IDENTIFICATION OF PARALLELISM IN NEURAL NETWORKS BY SIMULATION WITH LANGUAGE J

Alexei N. Skurihin
Mathematics Department, Institute of Physics and Power Engineering
249020 Obninsk, RUSSIA. Email: kvm@feimo.obninsk.su

Alvin J. Surkan
Department of Computer Science and Engineering, University of Nebraska
Lincoln, Nebraska 68588-0115 USA Email: surkan@cse.unl.edu

Abstract

Neural networks, trained by backpropagation, are designed and described in the language **J**, an *APL* derivative with powerful function encapsulation features. Both the languages **J** [4,6,7] and *APL* [5] help to identify and isolate the parallelism that is inherent in network training algorithms. Non-critical details of data input and derived output processes are de-emphasized by relegating those functions to callable stand-alone modules. Such input and output modules can be isolated and customized individually for managing communication with arbitrary, external storage systems. The central objective of this research is the design and precise description of a neural network training kernel. Such kernel designs are valuable for producing efficient reusable computer codes and facilitating the transfer of neural network technology from developers to users.

Key words: neural network, backpropagation, simulation, MIMD architecture, function arrays, nested arrays, gerund.

Introduction

A neural network model consists of a processing system with a densely interconnected network of interacting units. The programming language **J**, derived from *APL*, includes some especially attractive capabilities for the design and implementation of neural networks. Among the **J** features are tacit programming, function arrays, nested arrays, new operators, and a mechanism for defining composite functions that facilitate changing system behavior by modifying component functions without respecifying the composite function in which the function definition appears [1,2,9].

To a large extent, the expressive power of **J**- and *APL*-like languages for the implementation of neural networks comes from the ability of the languages to process arrays directly. This feature is becoming increasingly important as vector processors and parallel computing become more widely available. Because of the parallelism in neural network models, it is relatively easy to fine tune the necessary defining parameters for nearly optimal performance. The parallelism also lends itself naturally to distributed processing. **J** and *APL* implemented simulators of the neural nets allow the refinement and pruning of a neural network kernel and help in locating the inherent parallelism in the network model's structure and operation.

Because of its present popularity, we chose backpropagation training to explore future directions for alternative parallel implementations of neural networks. Networks were trained for diagnosing two modes of experimental nuclear reactor operation, namely, its steady and transient states. Improvement in the performance of these systems is expected to result from new methods of: (1) creating networks, (2) modifying their structure, and (3) evaluating networks of alternative architecture.

Beyond the need for designing and refining a neural network kernel, there are several important reasons for identifying parallelism in the neural network models. One reason is to increase their speed. A second reason is to increase robustness with respect to changes in their environment. For example, the loss of a process or a processor in a neural network's environment should not stop processing.

Neural Network Parallelism

Neural net models are specified by network topology, node characteristics, and training or learning rules. These rules instantiate an initial set of weights and indicate how weights should be adapted to improve performance during their use. Descriptions of neural network structures and their representation in **J** is the focus of this research.

Both J and *APL* descriptions of neural networks are helpful for finding and exploiting parallelism. This enables the implementation of SIMD (Single Instruction Multiple Data) and MIMD (Multiple Instruction Multiple Data) architectures.

Inherently, J has the necessary capabilities to describe parallel simulators [1,2]. Some interesting and powerful capabilities that J provides are tacit definition, gerunds, forks, and hooks. Consequently, J makes it possible to create function arrays as gerunds. This allows the association of functions with individual layers or subnetworks of the entire neural network. It also makes it easy to describe the arguments of the functions.

Introducing function arrays is one effective way of implementing parallel computation (MIMD architecture). At first it is necessary to arrange the subnetworks within the main network. Then there must be a specification of the classes of connections permitted between subnetworks, numbers, destinations, and initial strengths of connections desired in the subnetwork before each run.

As a general rule, the connectivity remains fixed once it is established, but the connection strengths vary in accordance with rules for the synaptic modification chosen as the network's learning or adaptation mechanism.

One interesting idea is the development of an algorithm that enables evolution of the neural network structure and/or its topology. For this purpose, genetic algorithms appear to be a promising approach for the solution of learning or adaptation problems and deserves careful investigation.

Each subnetwork can have quite different operating principles. The subsystems may interact by re-entry to form populations of subnetworks. For example, mutually interacting oscillators [11] working together can perform more complex functions than either could perform separately. The specificity of each subnetwork depends on a list of connections and their strengths. Also, it is suggested that presynaptic and postsynaptic changes can occur on different time scales. The changes also can be the result of various mechanisms (for example, heterosynaptic effects, in which activity at one synapse affects the strengths of nearby synapses). Networks controlled be these rules display interesting dynamic effects. Activities of all subnetworks in the main network are calculated and then cyclically updated simultaneously.

A simulator of neural networks must incorporate a number of features. Those considered most relevant to the topic are as follows:

(1) A neural network can be constructed from an arbitrary number of subnetworks. Each subnetwork may contain one or more layers of nodes of different kinds. Individually, each layer may have its own rules for connectivity and synaptic modification.

(2) A network kernel must operate in an environment or be interfaced to manage communication with external systems. We explore the question: Does J provide capabilities for implementing such a kernel? In general, the answer is yes. However, some limitations still require a detailed examination. The J language has been found to have significant potential for developing and evolving such a kernel.

f_wm(**f**-response function, **wm**-weight matrix).

```
NB.         nd - vector contains node's distribution at layers
NB.         wm - output nested array contains total weight matrix
iw0 =.'      wm=:'''' [ p=.0 [ cn=.#nd '
iw1 =.' m1) wl =. '''' [ k=.1 '
iw2 =.' m2) dim =. (1+p{nd),k{nd '
iw3 =.'     k   =. >:k '
iw4 =.'     wl  =. wl,< (dim $ 0 )'
iw5 =.' m3) $.  =. >(k=cn) { m2;>:m3 '
iw6 =.'     p =. >:p '
iw7 =.'     wm=: wm,<wl '
iw8 =.' m4) $. =. >(p=cn) { m1;>:m4 '
g_iw=.     (iw0;iw1;iw2;iw3;iw4;iw5;iw6;iw7;iw8) : ''
```

Figure 1. Generation of initial weight matrix.

Data Structures for Neural Net Simulation with the J Language

The data structures used to express the operation of neural network models are compatible with the capabilities of J. This permits J to serve as a very general network simulation tool. In these simulations, the user interface provides (at the level of subnetworks, nodes, and connections) explicit control over the structure and size of the simulated networks. Control statements define and name entities for each type of subnetwork. The features of version 6.1 of **J** facilitate the design of the desired network_kernel_environment interface.

With a sequence of executable statements, a neural network is structured as a nested array. More precisely, the structure is described by a boxed, nested array. In this nested structure, (1) the external(top) level array is the main(whole) neural network, (2) internal arrays (boxed and nested inside the main one) are subnetworks, and (3) the lower level has the nodes level and connections. The nodes at each level are allocated dynamically and linked by **J** verbs "Append", "Box", "Gerund". Using these, we can construct nested arrays, functions arrays, and boxed arrays. Each nested array contains parameters that define the properties of corresponding objects.

Network Connection Matrix, **cnm**

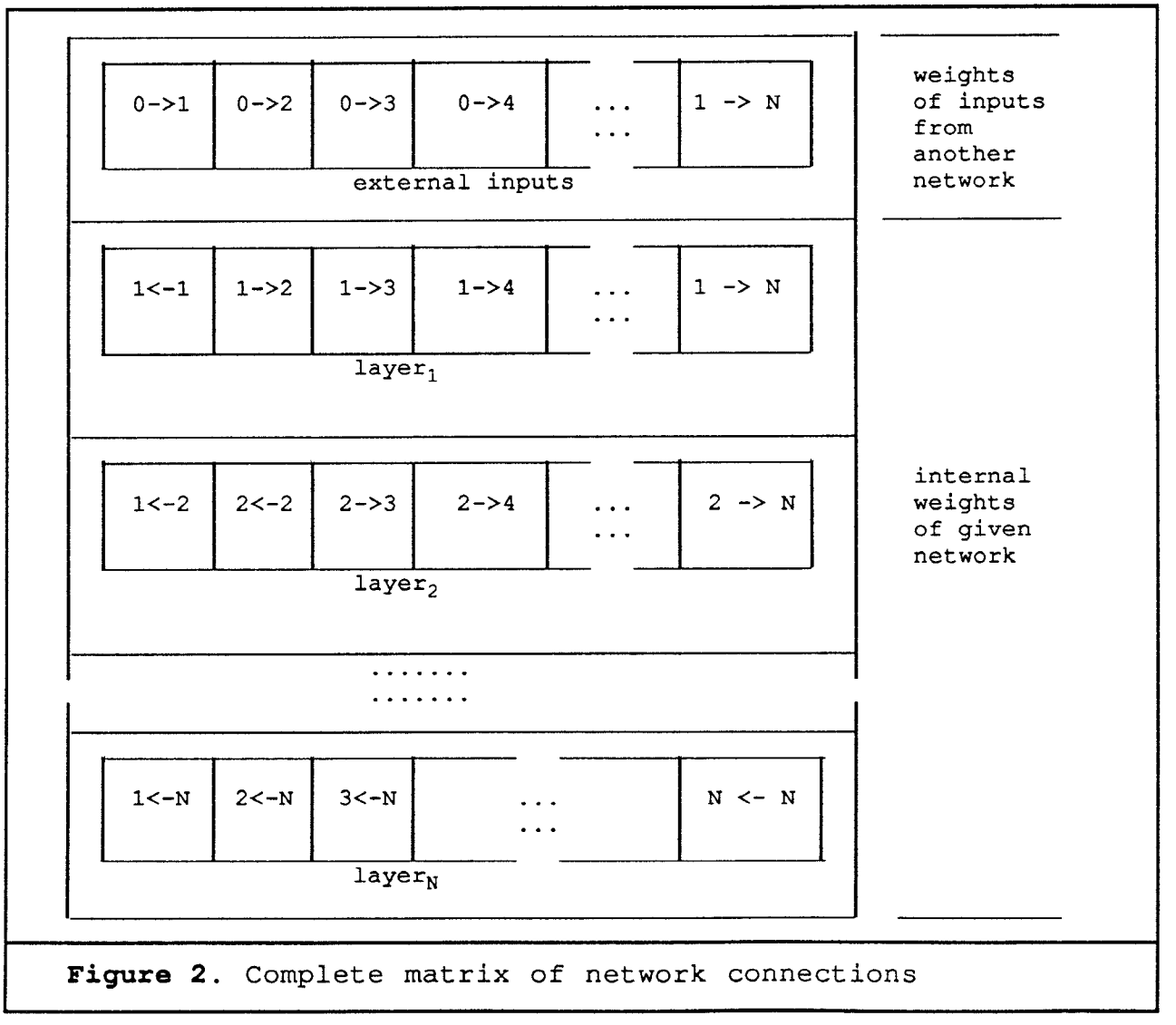

Figure 2. Complete matrix of network connections

The **J** implementation of the methods: (1) for the initialization or generation of the neural network, (2) for the implementation of the particular response function, and (3) for the net's learning are all similar. They are integrated with each other by appropriate choice of a network design method. When the network's kernel design is complete, it will be easy to construct different neural structures for supporting paradigms such as backpropagation or associative memory. Also, it will then be easier to integrate subnetworks into a single main network that provides different functions for the nodes, and to accommodate depression and refractory periods as well as lateral connectivity.

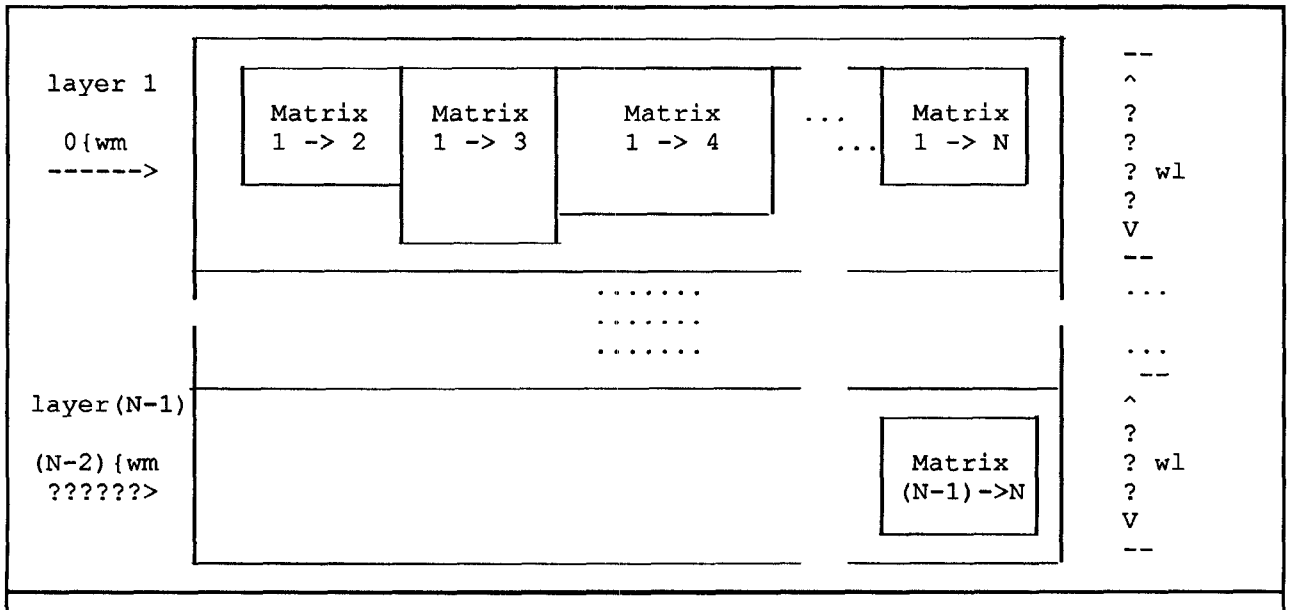

Figure 3. Complete connection structure for a feedforward network

Network Structures in the Language J

One simple example of a structure useful for representing neural network's connection matrices is shown in Figure 2 below. Let the main network consist of **N** layers and the vector **nd** for specifying the distribution of the nodes over the network's layers. The verb **g_iw** shown in Figure 1 describes the initialization of the weight matrix. Associations between the proper response functions and their respective layers are provided by a gerund. For example **f_wm** is defined in Figure 1 at the beginning.

Parallelism Identification in the J - Language Implementations

Implementation of neural networks in **J** can be structured ideally for parallel execution. For each time step, the processing at each node of the network is entirely independent of that at all other nodes. The only variables belonging to other nodes that are accessed during the evaluation of the state of node(i) at time **t** are **S(i,t)** and connection **C(i,j)** linking nodes **i** to **j**. **S(j,t)**'s are used for simulating parallel execution. **S(t)** and **S(t+1)** arrays are kept separately for each node type. This permits simultaneous updating of all activation values at the end of each cycle. In an actual multiprocessor hardware implementation, the same arrangement would permit **S(t)** to be read by other processors while **S(t+1)** is being calculated and written.

As a **J** program grows, difficulties in the construction and understanding of the system can become difficult. One way to overcome this difficulty is to introduce subspaces. Experience gained to date [3] shows that it is possible to design an Object-Oriented extended **J** interpreter which performs well and is adequately supported by tracing and browsing tools.

Beyond that, non-overlapping subspaces might be hosted on multiple processors. The **J** language allows the expression of parallel algorithms for machines with MIMD architectures.

J - Language Implementation of the Backpropagation Training Algorithm

As a starting point of this investigation, the basic backpropagation [8,10] was used. For this case, Figure 3 shows the structure of the network's connections above.

Figures 4 and 5 define two verbs **g_iwm** and **ag_wm** as variants which generate feed-forward connections.

```
NB.          GENERATION OF NESTED INITIAL WEIGHT MATRIX
NB.          USING   $. (Suite)
NB.          g_iwm ''
NB.    nhl -  number of hidden layers
NB.     n  -  vector contains numbers of nodes of input, hidden,
NB           output layers
NB.    wm  -  nested array contains total weight matrix

iw0 =.'      wm=:'''' [ p=.0 [ cwl=.2+nhl [ cwm=.1+nhl'
iw1 =.'  m1) wl =. '''' [ k=.>:p'
iw2 =.'  m2) dim =. (1+p{n),k{n '
iw3 =.'      k   =. >:k '
iw4 =.'      wl  =. wl,<( (dim $ %1+(*/dim)?(*/dim))*
                          (dim $ %1+(*/dim)?(*/dim))*(%2+?10)  )'
iw5 =.'  m3) $.  =. >(k=cwl) { m2;>:m3 '
iw6 =.'      p =. >:p '
iw7 =.'      wm=: wm,<wl '
iw8 =.'  m4) $. =. >(p=cwm) { m1;>:m4 '
g_iwm =.     (iw0;iw1;iw2;iw3;iw4;iw5;iw6;iw7;iw8) : ''
```

Figure 4. First variant for generating initial network connections

Second variant of J language program that generates a nested, initial-weight matrix with the verb **ag_wm.**

```
NB.        ALTERNATE GENERATION OF NESTED INITIAL WEIGHT MATRIX
NB.        USING   ^: (Power)
NB.        ag_wm ''
NB.        *********  IWL  **************
miw2 =.' m2) dim =: (1+p{n),k{n '
miw3 =.'   k   =: >:k '
miw4 =.'   wl  =: wl,<( (dim $ %1+(*/dim)?(*/dim))*
                        (dim $ %1+(*/dim)?(*/dim))*(%2+?10))'
iwl  =.    (miw2;miw3;miw4) : ''
NB.        *********  IWM  ***************
miw1 =.' m1) wl =: '''' [ k=:>:p '
mins =.'   iwl^:(cwl-k) ''''   '
miw6 =.'   p =: >:p    '
miw7 =.'   wm=: wm,<wl'
iwm  =.    (miw1;mins;miw6;miw7) : ''
NB.        ********* AG_WM **************
miw0 =.'   wm=:'''' [ p=:0 [ cwl=:2+nhl [ cwm=:1+nhl '
mt01 =.'   iwm^:cwm '''' '
ag_wm=.    (miw0;mt01) : ''
```

Figure 5. Second variant for generation of initial network connections

The result of the action of these verbs is the neural network weight matrix in the form of nested array. It consists of (number_of_layers - 1) elements similar to the ones shown in Figures 6 and 7 (the -1 diminishes the range because the output layer does not have any connections in this non-recurrent version). Each element contains a number of arrays corresponding to an interlayer connection matrix. The upper rows of the matrices represent thresholds.

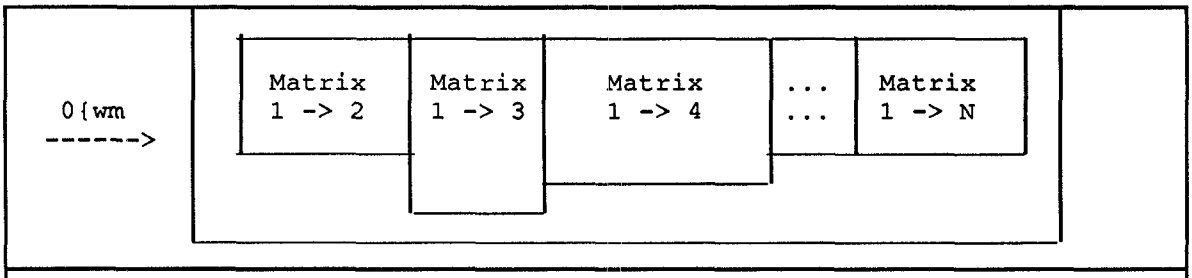

Figure 6. Boxed nested array containing connections between input layer and hidden layers, and output layer

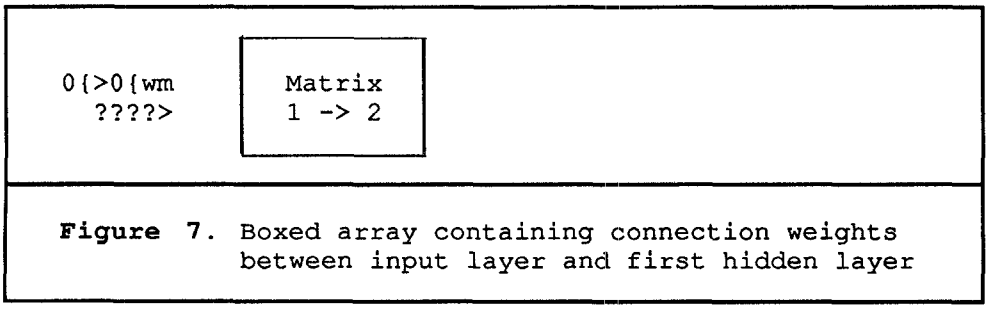

Figure 7. Boxed array containing connection weights between input layer and first hidden layer

To perform mathematical operations we have to open this matrix, i.e. >0{>0{wm. Then, after it is boxed, we can process it in reverse order. Shown below in Figure 8 is the calculation of output of nodes for all layers of the network:

```
NB.          CALCULATION OF OUTPUTS OF NET NODES AT ALL LAYERS
NB.          sn_all in
NB.          y. - right argument = in (in - boxed vector)
sn00 =.'    s_all =: y.[ psl=:'''' [ p=:0 [ csl=:2+nhl [ csa=:1+nhl'
sn01 =.' m1) sl =: '''' [ k=:0 [ csl=:<:csl '
sn02 =.' m2) sl =: sl,<( (1,>p{s_all)+/ .*(>k{>p{wm) )'
sn03 =.'     k  =: >:k'
sn04 =.' m3) $. =. >(k=csl) { m2;>:m3'
sn05 =.'     psl=: psl,<sl '
sn06 =.'     m =: p [ c =: 0 '
sn07 =.'     tsl=: >m{>c{psl '
sn08 =.' m4) $.=.>(c=p){ (>:m4);m5 '
sn09 =.'     m =: <:m [ c =:>:c '
sn10 =.'     tsl=:tsl+>m{>c{psl '
sn11 =.'     $.=.m4 '
sn12 =.' m5) s_all =: s_all,<(%1+^-tsl) '
sn13 =.'     p =: >:p '
sn14 =.' m6) $. =. >(p=csa) { m1;>:m6 '
signal =.  (sn00;sn01;sn02;sn03;sn04;sn05;sn06;sn07;sn08;sn09;sn10;
           sn11;sn12;sn13;sn14) : ''
```

Figure 8. Calculation of nodes outputs at all layers

Verb "signal" (see Figure 5) produces a boxed array **s_all** of (number_of_hidden_layers + 2) vectors of nodes output at each layer. For example, the number of nodes at net's layers =: 2 2 1 then **s_all** for some random input values is:

OUTPUT FROM VERB: SIGNAL

0.7 0.4	0.56 0.54	0.745

This method is extended easily. One can design both recurrent backpropagation, time-lagged recurrent networks and more complex network with a full coupling matrix. This matrix specifies a unique and modifiable coupling from each compartment of the neural model to each other compartment as in associative memory networks. Such a structure is implemented without any serious complications.

A future extension of this structure will add different function arrays to be associated with corresponding layers. Such structure resembles object-oriented programming to some extent. Also, the concept of subspaces is very attractive from this object-oriented viewpoint. At present the verbs "}" and "{" operate with nested arrays, but in future work the possibility of using rank ' " ' will be explore.

```
NB.  *  CALCULATION OF DELTA WEIGHT MATRIX, BACK-BACKPROPAGATION   *
NB.          mod_class_dw ''
NB.          out - class_dw
mo00 =.'     p =.>:nhl '
mo01 =.'     ds=.((>ic}desire)->p}s_all)*(>p}s_all)*(1->p}s_all) '
mo02 =.'m1)  p =.<:p '
mo03 =.'     class_dw=:(<(<(<(>_1{>p{class_dw)+(alfa*(1,>p{s_all)*/ds))
                                        _1} (>p{class_dw)) p} class_dw '
mo04 =.'m2)  $.=.>(p=0) { m1;>:m2 '
mh05 =.'     m=.1+nhl [ k=._1 [ ch=.nhl '
mh06 =.'m3)  m=.<:m '
mh07 =.'     ds=.(>m{s_all)*(1->m{s_all)*(ds +/ .*(|:(}.>k{>m{wm))) '
mh08 =.'     p=.m [ k=.<:k '
mh09 =.'m4)  p=.<:p '
mh10 =.'     class_dw=:(<(<(>k{>p{class_dw)+(alfa*(1,>p{s_all)*/ds))
                                      k} (>p{class_dw)) p} class_dw '
mh11 =.'m5)  $.=.>(p=0) { m4;>:m5 '
mh12 =.'     ch=.<:ch '
mh13 =.'m6)  $.=.>(ch=0) { m3;>:m6 '
me14 =.'     flag=:0 '
mod_class_dw =. (mo00;mo01;mo02;mo03;mo04;mh05;mh06;mh07;mh08;mh09;
                 mh10;mh11;mh12;mh13;me14) : ''
```

Figure 9. Calculation of the modification of network weight matrix for each class of patterns, **class_dw**.

In Figure 9, sentence **mh10** assigns a value to some element of the full nested array. However, it should not be necessary to reassign the total array in order to assign new values to several elements as was required with Version 4.1 of the **J** language). It is desirable to develop new alternative methods not requiring the reassignment all elements of the array. It should be different from:

array_a =. new_value (index of element)} array_a .

One possible solution to this problem is to organize the process so there is no use of the assignment operator. For this, our program will take the following form:

RESULT ← F_n ← ..... ← F_k ← ..... ← F_2 ← F_1 ← input_DATA

(result =. F_n ... F_k ... F_2 F_1 input_data) where F is a function.

But in that case our program must be executed in parallel. A version of feedforward batch backpropagation for a pattern recognition problem was implemented and tested. This focused on the application of neural networks to the monitoring and for decision making in nuclear power plant operation.

A two-category problem is used to test network learning. Two categories are sufficient for classifying the operating condition data that is monitored during nuclear reactor experiments. Patterns identified in the observed operating parameters characterize the operating conditions. The patterns need to be resolved into two classes. These classes must be defined so that they discriminate reliably between those data patterns representative of the reactor's steady and transient states. Temperature and neutron power spectral densities are introduced as input vectors. The frequency range is 0 - 1 Hz, and the layer size in the the structure of the neural network is described by the vector: 32 10 1.

Conclusions

Artificial neural network models are inherently highly parallel in their structure and operation. A direct approach to distributing their computational load is to provide a separate processor for each network node. The operations carried out at the different nodes are highly independent and do not require strict synchronization.

Features already provided in the J language such as gerunds, forks, and hooks support the implementation of parallel simulators of neural networks. The J language serves as a flexible tool for generating neural networks with a variety of architectures and for modeling them using many different simulation protocols. Such implementations are expected to provide powerful building blocks for design and refinement of neural networks to achieve improved speed and robustness.

Proposed improvements of J using the idea of subspaces [3] are expected to increase the efficiency of this language and of its implementation on machines with MIMD architecture.

Future Research Directions

Future directions planned for this research will address the development of the parallel simulators of neural networks. Equivalent simulations will be performed by J and *APL* implementations. This will provide comparisons and insights into how to benefit maximally from parallelism of the hardware of available platforms. Also, research into the applications of neural networks is to be expanded.

Besides applying neural networks to problems of monitoring and decision making at the nuclear power plants, their pattern recognition and prediction capabilities will be used for addressing economic problems. Both types of control applications will require the design of more robust estimations of the current states and the prediction of succeeding ones.

Acknowledgements

The authors thank Roger Hui and Robert Bernecky for their support and advice that resulted in the successful development of several variants of the neural network model expressed in the J language.

References

[1] Bernecky, R. "Functions Arrays", APL84 Conference Proceedings, APL Quote Quad, Vol. 14, No.4, (1984)

[2] Bernecky, R., Hui, R.K.W. "Gerunds and Representations", APL91 Conference Proceedings, APL Quote Quad, Vol.21, No.4, pp.39-46, 1991

[3] Frey, R.J. "Object Oriented Extensions to APL and J", Vector, Vol.9, No.2 1992

[4] Hui, R.K.W., Iverson, K.E., McDonnell, E.E., and Whitney A.T. "APL\?" APL Quote Quad, Vol. 20, No.4, pp. 192-200, (1990)

[5] Iverson, K.E. "A Personal View of APL", IBM Systems Journal, Vol.30, No.4, pp.582-593 1991

[6] Iverson, K.E. "Programming in J" ISI 1991

[7] McIntyre, D.B. "Language as an Intellectual Tool: From Hieroglyphics to APL", IBM Systems Journal, Vol 30, No. 4, pp.554-581 (1991)

[8] Rumelhart, D.E. and McClelland, J.L. (ed.) "Parallel Distributed Processing. Explorations in the Microstructure of Cognition" Vol.1: Foundations, Cambridge, MA MIT Press 1986

[9] Skurihin, A.N. "Neural Networks in J", APL92 Conference Proceedings, APL Quote Quad, Vol. 23, No.1, pp.216-220 (1992)

[10] Werbos, P.J. "Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences", Ph.D. Thesis, Harvard University, Cambridge, MA. August (1974)

[11] Yao, Y., Freeman, W., Burke, B., Yang, Q. "Pattern Recognition by a Distributed Neural Network: An Industrial Application", Neural Networks, Vol.4, pp.103-121 (1991)