```
// backprop.cpp
//layer.cpp
// compile for floating point hardware if available
#include <stdio.h>
#include <iostream.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>
#include "layer.h"


inline float squash(float input)
// squashing function
// use sigmoid -- can customize to something
// else if desired; can add a bias term too
//
{
if (input < -50)
    return 0.0;
else    if (input > 50)
        return 1.0;
    else return (float)(1/(1+exp(-(double)input)));

}


inline float randomweight(unsigned init)
{
int num;
// random number generator
// will return a floating point
// value between -1 and 1

if (init==1)    // seed the generator
    srand ((unsigned)time(NULL));

num=rand() % 100;

return 2*(float(num/100.00))-1;
}

// the next function is needed for Turbo C++
static void force_fpf()
{
    float x, *y;
    y=&x;
    x=*y;
}

// ------------------------------------------
//              input layer
//-------------------------------------------
input_layer::input_layer(int i, int o)
{



num_inputs=i;
num_outputs=o;

outputs = new float[num_outputs];
if (outputs==0)
    {
    cout << "not enough memory\n";
    cout << "choose a smaller architecture\n";
    exit(1);
    }
}

input_layer::~input_layer()
{
delete [num_outputs] outputs;
}


void input_layer::calc_out()
```

```
{
//nothing to do, yet
}




// ----------------------------------------
//              output layer
//-----------------------------------------




output_layer::output_layer(int i, int o)
{

num_inputs=i;
num_outputs=o;
weights = new float[num_inputs*num_outputs];
output_errors = new float[num_outputs];
back_errors = new float[num_inputs];
outputs = new float[num_outputs];
expected_values = new float[num_outputs];
if ((weights==0)||(output_errors==0)||(back_errors==0)
    ||(outputs==0)||(expected_values==0))
    {
    cout << "not enough memory\n";
    cout << "choose a smaller architecture\n";
    exit(1);
    }

}

output_layer::~output_layer()
{
delete [num_outputs*num_inputs] weights;
delete [num_outputs] output_errors;
delete [num_inputs] back_errors;
delete [num_outputs] outputs;

}


void output_layer::calc_out()
{

int i,j,k;
float accumulator=0.0;


for (j=0; j<num_outputs; j++)
    {

    for (i=0; i<num_inputs; i++)

        {
        k=i*num_outputs;
        if (weights[k+j]*weights[k+j] > 1000000.0)
            {
            cout << "weights are blowing up\n";
            cout << "try a smaller learning constant\n";
            cout << "e.g. beta=0.02    aborting...\n";


            exit(1);
            }
        outputs[j]=weights[k+j]*(*(inputs+i));
        accumulator+=outputs[j];
        }
    // use the sigmoid squash function
    outputs[j]=squash(accumulator);
    accumulator=0;
    }
```

```
}



void output_layer::calc_error(float & error)
{
int i, j, k;
float accumulator=0;
float total_error=0;

for (j=0; j<num_outputs; j++)
    {
    output_errors[j] = expected_values[j]-outputs[j];
    total_error+=output_errors[j];
    }

error=total_error;

for (i=0; i<num_inputs; i++)
    {
    k=i*num_outputs;
    for (j=0; j<num_outputs; j++)
        {
        back_errors[i]=
            weights[k+j]*output_errors[j];
        accumulator+=back_errors[i];
        }
    back_errors[i]=accumulator;
    accumulator=0;
    // now multiply by derivative of
    // sigmoid squashing function, which is
    // just the input*(1-input)
    back_errors[i]*=(*(inputs+i))*(1-(*(inputs+i)));
    }

}

void output_layer::randomize_weights()
{
int i, j, k;
const unsigned first_time=1;

const unsigned not_first_time=0;
float discard;

discard=randomweight(first_time);

for (i=0; i< num_inputs; i++)
    {
    k=i*num_outputs;
    for (j=0; j< num_outputs; j++)
        weights[k+j]=randomweight(not_first_time);
    }
}

void output_layer::update_weights(const float beta)
{
int i, j, k;

// learning law: weight_change =
//      beta*output_error*input

for (i=0; i< num_inputs; i++)
    {
    k=i*num_outputs;
    for (j=0; j< num_outputs; j++)
        weights[k+j] +=
            beta*output_errors[j]*(*(inputs+i));
    }

}

void output_layer::list_weights()
{
int i, j, k;
```

```
for (i=0; i< num_inputs; i++)
    {
    k=i*num_outputs;
    for (j=0; j< num_outputs; j++)
        cout << "weight["<<i<<","<<
            j<<"] is: "<<weights[k+j];
    }

}

void output_layer::list_errors()
{
int i, j;

for (i=0; i< num_inputs; i++)
    cout << "backerror["<<i<<
        "] is : "<<back_errors[i]<<"\n";

for (j=0; j< num_outputs; j++)
    cout << "outputerrors["<<j<<
            "] is: "<<output_errors[j]<<"\n";

}


void output_layer::write_weights(int layer_no,
        FILE * weights_file_ptr)
{
int i, j, k;

// assume file is already open and ready for
// writing

// prepend the layer_no to all lines of data
// format:
//      layer_no    weight[0,0] weight[0,1] ...
//      layer_no    weight[1,0] weight[1,1] ...
//      ...

for (i=0; i< num_inputs; i++)
    {
    fprintf(weights_file_ptr,"%i ",layer_no);
    k=i*num_outputs;
    for (j=0; j< num_outputs; j++)
    {
    fprintf(weights_file_ptr,"%f ",
                weights[k+j]);
    }
    fprintf(weights_file_ptr,"\n");
    }


}

void output_layer::read_weights(int layer_no,
        FILE * weights_file_ptr)
{
int i, j, k;


// assume file is already open and ready for
// reading

// look for the prepended layer_no
// format:
//      layer_no    weight[0,0] weight[0,1] ...
//      layer_no    weight[1,0] weight[1,1] ...
//      ...
while (1)

    {

    fscanf(weights_file_ptr,"%i",&j);
    if ((j==layer_no)|| (feof(weights_file_ptr)))
        break;
    else
```

```
        {
        while (fgetc(weights_file_ptr) != '\n')
            {;}// get rest of line
        }
    }

if (!(feof(weights_file_ptr)))
    {
    // continue getting first line
    i=0;
    for (j=0; j< num_outputs; j++)
            {

            fscanf(weights_file_ptr,"%f",
                    &weights[j]); // i*num_outputs = 0
        }
    fscanf(weights_file_ptr,"\n");



    // now get the other lines
    for (i=1; i< num_inputs; i++)
        {
        fscanf(weights_file_ptr,"%i",&layer_no);
        k=i*num_outputs;
    for (j=0; j< num_outputs; j++)
        {
        fscanf(weights_file_ptr,"%f",
            &weights[k+j]);
        }

    }
    fscanf(weights_file_ptr,"\n");
    }


else cout << "end of file reached\n";

}
void output_layer::list_outputs()
{
int j;

for (j=0; j< num_outputs; j++)
    {
    cout << "outputs["<<j
        <<"] is: "<<outputs[j]<<"\n";
    }

}


// -----------------------------------------
//              middle layer
//------------------------------------------


middle_layer::middle_layer(int i, int o):
    output_layer(i,o)
{

}

middle_layer::~middle_layer()
{
delete [num_outputs*num_inputs] weights;
delete [num_outputs] output_errors;
delete [num_inputs] back_errors;
delete [num_outputs] outputs;
}


void middle_layer::calc_error()
{
int i, j, k;
```

```
float accumulator=0;

for (i=0; i<num_inputs; i++)
    {
    k=i*num_outputs;
    for (j=0; j<num_outputs; j++)
        {
        back_errors[i]=
            weights[k+j]*(*(output_errors+j));
        accumulator+=back_errors[i];
        }
    back_errors[i]=accumulator;
    accumulator=0;
    // now multiply by derivative of
    // sigmoid squashing function, which is
    // just the input*(1-input)
    back_errors[i]*=(*(inputs+i))*(1-(*(inputs+i)));
    }

}

network::network()
{
position=0L;
}

network::~network()
{
int i,j,k;
i=layer_ptr[0]->num_outputs;// inputs
j=layer_ptr[number_of_layers-1]->num_outputs; //outputs
k=MAX_VECTORS;


delete [(i+j)*k]buffer;
}

void network::set_training(const unsigned & value)
{
training=value;
}

unsigned network::get_training_value()
{
return training;
}


void network::get_layer_info()
{
int i;

//-----------------------------------------
//
//  Get layer sizes for the network
//
// ----------------------------------------



cout << " Please enter in the number of layers for your network.\n";
cout << " You can have a minimum of 3 to a maximum of 5. \n";
cout << " 3 implies 1 hidden layer; 5 implies 3 hidden layers : \n\n";

cin >> number_of_layers;

cout << " Enter in the layer sizes separated by spaces.\n";
cout << " For a network with 3 neurons in the input layer,\n";
cout << " 2 neurons in a hidden layer, and 4 neurons in the\n";
cout << " output layer, you would enter: 3 2 4 .\n";
cout << " You can have up to 3 hidden layers,for five maximum entries :\n\n";

for (i=0; i<number_of_layers; i++)
    {
    cin >> layer_size[i];
    }
```

```
// ---------------------------------------------------------
// size of layers:
//      input_layer         layer_size[0]
//      output_layer        layer_size[number_of_layers-1]
//      middle_layers       layer_size[1]
//                  optional: layer_size[number_of_layers-3]
//                  optional: layer_size[number_of_layers-2]
//---------------------------------------------------------


}

void network::set_up_network()
{
int i,j,k;
//---------------------------------------------------------
// Construct the layers
//
//---------------------------------------------------------



layer_ptr[0] = new input_layer(0,layer_size[0]);

for (i=0;i<(number_of_layers-1);i++)
    {
    layer_ptr[i+1] =
    new middle_layer(layer_size[i],layer_size[i+1]);
    }

layer_ptr[number_of_layers-1] = new
output_layer(layer_size[number_of_layers-2],layer_size[number_of_layers-1]);

for (i=0;i<(number_of_layers-1);i++)
    {
    if (layer_ptr[i] == 0)
        {
        cout << "insufficient memory\n";
        cout << "use a smaller architecture\n";
        exit(1);
        }
    }

//---------------------------------------------------------
// Connect the layers
//
//---------------------------------------------------------
// set inputs to previous layer outputs for all layers,
//  except the input layer

for (i=1; i< number_of_layers; i++)
    layer_ptr[i]->inputs = layer_ptr[i-1]->outputs;



// for back_propagation, set output_errors to next layer
//      back_errors for all layers except the output
//      layer and input layer


for (i=1; i< number_of_layers -1; i++)
    ((output_layer *)layer_ptr[i])->output_errors =
        ((output_layer *)layer_ptr[i+1])->back_errors;

// define the IObuffer that caches data from
// the datafile
i=layer_ptr[0]->num_outputs;// inputs
j=layer_ptr[number_of_layers-1]->num_outputs; //outputs
k=MAX_VECTORS;

buffer=new
    float[(i+j)*k];
if (buffer==0)
    {
    cout << "insufficient memory for buffer\n";
    exit(1);
    }
}

void network::randomize_weights()
{
int i;

for (i=1; i<number_of_layers; i++)
    ((output_layer *)layer_ptr[i])
        ->randomize_weights();
}


void network::update_weights(const float beta)
{
int i;

for (i=1; i<number_of_layers; i++)
    ((output_layer *)layer_ptr[i])
        ->update_weights(beta);
}


void network::write_weights(FILE * weights_file_ptr)
{
int i;

for (i=1; i<number_of_layers; i++)
    ((output_layer *)layer_ptr[i])
        ->write_weights(i,weights_file_ptr);
}


void network::read_weights(FILE * weights_file_ptr)
{
int i;

for (i=1; i<number_of_layers; i++)
    ((output_layer *)layer_ptr[i])
        ->read_weights(i,weights_file_ptr);
}


void network::list_weights()
{
int i;

for (i=1; i<number_of_layers; i++)
    {
    cout << "layer number : " <<i<< "\n";
    ((output_layer *)layer_ptr[i])
        ->list_weights();
    }
}

void network::list_outputs()
{
int i;

for (i=1; i<number_of_layers; i++)
    {
    cout << "layer number : " <<i<< "\n";
    ((output_layer *)layer_ptr[i])
        ->list_outputs();
    }
}

void network::write_outputs(FILE *outfile)
{
int i, ins, outs;
ins=layer_ptr[0]->num_outputs;
outs=layer_ptr[number_of_layers-1]->num_outputs;
float temp;

fprintf(outfile,"for input vector:\n");
```

```
for (i=0; i<ins; i++)
    {
    temp=layer_ptr[0]->outputs[i];
    fprintf(outfile,"%f  ",temp);
    }


fprintf(outfile,"\noutput vector is:\n");

for (i=0; i<outs; i++)
    {
    temp=layer_ptr[number_of_layers-1]->
    outputs[i];
    fprintf(outfile,"%f  ",temp);

    }

if (training==1)
{
fprintf(outfile,"\nexpected output vector is: \n");

for(i=0;i<outs;i++)
    {
     temp =((output_layer*)(layer_ptr[number_of_layers-1]))->expected_values[i];
     fprintf(outfile,"%f  ",temp);
    }
}
fprintf(outfile, "\n------------------\n");

}

void network::list_errors( )
{
 int i;
for(i =1;i<number_of_layers;i++)
    {
    cout<<"layer number :  "<<i<<"\n";
    ((output_layer *)layer_ptr[i])->list_errors();
    }
}

int network::fill_IObuffer(FILE *inputfile)
{
//this routine fills memory with
//an array of input, output vectors
//up to a maximum capacity of
// MAX_INPUT_VECTORS_IN_ARRAY
//the return value is the number of read vectors

int i,k,count, veclength;
int ins,outs;

ins =layer_ptr[0]->num_outputs;
outs=layer_ptr[number_of_layers-1]->num_outputs;
if(training ==1)
    veclength = ins+outs;
else
    veclength = ins;
count=0;
while ((count<MAX_VECTORS)&&(!feof(inputfile)))
    {
    k=count*(veclength);
    for(i=0;i<veclength;i++)
        {
        fscanf(inputfile,"%f",&buffer[k+i]);
        }
    fscanf(inputfile,"\n");
    count++;
    }
if(!(ferror(inputfile)))
    return count;
else
    return -1; //error condition
}
```

```
void network::set_up_pattern(int buffer_index)
{
//read one vector into the network
int i,k;
int ins, outs;

ins=layer_ptr[0]->num_outputs;
outs=layer_ptr[number_of_layers-1]->num_outputs;
if(training==1)
    k=buffer_index*(ins+outs);
else
    k=buffer_index*ins;
for(i=0;i<ins;i++)
    layer_ptr[0]->outputs[i]=buffer[k+i];
if(training==1)
{
    for(i=0;i<outs;i++)
    ((output_layer *)layer_ptr[number_of_layers-1])->expected_values[i]= buffer[k+i+ins];
}

}
                                                                          void network::forward_pr
op()
{
int i;
for(i=0;i<number_of_layers;i++)
{
layer_ptr[i]->calc_out(); //polymorphic function
}
}

void network::backward_prop(float &toterror)
{
int i;
//error for the output layer
((output_layer*)layer_ptr[number_of_layers-1])-> calc_error(toterror);

//error for the middle layer(s)
for (i=number_of_layers-2;i>0;i--)
{
((middle_layer*)layer_ptr[i])-> calc_error();
}

}
#define TRAINING_FILE   "training.dat"
#define WEIGHTS_FILE    "weights.dat"
#define OUTPUT_FILE "output.dat"
#define TEST_FILE   "test.dat"


void main()
{

float error_tolerance=0.1;
float total_error=0.0;
float avg_error_per_cycle=0.0;
float error_last_cycle=0.0;
float avgerr_per_pattern=0.0; // for the latest cycle
float error_last_pattern=0.0;
float learning_parameter=0.02;
unsigned temp, startup;
long int vectors_in_buffer;
long int max_cycles;
long int patterns_per_cycle=0;

long int total_cycles, total_patterns;
int i;


// create a network object
network backp;


FILE * training_file_ptr, * weights_file_ptr, * output_file_ptr;
FILE * test_file_ptr, * data_file_ptr;
```

```
// open output file for writing
if ((output_file_ptr=fopen(OUTPUT_FILE,"w"))==NULL)
        {
        cout << "problem opening output file\n";
        exit(1);
        }

// enter the training mode : 1=training on     0=training off
cout << "------------------------------------------------\n";
cout << "   Backpropagation \n";
cout << "------------------------------------------------\n";
cout << "Please enter 1 for TRAINING on, or 0 for off: \n\n";
cout << "Use training to change weights according to your\n";
cout << "expected outputs. Your training.dat file should contain\n";
cout << "a set of inputs and expected outputs. The number of\n";
cout << "inputs determines the size of the first (input) layer\n";
cout << "while the number of outputs determines the size of the\n";
cout << "last (output) layer :\n\n";

cin >> temp;
backp.set_training(temp);

if (backp.get_training_value() == 1)
    {
    cout << "--> Training mode is *ON*. weights will be saved\n";
    cout << "in the file weights.dat at the end of the\n";
    cout << "current set of input (training) data\n";
    }
else
    {
    cout << "--> Training mode is *OFF*. weights will be loaded\n";
    cout << "from the file weights.dat and the current\n";
    cout << "(test) data set will be used. For the test\n";
    cout << "data set, the test.dat file should contain\n";
    cout << "only inputs, and no expected outputs.\n";
    }

if (backp.get_training_value()==1)
    {
    // ----------------------------------------
    //  Read in values for the error_tolerance,
    //  and the learning_parameter
    // ----------------------------------------
    cout << " Please enter in the error_tolerance\n";
    cout << " --- between 0.001 to 100.0, try 0.1 to start --\n";
    cout << "\n";
    cout << "and the learning_parameter, beta\n";
    cout << " --- between 0.01 to 1.0, try 0.5 to start -- \n\n";
    cout << " separate entries by a space\n";
    cout << " example: 0.1 0.5 sets defaults mentioned :\n\n";

    cin >> error_tolerance >> learning_parameter;
    //-----------------------------------------
    // open training file for reading
    //-----------------------------------------
    if ((training_file_ptr=fopen(TRAINING_FILE,"r"))==NULL)
        {
        cout << "problem opening training file\n";
        exit(1);
        }
    data_file_ptr=training_file_ptr; // training on

    // Read in the maximum number of cycles
    // each pass through the input data file is a cycle
    cout << "Please enter the maximum cycles for the simulation\n";
    cout << "A cycle is one pass through the data set.\n";
    cout << "Try a value of 10 to start with\n";

    cin >> max_cycles;

    }
else
    {
    if ((test_file_ptr=fopen(TEST_FILE,"r"))==NULL)
        {
        cout << "problem opening test file\n";
        exit(1);
        }

    data_file_ptr=test_file_ptr; // training off
    }




// the main loop
//
// training: continue looping until the total error is less than
//      the tolerance specified, or the maximum number of
//      cycles is exceeded; use both the forward signal propagation
//      and the backward error propagation phases. If the error
//      tolerance criteria is satisfied, save the weights in a file.
// no training: just proceed through the input data set once in the
//      forward signal propagation phase only. Read the starting
//      weights from a file.
// in both cases report the outputs on the screen


// intialize counters
total_cycles=0; // a cycle is once through all the input data
total_patterns=0; // a pattern is one entry in the input data



// get layer information
backp.get_layer_info();

// set up the network connections
backp.set_up_network();

// initialize the weights
if (backp.get_training_value()==1)
    {
    // randomize weights for all layers; there is no
    // weight matrix associated with the input layer
    // weight file will be written after processing
    // so open for writing
    if ((weights_file_ptr=fopen(WEIGHTS_FILE,"w"))
            ==NULL)
        {
        cout << "problem opening weights file\n";
        exit(1);
        }
    backp.randomize_weights();
    }
else
    {
    // read in the weight matrix defined by a
    // prior run of the backpropagation simulator
    // with training on
    if ((weights_file_ptr=fopen(WEIGHTS_FILE,"r"))
            ==NULL)
        {
        cout << "problem opening weights file\n";
        exit(1);
        }
    backp.read_weights(weights_file_ptr);
    }




// main loop
// if training is on, keep going through the input data
//      until the error is acceptable or the maximum number of cycles
//      is exceeded.
// if training is off, go through the input data once. report outputs
// with inputs to file output.dat

startup=1;
vectors_in_buffer = MAX_VECTORS; // startup condition
total_error = 0;
```

```
while (          ((backp.get_training_value()==1)
            && (avgerr_per_pattern
                    > error_tolerance)
            && (total_cycles < max_cycles)
            && (vectors_in_buffer !=0))
            || ((backp.get_training_value()==0)
            && (total_cycles < 1))
            || ((backp.get_training_value()==1)
            && (startup==1))
            )
{
startup=0;
error_last_cycle=0; // reset for each cycle
patterns_per_cycle=0;
// process all the vectors in the datafile
// going through one buffer at a time
// pattern by pattern



while ((vectors_in_buffer==MAX_VECTORS))
    {

    vectors_in_buffer=
        backp.fill_IObuffer(data_file_ptr); // fill buffer
        if (vectors_in_buffer < 0)
            {
            cout << "error in reading in vectors, aborting\n";
            cout << "check that there are on extra linefeeds\n";
            cout << "in your data file, and that the number\n";
            cout << "of layers and size of layers match the\n";
            cout << "the parameters provided.\n";
            exit(1);
            }

        // process vectors
        for (i=0; i<vectors_in_buffer; i++)
            {
            // get next pattern
            backp.set_up_pattern(i);

            total_patterns++;
            patterns_per_cycle++;
            // forward propagate

            backp.forward_prop();

            if (backp.get_training_value()==0)
                backp.write_outputs(output_file_ptr);

            // back_propagate, if appropriate
            if (backp.get_training_value()==1)
                {

                backp.backward_prop(error_last_pattern);
                error_last_cycle +=
                    error_last_pattern*error_last_pattern;
                backp.update_weights(learning_parameter);
                // backp.list_weights(); // can
                // see change in weights by
                // using list_weights before and
                // after back_propagation
                }

            }

    error_last_pattern = 0;
        }

avgerr_per_pattern=((float)sqrt((double)error_last_cycle/patterns_per_cycle));
total_error += error_last_cycle;
total_cycles++;

// most character displays are 25 lines
// user will see a corner display of the cycle count
// as it changes

cout << "\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n\n";
cout << total_cycles << "\t" << avgerr_per_pattern << "\n";

fseek(data_file_ptr, 0L, SEEK_SET); // reset the file pointer
                // to the beginning of
                // the file
vectors_in_buffer = MAX_VECTORS; // reset

} // end main loop

cout << "\n\n\n\n\n\n\n\n\n\n\n\n";
cout << "--------------------------------------------------------\n";
cout << "   done:   results in file output.dat\n";
cout << "       training: last vector only\n";
cout << "       not training: full cycle\n\n";
if (backp.get_training_value()==1)
    {
    backp.write_weights(weights_file_ptr);
    backp.write_outputs(output_file_ptr);
    avg_error_per_cycle= (float)sqrt((double)total_error/total_cycles);
    error_last_cycle=(float)sqrt((double)error_last_cycle);

cout << "       weights saved in file weights.dat\n";
cout << "\n";
cout << "---->average error per cycle = " << avg_error_per_cycle << " <---\n";
cout << "---->error last cycle = " << error_last_cycle << " <---\n";
cout << "->error last cycle per pattern= " << avgerr_per_pattern << " <---\n";

    }

cout << "------------>total cycles = " << total_cycles << " <---\n";
cout << "------------>total patterns = " << total_patterns << " <---\n";
cout << "--------------------------------------------------------\n";
// close all files
fclose(data_file_ptr);
fclose(weights_file_ptr);
fclose(output_file_ptr);

}
```