IMPLEMENTING NEURAL NETWORKS AND FUZZY LOGIC USING THE IMPORT SIMULATION LANGUAGE

Jeffrey W. Wallace
Space and Naval Warfare Command Systems Center San Diego
San Diego, CA
jwallace@spawar.navy.mil

ABSTRACT

Neural network technology has matured considerably over the last several years, but there are many potential application domains where the use of this technology is not widespread. In this case, we would like to explore the use of neural network modeling and fuzzy logic in the domain of process-based parallel and distributed simulation. In particular, we are interested in employing neural networks to cognitive modeling in command and control simulations.

In addition, one of our project goals is to explore the properties of the solution space using IMPORT [Whitehurst 95] neural network frameworks in conjunction with the DOME sub-language. To do so requires the establishment of a baseline research test-bed implementation. We present the API for an IMPORT CASE tool implementation of several traditional neural network paradigms, and work our way up to a discussion of the fuzzy associative memory (FAM) neural network, a technology which seems promising in the solution of cognitive modeling problems.

In closing, we illustrate our initial approach to a neural network simulation control framework based on the Model-View-Controller (MVC) paradigm.

INTRODUCTION

One of the key concepts in neural network modeling [Freeman 91] is that of *recall*. A neural network with the property of recall implies that one receives an expected output for a given input. In other words, if the same input is given to a properly trained neural network twice, the same output should result both times. The type of recall implemented in a given neural network is classified as *autoassociative* recall, or *heteroassociative* recall. Both concepts must be employed to implement a complete decision problem in cognitive modeling, and so we present layered neural network implementations.

Autoassociation is the ability to associate an input vector known to a neural system with itself. This implements the concept of *recognition*, e.g., given a known input, a neural system recognizes the input and outputs the same input vector.

Heteroassociation is the ability of a neural system to associate a known input vector with another vector. This is an inference mechanism which can be used to implement a stimulus-response model of cognition in a simulation.

The concept of *fuzzy remembrance* is the ability to recall the desired output vector correctly given incomplete or uncertain input data in an input vector. This is an extension of autoassociation, and the FAM framework we present provides us with a baseline implementation with which we may begin our advanced neural computing in modeling and simulation research.

We begin with an implementation discussion of the classical autoassociative neural network model, the *Hopfield Network*. The concept of layered networks is introduced in discussing the implementation of backpropagation networks. We then discuss the fuzzy associative memory (FAM) neural network model implementation [Kosko 92], and its role in a basic cognitive architecture.

In our implementations of neural networks there are three main issues we must address:

1) *Structure* - the architecture and topology of the network;
2) *Encoding* - the method of changing weights;
3) *Recall* - the method and capacity to retrieve information.

To utilize these neural systems in a simulation, we need a simulation interaction framework within which we can embed the recognition and decision-making capabilities of neural networks. The interaction framework is based on the Model-View-Controller (MVC) paradigm[Leonard 97]. The implementation of a *NeuralController* class demonstrates this concept.

HOPFIELD NEURAL NETWORKS

In our implementation of a simple Hopfield network we have two classes. The *Neuron* class and the *Network* class. In Figure 1 below we show a single layer network consisting of four neurons in a fully connected topology. In this implementation, as is common, a canonical weight matrix $\mathbf{W} = [w_{ij}]$ is formed based on the weights specified on the arcs.

We note that positive weights on the arcs tend to encourage agreement in a stable configuration, whereas negative weights tend to discourage agreement in a stable configuration.

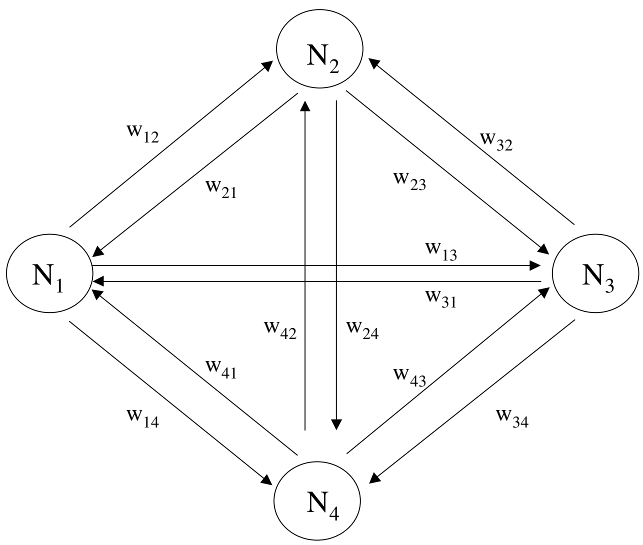

Figure 1. A single layer, fully connected four neuron network.

The *Neuron* class has an instance variable, *Activation*, which is the dot product of the input vector with the first row of the weight matrix, which is stored in the vector *Weightv* below. We assume that *Neurons* are not self connected.

The main purpose of the *Neuron* class is to maintain and calculate its state variable activation, and adjust the weights as training occurs and learning occurs. The method **Activation** takes an integer and an array of integers as arguments. The integer is the dimension of the input vector, and the array of integers is the input vector.

```
Neuron = OBJECT

   activation : INTEGER;

   Weightv : ARRAY[MAX] OF INTEGER;
   CONSTRUCTOR METHOD ();
   CONSTRUCTOR METHOD (INOUT j :
ARRAY[MAX] OF INTEGER);
   ASK METHOD act(IN a : INTEGER;
IN w : ARRAY[MAX] OF INTEGER): INTEGER;

END OBJECT;
```

The *HopfieldNetwork* class shown below maintains a collection of *Neurons* and the output pattern in the vector *output*. The method **Activation** fires the neurons and applies the **Threshold** method to their individual outputs and stores the results in the output vector. The method **GetResponse** can be accessed to provide the response pattern to the initial stimulus pattern.

```
HopFieldNetwork = OBJECT

   Neurons : ARRAY[MAX] OF Neuron;
   Output : ARRAY[MAX] OF INTEGER;
   ASK METHOD Threshold(IN thresh :
INTEGER): INTEGER;
   ASK METHOD Activation(IN stimulus :
ARRAY[MAX] OF INTEGER);
   ASK METHOD GetResponse(): ARRAY[MAX]
OF INTEGER;
CONSTRUCTOR METHOD(IN array : ARRAY[MAX]
OF INTEGER);

END OBJECT;
```

Hopfield networks are usually designed around the recall of a specific set of input vectors. These input vectors are referred to as exemplars, and are used in the case where iterative learning does not take place. This is an example of the prototypical memory cell.

BACKPROPAGATION NETWORKS

The feed-forward backpropagation neural network is a very popular model in neural networks. It does not have feedback connections, but errors are backpropagated during training. Various nonlinear objective functions can be used to measure error, but least mean square error is frequently used. This type of network is the prototypical of a neural network which must be trained, and learns from errors.

The distinguishing feature of the *FFBackProp* class is the use of neural *Layer* classes. These look similar to the Hopfield class, but they differ in that they have all the features necessary to incorporate error calulations from the following layer into calculations for the next iteration. The *Layer* class is specialized into *InputLayer*, *OutputLayer*, and *MiddleLayer* classes, due to the bookkeeping required of input vector and output vector marshalling at the beginning and end of a calculation sweep through the layers.

Thus, successful application of this technique requires the identification of a stimulus set **S**, and a response set **R**. A mapping between **S** and **R** must be defined, and a list of stimulus-response pairs enumerated into a set **E**, called the set of *exemplars*. The network is subjected to a training period where it is taught to recognize the correct response to a given stimulus. Figure 2 illustrates the action of the network.

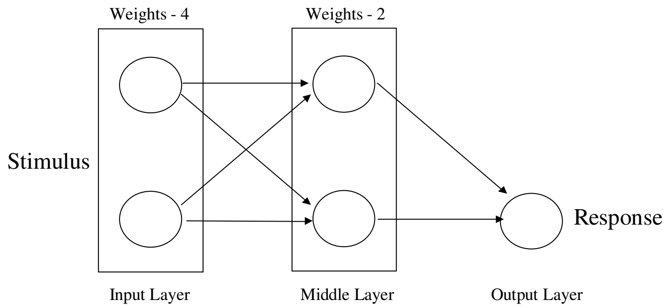

Figure 2. Basic stimulus-response architecture of a feed-forward backpropagation neural network.

The *Neuron* class is the same as before, except that a list of the outputs the neuron contributes to must be maintained, and it must have the ability to adjust its weights as it learns.

```
Neuron = OBJECT

   activation : REAL;
    numouputs : INTEGER;
    ids : ARRAY[MAX] OF INTEGER;

   Weightv : ARRAY[MAX] OF REAL;
   CONSTRUCTOR METHOD ();
   CONSTRUCTOR METHOD (INOUT j :
ARRAY[MAX] OF REAL);
   ASK METHOD act(IN a : REAL;
IN w : ARRAY[MAX] OF REAL): REAL;
ASK METHOD adjust(IN n : INTEGER; IN
delta : ARRAY[MAX] OF REAL);
ASK METHOD addoutput(IN id : INTEGER);
ASK METHOD getoutput(IN numid :
INTEGER;OUT ids : ARRAY[MAX] OF
INTEGER);

END OBJECT;
```

The base *Layer* class inherits from a base class *NeuralNetwork* similar in function to the Hopfield network class above, in terms of firing neurons and applying threshold functions to determine the response pattern.

```
Layer = OBJECT(NeuralNetwork)

   NumInputs : INTEGER;
   NumOutputs : INTEGER;

// array of output response

   outputs : ARRAY[MAXO] OF REAL;

// array of input stimuli, which
// are outputs of some other layer

   inputs : ARRAY[MAXI] OF REAL;

   ASK METHOD calculateOutput(): INTEGER;
END OBJECT;
```

The *InputLayer* class specializes the constructor and the **calculateOuput** method.

```
InputLayer = OBJECT(Layer)

   CONSTRUCTOR METHOD(IN in,out :
INTEGER);
   DESTRUCTOR METHOD();
   ASK METHOD calculateOutput(): INTEGER;

END OBJECT;
```

The *OutputLayer* inherits the basic *Layer* properties, but adds the necessary methods to begin error backpropagation.

```
OutputLayer = OBJECT(Layer)

   Outputerrors : ARRAY[MAXO] OF REAL;
   backerrors: ARRAY[MAXI] OF REAL;
   expectedvalues: ARRAY[MAXI] OF REAL;
   \/ to inputs

   CONSTRUCTOR METHOD (IN in,out :
INTEGER);
   DESTRUCTOR METHOD ();
   ASK METHOD calcout();
   ASK METHOD calcerror(INOUT accel:
REAL);
   ASK METHOD randomizeweights();
   ASK METHOD updateweights(IN tol:
REAL);
   ASK METHOD listweights();
   ASK METHOD writeweights(IN n :
INTEGER; IN outFile : OutFile);
   ASK METHOD readweights(IN n: INTEGER;
IN inFile : InFile);
   ASK METHOD listerrors();
   ASK METHOD listoutputs();
END OBJECT;
```

The *MiddleLayer* specializes *OuputLayer* in its initialization and method of calculating error for backpropagation.

```
MiddleLayer = OBJECT(OutputLayer)

   CONSTRUCTOR METHOD ();
   DESTRUCTOR METHOD ();
   ASK METHOD calcerror();
END OBJECT;
```

The main feature of the *FFBackProp* class is the use of the *Layer* classes. As illustrated in Figure 2 above the firing of neurons occurs in each layer from input through the middle and output layers in sequence, hence the name feed-forward. The errors in recognition made at each stage are simultaneously fed backward for use in the next firing wave. The *Training* instance variable controls error backpropagation. In the learning mode, this is true and backpropagation occurs. When the network is being used as a model in a simulation, this flag is false.

```
FFBackProp = OBJECT

     NumberMLayers : INTEGER;
     layersize : ARRAY[MAXL] OF INTEGER;
     inputLayer : InputLayer;
     middleLater : ARRAY[MAXL] OF
MiddleLayer;
     ouputLayer : OutputLayer;
     buffer : ARRAY[MAXS] OF REAL;
     Training : BOOLEAN;


     CONSTRUCTOR METHOD();
     DESTRUCTOR METHOD();
    ASK METHOD settraining(IN trn :
BOOLEAN);
    ASK METHOD gettraining():BOOLEAN;
    ASK METHOD getlayerinfo();
    ASK METHOD setupnetwork();
    ASK METHOD randomizeweights();
   ASK METHOD updateweights(IN w : REAL);
   ASK METHOD writeweights(IN out :
OutFile);
   ASK METHOD readweights(IN in :
InFile);
   ASK METHOD listweights();
   ASK METHOD writeoutputs(IN out :
OutFile);
   ASK METHOD listoutputs();
   ASK METHOD listerrors();
   ASK METHOD forwardprop();
   ASK METHOD backwardprop(INOUT w :
REAL);
   ASK METHODfillIObuffer(IN in :
   InFile):
      INTEGER;
   ASK METHOD setuppattern(IN p :
INTEGER);

END OBJECT;
```

The next step in developing a useful cognitive architecture is to illustrate how errors in the perception of inputs can be sharpened and then fed into FFBP networks to obtain a stimulus.

FUZZY ASSOCIATIVE MEMORY NEURAL NETWORKS

Real world data is never crisp, in terms of reliability, but fuzzy. Most inference procedures, including those above, would prefer crispness in the data which they operate, including the artificial intelligence inference engine in the DOME subsystem of IMPORT. Thus, a creditable cognitive architecture must have some method of focusing on (or providing crispness to) the nature of perceived truth, or input data, in models and simulations.

The fuzzy associative memory neural network is one technology with which to model this process. A set of responses and stimuli are encoded in the exemplar set **E** as above. A fuzzy input **X** from the stimulus set is presented to the network. A response **R** must be found from the exemplar set. For each **Y** in the response set, the following process occurs.

There are two layers of fuzzy neurons, **A** and **B**. The **B** fuzzy neuron layer is used with **X** as input to identify **Y** as a candidate for unification. If so, this potential pairing is then checked against the **A** layer fuzzy neurons to see if actual unification occurs. The basic algorithm for response determination to a stimulus is:

1) The **X** vector is given to the network. We find the activations of the **B** layer neurons and get the corresponding output pattern. If this is **Y** in the examplar pair, then the network has made an association in one direction, and we go on to step two. Otherwise we have a potential association pair, **X** and the output vector **O**. We call this a potential association, and collect this in a list and proceed to the next step to see if an actual association has occurred.
2) The potential associations collected so far are then run through the **A** fuzzy neuron layer to see if **X** is obtained in the opposite direction. The successful response unifications with **X** are then collected.
3) The methods **compare1** and **compare2** in the *FAMNetwork* class below verify if the pairs are in fact associated pairs.
4) The methods **compute1** and **compute2** in the *FAMNetwork* class carry out the calculations to get the activations and then find the final network output vector **R**.

The implementation below is somewhat simplified for illustration, but does contain the essential elements.

```
FuzzyNeuron = OBJECT

   nnbr : INTEGER;
   inn,outn : INTEGER;
   output : REAL;
   activation : REAL;
   outwt : ARRAY[MXSIZ] OF REAL;
   Name : STRING;

   CONSTRUCTOR METHOD ();
   ASK METHOD getNeuron(IN nb,in,out:
INTEGER);
END OBJECT;
```

The *Exemplar* class contains stimulus response pairs.

```
Exemplar = OBJECT

   xdim,ydim : INTEGER;
   v1, v2 : ARRAY[MXSIZ] OF REAL;

   CONSTRUCTOR METHOD ();
   ASK METHOD getExemplar(OUT xd,yd :
INTEGER; OUT a1,a2 : ARRAY[MXSIZ] OF
REAL);
   ASK METHOD prexmplr();
END OBJECT;
```

The *AssocPair* class below keeps track of the associated pairs.

```
AssocPair = OBJECT

   xdim,ydim,idn : INTEGER;
   v1, v2 : ARRAY[MXSIZ] OF REAL;

   CONSTRUCTOR METHOD ();
   ASK METHOD getasscpair(OUT xd,yd,id :
INTEGER; OUT a1,a2 : ARRAY[MXSIZ] OF
REAL);
   ASK METHOD prasscpair();
END OBJECT;
```

The potential pairs are represented by the *PotlPair* class.

```
PotlPair = OBJECT

   xdim,ydim : INTEGER;
   v1, v2 : ARRAY[MXSIZ] OF REAL;

   CONSTRUCTOR METHOD ();
   ASK METHOD getPotlPair(getExemplar(OUT
xd,yd : INTEGER; OUT a1,a2 :
ARRAY[MXSIZ] OF REAL);
   ASK METHOD prpotlpair();
END OBJECT;
```

The basic FAM network is implemented below, using the classes just defined, in the *FAMNetwork* class

```
FAMNetwork = OBJECT

   anmbr,bnmbr,flag,
   nexmplr,nasspr,ninpt : INTEGER;
   Aneurons,
   Bneurons : ARRAY[MXSIZ] OF
               FuzzyNeuron
   Exemplars : ARRAY[MXSIZ] OF Exemplar;
   Asscpair : ARRAY[MXSIZ] OF AssocPair;
   Potlpair : ARRAY[MXSIZ] OF PotlPair;
   outs1,outs2 : ARRAY[MXSIZ] OF REAL;
   mtrx1, mtrx2 : ARRAY[MXSIZ][MXSIZ] OF
                  REAL;
   CONSTRUCTOR METHOD ();
   ASK METHOD compare1(INTEGER,INTEGER);
   ASK METHOD compare2(INTEGER,INTEGER);
   ASK METHOD printweights();
   ASK METHOD iterate();
   ASK METHOD findassc(INOUT assoc:
ARRAY[MXSIZ] OF REAL);
   ASK METHOD assigninput(REAL *);
   ASK METHOD assignvector(IN id:
INTEGER,IN
          a1,a2 : ARRAY[MXSIZ] OF REAL);
   ASK METHOD compute1();
   ASK METHOD compute2();
   ASK METHOD prstatus();
END OBJECT;
```

NEURAL NETWORKS IN SIMULATION

To use the neural network constructs developed so far, we must indicate how we intend to integrate this cognitive modeling capability into a simulation. The Model-View-Controller (MVC) interaction paradigm in IMPORT [Leonard 97] is an ideal framework within which we may embed a neural network-based cognitive model.

Figure 3 below illustrates our use of neural networks in the decision process.

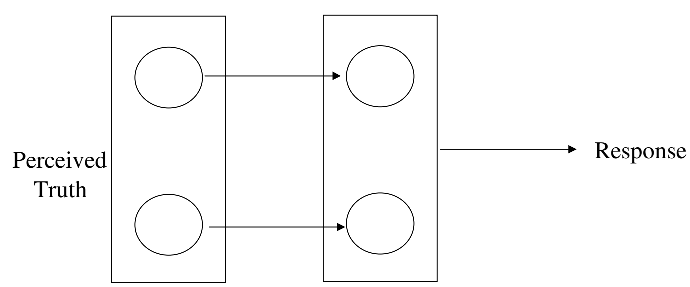

Figure 3. Neural network architecture.

Consider the following IMPORT pseudocode for the *NeuralController* class.

```
NeuralController = OBJECT(Controller)

    Model : ComplexModel;
    {***Associations that provide object
references which form the world view to
the controller are declared here***}

   FAM : FAMNetwork;
   FFBP : FFBackProp;

EXPORT ASK METHOD ActivelyControl()
BEGIN

WHILE( NOT ASK SELF TO GetDone() ) DO

   ASK FAM TO PerceiveWorld();
   ASK FFBP To CalculateResponse();
   ASK SELF TO WaitForChange();
   ASK SELF TO ResetChangeState();
END WHILE;

END METHOD;


EXPORT CONSTRUCTOR METHOD ()
INIT Controller();
BEGIN
END METHOD;

END OBJECT;
```

This allows the *NeuralController* to be activated when the *ComplexModel* which it controls changes state due to a change in the world view of the *ComplexModel*. Thus, the *FAM* filters the perception of the world to classify the stimulus, the autoassociation task, and *FFBP*, the feed-forward backpropagation neural network, calculates the response, the heteroassociation task.

FUTURE DIRECTIONS

We intend to extend these basic implementations into the field of hybrid neural networks, those which combine other artificial intelligence approaches, with neural networks. In particular, we are interested in describing the problem space addressed by the neural network concepts expressed above, and the expert system capabilities of the DOME subsystem of IMPORT.

ACKNOWLEDGEMENTS

The author wishes to acknowledge the Office of Naval Research 6.1 Basic Research program for their support.

REFERENCES

[Freeman 91] Freeman J. and D. Skapura. 1991. *Neural Networks Algorithms, Applications, and Programming Techniques*, Addison-Wesley, Reading, MA, 1991.

[Kosko 1992] Kosko B. 1992. Neural Networks and Fuzzy Systems: A Dynamical Systems Approach to Machine Intelligence, Prentice-Hall, New York, 1992.

[Leonard 97] Leonard G., L. Peterson, and J. Caldwell. 1997. “Using the Model-View-Controller Framework as a Simulation Development Methodology,” In *Object-Oriented Simulation 1997*, Wallace J. *et al* Eds., Simulation Councils, Inc., January 1997.

[Whitehurst 95] Whitehurst R. A. and V. P. Morrison. 1995. “IMPORT: A Language for Simulation of Complex Systems,” In *Object-Oriented Simulation 1995*, Roberts C. *et al* Eds., Simulation Councils, Inc., January 1995.