Differentiable Inductive Logic Programming in High-Dimensional Space

Stanisław J. Purgał[2 [0009−0000−1198−9430]], David M. Cerna[1 [0000−0002−6352−603X]], and Cezary Kaliszyk[3 [0000−0002−8273−6059]]

[1] Czech Academy of Sciences Institute of Computer Science, Prague, Czechia
dcerna@cs.cas.cz
[2] University of Innsbruck, Innsbruck, Austria
sjpurgal@gmail.com
[3] University of Melbourne, Melbourne, Australia
cezary.kaliszyk@unimelb.edu.au

Abstract. Synthesizing large logic programs through symbolic Inductive Logic Programming (ILP) typically requires intermediate definitions. However, cluttering the hypothesis space with invented predicates typically degrades performance. In contrast, gradient descent provides an efficient method to find solutions within high-dimensional spaces; a property not fully exploited by neuro-symbolic ILP approaches. We propose extending the *differentiable* ILP framework by large-scale (extending its small-scale) predicate invention to emulate search through a high-dimensional space, and thus allowing us to exploit the efficacy of gradient descent. We show that large-scale predicate invention is beneficial to differentiable inductive synthesis and results in learning capabilities beyond existing neuro-symbolic ILP systems. Furthermore, we achieve these results without specifying the precise structure of the solution within the *inductive bias*.

Keywords: Inductive Logic Programming · Differentiable Logics · Predicate Invention

1 Introduction

Neuro-symbolic ILP is quickly becoming one of the most important research domains in inductive synthesis [5]. Such systems can aid explainability research by providing logical representations of what was learned and providing noise-handling capabilities to symbolic learners. Systems such as δILP can consistently learn solutions for many standard inductive synthesis problems [12,25]. Nonetheless, searching through the hypothesis space remains a challenging task. To deal with this difficulty, inductive learners introduce problem-specific restrictions that reduce the size of the respective search space [26,24]; what is commonly referred to as *language bias*. While this is conducive to solving simple learning tasks, more complex synthesis tasks or tasks where the required language bias is not easily specifiable remain a formidable challenge.

Predicate invention (PI) is a technique that enables the creation of new predicates, thereby adding shortened programs to the hypothesis space and circumventing the limitations imposed by the user-provided background knowledge. However, in purely sym-

bolic inductive synthesis, reliance on large-scale PI is avoided, as it is time- and space-wise too demanding [19], and has only been used effectively in very restricted settings [1].

This paper introduces a novel approach to neuro-symbolic inductive synthesis, compatible with large-scale predicate invention, that leverages the power of the highly influential differentiable ILP [12]. We show that a few changes to the original architecture is enough to train models using a reduced language bias that tests with a significantly higher success rate. Our extension synthesizes a user-provided number of invented predicates during the learning process. Large-scale PI is either intractable for most systems due to memory requirements or results in a performance drop as it clutters the hypothesis space. In contrast, gradient descent methods generally benefit from large search spaces (high dimensionality). Thus, we propose introducing a large number of invented predicates to improve performance. We evaluate the approach on several standard ILP tasks (many derived from [12]), including several which existing neuro-symbolic ILP systems find to be a significant challenge (see *Hypothesis 1*).

Solutions found by our extension of δILP, in contrast to the usual ILP solutions, include large numbers of invented predicates. We posit the usefulness of large-scale PI for synthesizing complex logic programs. While adding many invented predicates can be seen as a duplication of the search space and, therefore, equivalent to multiple initializations of existing neuro-symbolic ILP systems, we demonstrate that our extension of δILP easily outperforms the re-initialization approach on a particularly challenging task (see *Hypothesis 2*). We compare to the most relevant existing approach, δILP (presented in [12]).

Unlike the experiments presented in [12], which specify the solution's precise structure, we assume a generic shape for all predicate definitions. Thus, our experiments force the learner to find both the correct predicates to reference within an invented predicate's definition and the structure of the definition. In this experimental setting, our approach is on par with δILP and, for challenging tasks, outperforms it. In particular, we outperform δILP on tasks deemed difficult in [12] such as *0=X mod 3* and *0=X mod 5*.

Furthermore, we propose an adjusted measure of task difficulty. In [12], the authors proposed the number of *learned* predicate definitions as a measure of learning complexity. While our results do not contradict this assertion, it is more precise to focus on the relation between input variables and body-only variables, i.e., we solve *0=X mod 5* (requires learning four predicate definitions) consistently but perform poorly on the seemingly simpler task $Y = X + 4$ which requires learning two predicates (*Hypothesis 3*). Improving our understanding of task difficulty will aid future investigations.

Our contributions are as follows: **(i)** an extension of δILP capable of large-scale PI (*Hypothesis 1*)[4], **(ii)** experimental verification of improved performance on challenging tasks (*Hypothesis 1*), **(iii)** experimental verification that large-scale PI differs from weight re-initialization (*Hypothesis 2*), **(iv)** proposing a novel complexity criterion and experimentally validating it (*Hypothesis 3*).

[4] Our implementation can be found at the following repository: github.com/Ermine516/DILP2.

2 Related Work

We briefly introduce Inductive logic programming [5], cover aspects of δILP [12] directly relevant to our increase in dimensionality, and compare our approach to related systems inspired by δILP. We assume familiarity with basic logic and logic programming; see [23].

2.1 Inductive Logic Programming (ILP)

ILP is traditionally a form of symbolic machine learning whose goal is to derive explanatory hypotheses from sets of examples (denoted E^+ and E^-) together with background knowledge (denoted BK). Investigations often represent explanatory hypotheses as logic programs [6,7,14,21,22]. A benefit of this approach is that only a few examples are typically needed to learn an explanatory hypothesis [9].

The most common learning paradigm implemented within ILP systems is *learning from entailment* [23]. The systems referenced above, including δILP, use this paradigm, which is succinctly stated as follows: A hypothesis H explains E^+ and E^- through the BK, if

$$\forall e \in E^+, BK \wedge H \models e \quad \text{and} \quad \forall e \in E^-, BK \wedge H \not\models e$$

Essentially, the hypothesis, together with the background knowledge, entails all the positive examples and none of the negative examples. In addition to the learning paradigm, one must consider how to search through the *hypothesis space*, the set of logic programs constructible using definitions from the *BK* together with the predicates provided as examples. Many approaches exploit *subsumption* ($\leq_{sub}$), which has the following property in relation to entailment: $H_1 \leq_{sub} H_2 \Rightarrow H_1 \models H_2$ where H_1 and H_2 are plausible hypotheses. Subsumption provides a measure of specificity between hypotheses and, thus, is used to measure progress. The FOIL [22] approach (*top-down*) iteratively builds logic programs using this principle. *Bottom-up* approaches, i.e., Progol [17], build the subsumptively most specific clause for each positive example and use FOIL to extend more general clauses towards it.

The ILP system *Metagol* [7] implements the meta-learning approach to search. It uses second-order Horn templates to restrict and search the hypothesis space. An example template would be $P(x, y)$:-$Q(x, z), R(z, y)$ where P, Q, and R are variables ranging over predicate symbols. This approach motivates the *template* representation of δILP and our work.

2.2 Differentiable ILP

Given that δILP plays an integral role in our work, we go into some detail concerning the system architecture. We refer the reader to the paper introducing δILP [12] for more details.

The δILP system provides a framework for differentiable *learning from entailment* ILP. Logic programs are represented by vectors whose components encode whether a particular code fragment is likely to be part of the solution. Such programs are "executed" in fuzzy logic, using a weighted average of fuzzy evaluations of code fragments.

The hypothesis space consists of all possible combinations of these code fragments captured by user-provided *templates*, a generalization of *metarules* [7]. In our setting *metarules* take the following form:

Definition 1 (V-Metarule). *Let x, y, z_1, z_2, z_3, z_4 be first-order variables, V a set of first-order variables such that $x, y \in V$ and $z_1, z_2, z_3, z_4 \notin V$, and P, Q, and R second-order variables ranging over predicate symbols. Then $P(x, y)$:-$Q(z_1, z_2), R(z_3, z_4)$ is a V-metarule.*

Unlike standard metarules, V-Metarules can have various instances.

Definition 2 (V-Metarule Instance). *Let M be a V-Metarule of the form $P(x, y)$:-$Q_1(x_1, x_2), R_1(x_3, x_4)$. Then $P(x, y)$:- $Q_1(y_1, y_2), R_1(y_3, y_4)$ is an Instance of M if $y_1, y_2, y_3, y_4 \in V$.*

Essentially, V-metarules are a generalization of metarules that allows for different variable configurations. A (V, p)-template is defined using V-Metarules as follows:

Definition 3 ((V, p)-template). *Let p be a predicate symbol, $M = P(x, y)$:- $Q_1(x_1, x_2), R_1(x_3, x_4)$ and $M' = P'(x, y)$:- $Q_2(x_1, x_2), R_2(x_3, x_4)$ be V-metarules. Then the (V, p)-template constructed from M and M' is $(M\{P \mapsto p\}, M'\{P' \mapsto p\})$.*

An instance of a (V, p)-template (M_1, M_2) is the pair (M_1', M_2') where M_1' and M_2' are V-Metarule Instances of M_1 and M_2.

An instantiation of a (V, p)-template (M_1, M_2) is $(M_1'\sigma, M_2'\sigma)$ where (M_1', M_2') is an instance of (M_1, M_2) and σ maps the second-order variables to predicate symbols.

Observe that we use templates consisting of two metarules as this is the minimal size, which allows for recursion and disjunction in the predicate definition.

In [12], additional restrictions were put on the instantiations of the second-order variables to simplify the hypothesis space for harder tasks. The authors designed the templates to allow precise descriptions of the solution structure, thus simplifying the search. We took a more general approach, which allowed us to define templates uniformly. This results in a larger hypothesis space and, thus, a more challenging experimental setting.

Example 1. Consider a $(\{x, y, z\}, p)$-template. Possible instances of the contained $(\{x, y, z\})$-metarules include

$$p(x, y)\text{:-}Q_1(x, y), R_1(x, y) \qquad p(x, y)\text{:-}Q_2(x, z), R_2(x, z)$$

A program fitting this template would be the following:

$$p(x, y) : \text{-}succ(x, y) \qquad p(x, y) : \text{-}succ(x, z), p(x, z)$$

where R_2 maps to p and Q_1, R_1, and Q_2 map to $succ$.

Templates are generalizable to higher-arity predicates; learning such predicates is theoretically challenging for this ILP setting [18]. It is common to restrict learning to dyadic predicates. Reducing template complexity is important when introducing many templates (up to 150). From now on, by template, we mean (V, p)-template.

As input, δILP requires a set of templates T (using pairwise distinct symbols, $p_1, \cdots, p_n$) and *BK*. From the input, it derives a satisfiability problem where each disjunctive clause $C_{i,j}$ denotes the range of possible choices for clause j given template $t \in T$, i.e., overall instantiations of t. The logical models satisfying this formula denote logic programs modulo the clauses derivable using the template instantiated by the *BK* and the symbols $p_1, \cdots, p_n$. Switching from a discrete semantics over $\{0, 1\}$ to a continuous semantics allows the use of differentiable logical operators when implementing differentiable deduction. Solving ILP tasks, in this setting, is reduced to minimizing loss through gradient descent.

δILP uses E^+ and E^- as training data for a binary classifier to learn a model attributing *true* or *false* to ground instances of predicates. This model implements the conditional probability $p(\lambda|\alpha, W, T, L, BK)$, where $\lambda \in \{$*true*,*false*$\}$, α is a ground instance, W a set of weights, T the templates, and L the symbolic language used to describe the problem containing a finite set of atoms.

Each $(\{x, y, z\}, p_i)$-template $(t_1, t_2) \in T$ is associated with a weight matrix whose shape is $d_1 \times d_2$ where d_j denotes the number of clauses constructible using the BK and L modulo the constraints of t_j. The number of weights may be roughly approximated (*quintic*) in terms of the number of templates (considering possible instances and the four second-order variables to instantiate). The weights denote δILP's confidence in an instantiation of a template being part of the solution; so-called *per template* assignment. We provide a detailed discussion of weight assignment in Section 3.

δILP implements differentiable inferencing by providing each clause c with a function $f_c : [0, 1]^m \to [0, 1]^m$ whose domain and range are valuations of grounded instantiations of templates. Note, m is not the number of templates; rather, it is the number of groundings of each template, a much larger number dependent on the *BK*, language bias, and the *atoms* of the symbolic language L. Consider a template (t_1, t_2) admitting the clause pair (c_1, c_2), and let the current valuation be $\mathcal{EV}_i$ and $g : [0, 1] \times [0, 1] \to [0, 1]$ a function computing $\vee$-*clausal* (disjunction between clauses). Assuming we have a definition of f_c, then $g(f_{c_1}(\mathcal{EV}_i), f_{c_2}(\mathcal{EV}_i))$ denotes one step of *forwards-chaining*. Computing the weighted average over all clausal combinations admitted by (t_1, t_2), using the *softmax* of the weights, and finally performing $\vee$-*step* (disjunction between inference steps) between their sums, in addition to $\mathcal{EV}_i$, results in $\mathcal{EV}_{i+1}$. This process is repeated n times (the number of forward-chaining steps), where $\mathcal{EV}_0$ is derived from the BK.

The above construction still depends on a precise definition of f_c. Let

$$c_g = p(x, y) \text{:-} Q_1(y_1, y_2), Q_2(y_3, y_4)$$

where $y_1, y_2, y_3, y_4 \in \{x, y, z\}$. We want to collect all ground predicates p_g for which a substitution θ into $Q_1, Q_2, y_1, y_2, y_3, y_4$ exists s.t. $p_g \in \{Q_1(y_1, y_2)\theta, Q_2(y_3, y_4)\theta\}$. These ground predicates are then paired with the appropriate grounding of the left-hand side of c_g. The result of this process can be reshaped into a tensor emphasizing which pairs of ground predicates derive various instantiations of $p(x_1, x_2)$. In the case of body-only variables, there is one pair per atom in the language. Pairing this tensor with some valuation $\mathcal{EV}_i$ allows one to compute $\wedge$-*literal* (conjunction between literals of a clause) between predicate pairs. As a final step, we compute $\vee$-*exists* (disjunction

between variants of literals with body-only variables) between the variants and thus complete computation of the tensor required for a step of *forward-chaining*.

Four operations parameterize the above process for conjunction and disjunction. We leave discussion to section 4.

2.3 Related Approaches

To the best of our knowledge, three recent investigations are related to δILP and build on the architecture. The *Logical Neural Network* (LNN) [24] uses a similar templating, but only to learn non-recursive *chain rules*, i.e. of the form $p_0(X, Y)$:-$p_1(X, Z_1)$, $\cdots, p_n(Z_n, Y)$; this is simulatable using δILP templates, especially for the short rules presented by the authors. They introduced particular parameterized, differentiable, logical operators that are optimizable for ILP.

Another system motivated by δILP is αilp [25]. The authors focused on learning logic programs that recognize visual scenes rather than general ILP tasks. The authors restricted the *BK* to predicates explaining aspects of the visual scenes used for evaluation. To build clauses, the authors start from a set of initial clauses and use a *top-k beam search* to iteratively extend the body of the clauses based on evaluation with respect to E^+. In this setting, predicate invention and recursive definitions are not considered. Additionally, learning a relation between two variables is not considered.

Feed-Forward Neural-Symbolic Learner [8] does not directly build on the δILP architecture but provides an alternative approach to one of the problems the δILP investigation addressed, namely developing a neural-symbolic architecture that can provide symbolic rules when presented with noisy input. In this work [8], rather than softening implication and working with fuzzy versions of logical operators, compose the Inductive *answer set programming* learners *ILASP* [14] and *FASTLAS* [13] with various pre-trained neural architectures for classification tasks. This data is then transformed into a weighted knowledge for the symbolic learner. While this work is to some extent relevant to the investigation outlined in this paper, we focus on improving the differentiable implication mechanism developed by the authors of δILP rather than completely replacing it. Furthermore, the authors focus on tasks with simpler logical structures, similar to the approach taken by αilp.

Earlier investigations, such as NeuraILP [29], experimented with various *T-norms*, and part of the investigation reported in [12] studied their influence on learning. The authors leave scaling their approach to larger, possibly recursive programs as future work, a limitation addressed herein.

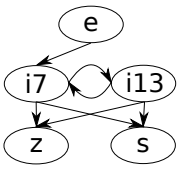
In [26], the authors built upon δILP but further restricted templating to add a simple term language (at most term depth 1). Thus, even under the severe restriction of at most one body literal per clause, they can learn predicates for list *append* and *delete*. Nonetheless, scalability remains an issue. *Neural Logical Machines* [11], in a limited sense, addressed the scalability issue. The authors modeled propositional formulas using multi-layer perceptrons wired together to form a circuit. This circuit is then trained on many (10000s) instances of a particular ILP task. While the trained model was accurate, interpretability is an issue, as it is unclear how to extract a symbolic expression. Our approach provides logic programs as output, similar to δILP.

Some related systems loosely related to our work are *Logical Tensor Networks* [10], *Lifted Relational Neural Networks* [27], *Neural theorem prover* [16], and *DeepProbLog* [15]. While some, such as Neural theorem prover, can learn rules, it also suffers from scaling issues. Overall, these systems were not designed to address learning in an ILP setting. Concerning the explainability aspects of systems similar to δILP, one notable mention is *Logic Explained Networks* [4], which adapts the input format of a neural learner to derive explanations from the output. However, the problem they tackle is only loosely connected to our work.

3 Contributions

The number of weights used by δILP is approximately *quintic* in the number of templates used, thus incurring a significant memory footprint (See Section 2.2). This issue is further exacerbated as computing evaluations requires grounding the hypothesis space. As a result, experiments performed in [12] use task-specific templates precisely defining the structure of the solution; this is evident in their experiments as certain tasks, i.e., the *even* predicate (See Figure 1), list multiple results, with different templating. These restrictions result in only a few of the many possible solutions being present in the search space. Furthermore, this *language bias* greatly influences the success rate of δILP on tasks such as *length*; the authors report low loss in **92.5**% of all runs, which significantly differs from our reduced *language bias* experiments, i.e. **0**% correct solutions and **0**% achieving low loss.

```
e(A,B):-i7(B,B),i7(B,B)
e(A,B):-i7(B,A),i7(A,B)
i7(A,B):-z(C,B),z(C,B)
i7(A,B):-i13(A,C),s(C,B)
i13(A,B):-s(C,B),i7(C,C)
i13(A,B):-z(A,B),z(C,A)
```




```
e(A,B):-i2(C,B),i2(C,B)
e(A,B):-i18(B,C),i2(C,A)
i2(A,B):-i2(C,C),i18(B,C)
i2(A,B):-z(A,B),z(A,B)
i18(A,B):-z(C,A),z(A,A)
i18(A,B):-s(B,C),s(C,A)
```

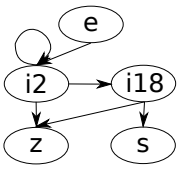


Fig. 1: *even* (e above) solutions trimmed to used templates. Note, s denotes successor, and z denotes zero.

To achieve low loss, in addition to the chosen *language bias*, δILP's authors assign weights *per template* resulting in a large vector v of learnable parameters, see Section 4. This seems to imply high dimensionality; however, intuitively, *softmax* is applied to v

during differentiable inferencing and thus transforms v into a distribution, effectively reducing its dimensionality.

Our investigation aims to: **(i)** increase the dimensionality of the search space while maintaining the efficacy of the differentiable inferencing, and **(ii)** minimize the *bias* required for effective learning. We proceed by adding many ({x,y,z},p)-templates (each with a unique symbol p) as discussed in subsection 2.2; this largely reduces the biases towards solutions of a particular shape. Nonetheless, given the significant number of weights required, large-scale PI remains highly intractable. Thus, we amend how weights are assigned to templates (See Section 4).

Assigning weights *per template* is the main source of the significant memory footprint. The authors discuss this design choice in Appendix F of [12]. In this Appendix, the authors describe assignment *per clause*, i.e., the weight denotes the likelihood of a given instantiation of a metarule instance occurring within an instantiation of a template. This approach was abandoned as it was "incapable of escaping local minima on harder tasks". Assigning weights *per clause* results in roughly a quadratic reduction in the number of weights.

Our system (δILP$_2$) assigns weights *per literal*, i.e., the weight denotes the likelihood of a given literal occurring within an instantiation of a template. Assignment *per literal* results in another roughly quadratic reduction in the number of assigned weights.

We observed that the most challenging tasks require learning a binary relation whose solution requires using a third (non-argument) variable, an additional body-only variable. This observation differs from the observations presented in [12] where complexity was measured purely in terms of the number of *learned predicate definitions* required. For example, consider *0=X mod 5*-hard and *Y=X+4*; our approach solves the former 61% of the time and the latter 4% of the time. Note, *0=X mod 5*-hard requires four learned predicate definitions while *Y=X+4* requires two, yet unlike *0=X mod 5*-hard, both predicates relate two variables through a third non-argument variable.

We test δILP$_2$ and support our observation through experimentally testing the following hypotheses (see Section 5):

- Hypothesis 1: Differentiable ILP benefits from increasing the number of templates used during training.
- Hypothesis 2: The benefit suggested by *hypothesis 1* is not solely due to the relationship between increasing the number of templates and training a multitude of times with a task-specific number of templates.
- Hypothesis 3: Learning binary predicates using body-only variables remains a challenge regardless of the weight assignment approach.

4 Methodology

We now outline the methodological differences between our implementations of differentiable inferencing and δILP; We **(i)** assign weights *per literal*, **(ii)** use slightly different logical operators, **(iii)** use more precise measures of training outcomes, and **(iv)** use a slightly different method of batching examples.

Weight assignment To exploit the benefits of large-scale PI, we need to reduce the large memory footprint incurred by δILP's weight assignment, *per template* assignment. We distinguish three types of weight assignment:

- *per template*: weight encodes the likelihood that a pair of clauses is the correct choice for the given template.
- *per clause*: weight encodes the likelihood that a clause occurs in the correct choice of clauses for the given template.
- *per literal*: weight encodes the likelihood a literal occurs in one of the correct clauses for the given template.

Per literal assignment is the coarsest of the three but also has the least memory footprint, thus allowing for large-scale PI. Our system, δILP_2, implements *per Literal* assignment.

T-norm for fuzzy logic Differentiable inferencing (Section 2) requires four differentiable logic operators. The choice of these operators greatly impacts overall performance. The Author's of δILP experimented with various *t-norms*, continuous versions of *classical* conjunction [2], from which continuous versions of other logical operators are derived. The standard t-norms are *max* ($x \wedge y \equiv \max\{x, y\}$), *product* ($x \wedge y \equiv x \cdot y$) and Łukasiewicz ($x \wedge y \equiv \max\{x + y - 1, 0\}$). For simplicity, we refer to all operators derived from a t-norm by **the conjunctive operator**, i.e., $x \vee y \equiv \min\{x, y\}$ is referred to as *max* when discussing the chosen t-norm.

	δILP	δILP_2
$\wedge$-Literal	product	product
$\vee$-Exists	max	max
$\vee$-Clausal	max	max
$\vee$-Step	product	max

When computing many inference steps, *product* produces vanishingly small gradients. Large programs require more inferencing, see Figure 3, thus, we use *max* for $\vee$-step.

Batch probability We require computing values for all predicates over all combinations of atoms, thus motivating an alternative approach to typical *mini-batching*. Instead of parameterization by *batch size*, we use a *batch probability* – the likelihood of an example contributing to gradient computation. When computing the loss, the example sets E^+ and E^- equally contribute. Regardless of the chosen examples, the loss is balanced (divided by the number of examples contributing). If batching results in no examples from E^+ (E^-), we set that half of the loss to 0 (with 0 gradient). Performance degrades when the *batch probability* is near 0.0 or 1.0. In our experiments, we used 0.5.

4.1 Considered outcomes

The experiments outlined in section 5 (See Table 1A & 1B) allow for five possible outcomes: *Correct on Test (**C**)*, *Fuzzily Correct on Test (**F**)*, *Correct on Training (**CT**)*, *Fuzzily Correct on Training (**FT**)*, and **FAIL**. We differentiate between test and training to cover the possibility of **overfitting** and differentiate between correct and fuzzy to cover the possibility of learning programs only correct using fuzzy logic.

Overfitting δILP avoids overfitting as the search space is restricted enough to exclude overfitting programs. However, this is no longer the case when 100s of templates are used. For example, when learning *even*, it is possible, when training with enough invented predicates, to remember all even numbers provided in E^+. Thus, we add a validation step to test our solutions on unseen data (i.e., numbers up to 20 after training on numbers up to 10). Given the types of tasks we evaluated and the structure of the resulting model, a relatively large number of unseen examples is enough to validate. Learning over-fitting solutions for large, unseen input is highly unlikely as the programs would be very large. During experimentation, we observed that δILP_2 rarely overfits, even when it clearly could. A plausible explanation is that shorter, precise solutions have a higher frequency in the search space.

Fuzzy solutions Another class of solutions observed in both our and earlier experimental designs is *fuzzy solutions*; that is, programs that made correct predictions using fuzzy logic but incorrect predictions when evaluated using classical logic (selected predicates with the highest weight). Typically, fuzzy solutions are worse at generalizing – they are correct when tested using the training parameters (for example, inference steps) and break on unseen input. Entirely correct solutions for *even* are translatable into a program correct for all numbers, while a fuzzy solution fails to generalize beyond training.

5 Experiments

We compare δILP_2 (*per Literal*) to δILP (*per Template*) on tasks presented in [12][5] plus additional tasks to experimentally test *Hypothesis 2 & 3*. The tasks are separated into four domains: *numeric*, *list*, *ancestors*, and *graphs*. Results are shown in Table 1A & 1B. Tasks annotated by *easy* contain extra background knowledge, simplifying the learning process, while *hard* versions do not use the extra background knowledge. Concerning *experimental parameters*, we ran δILP_2 (*Per Literal* assignment) using 150 templates to produce Table 1. For δILP (*Per Template* assignment) [12], we ran it with the precise number of templates needed to solve the task. Using more templates was infeasible for many tasks due to the large memory footprint of *per template* assignment. In both cases, we used $(\{x, y, z\}, p_i)$-templates with pairwise distinct p_i.

Other parameters are as follows: $2k$ gradient descent steps, early finish when loss reaches 10^{-3}, differentiable inference is performed for 25 steps, batch probability of 0.5, weights are initialized using a normal distribution, output programs are derived by selecting the highest weighted literals for each template.

We ran the experiments producing Figure 2 on a computational cluster with 16 nodes, each with 4 GeForce RTX 2070 (with 8 GB of RAM) GPUs. We ran the experiments producing Table 1A & 1B on a GPU server with 8 NVIDIA A40 GPUs (46GB each). We implemented both δILP and δILP_2 using PyTorch [20] (version 2.0). Our implementation can be found at the following repository: github.com/Ermine516/DILP2.

[5] Section 5 and Appendix G of [12].

Task	C	F	CT	FT	diff C	diff CT
predecessor/**2**	100	100	100	100	**+2%** †	**+2%** †
even/**1**	92	99	92	99	**+32%**	**+32%**
$(X \leq Y)$/**2***	30	31	35	38	-70%	-65%
(0=X mod 3)/**1**	91	97	91	97	**+91%**	**+91%**
(0=X mod 5)/**1**-easy	77	80	97	100	**+77%**	**+97%**
(0=X mod 5)/**1**-hard	61	65	61	65	**+61%**	**+65%**
(Y=X+2)/**2**	99	100	99	100	-1% †	-1% †
(Y=X+4)/**2**	4	12	5	13	**+4%** †	**+5%** †
member/**2***	17	19	37	43	-67%	-67%
length/**2**	25	26	31	38	**+25%**	**+31%**
grandparent/**2**	38	38	92	94	**+38%**	**+89%**
undirected_edge/**2**	94	94	100	100	**+75%**	**+81%**
adjacent_to_red/**1**	94	99	94	99	**+48%**	**+48%**
two_children/**1**	74	100	74	100	**+13%**	**+13%**
graph_colouring/**1**	83	85	96	100	-15%	-2% †
connectedness/**2***	40	41	98	99	**+16%**	**+74%**
cyclic/**1**	19	19	90	100	**+18%**	**+89%**

(A)

Task	C	F	CT	FT
predecessor/**2**	98	100	98	100
even/**1**	70	94	70	94
$(X \leq Y)$/**2***	100	100	100	100
(0=X mod 3)/**1**	0	0	0	0
(0=X mod 5)/**1**-easy	-	-	-	-
(0=X mod 5)/**2**-hard	-	-	-	-
(Y=X+2)/**2**	100	100	100	100
(Y=X+4)/**2**	0	0	0	0
member/**2***	84	100	84	100
length/**2**	0	0	0	0
grandparent/**2**	0	0	3	3
undirected_edge/**2**	19	100	19	100
adjacent_to_red/**1**	46	90	46	90
two_children/**1**	61	100	61	100
graph_colouring/**1**	98	100	98	100
connectedness/**2***	24	100	24	100
cyclic/**1**	0	0	1	1

(B)

Table 1: **(A)** *Per Literal* results: difference computed with respect to Table 1 (B). Significance computed using t-test and $p < e^{-4}$. We use † to denote differences that are not significant. Tasks are denoted *name/arity*, i.e. task *even* has arity 1 and *length* has arity 2. **(B)** *Per Template* result. Due to significant memory requirements, neither *(0=X mod 5)* task fits in GPU memory (46GB). Problems annotated with * are easy for *Per template* as the hypothesis space fits in one weight matrix.

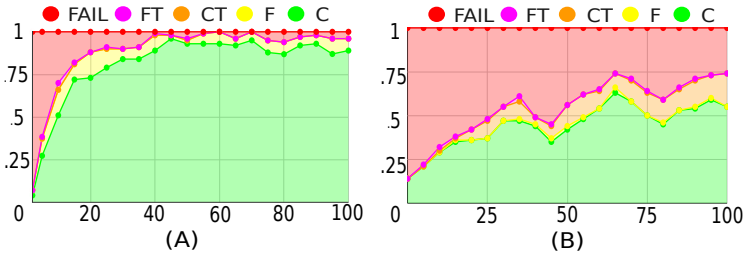


Fig. 2: Learning $0 = X mod 3$ **(A)** and $X \leq Y$ **(B)** varying the number of templates (X-axis). The Y-axis is the proportion of solutions in each category. All considered tasks show this pattern.

5.1 Hypothesis 1

Figure 2 illustrates the proportion of runs within each of our five categories (**C**, **CT**, **F**, **FT**, **FAIL**). As the number of templates increases, the proportion of the runs categorized as correct and generalizing increases. This pattern emerges even for tasks that remain hard to learn. Figure 2 clearly provides strong evidence supporting **Hypothesis 1**.

When comparing with δILP, out of the 17 tasks we tested δILP and δILP_2 on, δILP_2 showed improved performance on 13 tasks, and the improved performance was statistically significant for 12 of these tasks. Of the four remaining tasks, δILP showed a statistically significant performance difference on two, namely $X \leq Y$ and *member*. Both benefit from *per template* assignment as the entire search space fits into one weight matrix. Thus, δILP is essentially performing brute force search. While one would expect the same issue to occur for *connectedness*, there are fewer solutions to *member* and $X \leq Y$ in the search space than in the case of *connectedness*; it is a more general concept. Thus, even when δILP has an advantage, δILP_2 outperforms it when training on more complex learning tasks.

Notably, δILP_2 outperforms δILP on many challenging tasks. For example, *0=X mod 5* cannot be solved by δILP in our experimental setting. In [12], low loss was attained only 14% of the time when $+2$ and $+3$ are in the *BK* [12]. In contrast, we achieved a 61% success rate on this task even when the *BK* contained only *successor* and *zero*; for the dependency graphs of a solution learned, see Figure 3.

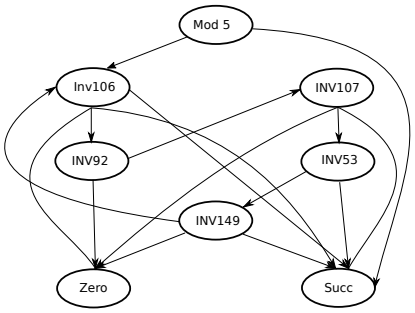
5.2 Hypothesis 2

To illustrate that large-scale predicate invention is not equivalent to re-initialization of weights, we ran δILP_2 on the *0=X mod 6* while varying the numbers of templates used during training (results shown in Figure 3**B**), i.e. improved performance is not the result of randomly initializing a small number of templates many times. Note, *0=X mod 6* is slightly more challenging than *0=X mod 5* and thus aids in illustrating the effect of larger-scale PI. According to Figure 3**B**, when training with three templates (minimum required), we would need to run δILP_2 5000 times to achieve a similar probability of success (**F** solution) as a single 50 predicate run.

When using five invented predicates, which is minimum required to avoid constructing binary predicates (see Hypothesis 3), we would need to run δILP_2 750 times to achieve a similar probability of success (finding a fuzzily correct solution) as a single 50 predicate run. These results do not necessarily imply that using more predicates is always beneficial over doing multiple runs; however, they show that repeated training with intermediary weight re-initialization is not a sufficient explanation of the observed benefits of large-scale PI.

5.3 Hypothesis 3

In Table 1**A** & 1**B**, one can observe that some tasks that require learning a relatively simple program (i.e., *length*) are more challenging than tasks such as *0=X mod 6* that require learning a much larger program.



task	**C**	**F**	**CT**	**FT**
3 templates	0%	0.02%	0%	0.02%
5 templates	0.11%	0.13%	0.11%	0.14%
50 templates	60%	65%	60%	65%

Fig. 3: **(A)** Template dependency graph of a correct program learned by δILP_2 for the (0= X mod 5)-hard task. **(B)** δILP_2 learning *0 = X mod 6*. Ran $10k$ times for 3 and 5 templates and 100 for 50.

As stated above, we hypothesize this is due to propagating the gradient through an existential quantification. This results in difficulties when learning predicates that relate two input variables through a body-only variable. The difficulty increases with the number of such predicates required to solve the task.

We introduced a task explicitly designed to test this hypothesis: ($Y = X + 4$). This task requires only one more predicate than $Y = X + 2$, yet the success rate drops significantly with respect to $Y = X + 2$ (from 99% to 4%). For *0=X mod 2* (*even*) and *0=X mod 5*, the change is gradual (from 92% to 61%). Thus, the number of relational predicate definitions that a given task requires learning is a more precise measure of complexity than the number of learned predicate definitions.

6 Conclusion & Future Work

The main contribution of this work (*Hypothesis* 1) is strong evidence that additional templating (beyond what is necessary) improves performance. Verification of this hypothesis used δILP_2, our modified version of δILP. We performed our experiments using reduced language bias compared to the experiments presented in [12]. Furthermore, we used the same generic template for all predicate definitions learned by the system. This choice makes some tasks significantly more difficult. Additionally, we verified that the performance gains were not simply due to properties shared with weight re-initialization when using a task-specific number of templates during learning (*Hypothesis* 2).

During experimentation, we noticed that the difficulty of the task did not correlate well with the number of learned predicate definitions needed to solve it but rather with the arity and the necessity of a body-only variable. Therefore, we tested this conjecture using the tasks $Y = X + 2$ and $Y = X + 4$. While both systems solve $Y = X + 2$, performance drastically drops for $Y = X+4$, which only requires learning two invented predicates. Note, *0=X mod 5* requires learning four invented predicates and is easily solved by δILP_2. This observation highlights the challenging tasks for such synthesis approaches (*Hypothesis* 3) and suggests a direction for future investigation.

As a continuation of our investigation, we plan to integrate ILP with Deep Neural Networks as a hybrid system that is trainable end-to-end through backpropagation. The Authors of δILP presented the first steps in [12]. One can imagine the development of a network inferring a discrete set of objects in an image [3], or integration with Transformer-based [28] language models that produce atoms δILP_2 can process. This research direction can lead to a network that responds to natural language queries based on a datalog database. Also, as part of planned investigations, we consider an ablation study to show that the improved performance is indeed due to large-scale predicate invention and a scalability analysis to test the limits of the approach.

Acknowledgments. Supported by the ERC starting grant no. 714034 SMART, Czech Science Foundation Grant No. 22-06414L, Math$_{LP}$ project (LIT-2019-7-YOU-213) of the Linz Institute of Technology and the state of Upper Austria, Cost action CA20111 EuroProofNet, and the Doctoral Stipend of the University of Innsbruck. We would also like to Thank David Coufal (CAS ICS) for setting up and providing access to the institute's GPU Server.

Disclosure of Interests. The authors have no competing interests.

References

1. Supplementary material from "hypothesizing an algorithm from one example: the role of specificity"
2. Monoidal t-norm based logic:towards a logic for left-continuous t-norms. Fuzzy Sets and Systems **124**(3), 271–288 (2001), fuzzy Logic
3. Carion, N., Massa, F., Synnaeve, G., Usunier, N., Kirillov, A., Zagoruyko, S.: End-to-end object detection with transformers. CoRR **abs/2005.12872** (2020)
4. Ciravegna, G., Barbiero, P., Giannini, F., Gori, M., Liò, P., Maggini, M., Melacci, S.: Logic explained networks. Artif. Intell. **314**, 103822 (2023)
5. Cropper, A., Dumancic, S.: Inductive logic programming at 30: A new introduction. J. Artif. Intell. Res. **74**, 765–850 (2022)
6. Cropper, A., Morel, R.: Learning programs by learning from failures. Mach. Learn. **110**(4), 801–856 (2021)
7. Cropper, A., Muggleton, S.: Metagol system (2016), `https://github.com/metagol/metagol`
8. Cunnington, D., Law, M., Lobo, J., Russo, A.: FFNSL: feed-forward neural-symbolic learner. Mach. Learn. **112**(2), 515–569 (2023)
9. Dai, W., Muggleton, S.H., Wen, J., Tamaddoni-Nezhad, A., Zhou, Z.: Logical vision: One-shot meta-interpretive learning from real images. In: Lachiche, N., Vrain, C. (eds.) ILP 2017. LNCS, vol. 10759, pp. 46–62. Springer (2017)
10. Donadello, I., Serafini, L., d'Avila Garcez, A.S.: Logic tensor networks for semantic image interpretation. In: Sierra, C. (ed.) Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI 2017. pp. 1596–1602. ijcai.org (2017)
11. Dong, H., Mao, J., Lin, T., Wang, C., Li, L., Zhou, D.: Neural logic machines. In: 7th International Conference on Learning Representations, ICLR 2019. OpenReview.net (2019)
12. Evans, R., Grefenstette, E.: Learning explanatory rules from noisy data. Journal of Artificial Intelligence Research **61**, 1–64 (2018)
13. Law, M., Russo, A., Bertino, E., Broda, K., Lobo, J.: Fastlas: Scalable inductive logic programming incorporating domain-specific optimisation criteria. In: The Thirty-Fourth AAAI. pp. 2877–2885. AAAI Press (2020)

14. Law, M., Russo, A., Broda, K.: Inductive learning of answer set programs. In: Proceedings of Logics in Artificial Intelligence. pp. 311–325. Springer (August 2014)
15. Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., Raedt, L.D.: Neural probabilistic logic programming in deepproblog. Artif. Intell. **298**, 103504 (2021)
16. Minervini, P., Bosnjak, M., Rocktäschel, T., Riedel, S., Grefenstette, E.: Differentiable reasoning on large knowledge bases and natural language. In: The Thirty-Fourth AAAI. pp. 5182–5190. AAAI Press (2020)
17. Muggleton, S.: Inverse entailment and progol. New Generation Computing **13**(3&4), 245–286 (December 1995)
18. Muggleton, S.H., Lin, D., Tamaddoni-Nezhad, A.: Meta-interpretive learning of higher-order dyadic datalog: predicate invention revisited. Mach. Learn. **100**(1), 49–73 (2015)
19. Muggleton, S.H., Raedt, L.D., Poole, D., Bratko, I., Flach, P.A., Inoue, K., Srinivasan, A.: ILP turns 20 - biography and future challenges. Mach. Learn. **86**(1), 3–23 (2012)
20. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In: Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., Garnett, R. (eds.) Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019)
21. Purgal, S.J., Cerna, D.M., Kaliszyk, C.: Learning higher-order logic programs from failures. In: Raedt, L.D. (ed.) Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022. pp. 2726–2733. ijcai.org (2022)
22. Quinlan, J.R.: Learning logical definitions from relations. Mach. Learn. **5**, 239–266 (1990)
23. Raedt, L.D.: Logical and relational learning. Cognitive Technologies, Springer (2008)
24. Sen, P., de Carvalho, B.W.S.R., Riegel, R., Gray, A.G.: Neuro-symbolic inductive logic programming with logical neural networks. In: Thirty-Sixth AAAI. pp. 8212–8219. AAAI Press (2022)
25. Shindo, H., Pfanschilling, V., Dhami, D.: αilp: thinking visual scenes as differentiable logic programs. Machine Learning (2023)
26. Shindo, H., Nishino, M., Yamamoto, A.: Differentiable inductive logic programming for structured examples. In: Thirty-Fifth AAAI. pp. 5034–5041. AAAI Press (2021)
27. Sourek, G., Svatos, M., Zelezný, F., Schockaert, S., Kuzelka, O.: Stacked structure learning for lifted relational neural networks. In: Lachiche, N., Vrain, C. (eds.) Inductive Logic Programming - 27th International Conference, ILP 2017, Revised Selected Papers. LNCS, vol. 10759, pp. 140–151. Springer (2017)
28. Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A.N., Kaiser, L., Polosukhin, I.: Attention is all you need. CoRR **abs/1706.03762** (2017)
29. Yang, F., Yang, Z., Cohen, W.W.: Differentiable learning of logical rules for knowledge base reasoning. In: Guyon, I., von Luxburg, U., Bengio, S., Wallach, H.M., Fergus, R., Vishwanathan, S.V.N., Garnett, R. (eds.) Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017. pp. 2319–2328 (2017)