Neuro-Symbolic Representation of Logic Programs Defining Infinite Sets

Ekaterina Komendantskaya[1], Krysia Broda[2], and Artur d'Avila Garcez[3]

[1] School of Computing, University of Dundee, UK *
[2] Department of Computing, Imperial College, London, UK
[3] Department of Computing, City University London, UK

Abstract. It has been one of the great challenges of neuro-symbolic integration to represent recursive logic programs using neural networks of finite size. In this paper, we propose to implement neural networks that can process recursive programs viewed as inductive definitions.
Key words: Neurosymbolic integration, Structured learning, Mathematical theory of neurocomputing, Logic programming.

1 Introduction

Neuro-symbolic integration is the area of research that endeavours to synthesize the best of two worlds: neurocomputing and symbolic logic. The area was given a start in 1943 by the pioneering paper of McCulloch and Pitts that showed how Boolean logic can be represented in neural networks; we will call these *Boolean networks*. Neuro-symbolism has since developed different approaches to inductive, probabilistic and fuzzy logic programming [2, 3, 11].

Various neuro-symbolic approaches that use logic programs run over finite domains have been shown effective as a hybrid machine learning system [3]. However, when it comes to recursive logic programs that describe infinite sets, Boolean networks become problematic, for they may require networks of infinite size. Some approaches to solve this problem use finite approximations of such networks, [1, 5], but the approximations may be difficult to obtain automatically.

In this paper, we propose to take a new look at recursive logic programs, that is, to approach them not from the point of view of *first-order logic*, but from the point of view of functional programming [9]. As an example, consider how a formal grammar generates the strings of a language. Grammars are inductive definitions, i.e. rules that generate a set. In [7], we have introduced neuro-symbolic networks that can process inductive definitions given in a functional language, and applied these networks to data type recognition. In this paper, we show how this neuro-symbolic construction can be applied to processing recursive logic programs. The idea is that inductive definitions can be used as set *generators* or term *recognisers*. The former can generate elements of a set from the inductive definition, the latter can recognise whether an element satisfies the inductive definition, and hence belongs to the defined set.

* The work was supported by EPSRC, UK; PDRF grant EP/F044046/2.

2 Neural Networks as Inductive Generators

In the standard formulations of logic programming [8], a logic program consists of a finite set of clauses (or rules) of the form $A \leftarrow A_1, \ldots, A_n$ where A and the A_i's are atomic formulae, typically containing free variables; and $A_1, \ldots, A_n$ denotes the conjunction of the A_i's. Note that n may be 0, such clauses are called *facts*. We assume that the logical syntax has a numerical encoding suitable for neural networks, cf. [6]. Let us start with an example.

Example 1. The program below corresponds to the inductive definition of the set of natural numbers in functional languages, where S(n)=n+1. Using the syntax below, number 3 will be given by a term `S(S(S(O)))`.

```
nat(O) <- // zero is a natural number
nat(S(n)) <- nat(n) // if n is a natural number, so is S(n)
```

Recursive clauses require the predicate (e.g. nat) appearing on the left-hand side (called the head) of a clause to appear also on the right-hand side (called the body) of the clause, and variables (e.g. n) appearing in the head to appear in the body within the same predicate. The head must contain a function symbol (e.g. `O` or `S`); such functions play the role of constructors in the inductive definitions of functional languages. Certain inductive definitions do not contain recursion of any kind. Such programs inductively define finite sets.

Example 2. Logic program defining the set of boolean values:

```
bool(t) <-  // true is a boolean
bool(f) <-  // false is a boolean
```

The last distinction we need to make is between *simple* and *dependent* definitions. All the inductive definitions we have considered so far were simple, in that they did not depend on other inductive definitions. Consider the example of the dependent inductive definition of lists of elements of a certain type, e.g. `nat`. The definition of this type not only involves recursion, but it is also dependent on another inductive definition: `nat`. See [7].

There are two common uses for inductive definitions: they can be used to *generate* the elements of a set - if they are read from right to left; and they can be used for type-checking expressions - if they are read from left to right. Both implementations require finite and terminating computations. Figure 1 shows some network architectures for generating and recognising expressions.

3 Neural Networks as Recursive Recognisers

We now turn to networks that can process recursive logic programs viewed as inductive definitions, see [7] for the full formal analysis of these networks. It has been shown in [7] that there exists a general method that allows to construct the networks from the specification of an inductive definition.

Given a recursive clause `X(C(y))` $\leftarrow$ `X(y)`, the *recursive recogniser* for `C` is a one layer network, consisting of $n > 1$ neurons with the following properties. The length n of the single layer is the length of the input vector that the network will process. Each neuron has one input connection, with weight 1. The biases of all

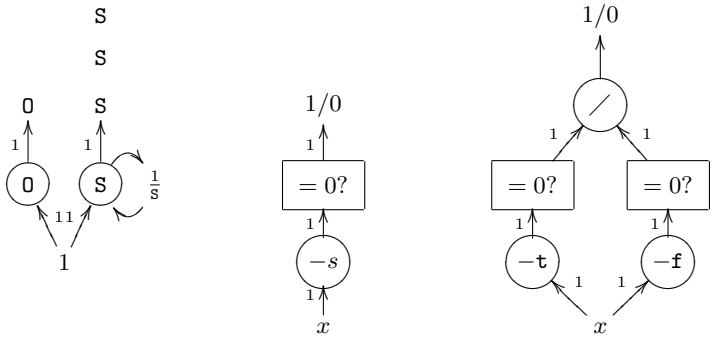

Fig. 1. Left: Generating elements of set `nat`. The input 1 is sent to the two neurons with zero biases and with activation functions $f(x) = x * O$ and $f(x) = x * S$, where O and S are the numerical encodings of `O` and `S`. The recursive weight is set to $\frac{1}{S}$. At time 1, the network outputs O and S standing for natural numbers 0 and 1, at time 2, it will generate another S, which will stand for $S(S(0))$ - or natural number 2. In the diagram - it is time 3, and the term $S(S(S(O)))$ is formed. **Centre:** Recognising symbol s. The input is sent to the neuron with bias $-s$; it outputs 0 if the input matches s and some non-zero value otherwise. This neuron may be connected to a zero-recogniser (in the square box) that outputs 1 whenever the signal 0 is computed. **Right:** Recognising expressions satisfying inductive definition `bool` from Example 2.

but the first neuron are set to 0; the bias of the first neuron is set to $-n_C$, where n_C is a numerical representation of C. The first neuron has an *output connection* that can be received by an external user. The outputs of the 2nd to nth neurons, called *recursive outputs*, are connected to the same layer, as follows: the output connection of the kth neuron ($k \in 2, \ldots, n$) is sent as an input to the $k-1$ neuron. Note that the first neuron of such network is the symbol recogniser for the function `C`. The other $n-1$ neurons in the layer are designed to recursively process the remaining $n-1$ elements of the input vector, see 2.

It is possible to connect vectors of neurons in a cascade in order to recognise dependent types. For example, to recognise a symbolic term of type `list(nat)`, composed of nested `cons` in the standard way, we can use two layers of neurons. The lower layer is used to recognise the list structure, and the upper layer to recognise `nat` structure, see [7].

4 Conclusions and Future Work

In this paper, we have applied the general method [7] to recursive logic programs. We have explained two neuro-symbolic methods that work with inductive definitions: *inductive generators* and *recursive recognisers*. The methods can be applied to logic programs with infinite Herbrand bases for which the traditional model-theoretic methods cannot be applied directly.

Taking on board the "logic programs as inductive definitions" idea, we wish to revise the traditional methods of building neuro-symbolic networks. The goal is to compare results with the traditional implementations of semantic operators

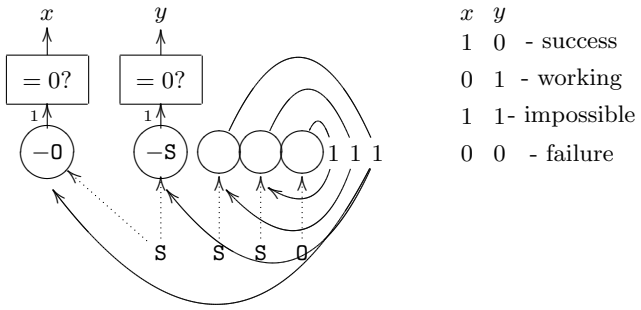

Fig. 2. This network decides whether an expression `S(S(S(O)))` satisfies the inductive definition of `nat`. Such network receives an input vector i, given by numerical encoding of the term - `S(S(S(O)))`. The dotted arrows show the initial input to the network; the solid arrows show the connections with weight 1. The network has two components: `O`-recogniser, and recursive `S`-recogniser: the recursive outputs from neurons in the `S`-recogniser layer are sent to the same layer.

and proof systems, resolution and unification. The hope is that the functional approach would be more natural to integrate with neural networks [4, 10].

References

1. S. Bader, P. Hitzler, and S. Hölldobler. Connectionist model generation: A first-order approach. *Neurocomputing*, 71:2420–2432, 2008.
2. A. d'Avila Garcez, K. B. Broda, and D. M. Gabbay. *Neural-Symbolic Learning Systems: Foundations and Applications*. Springer-Verlag, 2002.
3. A. d'Avila Garcez, L. C. Lamb, and D. M. Gabbay. *Neural-Symbolic Cognitive Reasoning*. Cognitive Technologies. Springer-Verlag, 2008.
4. T. Gartner, J. Lloyd, and P. Flach. Kernels and distances for structured data. *Machine Learning*, 3(57):205–232, 2004.
5. H. Gust, K.-U. Kühnberger, and P. Geibel. Learning models of predicate logical theories with neural networks based on topos theory. In *Perspectives of Neural-Symbolic Integration*, pages 233–264. Springer, 2007.
6. E. Komendantskaya. Unification neural networks: Unification by error-correction learning. *J. of Algorithms in Cognition, Informatics, and Logic (in print)*, 2009.
7. E. Komendantskaya, K. Broda, and A. d'Avila Garcez. Using inductive types for ensuring correctness of neuro-symbolic computations. In *CiE'10 booklet*, 2010.
8. J. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, 2nd edition, 1987.
9. L. C. Paulson and A. W. Smith. Logic programming, functional programming, and inductive definitions. In *ELP*, pages 283–309, 1989.
10. P. Smolensky and G. Legendre. *The Harmonic Mind*. MIT Press, 2006.
11. J. Wang and P. Domingos. Hybrid markov logic networks. In *Proc. AAAI'08*, pages 1106–1111, 2008.