Jl. of Computers in Mathematics and Science Teaching (2002) **21**(3), 287-306

Object-Oriented Implementation of the Backpropagation Algorithm

MOHAMED M. KHATIB
Girls College, Ain Shams University
Cairo, Egypt
mohamed_mkhatib@hotmail.com

ISMAIL. A. ISMAIL
Zagazig University
Egypt

The Error-Backpropagation (or simply, backpropagation) algorithm is the most important algorithm for the supervised training of multilayer feed-forward Artificial Neural Networks (ANN). It derives its name from the fact that error signals are propagated backward through the network on a layer-by-layer basis. In this article we use an object-oriented approach for implementing the backpropagation algorithm.

The backpropagation algorithm is based on the selection of a suitable error function or cost function, whose values are determined by the actual and desired outputs of the network, and which is also dependent on the network parameters such as the weights and the thresholds (Jacek, 1995). The basic idea is that the cost function has a particular surface over the weight space and therefore an iterative minimization process such as the gradient descent method can be used for its minimization. The method of gradient descent is based on the fact that, the gradient of a function always points in the direction of maximum increase of the function then, moving to the direction of the negative gradient induces a maximal "downhill" movement that

will eventually reach the minimum of the function surface over its parameter space. This is a rigorous and well-established technique for minimization of functions and has probably been the main factor behind the success of back-propagation.

A typical multi-layer feed-forward ANN is shown in Figure 1. This type of network is also known as a Multilayer Perceptron (MLP). The units (or nodes) of the network are nonlinear threshold units described by equations (1) and (2) and their activation function is given by equation (3):

$$u = \sum_{i=0}^{N} w_i x_i \quad (1) \qquad y = f\left(\sum_{i=0}^{N} w_i x_i\right) \quad (2)$$

$$f(u) = \frac{1}{1 + \exp(-\alpha u)} \quad (3)$$

Where $x_1, x_2, ..., x_n$ are the input signals, $w_1, w_2, ..., w_n$ are the synaptic weights, u is the *activation potential* of the neuron and α is the slope parameter of (3).

The units are arranged in layers and each unit in a layer has all its inputs connected to the units of a preceding layer (or to the inputs from the external world in the case of the units in the first layer), but it does not have any connections to units of the same layer to which it belongs. The layers are arrayed one succeeding the other so that there is an input layer, multiple intermediate layers and finally an output layer. Intermediate layers, that is those that have no inputs or outputs to the external world, are called hidden layers. Figure 1 shows a MLP with only one hidden layer. Backpropagation neural networks are usually fully connected. This means that each unit is connected to every output from the preceding layer (or to every input from the external world if the unit is in the first layer). Generally, the input layer is considered as just a distributor of the signals from the external world and is not therefore counted as a layer (Lefteri & Robert, 1997).

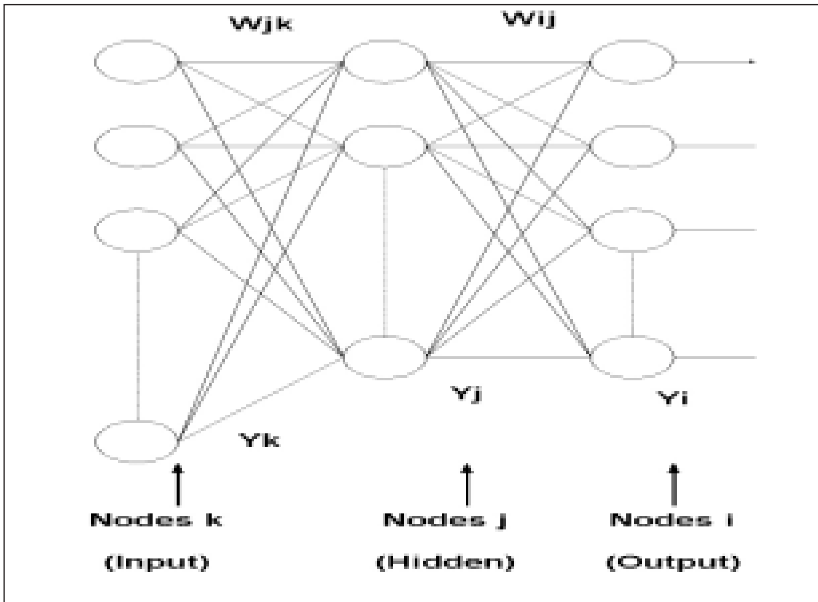

Figure 1. Multilayer feedforward ANN

The backpropagation training consists of two passes of computation: a *forward pass* and a *backward pass* (Jacek, 1995). In the forward pass an input pattern vector is applied to the sensory nodes of the network that is, to the units in the input layer. The signals from the input layer propagate to the units in the first layer and each unit produces an output according to equation (2). The outputs of these units are propagated to units in subsequent layers and this process continues until, finally, the signals reach the output layer where the actual response of the network to the input vector is obtained. During the forward pass the synaptic weights of the network are fixed. During the backward pass, on the other hand, the synaptic weights are all adjusted in accordance with an error signal which is propagated backward through the network against the direction of synaptic connections.

THE MATHEMATICAL ANALYSIS OF THE BACKPROPAGATION ALGORITHM

In the forward pass of computation, given an input pattern vector $y^{(p)}$, each hidden node j receives a net input:

$$x_j^{(p)} = \sum_k w_{jk} y_k^{(p)} \qquad (4)$$

Where w_{jk} represents the weight between hidden node j and input node k, thus node j produces an output:

$$y_j^{(p)} = f(x_j^p) = f(\sum_k w_{jk} y_k^p) \qquad (5)$$

Each output node i thus receives:

$$x_i^{(p)} = \sum_j w_{ij} y_j^{(p)} = \sum_j w_{ij} f(\sum_k w_{jk} y_k^{(p)}) \quad (6)$$

Where wij represents the weight between output node i and hidden node j. Hence it produces for the final output:

$$y_i^{(p)} = f(x_i^{(p)}) = f(\sum_j w_{ij} y_j^{(p)}) = f(\sum_j w_{ij} f(\sum_k w_{jk} y_k^{(p)})) \qquad (7)$$

The backpropagation algorithm can be implemented in two different modes: *on-line mode* and *batch mode*. *In the on-line mode* the error function is calculated after the presentation of each input pattern and the error signal is propagated back through the network modifying the weights before the presentation of the next pattern. This error function is usually the Mean Square Error (MSE) of the difference between the desired and the actual responses of the network over all the output units. Then the new weights remain fixed and a new pattern is presented to the network and this process continuous until all the patterns have been presented to the network. The presentation of all the patterns is usually called one epoch or one iteration. In practice many epochs are needed before the error becomes acceptably small.

In the batch mode the error signal is calculated for each input pattern and the weights are modified ever time the input pattern has been presented. Then the error function is calculated as the sum of the individual MSE errors for each pattern and the weights are accordingly modified (all in a single step for all the patterns) before the next iteration. Thus, in the batch mode, the error or cost function calculated as the MSE over all output units i and over all patterns p is given by:

$$E = \frac{1}{2} \sum_p \sum_i (d_i^{(p)} - y_i^{(p)})^2 = \frac{1}{2} \sum_p \sum_i (d_i^{(p)} - f(\sum_j w_{ij} f(\sum_k w_{jk} y_k^{(p)})))^2 \qquad (8)$$

Clearly E is a differentiable function of all the weights and therefore we can apply the method of gradient descent. For the hidden-to-output connections the gradient descent rule gives:

$$\Delta w_{ij} = -\eta \frac{\partial E}{\partial w_{ij}} \quad (9)$$

Where h is a constant that determines the rate of learning; it is called the *learning rate* of the backpropagation algorithm. Using the chain rule we have:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial \mathrm{E}}{\partial \mathrm{y}_{\mathrm{i}}^{(\mathrm{p})}} \cdot \frac{\partial y_i^{(p)}}{\partial w_{ij}}$$

$$(10)$$

$$\frac{\partial y_i^{(p)}}{\partial w_{ij}} = \frac{\partial y_i^{(p)}}{\partial x_i^{(p)}} \cdot \frac{\partial x_i^{(p)}}{\partial w_{ij}}$$

Giving:

$$\frac{\partial E}{\partial w_{ij}} = \frac{\partial E}{\partial y_i^{(p)}} \cdot \frac{\partial y_i^{(p)}}{\partial x_i^{(p)}} \cdot \frac{\partial x_i^{(p)}}{\partial w_{ij}} = -\sum_p (d_i^{(p)} - y_i^{(p)}) f'(x_i^{(p)}) y_j^{(p)} \qquad (11)$$

Thus:

$$\Delta w_{ij} = \eta \sum (d_i^{(p)} - y_i^{(p)}) f'(x_i^{(p)}) y_j^{(p)} = \eta \sum_p \delta_i^{(p)} y_i^{(p)} \qquad (12)$$

Where:

$$\delta_i^{(p)} = (d_i^{(p)} - y_i^{(p)}) f'(x_i^{(p)}) \qquad (13)$$

For the input-to-hidden connections the gradient descent rule gives:

$$\Delta w_{jk} = -\eta \frac{\partial E}{\partial w_{jk}} \qquad (14)$$

Using the chain rule we obtain:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial y_j^{(p)}} \cdot \frac{\partial y_j^{(p)}}{\partial w_{jk}}$$

$$(15)$$

$$\frac{\partial y_j^{(p)}}{\partial w_{jk}} = \frac{\partial y_j^{(p)}}{\partial x_j^{(p)}} \cdot \frac{\partial x_j^{(p)}}{\partial w_{jk}}$$

Giving:

$$\frac{\partial E}{\partial w_{jk}} = \frac{\partial E}{\partial y_j^{(p)}} \cdot \frac{\partial y_j^{(p)}}{\partial x_j^{(p)}} \cdot \frac{\partial x_j^{(p)}}{\partial w_{jk}} = \frac{\partial E}{\partial y_j^{(p)}} f'(x_j^{(p)})\, y_k^{(p)} \qquad (16)$$

Now for $\frac{\partial E}{\partial y_j}$ we have:

$$\frac{\partial E}{\partial y_j^{(p)}} = -\sum_p \sum_i (d_i^{(p)} - y_i^{(p)}) \frac{\partial f(x_i^p)}{\partial y_j^{(p)}}$$

$$= -\sum_p \sum_i (d_i^{(p)} - y_i^{(p)}) \frac{\partial f(x_i^p)}{\partial x_i^{(p)}} \cdot \frac{\partial x_i^{(p)}}{\partial y_j^{(p)}}$$

$$= -\sum_p \sum_i (d_i^{(p)} - y_i^{(p)}) f'(x_i^{(p)}) w_{ij} \qquad (17)$$

Thus:

$$\frac{\partial E}{\partial w_{jk}} = -\sum_p \sum_i (d_i^{(p)} - y_i^{(p)}) f'(x_i^{(p)}) w_{ij} f'(x_j^{(p)})\, y_k^{(p)} \qquad (18)$$

Which gives:

$$\Delta w_{jk} = \eta \sum_p \sum_i (d_i^{(p)} - y_i^{(p)}) f'(x_i^{(p)}) w_{ij} f'(x_j^{(p)})\, y_k^{(p)}$$

$$= \eta \sum_p \sum_i \delta_i^{(p)} w_{ij} f'(x_j^{(p)})\, y_k^{(p)} = \eta \sum_p \delta_j^{(p)} y_k^{(p)} \qquad (19)$$

With:

$$\delta_j^{(p)} = f'(x_j^{(p)}) \sum_i \delta_i^{(p)} w_{ij} \qquad (20)$$

From equation (11) and (18) we can see that if the activation function was not differentiable then we would be unable to implement the gradient-descent rule, as it would be impossible to calculate the partial derivatives of E with respect to the weights. It is for this reason that differentiability of the activation function is so important in backpropagation learning. Note also that the derivative of the sigmoid function is very easy to compute since:

$$f'(u) = f(u)[1 - f(u)] \ (21)$$

This means that it is not necessary to compute $f'(u)$ separately once we have found $f(u)$. This is one of the advantages of the sigmoid function as computation time can reduce significantly.

Although we have written the update rules for the batch mode of training it is clear that the weight updates in the online case would be given again by equations (12) and (19) without of course the summation over the training patterns. Note that equation (19) has the same form as equation (12) but with a different definition of the δ's. In general, with an arbitrary number of layers, the backpropagation update rule always has the form:

$$\Delta w_{lm} = \eta \cdot \delta_l \cdot y_m \qquad (23)$$

Where Δw_{lm} is the weight correction, η is the learning rate, ´ is the local gradient, and y_m is the input signal of node. This general rule for the adaptation of the weights is also known as *generalized delta rule.*

IMPLEMENTATION OF THE BACKPROPAGATION ALGORITHM

We have used C++ as object-oriented programming language for implementing the backpropagation algorithm. Object-oriented programming is a programming paradigm where a program is made up of a collection of independent "objects" which perform the actions of a program by interacting with each other (Lafore, 1999). An object is typically a collection of related data and routines. Our implementation makes extensive use of two important principles of OOP: encapsulation and abstraction (Mohamed & Laitinen, 1998).

Encapsulation describes the notion that all of the inner working of an object is hidden from other objects. OOP allows for information hiding, which means that the inner working of an object can literally be protected

from other objects. An object can be thought of as a "black box," meaning that other objects do not know and don't need to know how the duty of the object is being performed. By providing an "interface" to the outside world, an object describes how it is to be communicated with. This separation of the inner working from the interface also allows for abstraction.

Abstraction is a programming technique where the essential behavior of an object is separated from its specific purpose. This allows for efficiency when dealing with similar objects because common behavior can be shared through "inheritance." Another benefit of OOP is that, once created, objects are easy to reuse for other applications. Our implementation of the backpropagation algorithm has the following design objectives:

- allow the user to specify the number and size of all layers;
- allow the use of one or more hidden layers;
- be able to save and restore the state of the network;
- display key information at the end of the simulation; and
- drawing the network as specified by the user.

C++ Classes and Class Hierarchy

This section describes, in brief, the C++ classes of our system that implement the backpropagation algorithm. In this system we use a class hierarchy with the inheritance feature. Also, we use polymorphism with dynamic binding and function overloading with static binding. First let us look at the class hierarchy used for this program (Figure 2 shows the major classes using UML notation) (Grady, James, & Ivar, 1999). An *abstract* class is a class that is never meant to be instantiated as an object, but serves as a base class from which others can inherit functionality and interface definitions. The CBaseLayer class is such a class. One of its functions is set = zero (see appendix A), which indicates that this class is an *abstract base class.* From the CBaseLayer class are two branches. One is the CInputLayer class, and the other is the COutputClass. The middle layer class CMiddleLayer is very much like output layer in function and so inherits from the COutputClass class.

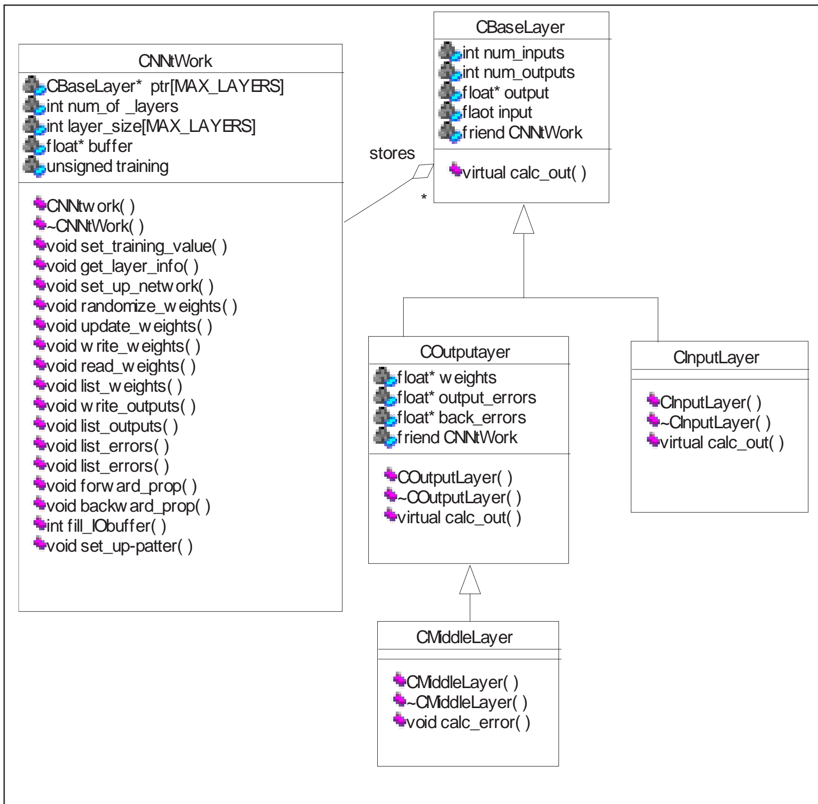

Figure 2. Class hierarchy using UML notation. A rectangle denotes a class. A solid directed line with a large open arrowhead means "inherits" from. A line denotes an association. A line with a small diamond denotes aggregation or "contains."

The CNNtwork class is used to set up communication channels between layers and to feed and remove data from the network. The CNNtwork class performs the interconnection of layers by setting the pointer of an input array of a given layer to output array of previous layer. Another connection that the CNNtwork class is responsible for is setting the pointer of an *output_error* array to the *back_error* array of the next layer (remember, errors flow in reverse, and the back_error array in the output error of the layer reflected at its inputs). The CNNtwork class stores an array of pointers to layers and an array of layer sizes for all the layers defined. These layer objects and arrays are dynamically allocated on the heap with the New and Delete functions in C++.

Details in Implementation of the Backpropagation Algorithm

Function overloading can be seen in the definition of the calc_error() function in the CBaseLayer class (see appendix A). It is used in the CInputLayer with no parameters, while it is used in the COutputClass (which the CInputLayer inherits from) with one parameter. Using the same function name is not a problem, and this is referred to as *overloading*. Besides function overloading, you may also have operator overloading, which is using an operator that performs some familiar function like + for addition, for another function, say, vector addition. When you have overloading with the same parameters and keyword *virtual,* then you have the potential for *dynamic binding*, which means that you determine which overloading function to execute at run time and not at compile time. Compile time binding is referred to as *static binding*. If you put a bunch of C++ objects in an array of pointers to the base class, and then go through a loop that indexes each pointer and executes an overloading virtual function that pointer is pointing to, then you will be using dynamic binding. This is exactly the case in the function calc_out(), which is declared with the virtual keyword in the CBaseLayer base class. Each descendant of CBaseLayer can provide a version of calc_out(), which differs in functionality from the base class, and the correct function will be selected at run time based on the object's identity. In this case calc_out(), which is a function to calculate the outputs for each layer, is different for in input layer than for the other two types of layers.

Note the following definitions in the CBaseLayer base class:

```
int num_inputs;
int num_outputs;
float* output; // pointer to array of outputs
float* inputs;//pointer to array of inputs, which
//are outputs of some other layer
friend CNNtwork;
```

There are also two pointers to arrays of floats in this class. They are the pointers to the outputs in a given layer and the inputs to a given layer. To get a better idea of what a layer encompasses, Figure 3 shows you a small feed forward backpropagation network, with a dotted line that shows you the three layers of the network.

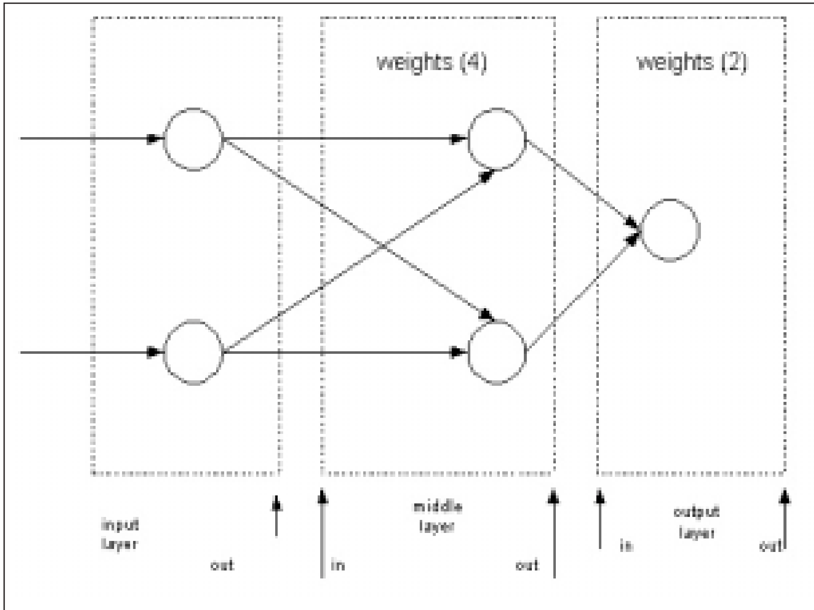

Figure 3. Organization of layers for backpropagation program

A layer contains neurons and weights. The layer is responsible for calculating its output (calc_out()), stored in the float *outputs array, and errors (calc_error()) for each of its respective neurons. The errors are stored in another array called float* output_errors defined in the COutputLayer class. Note that the CInputLayer class does not have any weights associated with it and therefore is a special case. It does not need to provide any data members or function members related to errors or backpropagation. The only purpose of the input layer is to store data to be forward propagated to the next layer.

With the output layer, there are a few more arrays present. First, for storing backpropagation errors, there is an array called float* back_errors. There is a weights array called float* weights, and finally, for storing the expected values that initiate the error calculation process, there is an array called flaot* expected_values. Note that the middle layer needs almost all of these arrays and inherits them by being a derived class of the COutputLayer class.

SYSTEM MODES

There are two modes of operation in our system: *Training Mode* and *Testing Mode.*

Training Mode:

In this mode the user provides the system with the following information:

- The training file in the current directory is called training.txt. This file contains exemplar pairs or patterns. Each pattern has a set of inputs followed by a set of outputs. Each value is separated by one or more spaces. Figure 4 shows an example of a training.txt file.
- the values for the error tolerance and the learning rate;
- the maximum number of cycles, or passes through the training data;
- the number of layers (the training mode layout is shown in Figure 5); and
- the size for each layer, from the input to the output.

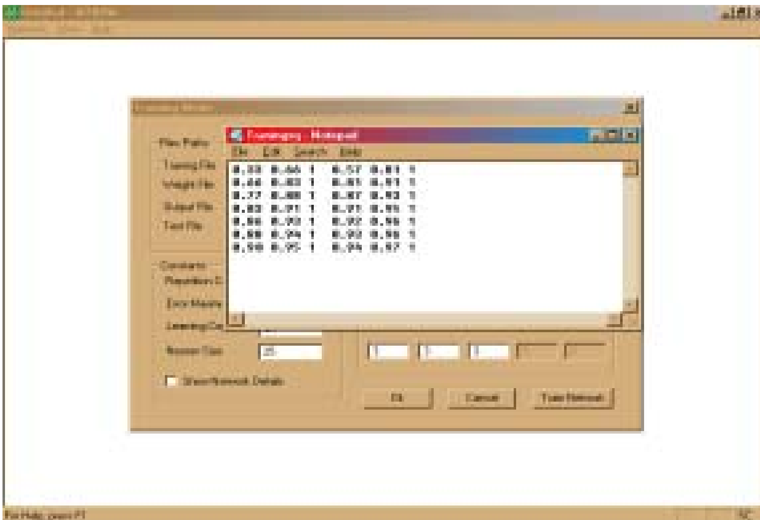

Figure 4. Training file

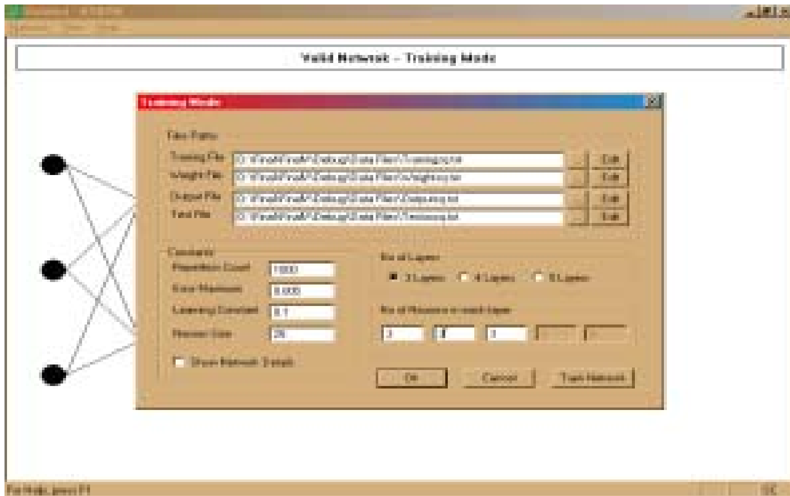

Figure 5. Layout of the training mode

After preparing previous information the system begins training and reports the current cycle number and the error for each cycle. Also it may watch the error to see that it is on the whole, decreasing with time, if it is not, then the system gives a chance to restart the training because this will start with a brand new set of random weights and another, possibly better, solution. Once the training is done the system gives information about the number of cycles and patterns used and the total error. The system can draw the network under training in two cases, without training data and with training data, as shown in Figures 6 and 7 respectively.

Another file that is used in training is the weights file. Once the system reaches the error tolerance that is specified by the user, or the maximum number of iterations, the system saves the state of the network, by saving all of its weight in a file called *weight.txt*. This file can then be used subsequently in another run of the system in testing mode. In addition, the output generated by the network is provided in an output file called *output.txt*.

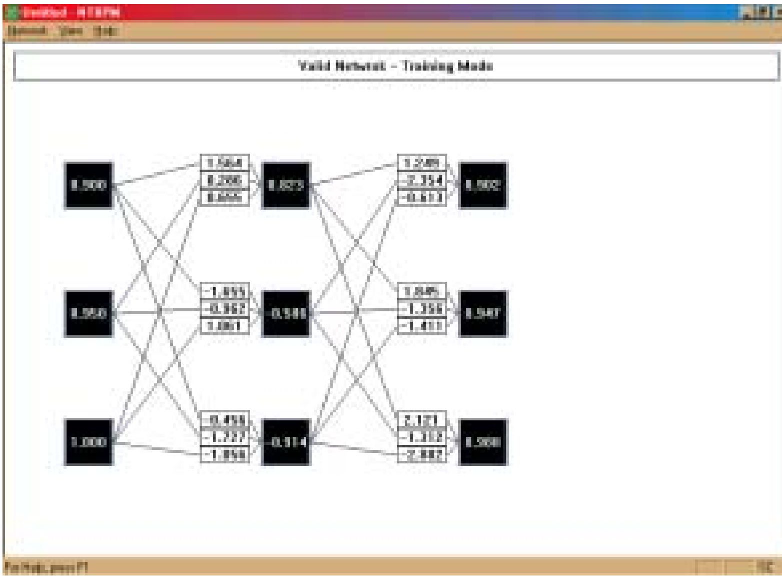

Figure 6. The network with training data

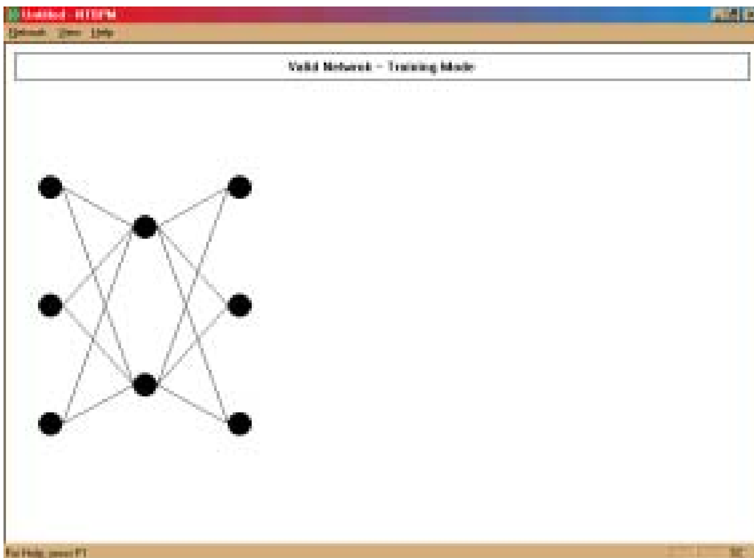

Figure 7. The network without training data

Testing Mode

In this mode, the user provides test data to the system in a file called *test.txt.* This file contains only input patterns. When this file is applied to an already trained network, an output.txt file is generated, which contains the output from the network for all of the input patterns. The network goes through one cycle of operation in this mode, covering all the patterns in the test data file. To start up the network, the weight file, weight.txt is read to initialize the state of the network. The user must provide the same network size parameters used to train the network.

CASE STUDY: CHARACTER RECOGNITION

The problem presented in this section deals with recognizing or categorizing alphabetic characters. We will input various alphabetic characters to our system and train the network to recognize these as separate categories.

Representing Characters

A 5x7 grid of pixels represents each character. To represent the character A, for example, we use the pattern shown in figure 8. Here the blackened boxes represent value 1, while blank boxes represent zeros. We can represent all characters this way, with a binary map of 35 pixel values.

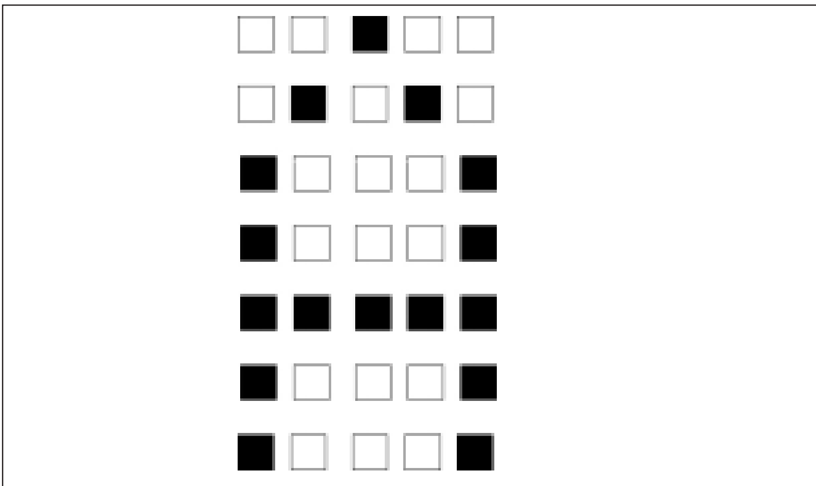

Figure 8. Representation of the letter A with a 5x7 pattern

We wish to distinguish alphabetic characters by assigning them to different bins. We would apply the inputs and train the network with anticipated responses. Here is the input file that we used for distinguishing five different characters, A, X, H, B, and I as shown in Table 1.

Table 1
Characters Representation

Letter	Representation
A	0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 0 0 0
X	1 0 0 0 1 0 1 0 1 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 1 0 1 0 0 0 1 0 1 0
H	1 0 0 0 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0
B	1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 0 0 1 1 0 0 0 1 1 1 1 1 1 1 0 1
I	0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 1 1 1

Each line has a 5x7-dot representation for each character. Now we need to name each of the output categories. We can assign a simple 3-bits representation as follows:

A	000
X	010
H	100
B	101
I	111

Let's train the network to recognize these characters. The training.dat file looks as in Figure 9:

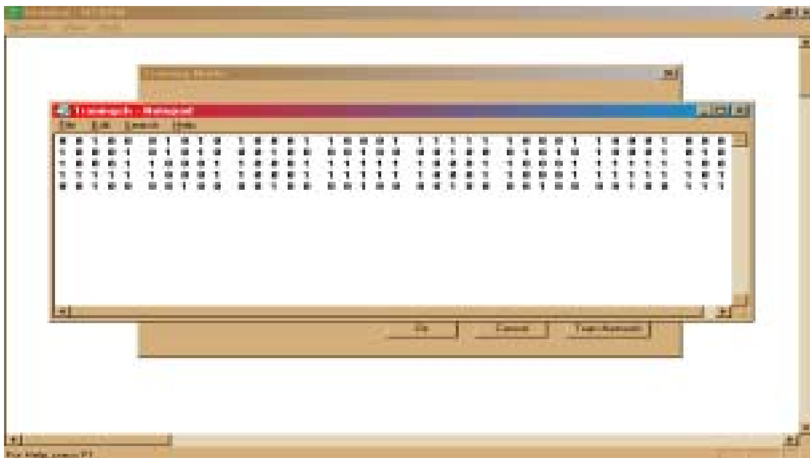

Figure 9. Training file for characters recognition

Now we can start training the network with parameters (learning rate 0.1, tolerance 0.001, and repetition count 1000) and with three layers of sizes 35 (input), 5 (middle), and 3 (output), we get after total cycles of 3322 a final error of 0.00164419. We note also that the tolerance specified is nearly met. The match for the last pattern is:

For input vector:

```
0.000000 0.000000 1.000000 0.000000 0.000000 0.000000 1.000000
0.000000 1.000000 0.000000 1.000000 0.000000 0.000000 0.000000
1.000000 1.000000 0.000000 0.000000 0.000000 1.000000 1.000000
1.000000 1.000000 1.000000 1.000000 1.000000 0.000000 0.000000
0.000000 1.000000 1.000000 0.000000 0.000000 0.000000 1.000000
Output vector is: 0.979606 0.985861 0.989526
Expected output vector is: 1.00000 1.00000 1.00000
```

To see the outputs of all the patterns, we need to copy the training.txt file to the test.txt file without the expected output field and rerun the system in test mode. Running the system in test mode shows the following result in output.txt file as shown in Figure 10. We noted that the training patterns are learned very well. If a smaller tolerance is used, it would be possible to complete the learning in fewer cycles.

Figure 10. Results of characters recognition in testing mode

CONCLUSION

In this article we have used an object-oriented approach for implementing one of the most powerful neural network algorithms called backpropagation algorithms. The algorithm uses the so-called generalized delta rule and trains the network with exemplar pairs of patterns. The main motivation for using object-oriented features as encapsulation and abstraction that we can have well packaged, reusable, extensible, and reliable programs and program segments.

References

Jacek, Z.M. (1995). *Introduction to artificial neural systemsy*. Boston: PWS Publishing.

Lefteri, T.H., & Robert, U.E. (1997). *Fuzzy and neural approaches in engineering*. New York: Wiley.

LaFore, R. (1999). *Object-oriented programming in C++*. Indianapolis, IN: Sams.

Mohamed, R., & Laitinen, M. (1998). *Transition to OO software development*. New York: Wiley

Grady, B., James, R., & Ivar, J. (1999). *The unified modeling language user guide*. Addison-Wesley.

Acknowledgements

I wish to express my sincere thanks, appreciation, and gratitude to Professor Samia Siad Al-Azab and Professor Farouk Aeob for their invaluable suggestions on this work.

APPENDIX A

The header file for the CBaseLayer class hierarchy and the network class CNNtwork:

```
#define MAX_LAYERS 5
class CNNtwork;
class CBaseLayer
{
protected:
        int num_inputs;
        int num_outputs;
        float* outputs; // pointer to array of outputs
        float* inputs;//pointer to array of inputs, which
            //are outputs of some other layer
        friend CNNtwork;
public:
        virtual void calc_out()=0;
};
class CInputLayer: public CBaseLayer
{
public:
        CInputLayer (int,int);
        ~ CInputLayer ();
        virtual calc_out();
};
class CMiddleLayer;
class COutputLayer: public CBaseLayer
{
protected:
        float* weights;
        float* output_errors;//array of errors at output;
        float* back_errors;//array of errors back-propagated to inputs
        float* expected_values; //to inputs
    friend CNNtwork;
public:
        COutputLayer (int,int);
        ~ COutputLayer ();
        virtual void calc_out();
        void calc_error(float &);
        void randomize_weights();
        void update_weights(const float);
        void list_weights();
        void write_weights(int, FILE*);
        void read_weights(int,FILE*);
        void list_errors();
        void list-outputs();
};
class CMiddleLayer: public COutputLayer
{
private:
```

```
        CMiddleLayer (int,int);
        ~ CMiddleLayer ();
        void calc_error();
};

class CNNtwork
{
private:
        CBaseLayer* ptr[MAX_LAYERS];
        int num_of_layers;
        int layer_size[MAX_LAYERS];
        float* buffer;
        unsigned training;
public:
        CNNtwork ();
        ~ CNNtwork ();
        void set_training_value(const unsigned &);
        unsigned get_training_value();
        void get_layer_info();
        void set_up_network();
        void randomize_weights();
        void update_weights(const float*);
        void write_weights(FILE*);
        void read_weights(FILE*);
        void list_weights();
        void write_outputs(FILE*);
        void list_outputs();
        void list_errors();
        void forward_prop();
        void backward_prop(float &);
        int fill_IObuffer(FILE*);
        void set_up_pattern(int);
};
```