Abstract

The complete definition of a Neuro-Symbolic Language, partially introduced in [1], for monotonic and non-monotonic logical inference by means of artificial neural networks (ANNs) is presented. Both the language and its compiler have been designed and implemented. It has been shown that the ANN model here adopted (NFC – Neural Forward Chaining [2]) is a massively parallel abstract interpreter of definite logic programs; moreover, inhibition is used to implement a neural form of logical negation. Previous compiler

1. Introduction

In this paper we introduce a Neuro-Symbolic Language (NSL) for describing monotonic and non-monotonic parallel logical inferences with ANNs. The language is formed by monotonic and non-monotonic operators (formally introduced in [1]), iterative and conditional constructs and a set of additional instructions. The neural network model here adopted is the one called NFC (Neural Forward Chaining). The model has been introduced in Aiello et al. [2] and it is based on a unified LCA (Localist Connectionist Architecture) approach [4] of rule-based systems proposed by Burattini et al. in 1992 [5]. The NFC model carries on a parallel forward inference process for propositional rule-based systems. Furthermore, for each NSL program there exists the corresponding NFC neural network, which is generated by a compiler program based on the Neuro-Symbolic approach introduced in [3]. The NFC computation captures the symbolic meaning of the corresponding NSL propositions.

The new extensions of the language introduced in [1] allows one to better describe the knowledge about a propositional rule-based system.

In this paper we briefly analyse non-monotonic NFC model, based on a proper use of inhibitions, to implement a parallel non-monotonic reasoning (in this case, inhibitions play a key role in asserting and retracting facts – see [1]).

In non-monotonic ANNs, reverberation, that is cyclic sequences of ANN states, could be present. Output and end-computation definitions for non reverberating networks give no problem and are easy to implement [3]. On the other hand, for reverberating neural networks, output and end-computation definitions have to be settled.

In the next sections we describe NSL formalism and the non-monotonic NFC model. Furthermore, we introduce the definitions of output and end-computation for reverberating neural networks and the logical aspects of the non-monotonic NFC model. Finally, we approach to the labyrinth problem writing an NSL program to find the shortest path length between two distinct points.

2. Knowledge representation

The neuron model adopted here is the Weighted-Sum non-Linear Thresholded Element of McCulloch and Pitts [6]. The two possible truth-values of each literal \( p \) are represented by means of two distinguished neurons \( p_e \) and \( \neg p \): the first is activated if and only if the corresponding literal is supposed to be true, the second if and only if it is supposed to be false. Inactivity of these neurons means that we do not have information about the truth-value of the corresponding literal. A similar approach has been already proposed by von Neumann (Double Line Trick – [7]).

Let \( P \) be a set of literals. We denote with \( P_n \) and with \( P_s \), respectively the conjunction and the disjunction of \( P \) elements (\( P_{ns} \) will denote either \( P_n \) or \( P_s \)). The basic operators we use to represent knowledge about a problem are classified into: a) monotonic operators \textbf{IMPLY} and \textbf{ATLEAST}; b) non-monotonic operators \textbf{UNLESS} and \textbf{ATMOST}. The statement “\( p_i \) is true if \( P_n \) is true” is denoted by \textbf{IMPLY}(\( P_{ns}, p_i \)). The statement “\( p_i \) is true if \( P_{ns} \) is true, unless \( Q_{ns} \) is true” is denoted by \textbf{UNLESS}(\( P_{ns}, Q_{ns}, p_i \)). The statement “\( p_i \) is true if at least \( h \) literals belonging to \( P \) are true” is described by operator

---

1 Copyright n. 001109/D801353-23/12/98
ATMOST($P_m, h, p_i$). Finally, we represent “$p_i$ is true if at most $h$ literals belonging to $P$ are true” with ATLEAST($P_m, h, p_i$).

These operators are represented in the NFC model by means of a set of neurons, using a Neuro-Symbolic approach formally discussed in Burattini et al. [1]. For instance, in fig. 1 is sketched the $\text{UNLESS}(P_\wedge, Q_\vee, p_i)$ neural representation, where $P_\wedge$ is the conjunction $p_1 \wedge \ldots \wedge p_n$ of $n$ literals and $Q_\vee$ the disjunction $q_1 \vee \ldots \vee q_m$ of $m$ literals.

![Fig. 1 – UNLESS($P_\wedge, Q_\vee, p_i$) neural representation](image)

The threshold value of neuron $p_i$ is $n \cdot \varepsilon$ (with $0 < \varepsilon < 1$), the coupling coefficients with $P_\wedge$ neurons are $a_{i,j} = 1$ and the coupling coefficients with $Q_\vee$ neurons are $a_{i,j} = -1$. The neuron $p_i$ will fire at time $t + 1$ if and only if all $P_\wedge$ neurons fire at time $t$ and no neurons belonging to $Q_\vee$ fire at time $t$. We remark that activation of one or more $Q_\vee$ neurons inhibits $p_i$ neuron, whatever all $P_\wedge$ neurons are firing. Thus, the neural network of fig. 1 captures the symbolic meaning of $\text{UNLESS}$ operator. The complete set of operator definitions with their corresponding neural representation are reported in [1].

3. NSL features and formalism

NSL is a context-free language and its grammar is written in the BNF (Backus-Naur) formalism [8]. The language is formed by: a) a set of monotonic and non-monotonic operators; b) a set of iterative and conditional constructs (based on a Pascal-like syntax); c) a set of additional instructions (input/output instructions, assign instruction, ...). The NSL language allows one to write programs in order to generate NFC networks for representing and solving logical inference problems.

Let $p$ be a literal and $S$ and $T$ two arithmetical expressions. Furthermore, let $s \in t$ be the results of evaluation of $S$ and $T$ expressions. The NSL language allows one to write some formal expressions as argument of language operators, such as: $p[S], p[S..T]$ and $p[S..T]$. Each of these expressions is translated by the NSL-compiler into the corresponding ones: $p, p_1 \wedge \ldots \wedge p_n$ and $p_{n+1}$. For instance, the instruction:

\[ \text{IMPLY}(a[1..i*2] \wedge b[j], c[i, j]) \]

for $i=2$ and $j=1$, is translated into the equivalent one:

\[ \text{IMPLY}(a[1..i*2], c[i, j]) \]

The instructions for setting the variable values are reported at the end of this section.

The language takes advantage of constructs such as: $\text{FOR}, \text{IF THEN ELSE}, \text{WHILE DO}$ and $\text{REPEAT UNTIL}$. For example, the following code:

\[ \begin{align*}
  \text{READ}(i) \\
  \text{IF} \ (i=1) \\
  \text{THEN} \\
  \text{BEGIN} \\
  \text{FOR} \ (j, 1, 6) \\
  \text{IMPLY}(a[1..j], c[j]) \\
  \text{END} \\
  \text{ELSE} \\
  \text{BEGIN} \\
  \text{FOR} \ (j, 1, 6) \\
  \text{UNLESS}(a[j], b[j+1], c[j]) \\
  \text{END}
\end{align*} \]

where \text{BEGIN} and \text{END} have to be interpreted as block delimiters. One can notice that the use of variables and one \text{FOR} instruction allows one to represent the knowledge about the logical problem in a more compact way.

Another example shows the use of \text{IF THEN ELSE} construct:

\[ \begin{align*}
  \text{READ}(i) \\
  \text{IF} \ (i=1) \\
  \text{THEN} \\
  \text{BEGIN} \\
  \text{FOR} \ (j, 1, 6) \\
  \text{IMPLY}(a[1, c]) \\
  \text{IMPLY}(a[1 \wedge a[2, c2]) \\
  \text{IMPLY}(a[1 \wedge a[2 \wedge a[3, c3]) \\
  \text{IMPLY}(a[1 \wedge a[2 \wedge a[3 \wedge a[4, c4]) \\
  \text{IMPLY}(a[1 \wedge a[2 \wedge a[3 \wedge a[4 \wedge a[5, c5]) \\
  \text{IMPLY}(a[1 \wedge a[2 \wedge a[3 \wedge a[4 \wedge a[5 \wedge a[6, c6]) \\
\end{align*} \]

otherwise, the following one:

\[ \begin{align*}
  \text{UNLESS}(a[1, b2, c1]) \\
  \text{UNLESS}(a[2, b3, c2]) \\
  \text{UNLESS}(a[3, b4, c3]) \\
  \text{UNLESS}(a[4, b5, c4]) \\
  \text{UNLESS}(a[5, b6, c5]) \\
  \text{UNLESS}(a[6, b7, c6]) \\
\end{align*} \]
In general, we are able to represent a logical problem in terms of a set of operators. Of course, even changing one parameter value of the problem we are obliged to rewrite part of the operators. With an NSL program, thanks to the introduction of variables, we can approach the same logical problem in a more general way. In fact, the previous example emphasises how the same NSL program generates different code with respect to an external parameter given as input (READ(i)).

We report here the remaining instructions of the NSL language:

- **ASSIGN**(<var>, <expr>) set the variable <var> to <expr> evaluation;
- **READ**(<var>) read <var> from the standard input stream;
- **WRITE**(<message>) write <message> to the standard output stream;
- **INPUT**(I₁, …, Iₙ) set the possible inputs for the corresponding NFC network;
- **OUTPUT**(O₁, …, Oₙ) set the possible outputs for the NFC network.

4. Non-monotonic NFC networks

Arithmetical evaluation mechanism, replacement operation on data structures and iterative and conditional construct interpretation (see §3), allow one to translate any NSL program into the corresponding set of NSL operators (called **NSLO version**). This set is formed only by monotonic and non-monotonic operators and by **INPUT** and **OUTPUT** instructions. Moreover, it can be shown that all operators can be represented only by **ATLEAST** and **UNLESS** operators.

Let P be an NSL program. We have designed and implemented a *Neuro-Symbolic compiler* for deriving automatically the NFC neural network corresponding to P (see [8]). This compiler is formed by three different modules (see fig. 2).

**Fig. 2 – Neuro-Symbolic compiler structure**

The **Syntactic Module** carries on a syntactic analysis of P. If no syntactic errors occur, the next module, called **Pre-Compiler Module** translates P into the corresponding NSLO version P₀. At last, the **Compiler Module** derives the NFC network, which can be both simulated by means of **Simulation Module**, or implemented on a FPGA device.

We call the NFC network generated by the program P:

- a) **monotonic** if P contains only monotonic operators;
- b) **non-monotonic** if P contains at least one non-monotonic operator.

The non-monotonic NFC model implements a non-monotonic reasoning, that is, the introduction of new premises (literals), initially supposed true, could invalidate previously inferred conclusions.

Since the monotonic NFC model is widely described in [3], we focus our discussion only on the non-monotonic one. Let F be a set of initial premises and let P be the following NSLO program:

\[
\begin{align*}
&> \text{UNLESS}(a, p, q) \\
&> \text{UNLESS}(a, q, p) \\
&> \text{IMPLY}(b, a)
\end{align*}
\]

In fig. 3 we report the non-monotonic NFC network related to P (dashed lines indicate inhibitory connections).

**Fig. 3 – A non-monotonic NFC network**

The first layer (IN) accepts external inputs to the network and it is used to assert the set of initial premises supposed to be true (also called *facts*). It could be formed by as many neurons as different literals appearing in P instructions (we can reduce their number by means of **INPUT** instruction). The second layer (DB) is a database formed by as many neurons as in the layer IN, and it stores the premises asserted in the IN layer. The third layer (KB) codifies the entire knowledge base and the inferential process (see fig. 4).

**Fig. 4 – KB layer**
This layer is based on neural representation of monotonic and non-monotonic operators, as discussed in §2 (see fig. 1 for the UNLESS neural representation).

At last, OUT layer represents the set of possible conclusions (as for the neurons belonging to the IN layer, we can reduce the number of output neurons by means of OUTPUT instruction). The activation of END node indicates that the end of computation has been reached.

The End-computation sub-network controls the completion of the inferential process while the Output sub-network selects the results of the computation; both sub-networks are designed and perform according to the definitions given in the next section.

In the previous example (see fig. 3), exciting the neuron \( b \) of the IN layer one obtains the \( \text{KB} \) neurons temporal evolution shown in fig. 5:

![Fig. 5 – KB layer temporal evolution](image)

One can notice that both neurons \( a \) and \( b \) keep firing indefinitely, while \( p \) and \( q \) neurons keep oscillating; that is, they pass from quiescent to exciting state, and vice versa indefinitely (reverberating network). Intuitively, we could say that facts \( a \) and \( b \) are the conclusions of the inferential process, while \( p \) and \( q \) are oscillating literals for which we are not able to determine their truth-value. In order to define the set of conclusions in case of reverberating networks, we need to define the output of such networks. In the next section, we propose a suitable End-computation and Output definitions.

### 5. End-Computation and Output definitions

Temporal evolution of a reverberating NFC networks is reported in fig. 6. \( I \) represents the input of the network and \( U(t) \) represents the generic network state at time \( t \) that is the set of \( \text{KB} \) nodes firing at time \( t \). The temporal evolution of such a network, whose states depend only on the immediately previous one, has a transient formed by states \( U(0), ..., U(t-1) \) and a reverberation formed by oscillating states \( U(\tau), ..., U(\tau + L) \), where \( L \) is called reverberation period of the network for input \( I \).

![Fig. 6 – Temporal evolution of a generic non-monotonic NFC](image)

If \( L > 0 \), an infinite cyclic sequences of oscillating states is present. If \( L = 0 \) the network is called reverberation free or non oscillating, and the unique state \( U(\tau) \) belonging to the reverberation is called stable state.

For non oscillating networks \( (L = 0) \) the output, that is formally the set of conclusions, is equal to the stable state; computation ends at time \( \tau \), that is, when stable state is reached. We wish now to introduce a suitable output and end-computation definitions for reverberating NFC networks \( (L > 0) \):

a) the output is the intersection \( U(\tau) \cap ... \cap U(\tau + L) \) of oscillating states, namely, the set of \( \text{KB} \) nodes always active in the reverberation period;

b) network computation ends when all oscillating states have been labeled. To label a state we must wait that network evolves twice into it. This is the only way to distinguish transient states by oscillating ones: in fact, a network evolves only one time in the former and more times in the latter.

From the previous definitions, the output of the network for the example reported in fig. 5 is \( \{a, b\} \).

In a reverberating NFC network there is a never-ending cyclic sequence of oscillating states, and we can consider a computation ended when all this states have been labeled. Then, we do not consider oscillating literals as belonging to the set of conclusions (they are asserted and retracted every reverberation period). We accept as conclusions of the logical inferential process only those literals belonging to each single reverberation state.

### 6. NFC model for logical inference

A propositional normal logic program \( P \) is a collection of clauses of the form \( a \leftarrow b_1, ..., b_m \neg c_1, ..., \neg c_n \) where \( m, n \geq 0 \), \( a \) is a literal called conclusion and \( b_1, ..., b_m, c_1, ..., c_n \) are literals, called premises. A program \( P \) is definite (or negation-free) if contains only clauses of the form \( a \leftarrow b_1, ..., b_n \) (Horn clauses). An abstract interpreter of logic programs is based on unification algorithm and resolution method, formulated by Robinson in 1965 [9]. Moreover, a normal logic program interpreter is based on the NFR (negation as failure rule) interpretation of \( \neg \) symbol, formulated by Clark in 1978 ([10]), that is only a partial form of logical negation.

A premises-free clause \( a \leftarrow \) is called fact, the remaining clauses are called rules. Let \( a \leftarrow b_1, ..., b_n \) be a rule of a definite logic program \( P \). We write it in our language with an IMPLY operator:

\[
\text{IMPLY}(b_1 \land ... \land b_n, a)
\]

In this way, rules are represented in the \( \text{KB} \) layer of the monotonic NFC model corresponding to \( P \), and facts are...
asserted by exciting nodes of the IN layer. Because all the applicable rules are processed and executed all at once, we can assert that the NFC model is a massively parallel abstract interpreter of definite logic programs. In [8] has been proved the correctness and completeness features of the monotonic NFC model.

Since NFC network structure depends exclusively on the set of logical rules (it does not depend on the set of facts), NFC network simulates a class of logic definite programs for any instance of the initial facts. Furthermore, there exists a one-to-one correspondence between logic program and NFC network when an instance of the set of initial facts is given as input to the network.

In a similar way, we can write a rule with negation, such as \[ a \leftarrow b_1, \ldots, b_n, \neg c_1, \ldots, \neg c_m, \] by means of UNLESS instruction:

\[
\text{UNLESS}(b_1 \land \ldots \land b_n, c_1 \lor \ldots \lor c_m, a).
\]

Since in the neural representation of UNLESS operator there are inhibitory connections (see fig. 1), it is clear that inhibition is used to implement a neural form of logical negation. Relationships between logic programs and non-monotonic NFC networks are still an open problem, the reporting of which falls, however, beyond our present scope.

7. An example

In this section, we approach the problem of finding the length of the shortest path, if exists, between two cells of a given labyrinth represented by an \(m \times n\) grid.

Let \(x_s\) and \(x_o\) be respectively the starting cell and the arrival cell, and let \(O = \{O_1, \ldots, O_k\}\) be a set of possible obstacles. Furthermore, let us suppose that only horizontal and vertical one-cell-moves are allowed.

We label with literals \(x_{i,j}\) and \(o_{i,j}\) the grid cells. When \(x_{i,j}\) is true, we can assert “the cell represented by \(x_{i,j}\) is reachable from one of its adjacent cells”, otherwise nothing can be said about that. In the same way, we assert that, when \(o_{i,j}\) is true, “in the cell represented by \(o_{i,j}\) there is an obstacle”.

We approach the labyrinth problem as a deductive one. The only rule to be written is the following:

\[ x_{i,j} \text{ is true IF}
\]

one of \(x_{i-1,j}, x_{i+1,j}, x_{i,j-1}, x_{i,j+1}\) is true, unless \(o_{i,j}\) is true.

The initial set of facts supposed to be true (input), will be formed by \(x_s\) and by the set \(o_{i,j}\) (representing respectively the starting cell and the obstacles in the labyrinth). If from the initial set of fact is possible to derive \(x_o\) then there exist at least one path between \(x_s\) and \(x_o\), otherwise \(x_o\) is unreachable.

The rule written in terms of UNLESS operator is:

\[
\text{UNLESS}(x_{i-1,j} \lor x_{i+1,j} \lor x_{i,j-1} \lor x_{i,j+1}, o_{i,j}, x_{i,j})
\]

while the corresponding NSL program for solving a generic \(m \times n\) labyrinth problem is the following:

```python
> WRITE('Insert grid dimension [m x n]: ')
> READ(m, n)
> FOR (i, 1, m)
>   FOR (j, 1, n)
>     UNLESS(x[i-1,j] \lor x[i+1,j] \lor x[i,j-1] \lor x[i,j+1],
>             o[i,j], x[i,j])
```

Once the related NFC network has been generated, the user can set the inputs (initial facts) by exciting the corresponding neurons in the IN layer. In the OUT layer, after the activation of the neuron \(\textit{END}\), the user can see whether the neuron \(x_o\) is active (\(x_o\) is reachable), and in this case, the number of NFC computation steps\(^2\) represents the length of the shortest path between \(x_s\) and \(x_o\).

Let us suppose that user chooses the 6x6 dimension grid labyrinth shown in fig.6 (the ticker path is the shortest one – 10 moves). The NFC network finds the solution after 17 computation steps. This number is the sum of 10 and 7. 10 is the number of \(KB\) layer computation steps, and 7 is the constant number of computation steps for any kind of problem approached by an NFC network (computation steps of the remaining layers).

In the above example, we have seen how to approach the labyrinth problem in a very general way (for any dimension grid and for different start and arrival cells. We have used just one NSL operator (UNLESS) to represent the knowledge related to the logical problem, and the corresponding NSL program is formed by few lines of code.

\[^2\] A neuron activated at time \(t\) will fire at time \(t+\tau\) We define \(\tau\) as NFC computation step.
8. Concluding remarks

We have outlined that NFC model is a massively parallel abstract interpreter of definite logic programs and NFC network structure does not depend on the set of initial facts but only on the set of logical rules. NFC network simulates a class of logic programs for any instance of initial facts. Furthermore, inhibition is used to implement a neural form of logical negation. While correctness and completeness have been proved for the monotonic NFC model [8], we cannot assert the same for the non-monotonic one.

The new features of the NSL language make the two corresponding compilers (1 – NLS program to NFC network; 2 – NFC network to VHDL code) more powerful than the previous ones [1]. With these compilers, logical problems can be translated directly into the corresponding VHDL code to set electronic devices like, for instance, FPGA (Field Programmable Gate Arrays) [11]. Each of these devices is reprogrammable and, should some of the NSL instructions be changed for some reason, the same reprogrammable device is ready to be used again, after that the modified NSL program has been compiled again.

In conclusion, we may assert that with FPGA devices we can effectively implement a parallel interpreter of logic programs.

References


