# Fast Relational Learning using Bottom Clauses in Neural Networks

Manoel V. M. Franca[1,2], Gerson Zaverucha[1], and Artur S. d'Avila Garcez[2]

[1] Programa de Engenharia de Sistemas e Computação
COPPE, Universidade Federal do Rio de Janeiro
21941-972 Rio de Janeiro, Brazil
gerson@cos.ufrj.br
http://www.cos.ufrj.br
[2] Department of Computing
School of Informatics, City University London
EC1V 0HB London, United Kingdom
{manoel.franca.1,a.garcez}@city.ac.uk
https://www.city.ac.uk/informatics

**Abstract.** In this paper we introduce a method and algorithm for ILP learning using neural networks, by extending a neural-symbolic system called C-ILP (Garcez and Zaverucha, 1999). It is a neural-symbolic system that uses a background knowledge encoded as an initial propositional logic program to build a recurrent neural network and furthermore, uses examples to apply standard back-propagation learning. This integration inherited both parallel learning from neural networks and explanatory power from propositional logic.
We will use the most specific clause of ILP systems that are based on Inverse Entailment, called Bottom Clause, to allow learning with neural networks. More precisely, Bottom Clauses will be generated for each training example and then, applied on an adapted version of C-ILP. Experimental comparisons with a traditional ILP system, Aleph, will be made using *alzheimers* benchmarks and the results show that our neural-symbolic approach for First-Order Logic has similar classification accuracy but it is considerably faster.

**Keywords:** Relational Learning, Inductive Logic Programming, Neural Network

## 1 Introduction

Connectionism and symbolic paradigms always struggled with each other to try to find out which one has better learning, efficiency and generalization capabilities. The fact is that what each one cannot achieve by itself, can be obtained by combining them together [1].

In this paper we introduce a method and algorithm for ILP learning using neural networks, based on the neural-symbolic system C-ILP [1], and report initial experimental results using *alzheimer* datasets. C-ILP is capable of building

a initial neural network with a background knowledge in the form of a propositional logic program, training it with examples, infer unknown data and extracting its acquired knowledge to update the current background data [2] and the results will show that we achieved fair testing accuracy in comparison with Aleph, a traditional and freely distributed ILP system, but having considerably lower processing time.

### 1.1 An Example

Consider a small family environment, in terms of ILP:

| **Background Knowledge:** | **Positive Examples:** |
|---|---|
| parent(mom1, daughter11); | motherInLaw(mom1, husband1); |
| wife(daughter11, husband1); | **Negative Examples:** |
| wife(daughter12, husband2); | motherInLaw(daughter11, husband2); |

The target predicate, *motherInlaw*, can be learned in many different ways, depending on the learning approach which one chooses to take: *inverse entailment* [3], *inverse resolution* [4], *answer set programming* [5], and so on. Considering *Progol*, one very known *inverse entailment*-based system, the first step to be taken in order to solve this problem would be to limit the hypothesis search space, and it does this by setting bounds.

Hypothesis search boundary in Progol is set by two special clauses: the most general clause $[motherInLaw(A, B) \leftarrow]$ and the *most saturated clause* that can be built with the background knowledge, called *Bottom Clause* ($\perp_e$). The small $e$ in the notation is to point out that a bottom clause is generated by saturating one example and thus, different examples can generate different bottom clauses.

Our work intends to use one of those search boundaries, $\perp_e$, to train a neural-symbolical system that works with propositional data, C-ILP [1]. Since it is the most complete representation that an example can have in ILP, a system capable of dealing properly with redundant data and extensive input representations maybe can learn from it. C-ILP, which uses a recursive neural network, is a very good candidate for this task. We will generate one bottom clause for each example, treat each variablized literal as a propositional atom for C-ILP and then do its training algorithm for learn from them the common "features" that positive and negative examples holds.

### 1.2 Related Work

Several studies have already been made regarding how neural networks can deal with relational learning [6]: they show that connectionism can be a good alternative for this kind of learning and this motivates further development of neural-symbolic systems.

FOLNN [7], for example, is a neural-symbolic system that defines a number of "free variables" that will be used to generate all combinations of variablized

predicates that exists in the background knowledge of a given problem, as well as the desired target predicate. It uses a standard neural network to learn and it builds it in a different way than we do: an input neuron is created for each variablized background knowledge predicate that has been created and an output neuron is created for each variablization of the target predicate.

Since we use Progol's *hash* function to variablize bottom clauses and apply them directly in a connectionist system, this approach can be seen as a kind of propositionalization. As such, well-known propositionalization algorithms such as RSD [8] and LINUS [9] can be related to this work as well.

The remainder of this paper is as follows: in Section 2 we will review two main areas that are involved in this work, ILP and Neural Networks, and briefly introduce C-ILP. In Section 3 we introduce our work: using bottom clauses as training examples for a neural network. In Section 4, all experiments made with the *Alzheimer* datasets will be shown and we conclude this with some final remarks and comments in the Section 5.

## 2 Preliminaries

All relevant theoretical aspects that involves this work will be briefly presented in this section. All notations that will be introduced here will be used for the remainder of this paper.

### 2.1 ILP and Inverse Entailment

Inductive Logic Programming (ILP) is a sub-area of both Machine Learning and Logic Programming, that makes use of logical languages to induce theory-structured hypothesis and by querying them, inference on unknown data can be done. Given a set of labeled examples $E$ and a background knowledge $B$, an ILP system will try to find a hypothesis $H$ that minimizes a specified loss function. More precisely, an ILP task is defined as <E, B, L>, where $E$ is a set of positive and negative literals, called *examples*, $B$ is a logic program called *background theory* and $L$ is a set of logic theories called *language bias*.

The set of all possible hypothesis for a given task, which we will call $S_H$, can be infinite [10]. One of the features that constrains $S_H$ in ILP is the language bias, $L$. It is usually composed by specification predicates, which will define how the search will be done and how far it can go. The most common specification language is called *mode declarations* [11], composed of: modeh predicates, that defines what can appear as a head of a clause; *modeb* predicates, that defines what can appear in the body of a clause; and *determination* predicates, that relates body and head literals. The *modeb* and *modeh* declarations also specifies what is considered an *input variable*, *output variable* and a *constant*. The language bias $L$, through those two predicates and the *determination* predicates, can restrict $S_H$ during hypothesis search to only allow a smaller set of candidate hypothesis $H_c$ to be analyzed. Formally, $H_c$ is a candidate hypothesis for a given ILP task <E, B, L> iff $H_c \in L$, and $B \cup H_c \models E$.

Another feature that is used to restrict hypothesis search space in algorithms based on inverse entailment, such as Progol [11], is the *most specific (saturated) clause*, $\bot_e$. Given an example $e$, Progol will firstly generate a clause that represents $e$ in the most specific way as possible, by searching in $L$ for *modeh* declarations that can unify with $e$ and if it finds one (let the found *modeh* define a predicate $h$), an initial $\bot_e$ is created:

$$\bot_e : h \leftarrow .$$

Then, passing through the *determination* predicates to verify which of the bodies specified in *modeb* can be added to $\bot_e$, more terms will be added to it (let them be named $b_1, \ldots, b_n$), resulting in a most saturated clause of the form

$$\bot_e : h \leftarrow b_1, b_2, \ldots, b_n.$$

Those *modeh* and *modeb* match-ups are done in such way that a *variable chaining* is sustained: each variable that is an input in a body predicate needs to be an input variable in a head predicate or an output variable in a body predicate.

### 2.2 Neural Networks and C-ILP

An neural network is a directed graph with the following structure: a unit (or neuron) in the graph is characterized, at time $t$, by its *input vector* $I_i(t)$, its *input potential* $U_i(t)$, its *activation state* $A_i(t)$, and its *output* $O_i(t)$. The units of the network are interconnected via a set of directed and weighted connections such that if there is a connection from unit $i$ to unit $j$ then $W_{ji} \in \mathbb{R}$ denotes the *weight* of this connection. The input potential of neuron $i$ at time $t$ ($U_i(t)$) is obtained by computing a weighted sum for neuron $i$ such that $U_i(t) = \sum_j W_{ij} I_i(t)$. The activation state $A_i(t)$ of neuron $i$ at time $t$ is then given by the neuron's *activation function* $h_i$ such that $A_i(t) = h_i(U_i(t))$.. In addition, $b_i$ (an extra weight with input always fixed at 1) is known as the *bias* of neuron $i$. We say that neuron $i$ is *active* at time $t$ if $A_i(t) > -b_i$. Finally, the neuron's output value is given by its activation state $A_i(t)$.

For learning from examples, *back-propagation* is the neural learning algorithm most successfully applied in industry [12] - an *error* is calculated as the difference between the network's actual output vector and the target vector, for each input vector in the set of examples. This error $\mathbf{E}$ is then propagated back through the network, and used to calculate the variation of the weights $\triangle\mathbf{W}$. This calculation is such that the weights vary according to the *gradient* of the error, i.e. $\triangle\mathbf{W} = -\eta\nabla\mathbf{E}$, where $0 < \eta < 1$ is called the *learning rate*. The process is repeated a number of times in an attempt to minimize the error, and thus approximate the network's actual output to the target output, for each example. Typically, a *validation set* is used for checking the network's *generalization* ability and avoid *overfitting* [13].

Lastly, there are several learning stopping criteria for back-propagation. Two of them are of interest for this work: *standard training* and *early stopping* [12]. In

*standard training*, the full training dataset is used to minimize an error function, while *early stopping* uses a validation set to measure data overfitting: training stops when the validation set error starts to get higher than previous steps. When this happens, the best validation configuration obtained so far is used as the learned model.

After these definitions, we are in position to introduce C-ILP [1]. It stands for *Connectionist and Inductive Learning and Logic Programming* and is a neural-symbolic system that builds a recursive neural with fixed recurrent weights by using a background knowledge composed of propositional clauses and afterwards, it is able to learn from examples using standard back-propagation, since its recursive connections have fixed weights. After training is finished, inference on unknown data can be done by querying the neural network and finally, knowledge extraction [2] can be used to obtain the new theory that has been built during training. To give an clearer idea of the entire learning cycle of C-ILP, let us analyze an example.

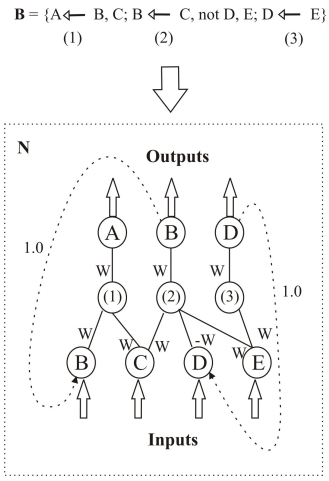


**Fig. 1.** C-ILP example

Starting from an empty network, C-ILP creates a hidden layer neuron for each clause inside $B$ (clause 1, clause 2 and clause 3). Thus, for the example above, C-ILP will have three hidden neurons. For the remaining layers: since each hidden neuron is related to a clause, each of its body literals will be related to an input neuron and the head literal will become an output neuron. Since clause 1 has three body literals, three input neurons will be created to represent them and then they will be connected to the hidden neuron corresponding to its original clause. Since clause 1 has only two bodies, $B$ and $C$, both will be connected to it with a calculated weight $W$ and clause 1's hidden neuron will have a bias $b_{c_1}$ such that it will only activate if both $B$ and $C$ are active. Both $W$

and the biases are functions of a third parameter, $A_{min}$: this parameter controls the activation of all the neurons in C-ILP by only allowing activation if the condition below holds, where $w_i$ is a network weight, $x_i$ is an input, $b$ is the neuron bias and $h_n$ is the activation function of neuron$n$, which is *linear* if it is an input neuron and *semi-linear bipolar*[1]if it is not. For formal definitions of each parameter described above, see [1].

$$h_n(\sum_{\forall i} w_i.x_i + b) \geq A_{min},$$

The same idea goes for the other clauses. Regarding the hidden/output layers, the only change is that the weights and bias will be set up in order to represent an *disjunction*: an output neuron will activate iff at least one of the hidden neurons that are connected to it is activated. Additionally, if any head literal happens to be identical to an literal represented by an input neuron, both are linked with a recursive connection, fixed with weight 1.0. The reason for this is to allow inference in rules that requires iterative deductions (chainings), like $B = \{a \leftarrow b; b \leftarrow c, d; d \leftarrow e\}$.

After building the initial network, standard back-propagation can be applied and then inference can be done with the underlying network. For C-ILP knowledge extraction, see [2].

## 3 C-ILP with First-Order Logic

Since neural-symbolic systems combines two algorithms from connectionism and symbolism, one natural way of improve those systems is to enhance one of its components. With this objective in mind, our work improved the symbolic side of this integration: we extended C-ILP to work with First-Order Logics through Bottom Clauses. Our algorithm is composed of four steps: *Bottom Clause Generation*, *Input Mapping*, *Network Building* and *Bottom Clause Training*. Each one of them will be explained hereafter and from now on, to differentiate the older version from this one, we will call our extension CILP++.

### 3.1 Bottom Clause Generation and Input Mapping

The first step of CILP++, Bottom Clause Generation, consists of transforming each first-order example into bottom clauses. In order to do that, a slightly modified version of Progol's bottom clause generation [14] was made, as shown in Algorithm 3.1. On it, *depth* is a parameter that sets up the *variable depth* of the bottom clause, which controls how deep the algorithm will go to try to find literals to saturate it. Also, $S_e$ is the set examples to be transformed and $S_\perp$ is the set of all generated $\perp_e$ by the algorithm. Note that this algorithm includes negative bottom clause generation as well: in practical terms, the only difference in dealing with negative bottom clauses is its classification value.

[1] Semi-linear bipolar activation function: $f(x) = \frac{2}{1+e^{-\beta.x}} - 1$, where $\beta$ is a slope parameter, usually set to 1.0

**Algorithm 3.1** Adapted Bottom Clause Generation

1: $S_\perp = \emptyset$
2: **for** each example $e$ of $S_e$ **do**
3: $currentDepth = 0$
4: Add $e$ to background knowledge and remove any previously inserted examples
5: $inTerms = \emptyset$, $\perp = \emptyset$
6: Find the first *mode declaration* with head $h$ which $\theta$-subsumes $a$
7: **for all** $v/t \in \theta$ **do**
8: If $v$ is of type #, replace $v$ in $h$ to $t$
9: If $v$ is of one of $\{+, -\}$, replace $v$ in $h$ to $v_k$, where $k = hash(t)$
10: If $v$ is of type $+$, add $t$ to $inTerms$
11: Split $c_i^B$ into a set of body literals $L = \{b_1, \ldots, b_n\}$ and a head literal $head$
12: **end for**
13: Add $h$ to $\perp$
14: **for** each body mode declaration $b$ **do**
15: **for all** substitutions $\theta$ of arguments $+$ of $b$ to elements of $inTerms$ **do**
16: **repeat**
17: **if** querying $b$ with substitution $\theta'$ succeeds **then**
18: **for** each $v/t$ in $\theta$ and $\theta'$ **do**
19: If $v$ is of type #, replace $v$ in $b$ to $t$
20: Else, substitute $v$ in $b$ to $v_k$, where $k = hash(t)$
21: If $v$ is of type $-$, add $t$ to $inTerms$
22: **end for**
23: Add $b$ to $\perp$
24: **end if**
25: **until** *recall* number of iterations has been done
26: **end for**
27: **end for**
28: Increment $currentDepth$; If it is less than $depth$, go back to line 15
29: If $e$ is a negative example, add a default negation symbol "$\sim$" to it
30: add $\perp_e$ to $S_\perp$
31: **end for**
32: **return** $S_\perp$

Continuing the example presented in the Subsecion 1.1, let us add some facts to the background knowledge:

**Background Knowledge Modes:**
:- *modeh(1, motherInLaw(+woman,-man))*
:- *modeb(*, parent(+woman,-woman))*
:- *modeb(1, wife(+woman,-man))*
:- *determination(motherInLaw/2, parent/2)*
:- *determination(motherInLaw/2, wife/2)*

**Background Knowledge Facts:**
man(husband1), man(husband2),
woman(mother1), woman(daughter11), woman(daughter12)

If Algorithm 3.1 is executed with $currentDepth = 1$ for each one of the examples presented in Subsection 1.1, it would output

$$S_{\perp} = \{motherInLaw(A, B) : -parent(A, C), wife(C, B);$$

$$motherInLaw(A, B) : -wife(A, C)\}.$$

After each example has been transformed into bottom clauses, they will be converted to an input that a neural network can process. The rule that we implemented for that was the following: given an example $e$ and its correspondent bottom clause $\perp_e$, initialize an input vector of same size as the number of literals of $\perp_e$ with 0.0 and for each body literal $b_i$ in $\perp_e$, insert 1.0 in the correspondent position.

### 3.2 Network Building

After all bottoms have been preprocessed, the original C-ILP building algorithm will take place to build a starting network, treating each variablized bottom clause literal as a single propositional term. Since C-ILP uses background knowledge to build the network and all we have is bottom clauses, we add a proportion of the example set as background knowledge clauses (we call this subset $S_{\perp}^{BK}$). The pseudo-algorithm for the building is as follows.

For each bottom clause $\perp_e$ of $S_{\perp}^{BK}$, do

1. Add a neuron $h$ in the hidden layer and label it as $\perp_e$;
2. Add input neurons with labels corresponding to each literal atom in the body of $\perp_e$;
3. Connect them to $h$ with a given weight $W$ if it the associated literals are positive, $-W$ otherwise;
4. Add an output neuron $o$ and label it as the head literal of $\perp_e$;
5. Connect $h$ with $o$, with a given weight $W$;
6. If $o$'s label is identical to a label in the input layer, add a recursive connection between them with fixed weight 1.0.

Alternatively, we wanted to see how important is this starting set-up, regarding C-ILP: in the propositional version, it is clearly important because its building algorithm could use it fully, in order to get a well-conditioned network. This is not our case: $\perp_e$ of $S_{\perp}^{BK}$ is just a sample of the examples dataset, it is not specifically a background knowledge. Thus, in order to evaluate that, we experimented a second version of the pseudo-algorithm, where only the input and output layers are built and we set up a specific number of initial hidden layer neurons, but just for convergence purposes: no initial background knowledge will be represented.

### 3.3 Learning From Bottom Clauses

After preparing the bottom clauses and building the network, now the neural network training algorithm will take place. CILP++ uses back-propagation with *validation set* (as explained in subsection 2.2) and optionally, *early stopping*: we use the validation set to measure generalization error during each training epoch. As soon as it starts to go up the training is stopped and the best model in terms of validation error is returned as the optimal. We apply a more "permissive" version of early stopping [15]: instead of finishing the training immediately after validation starts to go up, we stop when the criterion below is satisfied, where $\alpha$ is the stopping criteria parameter, $t$ is the current epoch number, $E_{va}(t)$ is the average validation error on epoch $t$ and $E_{opt}(t)$ is the least validation error obtained on epochs 1 up to $t$. The reason that it was chosen by us for this system is because of the complexity of the generated bottom-clauses: early stopping is really good to avoid overfitting [13] when the system complexity is a lot bigger than the numbers of examples. Since it is not guaranteed to converge, an additional criterion is added: the network will stop training after 300 training epochs.

$$GL(t) > \alpha, GL(t) = 0.1 * \left( \frac{E_{va}(t)}{E_{opt}(t)} - 1 \right)$$

The first action that CILP++ do when it receives an example set is to add all new predicates that may appear as input neurons in the network. After that, a small disturbance is applied to all the weights to avoid the problem of symmetry and all weights and then, they are normalized. Then back-propagation learning algorithm is done and afterwards, the network is ready to infer with unknown data. Firstly, the testing set is used as input for Algorithm 3.1 to generate a set of Bottom Clauses. They are converted to input vectors and then, CILP++ does inference in the same way as C-ILP: a feed-forward iteration is done, then when the outputs are calculated, if there are any recurrent connections that links output neurons with input ones, they are used to start another feed-forward iteration until all neurons that are connected through recurrent connections are having the same activation state.

## 4 Experimental Results

In this section we will show some preliminary results of CILP++ compared to a freely-distributed and well-known ILP system, Aleph. To do so, we have chosen the *Alzheimer* datasets[16], which consists of four benchmarks: *amine*, *choline*, *scopolamine* and *toxic*. We ran CILP++ on them in four different configurations, to make comparisons between different models and evaluate better how our approach for First-Order Logic neural-symbolic integration performs. Those configurations differs with regard to the chosen stopping criteria and building method:

- *st*: standard back-propagation stopping criteria;
- *es*: early stopping (see Subsection 2.2);
- *bk*: the network is built with a subset of 5% of the example set;
- $2h$: hidden layer with only 2 neurons (further details below);

The fourth configuration, $2h$, is explained with details in [12]: neural networks with only two neurons in the hidden layer generalizes binary problems approximately as well as more complex others. Furthermore, if a neural network has too many features to evaluate (if the input layer has too many neurons), it has already enough *degrees of freedom*, thus further increasings on it by adding hidden neurons would just increase the overfitting probability. Additionally, since bottom clauses are just "rough" representations of examples, we do not allow CILP++ to learn noise: we just want it to model the general characteristics of each example and thus, a simpler model would be required.

For Aleph, we used the same configurations as [17]: **minpos** = 2, **minacc** = 0.7, **minscore** = 0.6, **clauselength** = 5 and **noise** = 300. For CILP++, we ran **learning rate** = 0.1, **decay factor** = 0.995 and **momentum** = 0.1 on configuration *st* and **learning rate** = 0.5, **decay factor** = 0.999, **momentum** = 0.1 and $\alpha = 0.01$ on *es*.

### 4.1 Results

We present below two results tables: one with averages over 10-fold of accuracies, with standard deviations, for each *alzheimer* dataset and other, with complete runtime results (since program starting, until finish). In the accuracy table, values in bold are the highest ones obtained and the difference between them and the ones in italic are statistically significative by paired t-test. In the runtime table, only markings for highest values are done. All experiments were ran on a 3.2 Ghz Intel Core i3-2100 with 4 GB RAM.

| **Dataset** | ***Aleph*** | ***CILP++***$_{st,bk}$ | ***CILP++***$_{st,2h}$ | ***CILP++***$_{es,bk}$ | ***CILP++***$_{es,2h}$ |
|---|---|---|---|---|---|
| *amine* | **76.94%**(±3.51) | 76.16%(±2.44) | 75.59%(±4.66) | *67.95%*(±7.46) | *70.26%*(±7.1) |
| *choline* | **69.61%**(±3.6) | *60.09%*(±2.51) | *63.37%*(±4.28) | *65.08%*(±5.08) | 65.47%(±2.43) |
| *scopol.* | **68.57%**(±5.7) | *59.84%*(±6.26) | *58.11%*(±5.27) | *51.38%*(±3.55) | *51.57%*(±5.36) |
| *toxic* | *64.12%*(±4.83) | 79.78%(±4.15) | **79.81%**(±7.49) | 6*7.95%*(±7.46) | 74.48%(±5.62) |

**Table 1.** Accuracy/standard deviation results

As the results shows, most of CILP++ accuracy results (except *scopolamine*) are not far from the ones that Aleph reached. In the other hand, almost all four configurations (except *toxic* and *choline*, configuration (st,bk), which did not converged in any of the 10 folds) of CILP++ were faster, with special emphasis on the processing speed of the $(es, 2h)$ configuration that finished all four experiments at least two times faster than Aleph.

| Dataset | *Aleph* | *CILP++*$_{st,bk}$ | *CILP++*$_{st,2h}$ | *CILP++*$_{es,bk}$ | *CILP++*$_{es,2h}$ |
|---|---|---|---|---|---|
| *amine* | 1:31:05 | 1:18:29 | 1:07:56 | 0:14:24 | **0:10:14** |
| *choline* | 8:06:06 | 33:33:20 | 7:33:17 | 7:53:06 | **0:25:43** |
| *scopol.* | 3:47:55 | 2:24:53 | 1:39:22 | 4:59:23 | **1:33:35** |
| *toxic* | 6:02:05 | 17:38:57 | 0:38:32 | **0:17:24** | 0:28:39 |

**Table 2.** Total runtime results (*h:mm:ss* format)

### 4.2 Discussion

The slightly lower accuracy for CILP++ in general, with regard to Aleph, has two main reasons. One of them is the information loss that we have by just considering predicates that appears in training and ignoring all unknown predicates, that may appear when testing. This motivates us to devise a complete translation algorithm similar to the one that C-ILP uses, which would translate a complete language bias set into an initial network structure. The second one is *overfitting*: due to the large size of bottom clauses, the neural network of CILP++ ends up having hundreds, even thousands, of input neurons, which increases considerably the number of *degrees of freedom* of the model and makes it require an even large number of data to learn properly [13].

With regard to the *Alzheimer* datasets, a varied number of accuracy results using Aleph have been reported in the literature, such as [16, 17, 18]. As a result, we have decided to run our own comparisons between CILP++ and Aleph. We built 10 folds and both systems shared it. Additionally, all parameters that we have chosen for both were the ones that gave us better performance on *amine*. Thus, we do expect sub-optimal performance in the other three datasets.

## 5 Conclusion

In this paper we have introduced a method and algorithm for ILP learning using neural networks, by extending a neural-symbolic system called C-ILP. It managed to obtain expressive runtime performance, compared to Aleph. Our experimental results have shown that it is possible to keep up with ILP systems in terms of accuracy and having higher efficiency.

This approach for learning from relational data is a promising and simple way to apply connectionism into ILP, yet just a little of this potential has been explored. Future contributions to this approach includes: pruning techniques to reduce input dimensionality, noise–robustness analysis, a complete background knowledge and language bias mapping for CILP++, instead of using a sample set of pre-processed examples for network building, and study the feasibility of knowledge extraction, in order to complete its learning cycle [2] in the First-Order level. Current research is going towards implementing the first of those four suggestions.

C-ILP is currently an open-source project being hosted at *sourceforge.net* (*http://sourceforge.net/projects/cil2p*) and soon CILP++ will be available as well. Both are implemented in C++ and are under Apache 2.0 license.

# References


[1] Garcez, A.S.D., Zaverucha, G.: The connectionist inductive learning and logic programming system. Applied Intelligence **11** (1999) 59–77

[2] Garcez, A.S.D., Broda, K., Gabbay, D.M.: Symbolic knowledge extraction from trained neural networks: A sound approach. Artifcial Intelligence **125**(1-2) (2001) 155–207

[3] Muggleton, S.: Inverse entailment and progol (1995)

[4] Muggleton, S.: Inductive logic programming: Inverse resolution and beyond. In: IJCAI (1). (1995) 997

[5] Erdem, E.: Theory And Applications Of Answer Set Programming. PhD thesis (2002) AAI3101204.

[6] Uwents, W., Monfardini, G., Blockeel, H., Gori, M., Scarselli, F.: Neural networks for relational learning: an experimental comparison. Mach. Learn. **82**(3) (March 2011) 315–349

[7] Kijsirikul, B., Lerdlamnaochai, B.K.: First-order logical neural networks. Int. J. Hybrid Intell. Syst. **2** (December 2005) 253–267

[8] Zelezny, F., Lavrac, N.: Propositionalization-based relational subgroup discovery with rsd. Machine Learning **62** (2006) 33–63 10.1007/s10994-006-5834-0.

[9] Kramer, S., Lavrač, N., Flach, P.: Relational data mining. Springer-Verlag New York, Inc., New York, NY, USA (2000) 262–286

[10] Torre, F., Rouveirol, C.: Private properties and natural relations in inductive logic programming (1997)

[11] Russell, S.J., Norvig, P.: Artificial Intelligence - A Modern Approach (3. internat. ed.). Pearson Education (2010)

[12] Haykin, S.: Neural Networks and Learning Machines. Number v. 10 in Neural networks and learning machines. Prentice Hall (2009)

[13] Caruana, R., Lawrence, S., Giles, C.L.: Overfitting in neural nets: Backpropagation, conjugate gradient, and early stopping. In: in Proc. Neural Information Processing Systems Conference. (2000) 402–408

[14] Tamaddoni-Nezhad, A., Muggleton, S.: The lattice structure and refinement operators for the hypothesis space bounded by a bottom clause. Mach. Learn. **76**(1) (July 2009) 37–72

[15] Prechelt, L.: Early stopping - but when? In: Neural Networks: Tricks of the Trade, volume 1524 of LNCS, chapter 2, Springer-Verlag (1997) 55–69

[16] King, R.D., Srinivasan, A., Sternberg, M.J.E.: Relating chemical activity to structure: An examination of ilp successes (1995)

[17] Landwehr, N., Kersting, K., Raedt, L.D.: Integrating naive bayes and foil. J. Mach. Learn. Res. **8** (May 2007) 481–507

[18] Corapi, D.: Nonmonotonic Inductive Logic Programming as Abductive Search. Ph.D. thesis, Imperial College London, London, United Kingdom (2011)