```c
/*rete di tre layers; layer 1 con 2 neuroni; layer 2 con 5 (ne ho provati 200) neuroni; layer 3 con 1 neurone*/

#include<stdio.h>
#include<stdlib.h>
#include<math.h>
#include<time.h>

// Precisione dei valori
typedef double T_Precision;


// Struttura di un neurone
struct Neuron
{
 T_Precision value;// Uscita del percettrone
 T_Precision dEdy;//Errore del neurone
};
struct Neuron *i_neuron, *h_neuron, *o_neuron;//variabili di tipo Neuron


// Struttura di una connessione tra due neuroni (sinapsi)
struct Synapse
{
 T_Precision weight; // Peso della connessione
 //T_Precision dEdw;//Errore della connessione
};
struct Synapse *h_synapse, *o_synapse;//variabili di tipo Synapse




//DICHIARAZIONE FUNZIONI

float sigmoid(float x);//dichiarazione funzione sigmoide

float d_sigmoid(float x);//dichiarazione funzione derivata della sigmoide

void run_network( struct Neuron *i_neuron, size_t input_size,    //dichiarazione funzione che fa operare la rete di 3 strati,calcola le uscite di
struct Neuron *h_neuron, struct Synapse *h_synapse, size_t hidden_size, //ogni strato, un percettrone alla volta, fino a determinare le uscite
struct Neuron *o_neuron, struct Synapse *o_synapse, size_t output_size );//della rete

void init_weight( struct Synapse *synapse, size_t layer_size, size_t prev_layer_size );//dichiarazione funzione inizializzazione pesi

void backpropagation( struct Neuron *i_neuron, size_t input_size,
struct Neuron *h_neuron, struct Synapse *h_synapse, size_t hidden_size,
struct Neuron *o_neuron, struct Synapse *o_synapse, size_t output_size,
const T_Precision *dx, const T_Precision *dy, size_t sn,
const T_Precision eta, const T_Precision desired_error );//dichiarazione funzione di retropropagazione dell'errore

//FINE DICHIARAZIONI FUNZIONI



 int main( void )

{
    // Inizializzo il generatore di numeri pseudocasuali
```

```
    srand( (size_t) time( NULL ) );

    // Dimensioni della rete
    const size_t input_size = 2;
    const size_t hidden_size = 5;
    const size_t output_size = 1;

    // Creo i neuroni degli strati
    struct Neuron i_neuron[ input_size ];
    struct Neuron h_neuron[ hidden_size];
    struct Neuron o_neuron[ output_size];

    // Creo le sinapsi degli strati (+ quella del bias)
    struct Synapse h_synapse[ (input_size + 1) * (hidden_size) ];
    struct Synapse o_synapse[ (hidden_size + 1) * (output_size) ];

    // Iteratore
    size_t i, j;

    // Numero di esempi dell'insieme di addestramento
    const size_t sn = 4;

    // Preparo l'insieme di addestramento per l'operatore XOR//---sn*input_size
    const T_Precision dx[] = { 0,0, 0,1, 1,0, 1,1 };

    const T_Precision dy[] = { 0, 1, 1, 0 };//valori desiderati--- sn*output_size

    // Tasso di apprendimento
    const T_Precision eta = 0.3;

    // Errore desiderato
    const T_Precision desired_error = 0.0001;

    // Addestro la rete neurale
    backpropagation( i_neuron, input_size,
    h_neuron, h_synapse, hidden_size,
    o_neuron, o_synapse, output_size,
    dx, dy, sn, eta, desired_error );




    // Stampo l'intestazione della tabella
    printf( "[x1]\t[x2]\t[y]\n" );

    // Provo tutte le combinazioni degli ingressi
    for ( j = 0; j <= 1; j++ )
    {
        for ( i = 0; i <= 1; i++ )
        {

        // Imposto gli ingressi
        i_neuron[0].value = i;
        i_neuron[1].value = j;

        // Eseguo la rete neurale
        run_network( i_neuron, input_size,
        h_neuron, h_synapse, hidden_size,
        o_neuron, o_synapse, output_size );

        // Stampo gli ingressi e la rispettiva uscita
```

```
        printf( "%.0f\t%.0f\t%.0f\n", i_neuron[0].value, i_neuron[1].value, o_neuron[0].value
        );

        }
    }

 return 0;

}




float sigmoid(float x)//definizione funzione sigmoide
{
     float exp_value;
     float return_value;

     /*** Exponential calculation ***/
     exp_value = exp((double) -x);

     /*** Final sigmoid value ***/
     return_value = 1 / (1 + exp_value);

     return return_value;
}




float d_sigmoid(float x)//definizione funzione derivata della sigmoide
{
     float exp_value;
     float return_value;

     /*** Exponential calculation ***/
     exp_value = exp((double) -x);

     /*** Final sigmoid value ***/
     return_value = (1 / (1 + exp_value))*(1-(1 / (1 + exp_value)));


     return return_value;
}




void run_network(struct Neuron *i_neuron, size_t input_size,    //definizione della funzione
che fa operare la rete di 3 strati,calcola le uscite di
struct Neuron *h_neuron,struct Synapse *h_synapse, size_t hidden_size, //ogni strato, un
percettrone alla volta, fino a determinare le uscite
struct Neuron *o_neuron, struct Synapse *o_synapse, size_t output_size )//della rete

{

 // Potenziale di attivazione
 float potential;
 // Iteratori
 size_t j, i;
```

```
    // Calcolo le uscite dello strato intermedio hidden
    for ( i = 0; i < hidden_size; i++ )

        {
            // Azzero il potenziale di attivazione
            potential = 0;
                // Calcolo il potenziale di attivazione
                    for ( j = 0; j < input_size; j++ )
                    {
                        potential += i_neuron[j].value * h_synapse[i * ( input_size + 1 ) +
j].weight;
                    }
            // Aggiungo il valore del bias
            potential += h_synapse[i * ( input_size + 1 ) + j].weight;

            // Calcolo l'uscita del neurone sulla base del potenziale di attivazione
            h_neuron[i].value = sigmoid( potential ); //uscita del generico neurone nello
strato hidden
        }



    // Calcolo le uscite dello strato di uscita
    for ( i = 0; i < output_size; i++ )

        {
            // Azzero il potenziale di attivazione
            potential = 0;
                // Calcolo il potenziale di attivazione
                    for ( j = 0; j < hidden_size; j++ )
                    {
                        potential += h_neuron[j].value * o_synapse[i * ( hidden_size + 1 ) +
j].weight;
                    }

            // Aggiungo il valore del bias
            potential += o_synapse[i * ( hidden_size + 1 ) + j].weight;

            // Calcolo l'uscita del neurone sulla base del potenziale di attivazione
            o_neuron[i].value = sigmoid( potential );//uscita del neurone output
        }
}//fine funzione (procedura) run_network




void init_weight( struct Synapse *synapse, size_t layer_size, size_t prev_layer_size )//
definizione funzione inizializzazione pesi
{
    // Valore casuale
    T_Precision random_;

    // Iteratori
    size_t j, i = 0;

    // Inizializzo i pesi sinaptici con dei valori casuali (inizializzo anche il peso del
bias)
    for (i=0 ; i < layer_size; i++ )
    {
```

```
        for ( j = 0; j <= prev_layer_size; j++ )
        {
            // Prelevo un valore casuale
            random_ = rand();
            // Imposto il valore del peso tra 0 e 1
            synapse[i * (prev_layer_size + 1) + j].weight = 2*( sin(random_) * sin(random_));//
            col seno al quadrato
                                                    //ho valori tra zero e uno
        }
    }
}//fine definizione funzione inizializzazione pesi




void backpropagation( struct Neuron *i_neuron, size_t input_size,
struct Neuron *h_neuron, struct Synapse *h_synapse, size_t hidden_size,
struct Neuron *o_neuron, struct Synapse *o_synapse, size_t output_size,
const T_Precision *dx, const T_Precision *dy, size_t sn,
const T_Precision eta, const T_Precision desired_error )//definizione funzione di
retropropagazione dell'errore

{
     // Errore della rete
    T_Precision error;

    // Modifica del peso sinaptico
    T_Precision delta_w;

    // Contatore delle epoche
    size_t epochs = 0;

    // Iteratori
    size_t t, j, i;

    // Log di lavoro
    printf( "Inizio l'addestramento (eta = %.2f, errore desiderato = %f).\n", eta,desired_error
    );

    // Inizializzo i pesi sinaptici con dei valori casuali
    init_weight( h_synapse, hidden_size, input_size ); //inizializzazione peso neuroni dello
    strato nascosto
    init_weight( o_synapse, output_size, hidden_size );//inizializzazione peso neuroni dello
    strato output

    // Continuo l'addestramento finché non raggiungo
    // l'errore desiderato (max 50000 epoche)
    do
    {
    // Azzero l'errore della rete (somma degli errori della rete)
    error = 0.0;

    // Ripeto per tutti gli esempi nell'insieme di addestramento
    for ( t = 0; t < sn; t++ )
    {

            // Prendo gli ingressi dall'esempio
            for ( i = 0; i < input_size; i++ )

            {
                i_neuron[i].value = dx[t * input_size + i];
```

```
        }

        // Eseguo la rete neurale
        run_network( i_neuron, input_size,
        h_neuron, h_synapse, hidden_size,
        o_neuron, o_synapse, output_size );




    // Calcolo l'errore dello strato di uscita (in questo caso un neurone)
    for ( i = 0; i < output_size; i++ )
    {

        // Calcolo l'errore dell'uscita del neurone dE/dy_i = -(D_i - Y_i)
        o_neuron[i].dEdy = -( dy[t * output_size + i] - o_neuron[i].value );

        // Aggiungo l'errore al totale
        error += (( o_neuron[i].dEdy )*( o_neuron[i].dEdy)) ;
    }



    // Calcolo l'errore dello strato intermedio
    for ( i = 0; i < hidden_size; i++ )
        {

        // Azzero l'errore dei neuroni dello strato intermedio hidden
        h_neuron[i].dEdy = 0;

        // Calcolo l'errore dello strato intermedio dE/dz_k = SUM( dE/dy_j * dy_j/dP_j *
    dP_j/dz_k )
        for ( j = 0; j < output_size; j++ )
            {

            // Calcolo l'errore dell'uscita dei neuroni dello strato intermedio
            h_neuron[i].dEdy += o_neuron[j].dEdy *
            d_sigmoid ( o_neuron[j].value ) * o_synapse[j * ( hidden_size + 1 ) +
            i].weight;
            }

        // Calcolo l'errore dell'uscita del bias
        h_neuron[i].dEdy += o_neuron[j].dEdy *
        d_sigmoid ( o_neuron[j].value ) * o_synapse[j * ( hidden_size + 1 ) + i].weight;
        }




    // Aggiusto i pesi dello strato di uscita
    for ( i = 0; i < output_size; i++ )
    {

        // Correggo il peso sinaptico
        for ( j = 0; j < hidden_size; j++ )
        {

        // Calcolo la modifica del peso sinaptico delta_w = - eta * dE/dy_j * dy_j/dP_j *
    dP_j/dw_jk
        delta_w = - eta * o_neuron[i].dEdy *
        d_sigmoid ( o_neuron[i].value ) * h_neuron[j].value;
```

```
            // Applico la modifica al peso sinaptico w = w + delta_w dei neuroni in uscita (uno
            in questo caso)
            o_synapse[i * ( hidden_size + 1 ) + j].weight += delta_w;
            }

            // Calcolo la modifica del peso del bias delta_w = - eta * dE/dy_j * dy_j/dP_j *
            dP_j/dw_jk
            delta_w = - eta * o_neuron[i].dEdy *
            d_sigmoid ( o_neuron[i].value );

            // Aggiungo la correzione del bias w = w + delta_w
            o_synapse[i * ( hidden_size + 1 ) + j].weight += delta_w;
        }

        // Aggiusto i pesi dello strato intermedio
        for ( i = 0; i < hidden_size; i++ )
        {

            // Correggo il peso sinaptico dei neuroni dello strato intermedio
            for ( j = 0; j < input_size; j++ )
            {

            // Calcolo la modifica del peso sinaptico delta_w = - eta * dE/dz_k * dz_k/dP_k *
            dP_k/dw_ki
            delta_w = - eta * h_neuron[i].dEdy *
            d_sigmoid ( h_neuron[i].value ) * i_neuron[j].value;

            // Applico la modifica al peso sinaptico w = w + delta_w
            h_synapse[i * ( input_size + 1 ) + j].weight += delta_w;

            }

            // Calcolo la modifica del peso del bias delta_w = - eta * dE/dz_k * dz_k/dP_k *
            dP_k/dw_ki
            delta_w = - eta * h_neuron[i].dEdy *
            d_sigmoid ( h_neuron[i].value );

            // Aggiungo la correzione del bias w = w + delta_w
            h_synapse[i * ( input_size + 1 ) + j].weight += delta_w;
        }

}//fine del primo for


// Calcolo l'errore quadratico medio della rete (MSE) E(x) = SUM( e^2 ) / n_samples
error /= ( input_size * sn );

// Ogni 1000 epoche stampo il log di addestramento
if ( epochs % 1000 == 0 )
{
printf( "Epoca #%zu, MSE=%f\n", epochs, error );
}

// Incremento il numero delle epoche
epochs++;

} while ( error > desired_error && epochs < 50000 );//fine ciclo do-while

// Log di lavoro
printf( "Addestramento terminato dopo %zu epoche.\n\n", epochs );
```

```
}//fine definizione funzione di retropropagazione dell'errore
```