LEARNING OF CONTROL KNOWLEDGE FOR SYMBOLIC PROOFS WITH BACKPROPAGATION NETWORKS

A.Ultsch, R.Hannuschka, U.Hartmann, V.Weber

Institute of Informatics
University of Dortmund
D-4600 Dortmund, Germany

1. INTRODUCTION

This paper presents the application of a connectionist network to optimize symbolic proofs. By symbolic proofs we understand proofs in first order logic. Prolog interpreters are an implementation of theorem provers for special first order formulas, called horn clauses, based on the resolution principle [Robinson 65]. The usage of Prolog interpreters for symbolic proofs implies a certain proof strategy: first, selection of the leftmost partial goal in the resolvent as goal to be solved and second, selection of the clauses in written order for the resolution of the selected partial goal. In case of failure of a partial goal, the interpreter backtracks systematically to the last choice made without analyzing the cause of failure. Even for simple programs, this implicit control strategy is not sufficient to obtain efficient computations [Naish 84]. Explicit control by using control constructs like cut is, however, contradictory to Prolog as a declarative programming language. The need for the distinction of logic and control has therefore long been recognized [Kowalski 75].

We describe an approach to learn and store control knowledge in a connectionist network. The system utilizes a three-layer backpropagation network. A meta-interpreter generates training patterns encoding successful Prolog proofs. Trained with these examples of proofs the network generalizes a control strategy to select clauses. Another meta-interpreter uses the network to compute optimized proofs.

2. TRAINING THE SYSTEM

Fig.1a shows the generation of the training data. A generating meta-interpreter (GMI) containing the Prolog program is asked to prove a goal.

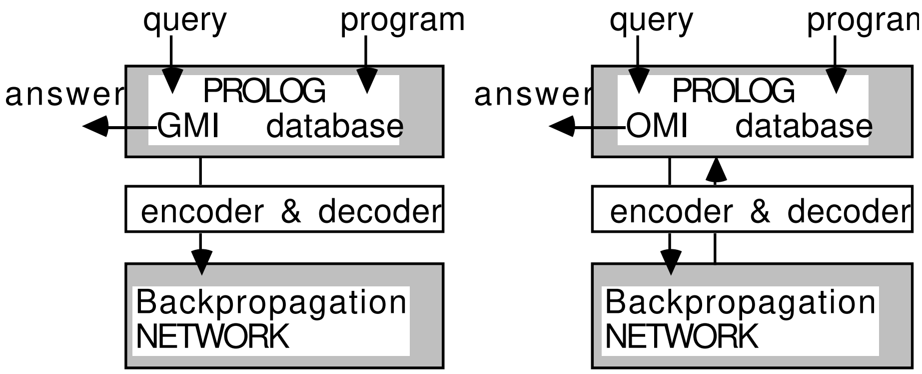

Fig. 1a: generation of training data b: using the network in proofs

The GMI constructs the optimal proof for the given goal, i.e. the proof with the minimal number of resolutions. The optimal proof is found by generating all possible proofs and comparing them with reference to the number of resolutions. For an optimal proof each clause-selection-situation is recorded. A clause-selection-situation is described by the features of the partial goal to be proved and the clause which is selected to solve that particular goal. The clause is described by a unique identification and two different sorts of information concerning the structure of arguments are used: the types of arguments and their possible identity.

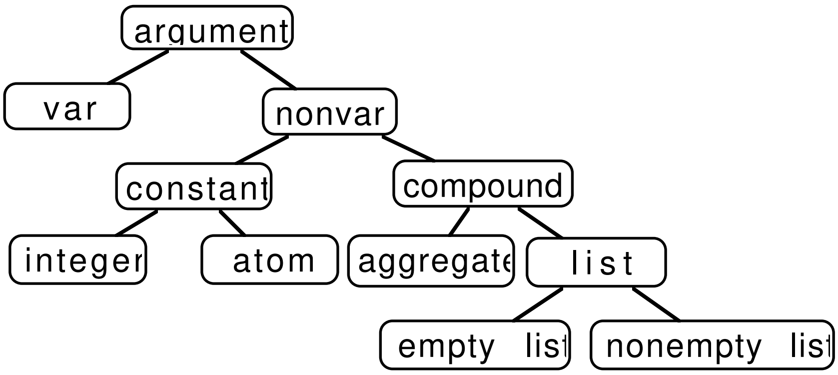

Fig. 2: type tree of Prolog types

For the types of arguments a hierarchical ordering of the possible argument types, as shown in Fig.2, is used. In the tree a parent-node represents a generalization of its son nodes. For example an argument may be a 'var' (variable) or a 'nonvar' (instantiated argument). The encoder (see Fig. 1a) takes the clause-selection-situation and produces a binary training pattern for the network. The encoding preserves similarities among the types, e.g. the code for 'integer' is more similar to the code for 'atom' than to the code for 'list'. The used network is a backpropagation network with an input layer, one hidden layer and an output layer. The activation function is the standard sigmoid function [Rumelhart/McClelland 86]. The number of hidden units is equal to the number of input units. The network is trained with the encoded training patterns until it is able to reproduce the choice of a clause for a partial goal. The usage of the trained net in a proof is shown in Fig. 1b. A query is passed to the optimizing meta-interpreter (OMI). For each partial goal the OMI presents the description of the partial goal as input to the net and obtains a candidate-clause for resolution. With this candidate the resolution is attempted. If resolution fails, the OMI uses Prolog search strategy as default.

3. FIRST RESULTS

To test our approach we use a Prolog program consisting of a predicate with 64 different clauses with 4 arguments. To produce training data three different classes of queries are used: class A, requiring two identical arguments, class B, requiring three identical arguments, and class C, requiring all arguments to be identical. The training set consists only of queries having atoms as argument type (see Fig.2) in order to test for generalization (see below).

The left part of Fig. 3a shows the mean number of unifications for Prolog to prove the goals. The network immediately finds the right clause for examples having been trained before, as illustrated in the right part of Fig. 3a. In a second experiment, goals are used with different types of arguments. This experiment tests the ability to generalize to proofs of similar structure. The results are given in Fig. 3b. While Prolog needs an average number of unifications between 33 and 64, our system demands a range of 15 to 38 unifications. Column D in Fig. 3b gives the results of a test involving only queries with type integer as arguments. This shows in particular that the number of unifications is drastically reduced, if the type in the tested query is very similar to the type the network is trained with.

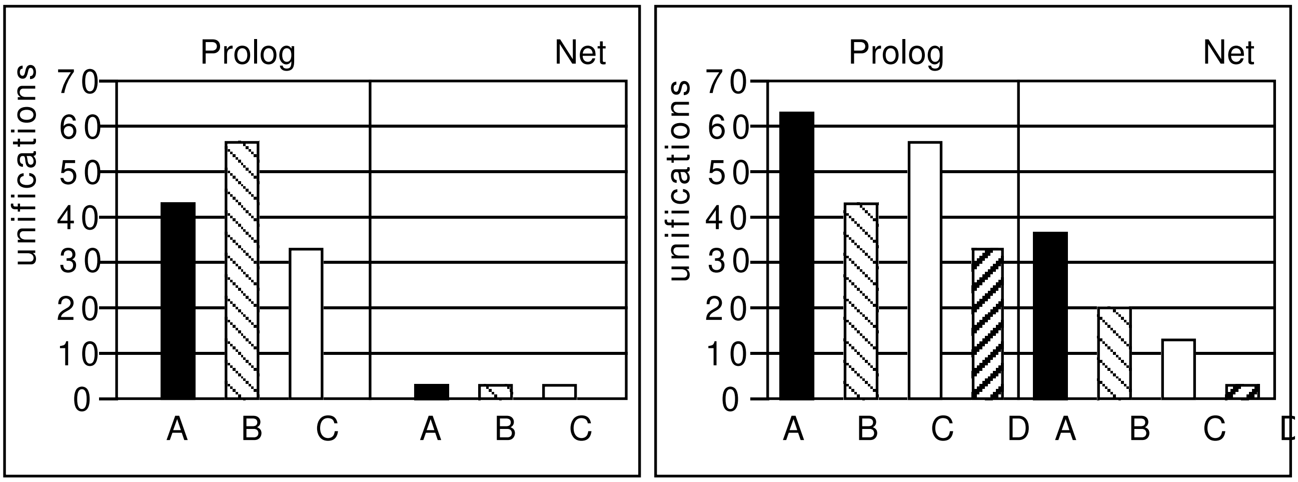

Fig. 3a: clause selection for trained queries b: clause selection for new queries

It has to be noted that all experiments are made with Prolog strategy as default if no clause is found by the network. This means in particular that there may be an overhead of one additional resolution compared with Prolog for some queries. The results in Fig. 3a and 3b, however, show that averaged over a number of queries the network heuristic for clause selection is superior to Prolog strategy.

4. DISCUSSION

Up to now our system improves simple Prolog programs. Transferring the results to arbitrary Prolog programs should be done with great care. Nevertheless a significant speedup for proofs can be observed. It is clear that in a symbolic system storing all proofs of the past would result in the same speedup. The latter method, however, would exhaust every storage media sooner or later. In contrast to such a method a network of limited size is used to store the proof information. We argue, however, that the ability of the network to extract to structure of the proof captures the essential features without having to remember all the details. In the network a distributed representation of the learned concepts is used. The interference of different concepts makes networks capable to learn new concepts without the need for new storage. The interference is also the reason of the ability to generalize. As the first results exemplify, the learned strategy is also applicable to other proofs of the same structure. Since we are learning proof heuristics, it is clear that there will be always examples where a learned strategy is not applicable at all.

5. OTHER APPROACHES

The representation of control knowledge for Prolog programs has been proposed before. For example Kasif suggests to represent control knowledge as expression in a procedural language [Kasif et al. 83]. Others propose a declarative representation of control knowledge [Gallaire/Lasserre 82, Genesereth/Ginsberg 85, Ultsch 87]. This means conditions are formulated on which certain clauses may be selectable to prove a partial goal. All these approaches require a human expert to formulate the control knowledge. Unlike the declarative approaches mentioned before, we don't limit the description of the arguments of partial goals to the instantiation status. Our approach uses a much more detailed description of instantiated arguments. To improve efficiency the selection of clauses often depends on instantiations of the arguments to certain structures.
Suttner introduces control to the selection of axiom in theorem provers [Suttner 89]. A proof is observed and certain features of the proof are counted, e.g. the number of variables in the theorem. Disadvantageous to this approach is the loss of information about the positions of the variables. Our approach considers the positions of arguments in encoding the features of a goal.

6. CONCLUSION

In this paper a new approach to learn and memorize control knowledge for Prolog programs is presented. Our system utilizes a three layer backpropagation network. Trained with examples of successful proofs, the network learns a control strategy. Proof strategies are declaratively described by certain features like the types and the possible identity of the arguments of a clause. Similarities of these features are used to train the network in order to generalize to similar clause-selection-situations. First experimental results show the ability of the network to learn the presented description of proofs. Our system is able to reproduce information without using too much memory, because the knowledge is stored in a connectionist network of limited size. The system is able to generalize a strategy for a clause selection to new, but similar proofs. An important advantage of the system is its ability to learn proof heuristics from examples. No programmer is needed to define explicit control knowledge for the program.

Because of the encouraging results described in this paper our further research will deal with the representation of other kinds of control knowledge for Prolog programs. We will extend the system to control the selection of literals and backtrack points. An important extension of our model should be a system which observes a Prolog program and recognizes situations in which it is likely that the proof will not terminate.

ACKNOWLEDGEMENTS

We would like to thank all members of the student research group PANDA (Prolog And Neural Distributed Architectures, Institute of Informatics, University of Dortmund) for their encouragement. Special thanks to M. Mandischer for his support concerning the usage of programming tools.

REFERENCES

[Gallaire/Lasserre 82] H. Gallaire, C. Lasserre: *Metalevel Control for Logic Programs,* in: Clark/Tärlund (Eds.): Logic Programs, Academic Press, London, 1982, pp.173-188

[Genesereth/Ginsberg 85] M.R. Genesereth, M.L. Ginsberg: *Logic Programming* in: Communications of the ACM, Vol. 28 Nº 9, 1985, pp.933-941

[Kasif et al. 83] S. Kasif, M. Kohli, J. Minker: *Prism: A Parallel Inference System for Problem Solving,* in: Proceedings of the 8th IJAI, Karlsruhe 1983, pp.544-546

[Kowalski 75] R. Kowalski: *Algorithm = Logic + Control,* in: Communications of the ACM, Vol. 33 Nº 7, July 1979, pp.424-436

[Naish 84] L. Naish: *Prolog Control Rules,* Technical Report, No.13, University of Melbourne, Australia, 1984

[Robinson 65] J.A. Robinson: *A machine-oriented Logic based on the Resolution Principle,* in: Journal of the ACM, Vol. 12 Nº 1, Jan. 1965, pp.23-44

[Rumelhart/McClelland 86] D.E. Rumelhart, J.L. McClelland: *Parallel Distributed Processing : Explorations in the Microstructure of Cognition,* MIT Press, 1986

[Suttner 89] C.B. Suttner: *A feature-based heuristic module for the SETTHEO theorem prover,* Forschungsgruppe Künstliche Intelligenz, TU München, 1989

[Ultsch 87] A. Ultsch: *Control for Knowledge-based Information Retrieval,* Dissertation, Swiss Federal Institute of Technology, Zurich, 1987