Pp. 231–240 in: *Proceedings of the Finnish Artificial Intelligence Conference* (Otaniemi, Finland, June 1992), edited by E.Hyvönen, J.Seppänen and M.Syrjänen. Finnish Artificial Intelligence Society, Helsinki 1992.

Integrating Symbolic Reasoning with Neurally Represented Background Knowledge

Petri Myllymäki Pekka Orponen Tomi Silander
Department of Computer Science, University of Helsinki
Teollisuuskatu 23, SF–00510 Helsinki, Finland*

Abstract

In an experimental implementation of a hybrid neural-symbolic programming environment, we have interfaced a standard Prolog system with a neurally implemented facility for representing probabilistic background knowledge. In inferential processing, the Prolog engine first searches for explicitly given facts in its symbolic knowledge base, and then queries the neural component for the most likely values of propositions not resolved by the explicitly given information. The neural component performs approximate Bayesian reasoning to answer these queries on the basis of a given probabilistic taxonomy of concepts and their attributes.

1 Introduction

With the current advance of research on neural computation models, there has been increasing interest in the infusion of neural network techniques into more conventional symbolic information processing. The potential advantages of such *hybrid neural-symbolic* systems would be great. Combining the robust, adaptive neural knowledge representations with high-level programming techniques should help in developing information processing systems that are less brittle and more adaptive to changing environments than what can be achieved by present means.

Some recent approaches to developing such hybrid systems have been reported in the collections [2, 7]. A crucial issue differentiating between the various approaches is how the symbolic level of such a system should relate to the neural level. While some researchers (e.g. [13, 17]) work on the foundational problems of performing symbolic operations

*E-mail: Firstname.Lastname@Helsinki.FI

on distributed representations, others (e.g. [1]) take a more implementational approach, using localized representations and designing neural techniques for performing symbolic reasoning upon them. Yet others (e.g. [8]) forgo the need for neural knowledge representations altogether, and use neural networks to simply implement existing symbolic algoritms.

None of the proposed approaches is quite mature for application yet: a problem with the foundational approach is that building a full-strength inference system thoroughly based on distributed representations is very difficult; on the other hand the more localized approaches stand in danger of losing the robustness that comes from distributed representations. (For instance, while Shastri in his early work [14, 15] presented a system for neural knowledge representation capable of evidential reasoning on incomplete data, his newer system [1], which performs more complicated logical reasoning, only deals with crisp data.)

All their differences notwithstanding, a striking similarity in most of the current hybrid system approaches is their concentration on developing neural realizations of high-level symbolic processing functions. Contrary to this trend, we propose that in the present situation, until the techniques for manipulating distributed representations mature, the most promising approach to take is to build systems that consist of two different, but complementary components interfaced together: a purely neural component that provides a basic robustness to the knowledge representation, and a purely symbolic component that handles the complicated high-level inferential tasks. To rephrase this idea in hybrid systems terminology, we suggest that the notorious "variable binding problem" [1, 3, 17] be provisionally solved by using purely symbolic variable bindings in high-level reasoning, with the option of querying a neural representation for background knowledge about the variable referents[1]. Of the hybrid system schemes reported in [2, 7], the one closest in spirit to this view seems to be that of Hendler's [5, 6], but the marker-passing/back-propagation components in his scheme are unnecessarily restricted.

We are currently experimenting with a functioning bi-partite hybrid system built out of two implementations of the subcomponents that were readily available to us: the SICStus Prolog interpreter developed at the

[1] We have been somewhat motivated in this division of work by the analogy of the neural and linguistic domains of human information processing, but we wish to make no claims about the cognitive aspects of our model.

Swedish Institute of Computer Science [4], and a neural knowledge representation scheme NEULA developed by us [9, 10] on the basis of Smolensky's [16] Harmony Theory model.

In the following sections we present first a brief outline of the NEULA system and the interface between SICStus Prolog and NEULA, then a small example of using the NEULA representation to provide background knowledge to a Prolog program, and finally some concluding comments.

2 The NEULA System

The knowledge representation system NEULA consists of a compiler and a simulator. The compiler translates a description of a probabilistic taxonomy of concepts and their associated attributes, presented in a generic high-level knowledge representation language, into a representation as a harmony network [16], whose operation may then be simulated with the simulator. As an example, consider the taxonomy given in Figure 1 of the inhabitants of a zoo.

In the NEULA description language used in the figure, a description of a concept consists of a reference to its immediate ancestor (if any), together with a list of attributes and their value distribution for objects belonging to this conceptual class. There are two types of attributes: exclusive (indicated by the square brackets "[ ]" enclosing the list of possible values) and multivalued (indicated by the curly braces "{ }"). For any object, an exclusive attribute must be assigned exactly one value from its list of possible values; a multivalued attribute may possess any number of values (including zero) from its list. The parenthesized numbers indicate the "frequency" of a given value for an attribute, or at the header of a concept declaration, the "frequency" of objects falling into that concept class. These numbers may either be actual objective frequencies, or subjective estimates of the "typicality" of certain contingencies. If the distribution of attribute values for a concept is the same as for its ancestor, then knowledge of this is inherited, and the distribution need not be given explicitly.

The NEULA system first translates a taxonomy such as that in Figure 1 into a Bayesian belief network [11] representation, which is then implemented as a harmony network (for details of the translation scheme, see [9, 10]). The harmony network [16] consists of two layers of stochastic binary valued units, the *feature* units and the *pattern* units ("knowl-


```
concept animal is basic (1000) with
    offspring : [ living (100), eggs (900) ];
    can       : { swim (900), fly (300), walk (400) };
    eats      : { fish (350), meat (120), plants (650) };
concept mammal is animal (100) with
    offspring : [ living (100) ];
    can       : { swim (50), walk (90), fly(0) };
    eats      : { fish (50), meat (20), plants (50) };
concept dolphin is mammal (10) with
    can       : { swim (10), walk (0) };
    eats      : { fish (10), meat (0), plants (0) };
concept lion is mammal (5) with
    can       : { walk (5) };
    eats      : { fish (0), meat (5), plants (0) };
concept zebra is mammal (10) with
    can       : { walk (10) };
    eats      : { fish (0), meat (0), plants (10) };
concept bird is animal (300) with
    offspring : [ eggs (300) ];
    can       : { swim (200), fly (280), walk (300) };
    eats      : { fish (100), meat (50), plants (200) };
concept penguin is bird (20) with
    can       : { swim (20), fly (0) };
    eats      : { fish (20), meat (0) };
concept flamingo is bird (30) with
    can       : { swim (30) };
    eats      : { fish (30), meat (0) };
concept fish is animal (600) with
    offspring : [ eggs (600) ];
    can       : { swim (600), fly (0), walk (0) };
    eats      : { fish (200), meat (50), plants (400) };
concept shark is fish (10) with
    eats      : { fish (10), meat (10), plants (0) };
concept goldfish is fish (100) with
    eats      : { fish (0), meat (0), plants (100) };
```

Figure 1: A NEULA description of a zoo.

edge" units in [16]). The edges in such a network are undirected and connect only units in different layers. In the case of a harmony network compiled from a NEULA representation, there is one feature unit corresponding to each possible attribute:value or concept:subconcept pair, such as *can:swim*, *eats:plants*, *fish:goldfish*, etc. The pattern units implement the probabilistic relations between the features in a relatively complicated manner into which we shall not go here (see [9]). From the zoo description in Figure 1, the NEULA compiler produces a harmony network containing 24 feature nodes and 285 pattern nodes.

Queries may be performed against knowledge represented in the harmony network by "clamping" the values of some of the feature units into fixed values and running a simulated annealing computation on the network. If the annealing is performed sufficiently slowly[2], it can be shown [16] that with high probability, the values of the unclamped feature units will converge to their probabilistically most likely configuration, given the clamped values.

3 Integrating Prolog and NEULA

The integration of the Prolog interpreter and the NEULA system is implemented using the ability of SICStus Prolog to call external subroutines written in C. First, the NEULA compiler, which is written in C, is invoked from the Prolog interpreter and the necessary C-modules are linked to the Prolog load image. The NEULA compiler then produces, from the given knowledge description, a file containing a patch of NEULA specific Prolog code for the integration. In particular, this file contains Prolog facts named `nunit`. One `nunit` fact is generated for each feature unit in a NEULA harmony network, for the purpose of linking Prolog names to their correspondents in the network. For example, the `nunit/4`[3] predicate for a *fish:goldfish* unit from the zoo network would be `nunit(animalNKB,fish,goldfish,fish_goldfish)`.

After compilation, the fourth arguments of the `nunit` facts (actually the corresponding integer codes) are transmitted to a C function, which creates a global table mapping Prolog terms to the corresponding feature units in the NEULA network.

[2]Which, unfortunately, occasionally means *excruciatingly* slowly. However, the computations are parallelizable, and approximations such as "mean field annealing" [12] are available.

[3]Following standard Prolog meta-notation, the number following the name of a predicate indicates its arity.

The low-level interface between Prolog and NEULA is effectively reduced to three predicates implemented in C. First there is a predicate `n_set/3`, with which one can clamp a NEULA feature node to be true or false, or set an already clamped node free. Another predicate `n_run/1` is used to start a simulation in a NEULA network. With the last predicate `n_ask/3` one can query the state of any NEULA feature unit. A more sophisticated interface can be built by introducing new Prolog predicates, which use the low level interface described above together with the `nunit` facts. For example:

```
set_true(NKB,X,Y) :-
    nunit(NKB,X,Y,Node),
    n_set(NKB,Node,1).            % 1 for true

set_false(NKB,X,Y) :-
    nunit(NKB,X,Y,Node),
    n_set(NKB,Node,0).            % 0 for false

ask_true(NKB,X,Y) :-
    nunit(NKB,X,Y,Node),
    n_ask(NKB,Node,Answer),
    Answer =:= 1.

ask_false(NKB,X,Y) :-
    nunit(NKB,X,Y,Node),
    n_ask(NKB,Node,Answer),
    Answer =:= 0.
```

Using these predicates, one can make for instance the following Prolog query to find out about the diet of a bird incapable of flying:

```
?- set_true(animalNKB,animal,bird),
   set_false(animalNKB,can,fly),
   n_run(animalNKB),
   ask_true(animalNKB,eats,X).

X = fish ;

no
```

The query above shows a simple example of how Prolog atoms and logical variables can be used in a query to a neural knowledge base. Prolog's backtracking mechanism causes three different questions being asked from the neural network: `n_ask(animalNKB,eats_meat,Answer)`, `n_ask(animalNKB,eats_fish,Answer)` and `n_ask(animalNKB, eats_plants,Answer)`.

Using the `set_true` and `set_false` predicates with logical variables gives rise to certain inconveniences due to the mismatch between the Prolog backtracking mechanism and the NEULA notion of a global state (i.e., clamping nodes in a sequence and then running a simulation). For instance, one has to explicitly control the need to run a new simulation. In Prolog, this can be achieved by introducing a global boolean variable `NeedsRunning`, which is set to true each time something is clamped, and to false after every simulation. This way one can make a query trigger a new simulation only if something has been set in a network after the previous query. Conceivably, the NEULA system might fit in more naturally with some forward-chaining production system than Prolog.

4 An Example

In this section, we show by means of a simple example how a NEULA network can be used to provide background knowledge to an incomplete Prolog knowledge base. The Prolog knowledge base contains information about some of the individual inhabitants of the zoo from Section 2. Each individual is introduced by its name in a Prolog predicate `habitant/1`. The explicitly known facts about these individuals are given in terms of Prolog predicates `fact/3` and `fact_not/3`. The predicate `fact` tells that something is known to be true of an individual animal, while the predicate `fact_not` tells that something is known to be false.

The Prolog knowledge base could look like this:

```
habitant(tweety).
   fact(tweety,bird).
   fact(tweety,eats,plants).
   fact_not(tweety,can,fly).
```



```
habitant(cleo).
   fact(cleo,goldfish).

habitant( ...
```

Now, by using the low-level interface described earlier, we have built a `query/3` predicate, with which one can query things about the inhabitants of our zoo, e.g., `query(tweety,offspring,eggs)`, `query(tweety, eats,X)`, or `query(X,eats,Y)`. The predicate `query` first looks up the explicitly given `fact` facts in the Prolog knowledge base. If no answer is found in the Prolog base or if there could be other answers to be found, the predicate then collects all the facts concerning an inhabitant, feeds them to the NEULA network, runs a simulation and collects the results from the net. For example, the query `query(tweety,eats,X)` first yields the result `X = plants`, which is given explicitly in the Prolog knowledge base. Then the background network is set up by the operation sequence `set_true(animalKB,animal,bird)`, `set_true(animalKB,eats,plants)` and `set_false(animalKB,can,fly)`, and a simulation is run by calling `n_run(animalKB)`. Upon convergence the NEULA network has concluded, based on the given feature values, that the animal in question is a penguin, and the Prolog computation retrieves information about its diet by calling `ask_true(animalKB,eats,X)`. This gives two solutions `X = plants` and `X = fish`, the first of which is a duplicate and can be omitted.

Using the `query` predicate, one can now easily build more sophisticated Prolog rules, such as the following ones to find out whether an animal eats another:

```
eats(X,Y) :- query(X,eats,fish), query(Y,animal,fish).
eats(X,Y) :- query(X,eats,meat), query(Y,animal,mammmal).
eats(X,Y) :- query(X,eats,meat), query(Y,animal,bird).
```

Here, asking for example `eats(tweety,X)` yields a solution `X = cleo`, and possibly also other answers depending on the contents of the Prolog base. Note again that the fact "tweety eats fish" needed in this computation is inferred from the background knowledge, based on the penguin-likeness of the explicitly given information about "tweety".

5 Conclusion and Further Work

We have implemented a bi-partite hybrid neural-symbolic system by interfacing together the neural knowledge representation system NEULA and the SICStus Prolog system. The combination yields a working interim solution of the "variable binding" problem in hybrid systems by using purely symbolic variables in high-level reasoning, but with access to a neural representation encoding background knowledge about the properties and relations of the variable referents.

The main problems encountered concern the computational mechanism of the NEULA system, which requires excessive computation times to give reliable results. We are currently working on improving the simulated annealing schedules, and eperimenting with the mean field annealing approximation. A parallel implementation of the system is also planned, as well as experiments with an inherently more efficient variant of the NEULA system [10]. Lesser problems concern the mismatch of the design philosophies of NEULA and Prolog; in a different combination of systems these problems could conceivably be completely avoided.

References

[1] V. Ajjanagadde and L. Shastri, Rules and variables in neural nets. *Neural Computation* 3 (1991), 121–134.

[2] J. A. Barnden and J. B. Pollack (eds.), *Advances in Connectionist and Neural Computation Theory, Vol. 1: High Level Connectionist Models*. Ablex Publ. Co., Norwood, NJ, 1991.

[3] J. A. Barnden and J. B. Pollack, Introduction: Problems for high-level connectionism. Pp. 1–16 in [2].

[4] M. Carlsson, J. Widén, SICStus Prolog User's Manual. Res. Rep. R88007, Swedish Institute of Computer Science, Stockholm/Kista, 1988.

[5] J. A. Hendler, Developing hybrid symbolic/connectionist models. Pp. 165–179 in [2].

[6] J. A. Hendler, Marker-passing over microfeatures: Towards a hybrid symbolic/connectionist model. *Cognitive Science* 13 (1989), 79–106.

[7] G. E. Hinton (ed.), Special Issue on Connectionist Symbol Processing. *Artificial Intelligence* 46:1–2 (1990).

[8] F. Kurfeß, Unification on a connectionist simulator. Pp. 471–476 in: *Proc. of the 1991 International Conference on Artificial Neural Networks (ICANN-91) (Espoo, Finland, June 1991), Vol. 1* (T. Kohonen et al., eds.). Elsevier–North-Holland, Amsterdam, 1991.

[9] P. Myllymäki and P. Orponen, Programming the Harmonium. Pp. 671–677 in: *Proc. of the International Joint Conf. on Neural Networks (Singapore, Nov. 1991), Vol. 1.* IEEE, New York, NY, 1991.

[10] P. Orponen, P. Floréen, P. Myllymäki, H. Tirri, A neural implementation of conceptual hierarchies with Bayesian reasoning. Pp. 297–303 in: *Proc. of the International Joint Conf. on Neural Networks (San Diego, CA, June 1990), Vol. I.* IEEE, New York, NY, 1990.

[11] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann, San Mateo, CA, 1988.

[12] C. Peterson and J. R. Anderson, A mean field theory learning algorithm for neural networks. *Complex Systems* 1 (1987), 995–1019.

[13] T. Plate, Holographic reduced representations: Convolution algebra for compositional distributed representations. Pp. 30–35 in: *Proc. of the International Joint Conf. on Artificial Intelligence (Sydney, August 1991), Vol. 1* (J. Mylopoulos and R. Reiter, eds.). Morgan Kaufmann, San Mateo, CA, 1991.

[14] L. Shastri, A connectionist approach to knowledge representation and limited inference. *Cognitive Science* 12 (1988), 331–392.

[15] L. Shastri, *Semantic Networks: An Evidential Formalization and Its Connectionist Realization.* Pitman, London, 1988.

[16] P. Smolensky, Information processing in dynamical systems: Foundations of Harmony Theory. Pp. 194-281 in: *Parallel Distributed Processing, Vol. 1* (D. E. Rumelhart and J. L. McClelland, eds.). The MIT Press, Cambridge, MA, 1986.

[17] P. Smolensky, Tensor product variable binding and the representation of symbolic structures in connectionist systems. Pp. 159–216 in [7].