Forward-Backward Building Blocks for Evolving Neural Networks with Intrinsic Learning Behaviours

S.M. Lucas

Department of Electronic Systems Engineering,
University of Essex,
Colchester CO4 3SQ, UK
email sml@essex.ac.uk

Abstract. This paper describes the forward-backward module: a simple building block that allows the evolution of neural networks with intrinsic supervised learning ability. This expands the range of networks that can be efficiently evolved compared to previous approaches, and also enables the networks to be invertible i.e. once a network has been evolved for a given problem domain, and trained on a particular dataset, the network can then be run backwards to observe what kind of mapping has been learned, or for use in control problems. A demonstration is given of the kind of self-training networks that could be evolved.

1 Introduction

Despite much research into evolving neural networks, there is relatively little work published on evolving learning behaviours for neural networks. Within this area we can identify four kinds of approach:

- Write down a set of learning rules with variable parameters for a given network architecture, then evolve the choice of rules and their parameters.
- Evolve some arbitrary kind of network, and hope that it learns.
- Specify the feedforward part of an architecture; allow the feedback part (responsible for learning) to evolve in some arbitrary way.
- Evolve some arbitrary kind of network, but with some special restrictions that force a type of learning behaviour to be built into it.

Examples of the first type are the work of Chalmers [1] and also Bengio [2], while an example of the second type can be found in [3]. The Author has worked on the third type [4], while this paper introduces the concept of a set of modules to facilitate the fourth type.

2 Background

There are two distinct methods of encoding neural networks in chromosomes: direct and indirect. In the direct case, we can identify a direct correspondence between each part of the chromosome and each part of the network. The indirect encoding method is where no such correspondence exists. Most indirect encodings that have been presented in the literature have the property that small chromosomes can generate large networks.

Evolutionary methods have been used most extensively in conjunction with a direct encoding, and there are hundreds of papers in the literature that have adopted this approach. Some have used a fixed architecture, and evolved only the weights of that architecture [3, 5], others have evolved both the structure and the weights [6, 7]. Interesting neural solutions have been evolved to problems in time series prediction [8, 9], playing tic-tac-toe [5] and in evolving dynamical networks that *learn* to output simple sequences [3]. This latter approach is of particular relevance to the work reported here, since a learning behaviour was evolved. However, due to the use of a direct encoding, the learning behaviour is not transferable to other problems of a different size. Furthermore, while the direct encoding method has the virtue of simplicity it scales poorly, and hence is only suited to evolving relatively small networks.

For this reason, there has been some interesting work done on indirect encoding. Some of the earliest work in this area was reported by Kitano in 1990 [10], and has since been followed up with superior network generation languages and grammars designed by Gruau [11, 12], Boers and Kuiper [13], Muhlenbein [14, 15] and Sharman *et al* [16]. All of these however, either use the GA (operating on strings or graphs in the neural description language or chromosome) to evolve a hard-wired neural network, or use the GA to evolve a good topology (or evolve a good topology and weight set) which is then trained by error backpropagation or simulated annealing.

Kitano [17] has also developed a unified framework in which the network structure and weights are allowed to evolve, and followed this up with a more biologically detailed simulation [18]. Even in [17], however, despite claims that all details of the network are allowed to evolve, this is not quite true, since the learning algorithm is fixed in advance to be error back propagation – although the network topology and initial weights are evolved.

In contrast to this, the Author [19, 20, 4] has shown how it might be possible to evolve the learning algorithm within the same unified framework in which the other network details are evolved. Also, related work of interest is the Strongly Typed Genetic Programming of Montana [21], which he uses to evolve programs that explicitly operate on vector, matrix and list data structures, as well as simpler data types. Within this framework he evolves (among other things) the algorithm for updating the input tracking estimate in a Kalman Filter.

Finally, on the subject of encoding, we present some ideals which artificial chromosomes used for neural architecture evolution should aspire to:

- The chromosome should be modular. This will allow evolution to proceed by putting together existing building blocks in new ways as well as developing new building blocks.
- The chromosome should be human-readable — after spending a good deal of time evolving a novel solution to an interesting problem, it would be a shame we could only appreciate it at the connection-matrix level.

- The chromosome should be parameterised. Hence, having evolved a new type of architecture for a class of problem, we should be able to parametrically alter it to tackle related problems of a different size.

Modularity is of paramount importance in good engineering design. Systems that exhibit complex and useful behaviour usually achieve such behaviour through the interaction of simpler modules. Many of the previous indirect neural coding schemes have claimed modularity, but have only achieved it in a very limited sense. Consider for example the encoding scheme of Kitano [10]. This is based on the context-free rewriting of matrices. While it possesses a good deal of modularity in the genotype, most of this does not carry over to the phenotype, which is the critical place where modularity is needed. In designing chromosomes for neural network evolution the best place to seek inspiration is in hardware or software engineering, where the efficient use of modularity is of paramount importance. It is not enough to allow useful substructures to be repeated – they must also be properly interconnected.

3 The Forward-Backward Modules

The proposed building block is the forward-backward module. These are composed of two specific behaviours: the forward part and the backward part. The details of each forward-backward module could be evolved, or they could be pre-specified. Here, we show how to specify them manually.

A forward-backward module implements two behaviours, or functions – a forward behaviour and a backward behaviour. Each behaviour is achieved by simply updating a set of cells in a particular order. Unmarked cells compute the identity function, and act as local storage. In Figures 1–5 the cells are illustrated by white circles (forward) and grey circles (backward).

A complete forward-backward network of modules is then simulated in the forward mode, by executing the forward action of each module, in order from bottom-to-top as sketched in Figure 7. Similarly, the backward mode is simulated by executing the backward action of each module in top-to-bottom order.

Forward backward modules can be designed for any differentiable functions; here we just give a small sample, and an example application.

3.1 A general forward-backward node

The general form of the forward-backward module is shown in Figure 1. It computes function f for its forward behaviour. For its backward behaviour it computes the derivate f' of the function with respect to the input (or that the partial derivative w.r.t. that particular input), multiplies it by the accumulated backward error (in the $\sum$ cell) and passes it back to its backward output.

Note that all connections to or from the forward-backward module are grouped into sockets, where each socket contains two connection cells; one for input and

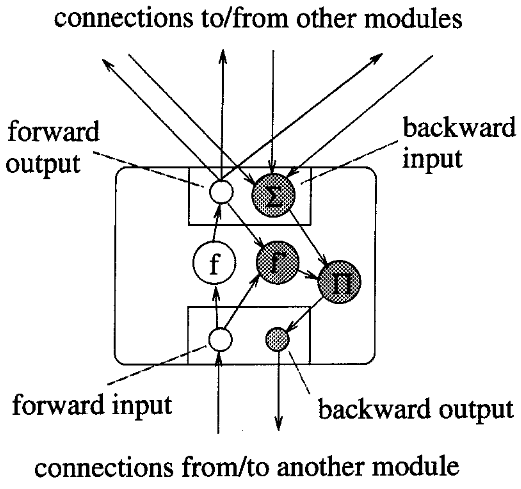
connections to/from other modules
forward output
backward input
Σ
f
Γ
Π
forward input
backward output
connections from/to another module

Fig. 1. A general forward-backward node. The white circles are the nodes evaluated during the forward mode, in bottom-to-top order. The shaded nodes are evaluated in the backward pass in top to bottom order.

one for output. The upper socket contains the forward output and the backward input, while the lower socket(s) contain the forward input(s) and the backward output(s).

3.2 Sine and cosine modules

Sine and cosine function nodes are not frequently used in artificial neural networks, though for many applications they can play a useful role. Figure 2 shows the forward-backward module for the sine function. The forward-backward module for cosine can be obtained by swapping the sine and cosine nodes, and negating the backward output.

Figure 2 On the forward pass the node computes the *sin* of its single input, and passes the output on to (possibly) many nodes. On the backward pass, it accumulates an error signal from all the nodes to which its output is connected, multiplies this by the derivative of the output with respect to the forward input, and passes this back to the node that it took input from on the forward pass.

3.3 Product and summation modules

The product node is an example of a node that can take multiple forward inputs. Since the partial derivative of the network error with respect to each input is different in the case of this type of node, then so must the corresponding backward output be different.

The summation node (Figure 4) also takes multiple inputs, but in this case the backward input is fed directly to all the backward outputs.

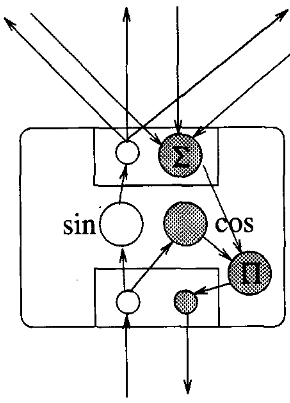

Fig. 2. A *sin* forward-backward node.

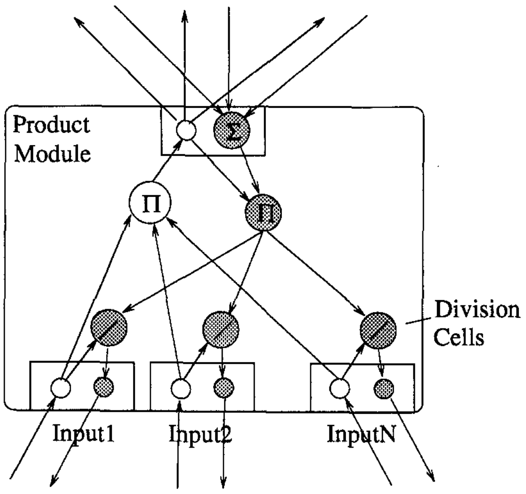

Fig. 3. A more complex example: a product module with multiple inputs. In this case, it must pass back a different partial derivative to each input.

3.4 Variable modules

The variable module provides a way of modelling the important parameters of a particular neural architecture within the same framework of forward-backward modules. The variable module does nothing for its forward behaviour. Its backward behaviour is to update itself in the opposite direction of the gradient of the accumulated error at its backward input. Variable modules do not have a forward input or a backward output. What they do have is a *rate* parameter which is used as a learning rate, and serves to attenuate or amplify the level of adjustment to the value stored in the node. This is illustrated in Figure 5.

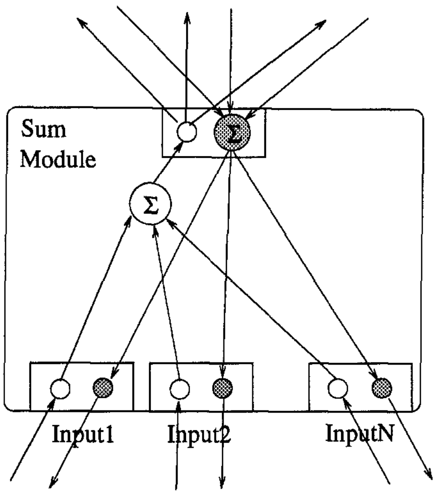

Fig. 4. The summation module. In this case, the same derivative is fed back to each backward output.

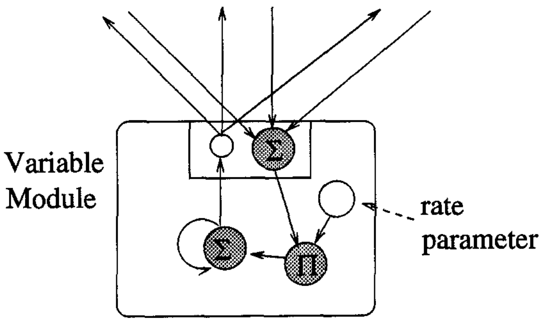

Fig. 5. The variable module. The rate parameter is some negative constant, to be determined by trial and error or evolution.

4 Example: modelling a simple robot arm

This example demonstrates the potential benefits of the approach. Figure 6 illustrates a simple robot arm with two movable sections. The tip of the arm is moved by adjusting the angles α and β. Suppose that the lengths l and m of each section are fixed in reality, but unknown to the network (Figure 7). Then, the system can be trained as follows to estimate the lengths. For a set of training samples, an angles α and β are set up, and the position of the tip in x and y coordinates are paired with these to form target outputs for these inputs. For each training sample, the error at the output given the current estimates of l and m is propagated back through the network, by the process of each module performing its backward action. This training process proceeds until the values of l and m converge to the actual values of the robot arm sections.

The mapping from (α, β) coordinates to (x, y) coordinates is a functional one (many-to-one) that can be represented by the forward action of the network. Suppose now, as would be more usefully the case, that we wish to map from (x, y) coordinates to find the α and β angles that can position the tip at a particular point. Since this is a one-to-many mapping, we cannot represent this by the normal forward action of the net. Instead, we use the net in backward mode. For a desired target point (t_x, t_y) the error between the target and the current output point (o_x, o_y) is fed into the backward inputs of nodes o_x and o_y respectively. Then, the network is run backwards to adjust the angles α and β, then forward to compute the current error, then backwards again, and this process is repeated until the error is sufficiently small. Figure 8 illustrates the behaviour of the system. Each sudden increase in error corresponds to a movement of the target. Note that after each movement, the system responds to move the tip towards the target quite rapidly.

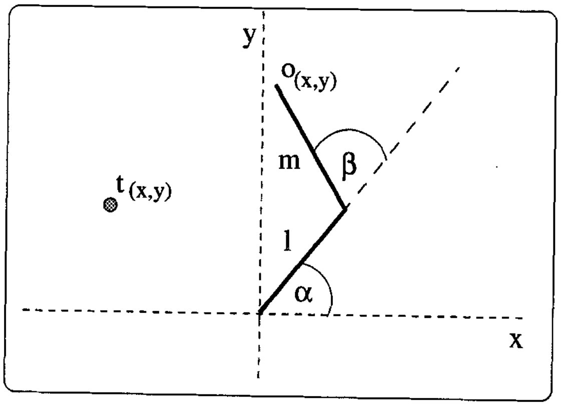

Fig. 6. A simple 2-section robot arm.

Figure 7 illustrates the forward computations necessary to calculate the position of a two arm planar robot with arm lengths l and m and arm angles α and β. β is taken relative to α.

5 Discussion and Conclusions

This paper has described a new and simple method for specifying invertible feedforward networks with intrinsic learning abilities. Each node in the network is now a module that has a forward and backward part. The forward part computes some differentiable function of its input while the backward part accumulates the error from the nodes successors and multiplies it by the derivative of the

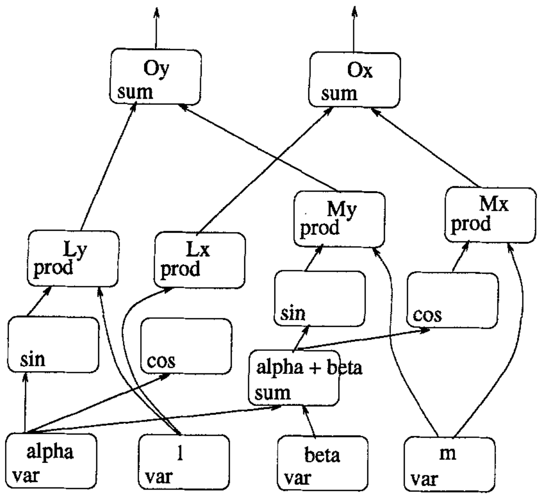

Fig. 7. A network of forward-backward modules for specifying the robot arm of Figure 6. Note that for each forward connection on the diagram, there is also a backward connection that is not shown to keep the diagram simpler.

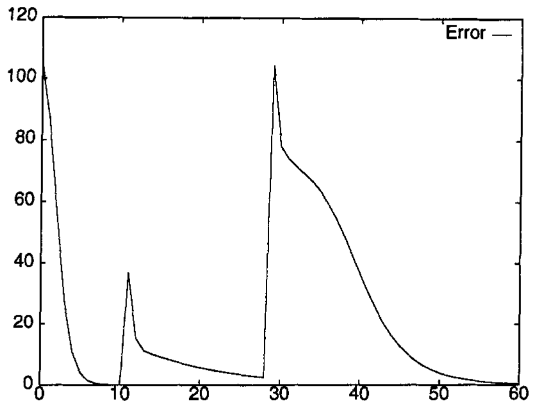

Fig. 8. Plot of error (Euclidean distance between target and tip of robot arm) with respect to time. Each sudden increase in the error corresponds to the target being moved.

function with respect to the nodes forward input. On the one hand, this can be seen as simply a re-statement of the generalised delta rule [22]. On the other hand, this simple modular way of viewing it would appear to be a significant step forward for researchers interested in evolving neural networks, since previous work has either restricted the type of network evolved to more or less a standard multi-layer perceptron (typically with a non-standard connection pattern), or other types of network (such as higher-order nets [14]) but with the restriction that the free parameters of the network (e.g. the weights) be found through a process of random hill-climbing [14] or simulated annealing [16] Generally speaking, gradient methods converge much faster than perturbation methods, hence the perspective offered by this paper has important implications.

There are two ways that this framework can be applied to evolving neural architectures. First, any direct or indirect encoding that would normally specify a network of simple cells could equally well be used to specify a network of forward-backward modules – with the immediate benefit that the network would have its own in-built learning behaviour and be potentially invertible. The second possibility is to evolve details of each module, rather than pre-specify them. Each forward-backward module presented here was specified in terms of simple cells, but clearly, an alternative would be to specify it in terms of complex modules, where each of these could also be specified either in terms of simple cells or its own complex modules. This sort of multi-level hierarchical abstraction is standard practice in hardware and software design. It has the potential to have a significant impact on the artificial evolution of neural networks. The forward-backward modules presented here are a useful step in that direction.

References

1. D. Chalmers, "The evolutions of learning: an experiment in genetic connectionism," in *Proceedings of the 1990 Connectionist Models Summer School* (D. Touretzky, J. Elman, T. Sejnowski, and G. Hinton, eds.), San Francisco: Morgan Kaufman, (1990).
2. S. Bengio, Y. Bengio, and J. Cloutier, "Use of genetic programming for the search of a new learning rule for neural networks," in *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 324 – 327, Orlando: IEEE, (1994).
3. B. Yamauchi and R. Beer, "Sequential behaviour and learning in evolved dynamical neural networks," *Adaptive Behaviour*, vol. 2, pp. 219 – 246, (1994).
4. S. Lucas, "Evolving neural network learning behaviours with set-based chromosomes," in *Proceedings of European Symposium on Artificial Neural Networks (ESANN '96)*, pp. 291 – 296, Brussels: D facto, (1996).
5. D. Fogel, "Using evolutionary programming to create networks that are capable of playing tic-tac-toe," in *Proceedings of IEEE International Conference on Neural Networks*, pp. 875 – 880, San Francisco: IEEE, (1993).
6. D. Dasgupta and D. McGregor, "Designing application specific neural networks using the structured genetic algorithm," in *Proceedings of COGANN-92 - IEEE International Workshop on Combinations of Genetic Algorithms and Neural Networks*, pp. 87 – 96, Baltimore: IEEE, (1992).
7. L. Marti, "Genetically generated neural networks ii: searching for an optimal representation," in *Proceedings of the International Joint Conference on Neural Networks (Baltimore '92)*, pp. I, 221 – 226, San Diego, CA: IEEE, (1992).
8. J. McDonell and D. Waagen, "Neural network structure design by evolutionary programming," in *Proceedings of the Third Annual Conference on Evolutionary Programming* (D. Fogel and W. Atmar, eds.), pp. 79 – 89, Evolutionary Programming Society, (1993).

9. J. McDonell, W. Page, and D. Waagen, "Neural network construction using evolutionary search," in *Proceedings of the Third Annual Conference on Evolutionary Programming* (A. Sebald and L. Fogel, eds.), pp. 9 – 16, World Scientific, (1994).
10. H. Kitano, "Designing neural networks using genetic algorithm with graph generation system," *Complex Systems*, vol. 4, pp. 461 – 476, (1990).
11. F. Gruau, "Cellular encoding of genetic neural networks," *Laboratoire de l'Informatique du Parallelisme Technical Report 92-21, Ecole Normale Superieure de Lyon*, (1992).
12. F. Gruau, "Automatic definition of modular neural networks," *Adaptive Behaviour*, vol. 3, pp. 151 – 183, (1994).
13. E. Boers and H. Kuiper, "Biological metaphors and the design of modular artificial neural networks," *Masters thesis, Department of Computer Science and Experimental and Theoretical Psychology, Leiden University, the Netherlands*, (1993).
14. H. Muhlenbein and B. Zhang, "Synthesis of sigma-pi neural networks by the breeder genetic programming," in *Proceedings of IEEE International Conference on Evolutionary Computation*, pp. 318 – 323, Orlando: IEEE, (1994).
15. B. Zhang and H. Muehlenbein, "Balancing accuracy and parsimony in genetic programming," *Evolutionary Computation*, vol. 3, pp. 17 – 38, (1995).
16. K. Sharman, A. Esparcia-Alcazar, and Y. Li, "Evolving signal processing algorithms by genetic programming," in *Proceedings of IEE 1st International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pp. 473 – 480, London: IEE, (1995).
17. H. Kitano, "Neurogenetic learning: An integrated model of designing and training neural networks using genetic algorithms," *Physica D*, vol. 75, pp. 225 – 238, (1994).
18. H. Kitano, "A simple model of neurogenesis and cell differentiation," *Artificial Life*, vol. 2, pp. 79 – 99, (1995).
19. S. Lucas, "Growing adaptive neural networks with graph grammars," in *Proceedings of European Symposium on Artificial Neural Networks (ESANN '95)*, pp. 235 – 240, Brussels: D facto, (1995).
20. S. Lucas, "Towards the open-ended evolution of neural networks," in *Proceedings of IEE 1st International Conference on Genetic Algorithms in Engineering Systems: Innovations and Applications*, pp. 388 – 393, London: IEE, (1995).
21. D. Montana, "Strongly typed genetic programming," *Evolutionary Computation*, vol. 3, pp. 199 – 230, (1995).
22. D. Rumelhart, G. Hinton, and R. Williams, "Chapter 8: Learning internal representations," in *Parallel Distributed Processing: Explorations in the Microstructure of Cognition, Volume 1: Foundations* (D. Rumelhart and J. McClelland, eds.), pp. 319 – 362, London: The MIT Press, (1986).