Object-oriented programming provides an interesting approach to implementing an already interesting collection of artificial intelligence algorithm known as neural networks. Neural networks are learning algorithms capable of generalizing a set of examples into categories, discovering inherent relationships in data, and recognizing patterns from incomplete or distorted inputs. The applications of neural networks are countless ranging from OCR, stock price analysis, data interpretation, to user identification; however, this article is concerned with the implementation of neural networks rather than their applications. The architecture of neural networks is ideally suited for object oriented programming easily exploiting encapsulation, inheritance, and polymorphism. Part 1 of this article will define the base classes for a generic neural network architecture and illustrate how the class can be used for build neural networks. Part 2 of this article will use the base classes to implement two different types of neural network and demonstrate well object-oriented programming and neural networks work together.

Neural networks fit into the object-oriented paradigm so well because of the basic structure of what neural networks are modeling. On a very simplistic level, artificial neural networks are simulating the functionality of the cells in the human brain. Just like neuron in the brain are connected to other neuron via synapse forming a biological neural network, a node is connected to other nodes via links to form an artificial neural network. Both types of neural networks (biological or artificial) take stimulus or set of inputs, process them, and produce a reaction or set of outputs.

Other Object-Oriented Approaches

Several other object-oriented implementations of neural networks exists. Masters[3] presents a very coarse-grained approach using a base network class as the foundation from which the different neural network architectures are derived.

Another approach by Blum[4] uses a base network class as its foundation, but it also uses a vector class to implement nodes and a matrix class to implement connections.

There are several disadvantages to Blum's and Masters' approaches. They are both designed from the top down which from a software engineering standpoint does not easily support reuse or a fine-grained object design. These approaches do not take full advantage of the object-oriented nature inherent in a neural network; the network nodes are simply data structures embedded into the derived classes. Therefore, all nodes will be homogenous. Also, since the derived network classes must establish their own node representation, operating methods, and training procedures, a great deal of coding is required anytime a new network architecture is needed. Granted a new network class could be derived from another class; however, the modifications that can be made may be limited. Also, the networks that can be created with these classes work very well for the generic implementations they are designed for; however, if a more creative architecture, such as one using different node types, these classes do not readily support this feature.

A third approach by O'Brien[5] does utilize finer-grained objects to construct neural networks. He uses a network class, a node class, and a connection class. However, the network class does more work than is needed; the nodes are defined as layers in the network class. Therefore, the base class is
geared toward a neural network architecture like the backpropagation neural network. Networks like the Kohonen self-organizing network would be very difficult to implement with these base classes. Also, the connection class performs most of the work for the network; this design decision is not wrong, but it does seem to drift away from the natural object nature of network nodes.

Fine-Grained Object-Oriented Approach

As mentioned above, the node will be the base object for this approach. There will also be a network class and a link class (similar to O'Brien's connection class). The backpropagation neural network is used to illustrate the fine-grained objects since it is the most popular neural network; however, the set of objects works equally well for other architectures such as recurrent-backpropagation, counter-propagation, and Kohonen self-organizing.

The Base Node Class

A neural network is a collection of interconnected nodes. The node is the most primitive element; therefore, a base node class is provided to derive other node classes from. A node has two major responsibilities: producing an output based on the given inputs and adjusting itself so that it can learn. A node must also maintain a value (or set of values), an error value to be used in the learning process, and a set of connections or links to other nodes. The base node class is shown below:

```cpp
class Base_Node
{
  private:
    struct Link_List    // List of node links
    {
      Base_Link *link;  // Link Instances
      Link_List *next;  // Next link in list
    }
  protected:
    double *value;              // Values stored by this node
    int size;                   // Num of Values stored
    double error;               // Error value of Node
    Link_List *from_list;       // From Connections
    Link_List *to_list;         // To Connections
  public:
    Base_Node( int Value_Size=1 )  // Constructor
    {
      size=Value_Size;
      value=new double[size];
      from_list=NULL;   to_list=NULL;
    }

    virtual void Run( void ){};     // Compute Value
    virtual void Learn( void ){};   // Adjust node
    double Get_Value( int id )      // Return value
    { return value[id]; };
    void Set_Value(int id,double new_val) // Set value
    { value[id]=new_val; }
    double Get_Error( void )   // Return Network Error
    { return error; }
    void Set_Error( double new_error )
    { error=new_error; }
};
```
void Add_To_Link( Base_Link *link )
{
    Link_List *temp=new Link_List;
    temp->link=link;   temp->next=to_list;
    to_list=temp;
}
void Add_From_Link( Base_Link *link )
{
    Link_List *temp=new Link_List;
    temp->link=link;   temp->next=from_list;
    from_list=temp;
}

The base node provides operations to make connections between it and other nodes and operations to set
and retrieve the node's values. Run and Learn are virtual functions that must be defined by the derived
classes. Having the two virtual functions allows the basic structure of the node to be established at a
very low level while leaving implementation details of the specific architecture to a derived class. The
two functions are not pure virtual functions because the base node can be used as it is with nothing ex-
ecuted for Run and Learn. The data housed in the node class is completely encapsulated (unlike
O'Brien's classes) and the other operations supplied by the base node class will free the derived classes
from code that is common to all nodes in the network.

The Base Link Class

The base node class supplies an operation to link the node it instantiates to other nodes. The pa-
rameter required by the connection operations is a pointer to a base link object shown below:

class Base_Link
{
    protected:
        double *value;    // Value(s) for Link
        Base_Node *from;  // Link is coming from
        Base_Node *to;    // Link is going to

    public:
        Base_Link( Base_Node *from_node, 
                    Base_Node *to_node,int size )
        {
            value=new double[size];
            from=from_node; to=to_node;
        }

        virtual double Get_Value( int id )
        { return value[id]; };   // Return Link value
        virtual void Set_Value( double new_val,int id)
        { value[id]=new_val; }; // Set Link value
        Base_Node *To_Node( void )
        { return to;};   // Return ptr to node pointing to
        Base_Node *From_Node( void )
        { return from; }; // Return ptr to from node
    }

One may question why the link class is necessary at all since this information could easily be stored in
the base node class. However, having the link class separate from the node class allows a variety of de-
derived links to be used simultaneously in the same network. Also, the virtual Get_Value and Set_Value functions can be redefined in a derived class allowing more functionality to be placed in the link itself if needed.

The Base Network Class

class Base_Network
{
  protected:
    int net_size; // Max number of nodes in Network
    Base_Node **Node_List; // List of network nodes
  
  public:
    Base_Network( int size ) // Constructor
    {
      net_size=size;
      Node_List=new Base_Node*[net_size];
    }

    void Define_Node( int id, Base_Node *new_node )
    {
      Node_List[id]=new_node;
    }

    Base_Node *Get_Node( int id )
    {
      return Node_List[id];
    }
};

The base network class does very little. Its main task is to create an array of pointers to base nodes. Through the use of dynamic binding, any node derived from the base node class can be used in the network. This feature gives a tremendous amount of power and flexibility to the neural network. The base network class also provides two more operations: one to assign a network pointer to a node and the other to return a pointer to a node in the network.

Backpropagation Neural Network Example

The backpropagation network will now be constructed using the three base classes. The architecture of a backpropagation neural network is arranged into three or more layers: an input layer, an output layer, and one or more middle layers. Each layer contains some number of nodes each connected to the nodes in the neighboring layers. The nodes in the different layers are slightly different from one another; however, they are easily derived from the base node class or from each other. The simplest nodes are found in the input layer. These nodes only need to pass the input values from the problem being solved to the nodes in the first middle layers; therefore, the base node class can be used as the input node class.

The output node class can be derived next. Here the output of each node is computed by taking the sum of each node's value that is linked to the output node multiplied by the link's value. The sum is then run through a transfer function (usually a sigmoid function) and then becomes the output node's value.

class BP_Output_Node : public Base_Node
{
  protected:
    double Alpha; // Momentum Term
    double Beta;  // Learning Rate
    double bias;  // Node's Bias Term
The output node class simply fills in the Run and Learn functions with the appropriate actions from the backpropagation neural network algorithm. The transfer function is virtual because it is often desirable to change it to something other than the sigmoid function. Also, another node class could be derived from the output node class, the transfer function could be replaced with something other than the sigmoid function, and both node classes could be used in the same network. The Compute_Error function is virtual because the only difference between an output layer node and a middle layer node is the way the error is computed for the delta rule. Therefore, the middle layer node class is as follows:

```cpp
class BP_Middle_Node: public BP_Output_Node
{
    public:
        BP_Middle_Node( double Learning_Rate,  // Constructor
                        double Momentum );
    virtual double Compute_Error( void )
    { return value[0]*(1.0-value[0])*(error-value[0]);};
};
```
The base link could be used as it is for the backpropagation neural network; however, to simplify the coding during instantiation of the links and initialization, the following class is provided:

#define LINK 0
#define PREV_DELTA 1
class BP_Link : public Base_Link
{
public:
    BP_Link( Base_Node *from, Base_Node *to ):Base_Link(from,to,2)
    {
        value[LINK]=Random(-1,1); // Init weight
        value[PREV_DELTA]=0.0; // Set prev delta to 0
    }
};

The BP_Network class is derived from the base network class. This class simply directs the actions that take place. It does not perform any of the computations itself, nor does it dictate how the network is put together. The BP_Network class is shown below:

class BP_Network : public Base_Network
{
public:
    BP_Network( int Max_Nodes ):Base_Network(Max_Nodes){};  // Constructor
    void Load_Input( Data_Class &data ) // Load Data into // the Input Layer
    {
        for (int i=0; i<data.Get_In_Size(); I++)
            Node_List[i]->Set_Value(0,data.Get_In(i));
    }
    void Load_Output( Data_Class &data ) // Load Data into // the Output Layer
    {
        for (int i=0; i<data.Get_Out_Size(); i++)
            Node_List[net_size-data.Get_Out_Size() + i]->
                Set_Error(data.Get_Out(i));
    }
    void Run( Data_Class &data ) // Forward Pass
    {
        Load_Input(data);
        for (int i=0; i<net_size; i++)
            if (Node_List[i]!=NULL) Node_List[i]->Run();
void Learn( Data_Class &data ) // Backward Pass
{
    Load_Output(data);
    for (int i=net_size-1; i>=0; i--)
        if (Node_List[i]!=NULL)
            Node_List[i]->Learn();
};

void Make_Link( Base_Node *from, Base_Node *to )  // Link 2 nodes
{
    Base_Link *link=new BP_Link(from,to);
    from->Add_From_Link(link);
    to->Add_To_Link(link);
};

double Get_Error( int id )                   // Get Network error
{ return Node_List[id]->Get_Error(); }; }

Note that the Run and Learn functions simply go through the list of nodes and execute the lower level Run and Learn functions. Dynamic binding is used to execute the appropriate functions since the network can use any combination of derived node classes. The Load_Input and Load_Output functions are provided to simplify transferring the training set information.

Everything needed to create a backpropagation neural network has now been presented (see the class hierarchy in figure 1). The fine-grained objects allow a tremendous amount of control over the backpropagation network that will be constructed. For example, the user can establish however many layers he or she desires, make any set of connections between the nodes, and can even set the learning rate and momentum term to different values for each node. Most commercial packages do not allow the user this much control over the network.

The actual use of the objects is shown below:

void main( void )
{
    Data_Class data; // Class to hold training set
    int data_count, inputs, outputs;

    data.Load_Data("xor.trn", // Load training set from file
        data_count,inputs,outputs);

Figure 1: Class Hierarchy
BP_Network net(6); // Instantiate BP network
    // with 6 nodes

int i, j;
double Learning_Rate=0.4;
double Momentum=0.9;

for (i=0; i<2; I++) // First two are Input layer nodes
    net.Define_Node(i, new Base_Node);

for (i=2; i<5; I++) // Next 3 nodes are middle layer nodes
    net.Define_Node(i,
        new BP_Middle_Node(Learning_Rate,Momentum));

net.Define_Node(5, new // Last node is an Output layer node
    BP_Output_Node(Learning_Rate,Momentum) );

for (i=0; i<2; i++)  // Connect each input layer node
    for (j=2; j<5; j++)  // to each middle layer node
        net.Make_Link(net.Get_Node(i),net.Get_Node(j));

for (i=2; i<5; I++) // Connect each middle layer node
    // to the output layer node
    net.Make_Link(net.Get_Node(i),net.Get_Node(5));

double net_error=1.0;
int good=0;
int iteration=0;
while (good<4) // Train until all 4 items in
    // training set are learned
{
    net_error=0.0;  // Reset network error
    good=0;
data.Reset_to_Head();

    for (i=0; i<data_count; i++)  // Present each
        // training set // item to network
{
        net.Run(data);  // Perform forward pass
        net.Learn(data); // Perform backward pass

        if (fabs(net.Get_Node(5)->Get_Value(0)-
            data.Get_Out(0))<0.5) good++;

        net_error+=pow(net.Get_Error(5),2);
        data.Next_Data();
    }
}
Here a training set is loaded from disk (this example learns the classic XOR problem). A network is created with six nodes, the first two nodes are instantiated as input layer nodes, the next three nodes are instantiated as middle layer nodes, and the last node is instantiated as an output layer node. Connections are made in the traditional manner, every node in the input layer is connected to every node in the middle layer, and every node in the middle layer is connected to every node in the output layer. A loop is then performed to execute the Run and Learn functions until all patterns in the training set are learned within a specified tolerance.

**Conclusion**

The neural network objects presented in this paper facilitate the design and creation of neural network architectures. Unlike other object-oriented approaches designed from the top down, this approach attempts to exploit the inherent object-oriented nature of neural networks while at the same time giving the user more control and flexibility in his or her implementations. The objects are not restricted to or biased toward any specific neural network architectures. Also, the objects provide a consistent interface to neural network design operations which could easily be exploited by a design tool or a neural-network description language like NeuDL[6]. While the power and flexibility of these simple objects illustrate the suitability of object-oriented languages to problems like neural networks, it should be clear how important the design of such objects is and how greatly this design will impact the overall quality of the program design.

**References**