EFFICIENT NEURAL NETWORK MODELING BASED ON DATAFLOW COMPUTING LANGUAGES

D. A. Karras
Hellenic Aerospace Industry and
Hellenic Open University,
Rodu 2, Ano Iliupolis, Athens 16342,
Greece,
tel: +30 6944
226018, fax: +30 210 9945231

ABSTRACT

We present an efficient dataflow modeling of Multilayer Perceptron (MLP) algorithm based on the directed graph concept but with controlled global states. More specifically, we investigate the dataflow implementation of two efficient and widely used MLP training algorithms, namely, on-line Backpropagation and Conjugate Gradient in its major versions. The proposed MLP dataflow modeling approach is applied to a medical diagnosis problem, namely, psychiatric case categorization based on evoked potential data classification. The whole system is implemented in the Labview-G programming environment and it is found that the modeling capabilities along with the results obtained from the proposed MLP dataflow implementation in G demonstrate its efficiency for medical diagnosis applications.

KEY WORDS

Dataflow Modeling, Multilayer Perceptrons, Computer Aided Diagnosis.

1. INTRODUCTION

The use of dataflow programming tools for system prototyping and development predates some of the recent work in compiling and scheduling dataflow graphs [1]. The dataflow paradigm is currently the approach to visual programming used most widely in industry [1-3]. One way in which the dataflow paradigm distinguishes itself from many other paradigms is through its explicitness (through the explicit rendering of the edges in the graph) about the dataflow relationships in the program [3]. In this paper we consider LabVIEW-G capabilities and discuss how we can take advantage of them in order to efficiently map MLP (Multilayer Perceptrons) training algorithms.

LabVIEW (Laboratory Virtual instrument Engineering Workbench) is a graphical application development environment (ADE) developed by National Instruments Corporation for the Data Acquisition (DAQ), Test and Measurement (T&M) and the Industrial Automation (IA) markets. It is composed of several sub-tools targeted at making the development and prototyping of instrumentation applications very simple and efficient. One of its most important components is a compiler for the G programming language. G is a dataflow language that due to its intuitive graphical representation and programmatic syntax has been well accepted in the instrumentation industry, especially by scientists and engineers that are familiar with programming concepts but are not professional software developers but rather domain experts. Though it is easy to use and flexible, it is built on an elegant and practical model of computation. This model could be described as the dataflow graph programming methodology [1-4].

Dataflow graph methodology has been a successful representation for many computational procedures, and most importantly in DSP algorithms, since dataflow semantics is well matched with algorithmic function flow in DSP applications. In a dataflow graph, a *node* represents a function block such as an FIR filter or a Gain, and an *arc* between nodes represents the flow dependency. A function block may contain *states* inside, called local states or parameters. When a node is executed, it consumes a fixed number of data samples from each input arc, and produces a fixed number of data samples to each output arc. These concepts of dataflow graph representation are very well suited to algorithmic procedures based on local computations. Neural Networks, and especially MLPs are ideal candidates for being implemented using the dataflow architecture, since they mainly involve local processing of information. However, they also, involve global information processing features, like learning parameters.

It is well known that Artificial Neural Networks (ANNs) posses very interesting function approximation capabilities making them a very powerful tool in many scientific disciplines. More specifically ANNs have the theoretical ability to approximate arbitrary nonlinear mappings [4]; there is also the possibility that such an ANN approximation is more parsimonious , i.e. it requires less parameters, than other competitive techniques such as orthogonal polynomials, splines, or Fourier series. Moreover since ANNs can have multi-inputs and multi-outputs, they can be naturally used for identification/control of multivariable systems; they can be trained off-line using past data records of the system to be controlled or they can be adapted on-line in order to compensate for changes in the controlled system. Also, since ANNs are parallel distributed processing devices [4] they can be implemented in parallel hardware. While MLPs have been successfully employed in a multitude of function approximation problems in general and, more specifically, in control applications, they suffer from some major drawbacks. First of all the lack of universally efficient training algorithms results in their poor approximation performance. Second, starting from a random initial point in the weight space, the path to the global minimum is often strewn with many local minima, imposing an oscillatory convergence for the weight update algorithm and therefore, resulting in a very slow learning process. These two drawbacks clearly emerge in real world problems [4]. Therefore, investigation for some means of improving MLP function approximation capabilities is demanded. Some ways to achieve this could be the development of better simulation models.

The above mentioned MLP characteristics, local processing, intrinsic parallelism, global computation parameters, like learning parameters, their applicability in a broad set of tasks as well as their learning deficiencies make their efficient simulation an important research issue. Actually many computing environments, like the high level programming languages (C, Fortran etc.), the object oriented framework (e.g C++) and the web programming/object oriented framework (e.g Java) have been involved so far [4]. To the best of our knowledge the visual programming/ dataflow methodology has not been involved up to now in ANN modeling except from the Matlab/Simulink environment. However, this latter paradigm can model ANN following the black box methodology and not the dataflow one. Why to model ANN following the dataflow methodology? The main rationale from the research point of view is that in order to understand e.g. ANN learning deficiencies it is important to monitor the flow of network parameters and their changes in the intuitive manner the dataflow paradigm provides. In addition, the intrinsic parallelism of MLPs can be naturally modeled through it. We provide here the notion that the natural way of modeling ANNs and more specifically MLPs could be achieved by using the dataflow paradigm.

To this end, we employ the known directed graph approach, extended with a mechanism for controlled global states, in MLP modeling as the step towards their dataflow graph architecture design. We suggest that such an MLP dataflow modeling approach, which enables periodic parameter update and dynamic behavior of function blocks, is quite effective.

The proposed MLP dataflow modeling approach is applied to a medical diagnosis problem, namely, psychiatric case categorization based on evoked potential data classification. The whole system is implemented in the Labview-G programming environment and it is found that the modeling capabilities along with the results obtained from the proposed MLP dataflow implementation in G demonstrate its efficiency for medical diagnosis applications.

2. MLP MODELING AND LABVIEW-G

The implementation of MLP data flow modeling has been herein carried out in the G visual programming/ dataflow oriented language [5]. G uses “structured dataflow” semantics to specify high level concepts (e.g. loops). G could be also, examined in the context of other models of computation, such as cyclostatic, dynamic dataflow, and process networks which could find application in MLP/ANN modeling. G has useful subsets that can be statically or quasi-statically scheduled. Parallelism can be further exploited by allowing overlapping execution of loops, and adding array auto-subsetting. Another useful addition would be execution relative to a global clock. Finally, a view manager could present a G based dataflow architecture using a different model of computation. In the present paper we consider how these G concepts might be applied to the MLP training algorithm implementation and evaluation.

To be more specific and understand the basic G characteristics we should first discuss the structure of G. The first thing to note is that the basic difference between G and other programming languages is that the other programming systems use text-based languages to create lines of code, while G uses a graphical programming language to create programs in block diagram form.

G programs are called virtual instruments (VIs) because their appearance and operation can imitate actual instruments. However, VIs are similar to the functions of conventional programming languages. A VI consists of an interactive user interface, a dataflow diagram that serves as the source code, and icon connections that set up the VI so that it can be called from higher level VIs. So, VI’s structure contains:

a. The interactive user interface of a VI is called the front panel, because it simulates the panel of a physical instrument. The front panel of a VI is primarily a combination of controls and indicators. Controls simulate instrument input devices and supply data to the block

diagram of the VI. Indicators simulate instrument output devices that display data acquired or generated by the block diagram of the VI. The front panel can contain knobs, pushbuttons, graphs, and other controls and indicators. You enter data using a mouse and keyboard, and then view the results on the computer screen. You can change the size, shape, and position of a control or indicator. In addition, each control or indicator has a pop-up menu you can use to change various attributes or select different menu items.

b. The VI receives instructions from a block diagram, which you construct in G. The block diagram is a pictorial solution to a programming problem. The block diagram is also the source code for the VI.

c. VIs are hierarchical and modular. You can use them as top-level programs, or as subprograms within other programs or subprograms. A VI, when used within another VI, is called a subVI. The icon and connector of a VI work like a graphical parameter list so that other VIs can pass data to a subVI.

With these features, G makes the best use of the concept of modular programming. You divide an application into a series of tasks, which you can divide again until a complicated application becomes a series of simple subtasks. You build a VI to accomplish each subtask and then combine those VIs on another block diagram to accomplish the larger task. Finally, the top-level VI contains a collection of subVIs that represent application functions. Because each subVI could be executed by itself, separately from the rest of the application, debugging is much easier. Furthermore, many low-level subVIs often perform tasks common to several applications, a specialized set of subVIs well-suited to certain applications like MLP modeling could be developed. Actually, the principal goal of the authors is to develop an efficient library of ANN algorithms in G, like an ANN-G toolbox

In summary, G is a homogeneous, dynamic, multidimensional dataflow language [5,6,7]:

• Homogeneous - G actors produce and consume a single token for each edge in the graph.
• Dynamic - G includes constructs that allow portions of the graph to be conditionally executed based on the input data.
• Multidimensional - G has full support for multidimensional arrays. Loop constructs in G can be used to combine individual tokens into arrays of tokens, or to separate array elements back into individual tokens. This is known as "auto-indexing"
• Turing Complete: It has been demonstrated that if you can implement a Turing machine in a language, that language is Turing complete. A Turing machine has been implemented in G, so G satisfies this condition.
• Bounded communication queues: Although the data structures contained in a token can be arbitrarily large, there can only be one token on any wire at any time
• Structured dataflow: Instead of switch, select, and feedback loops, G has programming structures to control program flow. There is a structured case statement that will select one subgraph to execute based on a single input. There are while and for loops in which the user can specify feedback from one iteration to the next. (No other feedback allowed in G.)
• Composability: Because load balancing is not an issue in scheduling homogenous dataflow, G diagrams can be clustered into sub-diagrams without affecting the correctness of the diagram. The only exception is that since G only allows feedback in a loop structure, the partitioning cannot be allowed to create a feedback loop. Furthermore, a node in G can be a VI written entirely in G. The sub-VI can be a binary compiled from within Lab VIEW, which allows libraries to be distributed without source. G does not need to know the internal implementation of a sub-VI to schedule it.
• Explicit coupling: G supports non-dataflow communications directly in the diagram. Global variables, local variables, and synchronization primitives can be used to explicitly send data or control scheduling in a VI. This reduces the need to have hidden communication between nodes that might affect the scheduling algorithm.

After this discussion about main G features we concentrate on MLP training through the most widely used leaning algorithm for large scale networks and problems, namely, the Conjugate Gradient algorithm [4] and its main characteristics related to the G suggested implementation. Similar characteristics hold for other MLP algorithms like Online Backpropagation etc. The algorithm is as follows [4]

1. We choose an initial weight vector w1.
2. Evaluate the gradient vector g1 and set the initial search direction d1 = -g1, for the weight multidimensional weight vector.
3. At step j, minimize E(wj+Adj) with respect to A to give wj+1 = wj + Amin dj .
4. Test to see if the stopping criterion is satisfied (forward pass).
5. Evaluate the new gradient vector gj+1.
6. Evaluate the new search direction using the form:
   dj+1 = gj+1 + Bjdj
in which Bj is given by the Hestenes Stiefel formula, the Polak – Ribiere formula or the Fletcher – Reeves formula.
7. We set j = j+1 and go to 3.

As it can be seen, the algorithm incorporates not only local computations but, also, global ones. Estimation of the Error function, the search direction and the parameters Bj implies global computation and exchange of information between local processing/ local States Update and global States Update. If we, also, consider parallel implementation of this algorithm with all synchronization aspects we understand that the dataflow paradigm for such modeling purposes should be extended to allow, most importantly, global state manipulation.

Intuitively, it is natural to implement the state update request using shared *global states* for coding parameters since the state update request is shared by several blocks and remains unchanged within the time frame of an epoch for instance. However, the basic dataflow model does not allow global states since they are the sources of side effects. In this paper, we present a modeling solution to fuse global states into dataflow model without side effects.

The key idea of the proposed approach is to make a global table for global states with limited access and to piggyback a pointer to a global table entry on each local processing unit. As a result, a State Update (SU) request is implicitly delivered to the node with a global state. When an SU request is applied to the node with a global state, the node will update its local state with a new value of the global state before processing the data samples

To manage a parameter of a block as a global state, we define a *global state table* (GST). The GST maintains the values of the parameters that may be changed from the outside through the SU request. And, it has limited access from outside; in short, it allows only one writer and many readers. An entry of the GST is a tuple *{state_name, an array of state_values}*. In a tuple, *state_name* is the name of the global state that is referred to by the associated blocks. And *an array of state_values* is a circular buffer to maintain state values. Since dataflow modeling allows concurrent executions of multiple iterations, we may keep all outstanding instances of the associated global states. The size of the array is determined by the scheduling result. Each global state is accessed by its name and an array of state values maintains the updated values of global states.

Based on the above concepts we have achieved efficient dataflow modeling of MLP training / testing algorithms for the basic online/offline BP (Back Propagation) algorithm as well as the Conjugate Gradient algorithm in its different major versions.

In the following figures 1-4 some basic screen-shots of the proposed LabView-G implementation of the MLP training algorithms are illustrated.

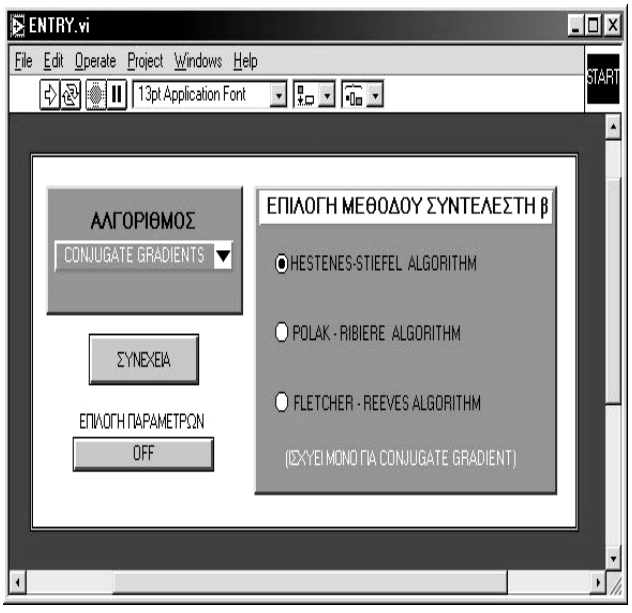

Fig. 1 Basic entry VI

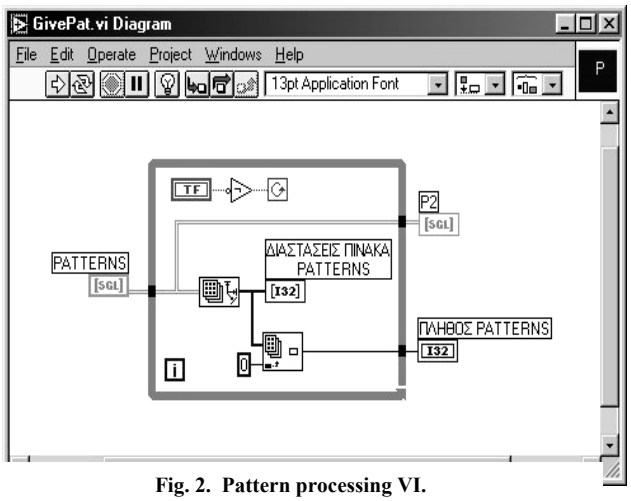

Fig. 2. Pattern processing VI.

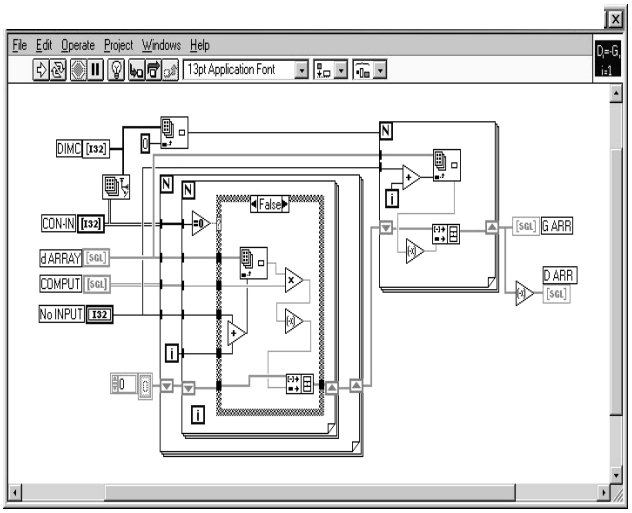

Fig. 3. Search Direction Update in the conjugate algorithm VI.

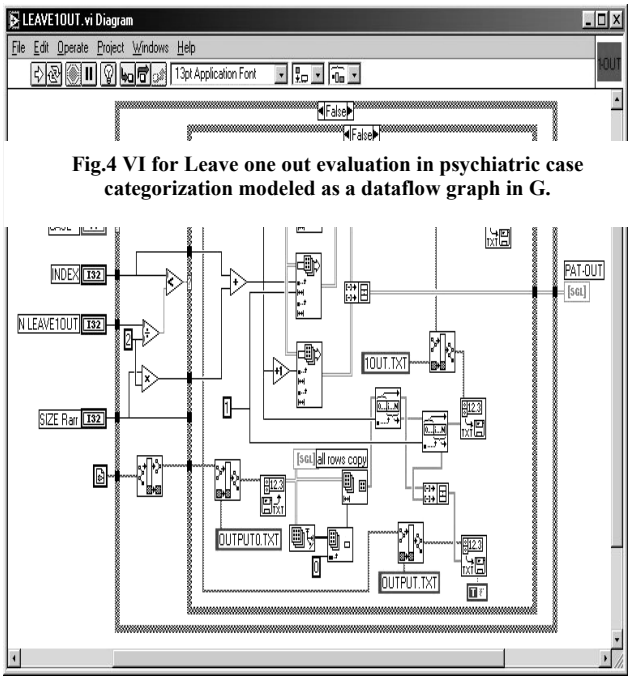

Fig.4 VI for Leave one out evaluation in psychiatric case categorization modeled as a dataflow graph in G.

3. EXPERIMENTAL STUDY

Our first set of tests were made with the examination of simple logical problems, like XOR and AND. We used a small network that had two inputs, two neurons in one middle layer and one output. Every neuron was connected with all the neurons of the next layer.
The training process on the Back Propagation algorithm was almost 100% successful. It needed a very short time to learn and the structure of the network could be said to be ideal.

Likewise, the Conjugate Gradients algorithm also had a very high percent of success, over 80%, on the tests. The outcome could be better with the control of some parameters, but due to the fact that all of the initial parameters were random some tests failed to produce the desired outcome.

The second set of tests was real data taken from medical brain activity measurements. There were six different sets of patterns on healthy and mentally ill people, measured and taken with electrodes and recorded as amplitude and latency points. We used the statistical method of Leave-One-Out to train our networks. That is, one of the patterns of each set was left aside and the network trained and learned the other patterns. At the end, the pattern that was randomly left aside was imported as an input to out trained network to produce an evaluaton outcome. From the comparison of the result and the desired output that was known to us, we could establish the functionality of the test.

From these tests, we managed to remark the following conclusions. If the network was trained for all of the patterns of each set and the final random pattern was one of the training set, then in most cases the network had an almost 100% success. This outcome, of course, was accomplished depending on the right initial parameters and training time given.

For the method of Leave-One-Out, the percent was in range of 53% to 71%, depending on the different set, random pattern, initial parameters and time for the network to be trained. We could say that overall we had an average value of about 62% of success, quite reasonable for the difficult problem examined.

We could mention that in many cases that we didn't have a desired output, the final outcome with regards to the factor E of the Error function of the training algorithm showed us, that the more the time given, the better the result. That is, our network was under-trained. On the other side, in other cases our network was over-trained, that is it could not be in a balance and produced wrong results no matter the time given to be trained. In that situation, our network was unstable and was not capable of computing the real and right parameters and values for the algorithm to work properly.

Thus, the network was depending on structure and its values of initialization. In other words, various coefficients and number of Loops of each algorithm affect the training process. Also the n-Factor (critical value and parameter of the algorithms) affected the speed of the training process. For a small n-Factor (e.g. 0.1 or 0.3) we had a slow change of the E (Error function 9). That is, the network trained in a slower rhythm but it seemed to be more stable. On the other hand, a large n-Factor (e.g. 0.9 or 1.0) we had a faster change of the E. The network learned faster, but there was the danger of it to become unstable. The time given for training, number of training Loops, affected the outcome. Also the structure of the network itself affected the result. We used a lot of different network "trees" in our tests. In most cases we used a net with 100 inputs, seven neurons in one middle layer and one output, with the neurons of its layer connected with all the neurons of the next layer. This structure has given us a lot of satisfying results.

At the end, we must remark the fact that the best structure for the biomedical problem at hand of a neural network depends on the problem and method we follow to solve it. Finally, experimentation with MLPs modeled with the proposed LabView-G approach has revealed the easiness to debug and, moreover, to understand the MLP processing in a completely hierarchical/ visual way as well as to understand when/why such a process fails or succeeds.

4. CONCLUSIONS

An efficient implementation of MLP training/evaluation has been proposed in the LabView-G dataflow programming environment and some experimentation has been carried out with medical data. The future prospects of this work are to develop a complete library for ANN dataflow modeling.

REFERENCES

[1] M. Burnett and D. McIntyre, “Visual Programming.”, Computer 28(3):14-16, 1995
[2] R. Pandey and M. Burnett, “Is it easier to write matrix manipulation programs visually or textually ? An empirical study.”, IEEE Symposium on Visual Languages, Bergen, Norway, 344-351, 1993
[3] K. Whitley, “Visual Programming Languages and the empirical evidence for and against”, J. Visual Language and computing 8 (1), 1997, pp. 109-142
[4] S. Haykin, “Neural Networks: A comprehensive foundation”, 1999, 2nd edition
[5] National Instruments, LabVIEW Software Reference and User Manual, National Instruments, Feb. 1998.
[6] B. Lee, and A. R. Hurson, "A Hybrid Scheme for Processing Data Structures in a Dataflow Environment," IEEE Transactions on Parallel and Distributed Systems, vol. 3, no. 1, pp. 83-96, Jan. 1992.
[7] B. Lee, and A. R. Hurson, "Dataflow Architectures and Multithreading," Computer, Aug. 1994, pp. 27-39.