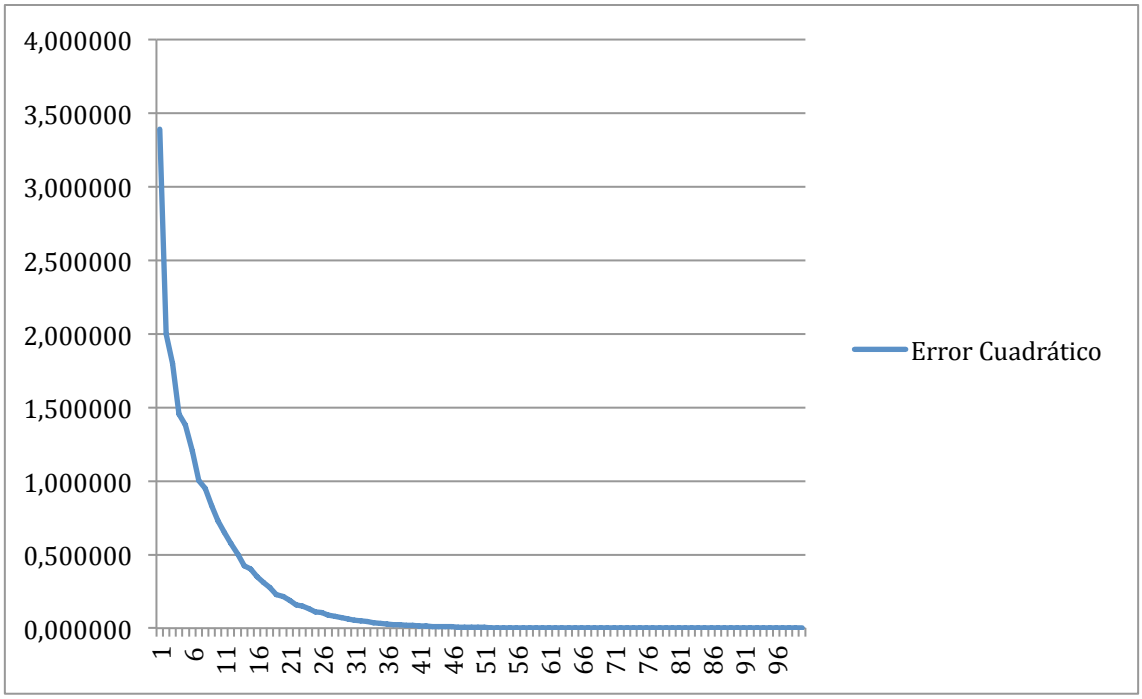
En el XOR monocapa, no es posible calcular la función lógica con una única neurona, como podemos comprobar en los datos obtenidos, ya que el error nunca tiende a 0.



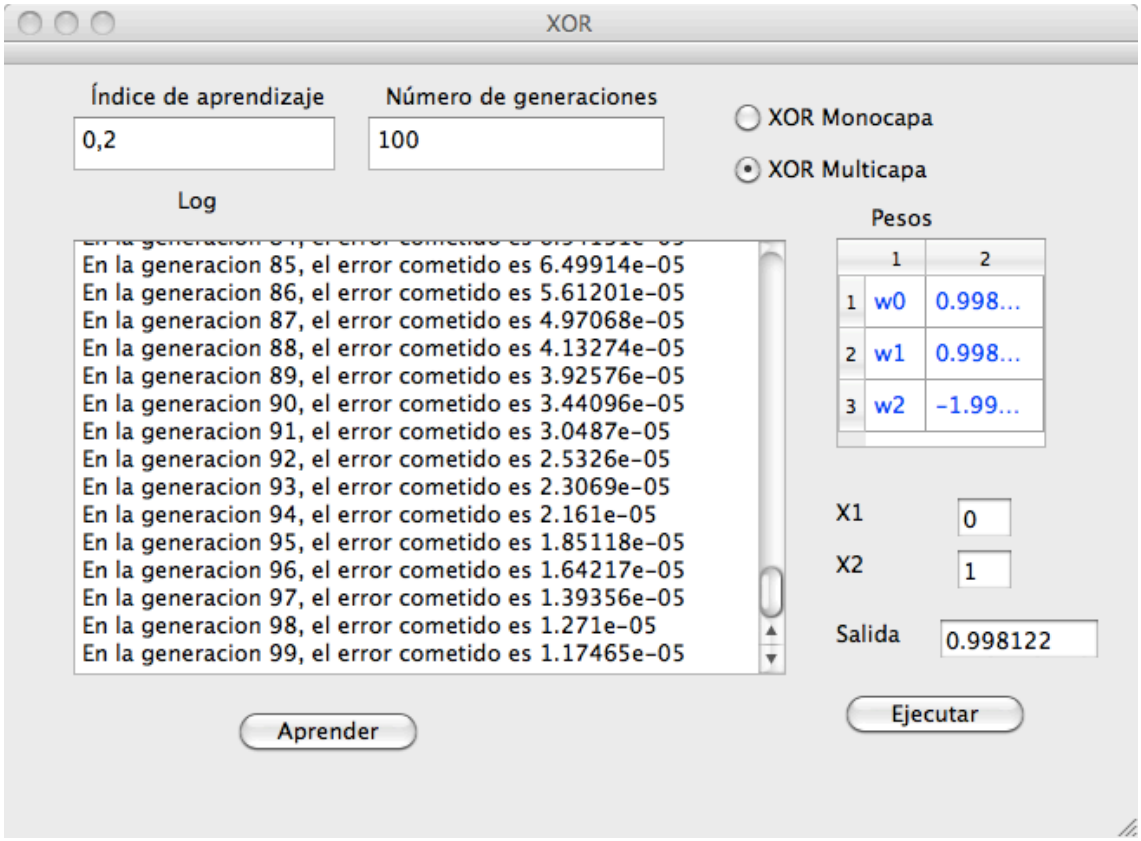


En el XOR multicapa, si es posible (y lo hacemos) calcular la función lógica con una red multicapa, como podemos comprobar con la convergencia del error a 0. Los pesos finales obtenidos al final del aprendizaje se muestran en la imagen la interfaz obtenida:



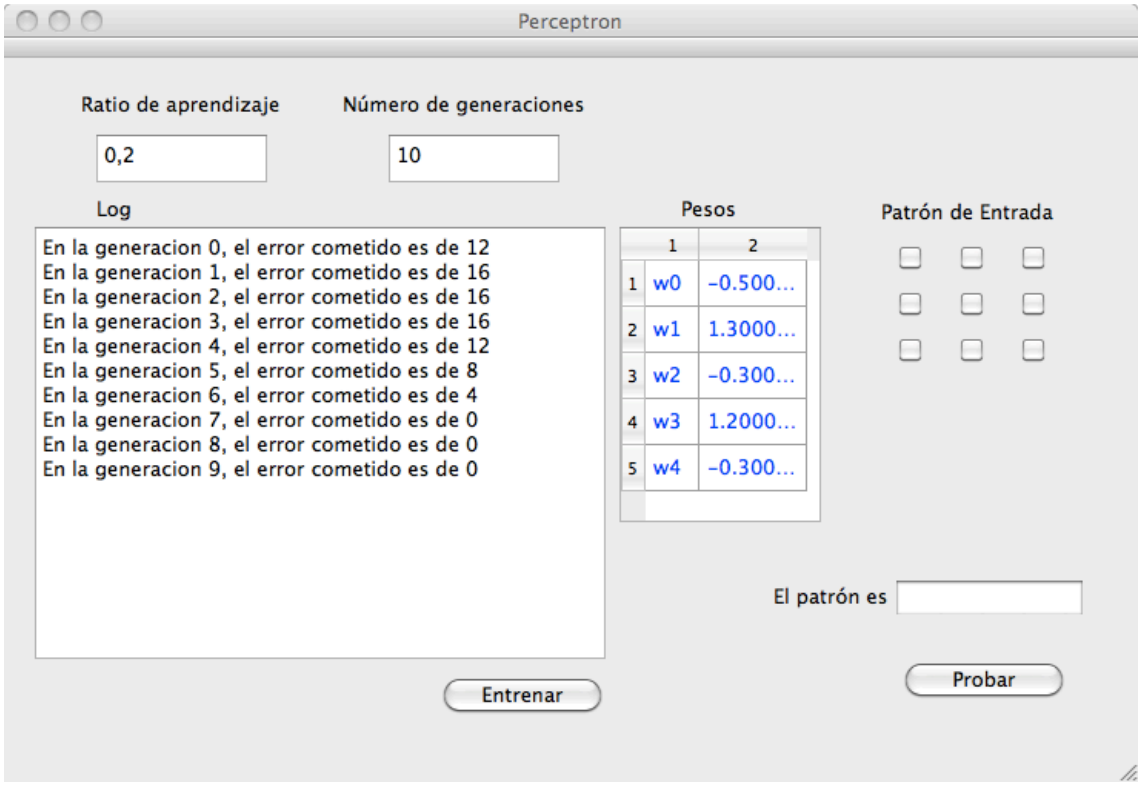


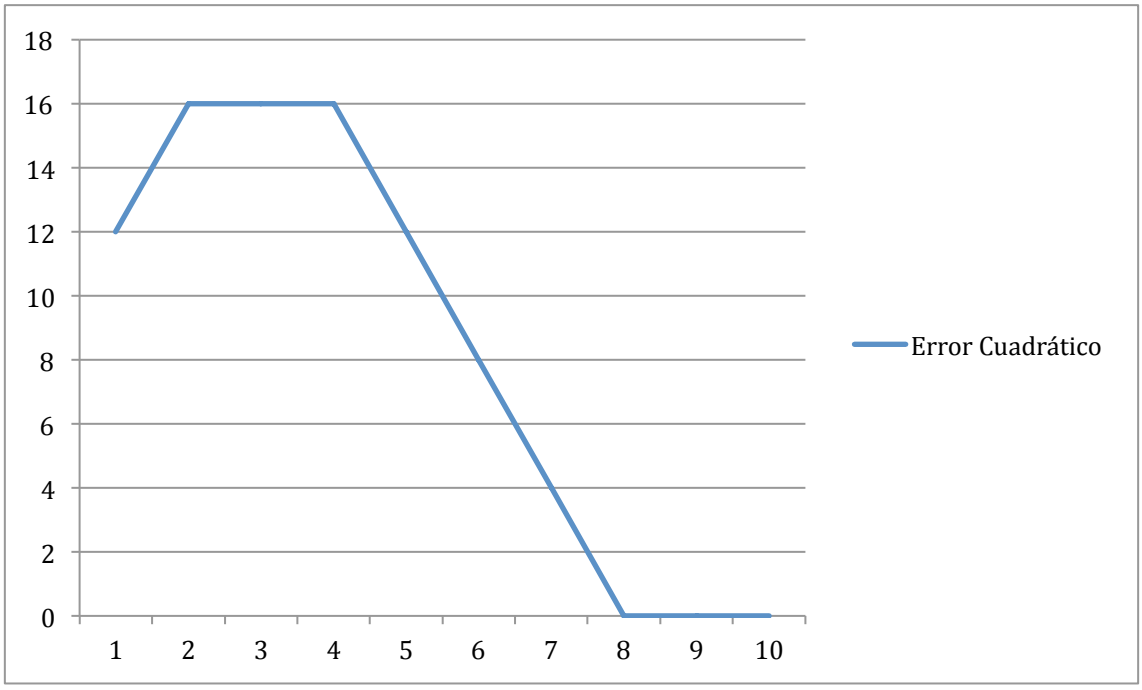
También añadir que, una vez completado el aprendizaje, se puede utilizar la red para el cálculo de la función XOR:



- Segundo ejercicio

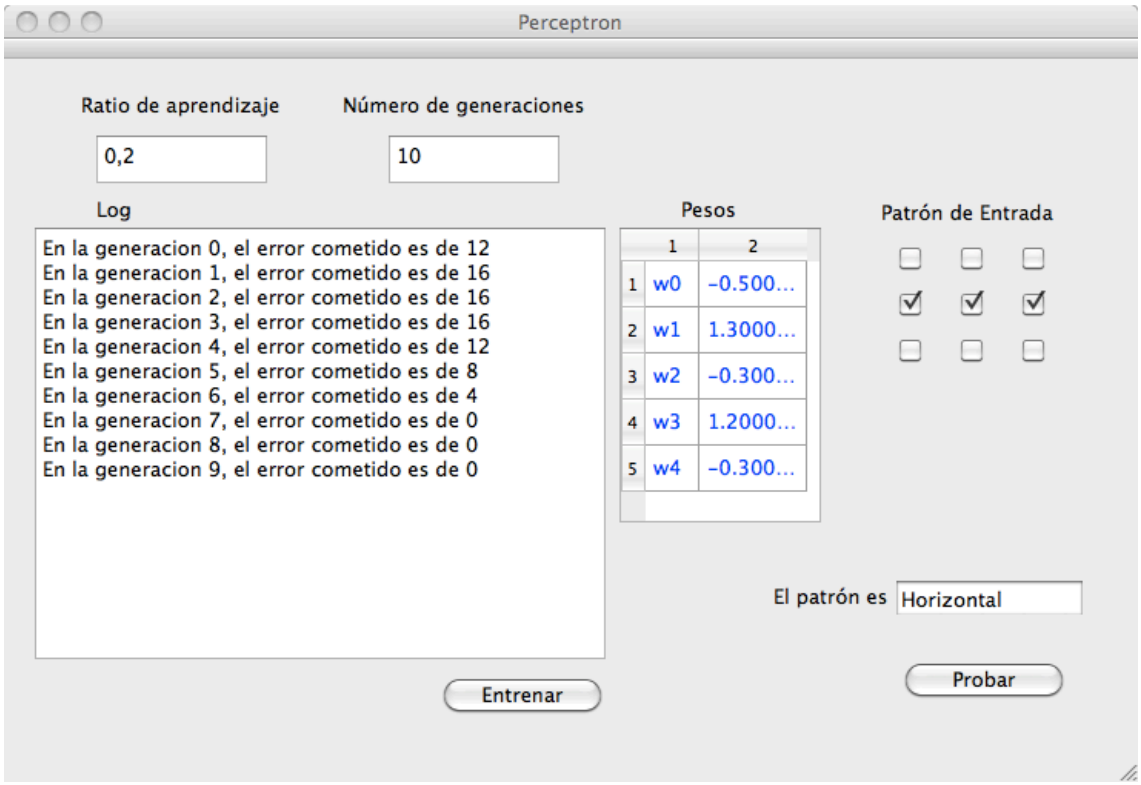
Con el perceptrón, hemos conseguido que la red se estabilice en sólo 7 iteraciones.

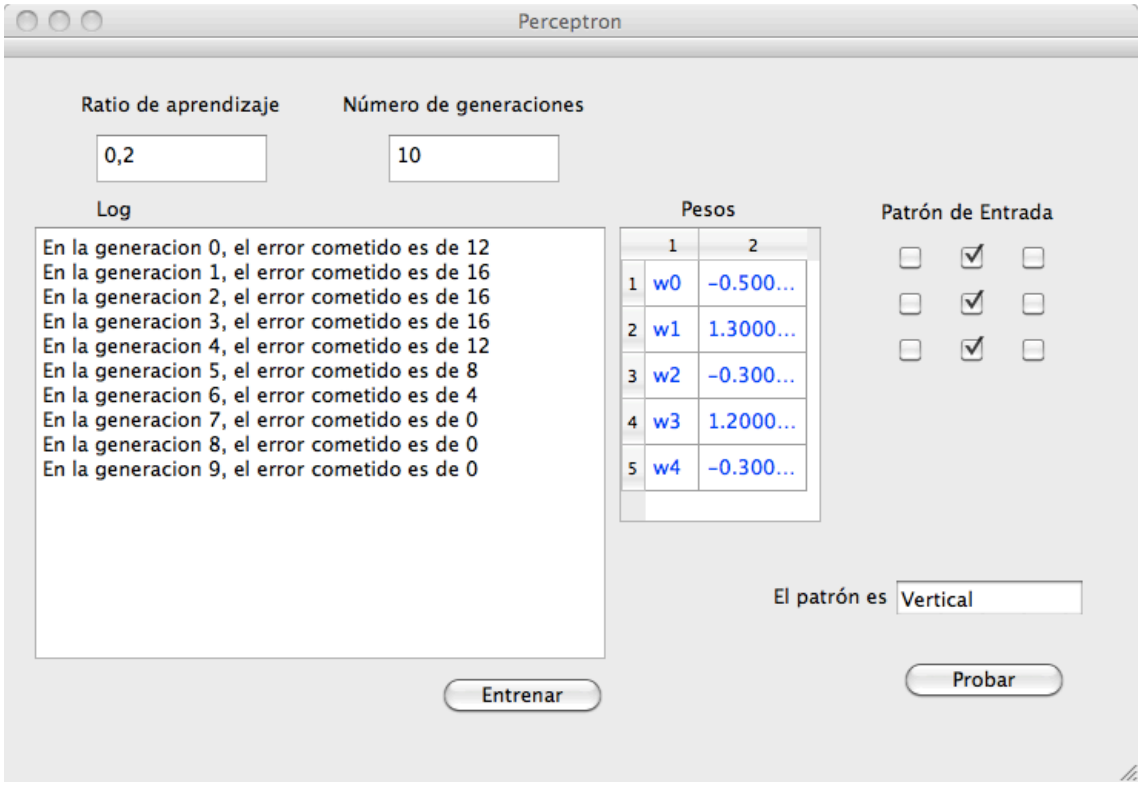




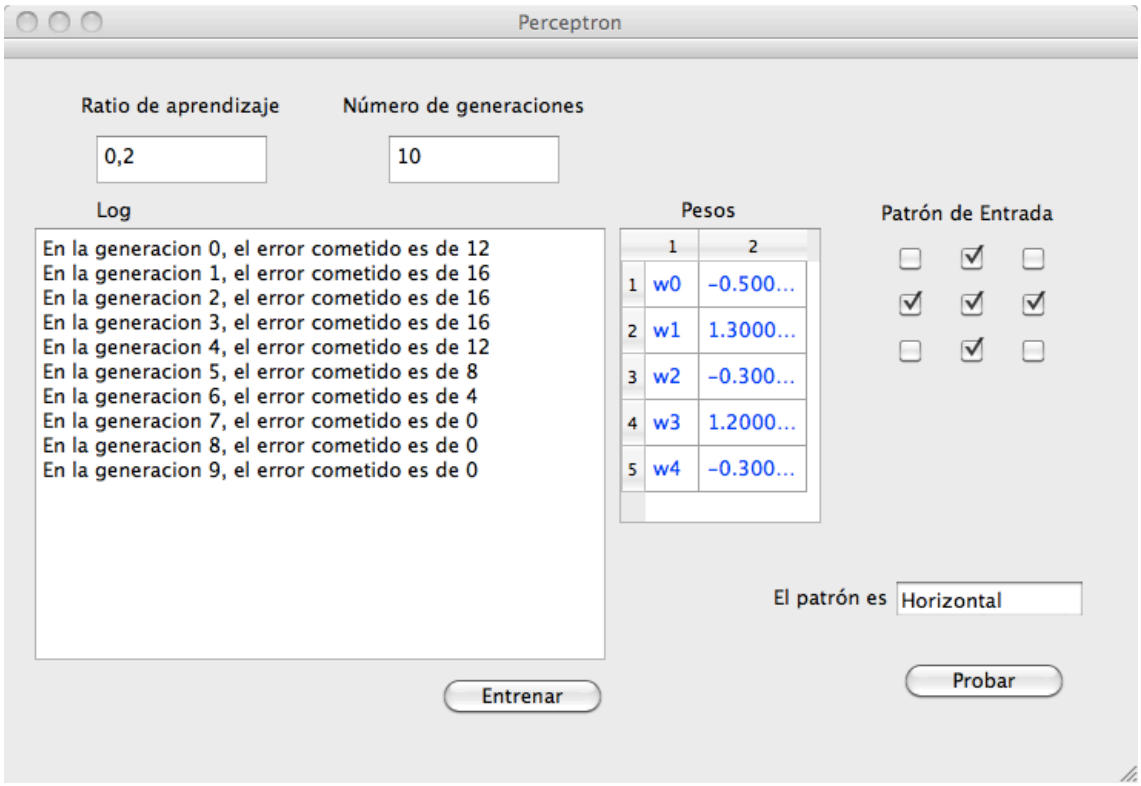
Como podemos observar, el error cometido a lo largo de las iteraciones acaba tendiendo a 0, es decir, la red ha aprendido. A su vez, cuando probamos la red

sometiéndole a su entrada un patrón horizontal o vertical, vemos que es capaz de reconocerlos.





En el caso en el que introduzcamos un patrón que la red no conoce, la red falla pues no generaliza ya que no reconoce líneas que no estén bien definidas.



- Bibliografía

-Apuntes de la asignatura

-Raul Rojas – Neural Networks

-James A. Freeman – Simulating Neural Networks with Matematica

- Anexo

Xor

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

double pesos_mon[2];
double pesos_mul[3];

MainWindow::MainWindow(QWidget *parent) :
QMainWindow(parent),
ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    // Inicializamos y mostramos la tabla de pesos de la
interfaz
    ui->pesos->setRowCount(3);
    ui->pesos->setColumnCount(2);
    ui->pesos->setColumnWidth(0, 35);
    ui->pesos->setColumnWidth(1, 68);

    QTableWidgetItem *letras0 = new QTableWidgetItem("w0");
    QTableWidgetItem *letras1 = new QTableWidgetItem("w1");
    QTableWidgetItem *letras2 = new QTableWidgetItem("w2");

    letras0->setFlags(letras0->flags() & (~Qt::ItemIsEditable));
    letras0->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(0,0,letras0);

    letras1->setFlags(letras1->flags() & (~Qt::ItemIsEditable));
    letras1->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(1,0,letras1);

    letras2->setFlags(letras2->flags() & (~Qt::ItemIsEditable));
    letras2->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(2,0,letras2);

    // Inicializamos los vectores de pesos
    pesos_mon[0] = pesos_mon[1] = 0.5;
    pesos_mul[0] = pesos_mul[1] = pesos_mul[2] = 0.5;
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::neurona_mono(int ciclos,double ratio_aprend){

    //Inicializacion
    double patrones[4][2] = {{0.,0.},
     {0.,1.},
     {1.,0.},
     {1.,1.}};
    double peso[2]={0.5,0.5};
```

```
    double salida[4] = {0.,1.,1.,0.};
    double Salida_Act, delta, error;

    //mostrar el error para ver si converge
    for (int i=0;i<ciclos;i++){
        int Ciclo_Act=0;
        int j=rand()%4; //Patron aleatorio incial
        error=0.;
        while(Ciclo_Act < 4){
            Salida_Act=0.;
            for(int s=0;s<2;s++) //Salida de cada iteración
                Salida_Act+=peso[s]*patrones[j][s];
            //Calculo del error
            delta=salida[j]-Salida_Act;
            //Actualizamos los pesos
            for(int s=0;s<2;s++)
                peso[s]+=ratio_aprend*delta*patrones[j][s];
            //Elegimos otro patrón
            j=(j+1)%4;
            error += delta*delta;
            Ciclo_Act++;
        }
        //Imprimimos el error en el LOG
        ui->log->append(QString("En la generacion %1, el error
    cometido es %2").arg(i).arg(error));
    }
    //mostrar pesos

    QString proba;

    QTableWidgetItem *prob0 = new
QTableWidgetItem(proba.sprintf("%f",peso[0]));
    QTableWidgetItem *prob1 = new
QTableWidgetItem(proba.sprintf("%f",peso[1]));



    prob0->setFlags(prob0->flags() & (~Qt::ItemIsEditable));
    prob0->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(0,1,prob0);

    prob1->setFlags(prob1->flags() & (~Qt::ItemIsEditable));
    prob1->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(1,1,prob1);

    // Actualizamos el vector de pesos global
    pesos_mon[0] = peso[0];
    pesos_mon[1] = peso[1];

}


void MainWindow::neurona_multi(int ciclos,double ratio_aprend){
    //Inicializamos
    double patrones[4][3] = {{0.,0.,0.},
     {0.,1.,0.},
     {1.,0.,0.},
```

```
      {1.,1.,1.}};
    double peso[3]={0.5,0.5,0.5};
    double salidas[4]={0.,1.,1.,0.};
    double Salida_Act, delta, error;

    for(int i=0;i<ciclos;i++){
        int Ciclo_Act=0;
        int j=rand()%4; //patrón aleatorio
        error=0.;
        while(Ciclo_Act<4){
            Salida_Act=0.;
            for(int s=0;s<3;s++) //Salida de la iteracion
                Salida_Act+=peso[s]*patrones[j][s];
            //Error de la iteracion
            delta=salidas[j]-Salida_Act;
            //Actualizamos pesos
            for(int s=0;s<3;s++)
                peso[s]+=ratio_aprend*delta*patrones[j][s];
            //Siguiente patrón
            j=(j+1)%4;
            //Acumula error cuadrático
            error+=delta*delta;
            Ciclo_Act++;
        }
        // Mostramos el error cuadrático cometido
        ui->log->append(QString("En la generacion %1, el error
cometido es %2").arg(i).arg(error));
    }
    //mostrar pesos("En la generación %d, el error cometido
es %f",ciclos,error);

    QString proba;

    QTableWidgetItem *prob0 = new
QTableWidgetItem(proba.sprintf("%f",peso[0]));
    QTableWidgetItem *prob1 = new
QTableWidgetItem(proba.sprintf("%f",peso[1]));
    QTableWidgetItem *prob2 = new
QTableWidgetItem(proba.sprintf("%f",peso[2]));



    prob0->setFlags(prob0->flags() & (~Qt::ItemIsEditable));
    prob0->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(0,1,prob0);

    prob1->setFlags(prob1->flags() & (~Qt::ItemIsEditable));
    prob1->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(1,1,prob1);

    prob2->setFlags(prob2->flags() & (~Qt::ItemIsEditable));
    prob2->setTextColor(Qt::blue); // color de los items
    ui->pesos->setItem(2,1,prob2);

    // Actualizamos el vector de pesos global
    pesos_mul[0] = peso[0];
    pesos_mul[1] = peso[1];
```

```
    pesos_mul[2] = peso[2];
}

void MainWindow::on_pushButton_clicked()
{
    double aprendizaje;
    int ciclos;
    // Limpiamos la ventana de Log
    ui->log->clear();
    aprendizaje=ui->aprend->toPlainText().toDouble();
    ciclos=ui->gener->toPlainText().toInt();
    if (aprendizaje>0 && ciclos>0){
        if (ui->mono->isChecked()) // Iniciamos la red monocapa
            neurona_mono(ciclos,aprendizaje);
        else // Iniciamos la red multicapa
            neurona_multi(ciclos,aprendizaje);
    }
}

// Nos aseguramos que sólo se active una de las dos opciones
void MainWindow::on_mono_clicked()
{
        ui->multi->setDown(true);
}

void MainWindow::on_multi_clicked()
{
        ui->mono->setDown(true);
}

// ejecutamos la red
void MainWindow::on_pushButton_2_clicked()
{
    int X1 = ui->x1->text().toInt();
    int X2 = ui->x2->text().toInt();
    double salida = 0.;
    if(ui->mono->isChecked()) // En el caso que la red sea
monocapa
        salida = X1*pesos_mon[0]+X2*pesos_mon[1];
    else
    {   // En el caso que la red sea multicapa
        int X3 = X1 & X2;
        salida = X1*pesos_mul[0] + X2*pesos_mul[1] +
X3*pesos_mul[2];
    }
    ui->out->setText(QString("%1").arg(salida));
}
```

Perceptrón

```
#include "perceptron.h"
#include "ui_perceptron.h"

double pesitos[5];


Perceptron::Perceptron(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::Perceptron)
{
    ui->setupUi(this);

    // Inicializamos la tabla de pesos de la interfaz
    ui->tpesos->setRowCount(5);
    ui->tpesos->setColumnCount(2);
    ui->tpesos->setColumnWidth(0, 35);
    ui->tpesos->setColumnWidth(1, 70);

    QTableWidgetItem *letras0 = new QTableWidgetItem("w0");
    QTableWidgetItem *letras1 = new QTableWidgetItem("w1");
    QTableWidgetItem *letras2 = new QTableWidgetItem("w2");
    QTableWidgetItem *letras3 = new QTableWidgetItem("w3");
    QTableWidgetItem *letras4 = new QTableWidgetItem("w4");

    letras0->setFlags(letras0->flags() & (~Qt::ItemIsEditable));
    letras0->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(0,0,letras0);

    letras1->setFlags(letras1->flags() & (~Qt::ItemIsEditable));
    letras1->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(1,0,letras1);

    letras2->setFlags(letras2->flags() & (~Qt::ItemIsEditable));
    letras2->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(2,0,letras2);

    letras3->setFlags(letras3->flags() & (~Qt::ItemIsEditable));
    letras3->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(3,0,letras3);

    letras4->setFlags(letras4->flags() & (~Qt::ItemIsEditable));
    letras4->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(4,0,letras4);

    // Inicializamos el vector de pesos global
    pesitos[0]= 0.5;
    pesitos[1]= 0.3;
    pesitos[2]= 0.2;
    pesitos[3]= 0.7;
    pesitos[4]= 0.9;

}

Perceptron::~Perceptron()
{
```

```
    delete ui;
}

void Perceptron::entrenar(int iter, double pesos[]){

    patron patrones[6];
    // Creamos y definimos los patrones de entrada para el
aprendizaje
    patrones[0].mat[0][0] = patrones[0].mat[0][1] =
patrones[0].mat [0][2] = 1;
    patrones[0].mat[1][0] = patrones[0].mat[1][1] =
patrones[0].mat [1][2] = 0;
    patrones[0].mat[2][0] = patrones[0].mat[2][1] =
patrones[0].mat [2][2] = 0;

    patrones[1].mat[0][0] = patrones[1].mat[0][1] =
patrones[1].mat [0][2] = 0;
    patrones[1].mat[1][0] = patrones[1].mat[1][1] =
patrones[1].mat [1][2] = 1;
    patrones[1].mat[2][0] = patrones[1].mat[2][1] =
patrones[1].mat [2][2] = 0;

    patrones[2].mat[0][0] = patrones[2].mat[0][1] =
patrones[2].mat [0][2] = 0;
    patrones[2].mat[1][0] = patrones[2].mat[1][1] =
patrones[2].mat [1][2] = 0;
    patrones[2].mat[2][0] = patrones[2].mat[2][1] =
patrones[2].mat [2][2] = 1;

    patrones[3].mat[0][0] = patrones[3].mat[1][0] =
patrones[3].mat [2][0] = 1;
    patrones[3].mat[0][1] = patrones[3].mat[1][1] =
patrones[3].mat [2][1] = 0;
    patrones[3].mat[0][2] = patrones[3].mat[1][2] =
patrones[3].mat [2][2] = 0;

    patrones[4].mat[0][0] = patrones[4].mat[1][0] =
patrones[4].mat [2][0] = 0;
    patrones[4].mat[0][1] = patrones[4].mat[1][1] =
patrones[4].mat [2][1] = 1;
    patrones[4].mat[0][2] = patrones[4].mat[1][2] =
patrones[4].mat [2][2] = 0;

    patrones[5].mat[0][0] = patrones[5].mat[1][0] =
patrones[5].mat [2][0] = 0;
    patrones[5].mat[0][1] = patrones[5].mat[1][1] =
patrones[5].mat [2][1] = 0;
    patrones[5].mat[0][2] = patrones[5].mat[1][2] =
patrones[5].mat [2][2] = 1;

    //Inicializamos las salidas
    int salida[6]={1,1,1,-1,-1,-1};

    double hide[5];

    double resultado=0;
    double alfa=ui->ratio->toPlainText().toDouble();
```

```
    int j=0;
    int cont=0;
    double delta=0.;
    double error=0.;

    for (int i=0;i<iter;i++){

        j=0;
        while(cont<6){
            resultado=0.;
            // Insertamos en el vector hide los valores de las
conexiones entre la entrada
            // y la capa oculta

hide[0]=(patrones[j].mat[0][0]+patrones[j].mat[0][2]+patrones[j]
.mat[1][0]);

hide[1]=(patrones[j].mat[0][0]+patrones[j].mat[0][1]+patrones[j]
.mat[1][2]);

hide[2]=(patrones[j].mat[0][2]+patrones[j].mat[1][1]+patrones[j]
.mat[2][2]);

hide[3]=(patrones[j].mat[0][1]+patrones[j].mat[1][0]+patrones[j]
.mat[2][2]);

hide[4]=(patrones[j].mat[0][1]+patrones[j].mat[0][2]+patrones[j]
.mat[1][1]);

            // Calculamos el resultado total del sumatorio
pesado en la salida
            for(int i=0;i<5;i++){
                int umbral=(i==3)?1:2;
                hide[i]=(hide[i]>=umbral)?1:0;
                resultado+=pesos[i]*hide[i];
            }
            // Comprobamos con el umbral y separamos
            resultado=(resultado>=1)?1:-1;
            delta=salida[j]-resultado;
            error+=delta*delta;
            cont++;
            // Modificamos los pesos de la red
            for(int i=0;i<5;i++){
                pesos[i]+=alfa*delta*hide[i];
            }
            j++;
        }
        //mostrar error
        ui->log->append(QString("En la generacion %1, el error
cometido es de %2").arg(i).arg(error));
        error=0.;
        cont=0;
    }
    //actualizamos pesos en global
    pesitos[0]= pesos[0];
    pesitos[1]= pesos[1];
    pesitos[2]= pesos[2];
```

```
    pesitos[3]= pesos[3];
    pesitos[4]= pesos[4];
    //mostrar pesos
    QString proba;

    QTableWidgetItem *prob0 = new
QTableWidgetItem(proba.sprintf("%f",pesos[0]));
    QTableWidgetItem *prob1 = new
QTableWidgetItem(proba.sprintf("%f",pesos[1]));
    QTableWidgetItem *prob2 = new
QTableWidgetItem(proba.sprintf("%f",pesos[2]));
    QTableWidgetItem *prob3 = new
QTableWidgetItem(proba.sprintf("%f",pesos[3]));
    QTableWidgetItem *prob4 = new
QTableWidgetItem(proba.sprintf("%f",pesos[4]));

    prob0->setFlags(prob0->flags() & (~Qt::ItemIsEditable));
    prob0->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(0,1,prob0);

    prob1->setFlags(prob1->flags() & (~Qt::ItemIsEditable));
    prob1->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(1,1,prob1);

    prob2->setFlags(prob2->flags() & (~Qt::ItemIsEditable));
    prob2->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(2,1,prob2);

    prob3->setFlags(prob3->flags() & (~Qt::ItemIsEditable));
    prob3->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(3,1,prob3);

    prob4->setFlags(prob4->flags() & (~Qt::ItemIsEditable));
    prob4->setTextColor(Qt::blue); // color de los items
    ui->tpesos->setItem(4,1,prob4);

}

int Perceptron::calculo(int patron[][3], double pesos[5]){

    double hide[5];
    double resultado=0;
    // Calculamos los valores de la capa oculta
    hide[0]=(patron[0][0]+patron[0][2]+patron[1][0]);
    hide[1]=(patron[0][0]+patron[0][1]+patron[1][2]);
    hide[2]=(patron[0][2]+patron[1][1]+patron[2][2]);
    hide[3]=(patron[0][1]+patron[1][0]+patron[2][2]);
    hide[4]=(patron[0][1]+patron[0][2]+patron[1][1]);

    // Calculamos el valor en la salida
    for(int i=0;i<5;i++){
        int umbral=(i==3)?1:2;
        hide[i]=(hide[i]>=umbral)?1:0;
        resultado+=pesos[i]*hide[i];
    }
    resultado=(resultado>=1)?1:-1;
    return resultado;
```

```
}

void Perceptron::on_pushButton_clicked()
{
    // Inicializamos los valores del vector
    // de pesos global para el aprendizaje
    ui->log->clear();
    pesitos[0]= 0.3;
    pesitos[1]= 0.5;
    pesitos[2]= 0.1;
    pesitos[3]= 0.8;
    pesitos[4]= 0.9;
    entrenar(ui->gener->toPlainText().toInt(), pesitos);
}

void Perceptron::on_pushButton_2_clicked()
{
    patron entrada;
    // Obtenemos el patrón de entrada
    if(ui->c00->isChecked()) entrada.mat[0][0]=1;
    else entrada.mat[0][0]=0;
    if(ui->c01->isChecked()) entrada.mat[0][1]=1;
    else entrada.mat[0][1]=0;
    if(ui->c02->isChecked()) entrada.mat[0][2]=1;
    else entrada.mat[0][2]=0;
    if(ui->c10->isChecked()) entrada.mat[1][0]=1;
    else entrada.mat[1][0]=0;
    if(ui->c11->isChecked()) entrada.mat[1][1]=1;
    else entrada.mat[1][1]=0;
    if(ui->c12->isChecked()) entrada.mat[1][2]=1;
    else entrada.mat[1][2]=0;
    if(ui->c20->isChecked()) entrada.mat[2][0]=1;
    else entrada.mat[2][0]=0;
    if(ui->c21->isChecked()) entrada.mat[2][1]=1;
    else entrada.mat[2][1]=0;
    if(ui->c22->isChecked()) entrada.mat[2][2]=1;
    else entrada.mat[2][2]=0;

    ui->out->clear();
    int vsalida=calculo(entrada.mat, pesitos);
    // Mostramos el tipo de patrón
    if(vsalida==1) ui->out->setText(QString("Horizontal"));
    else
        if(vsalida==-1)ui->out->setText(QString("Vertical"));


}
```