Learning Rules with Stratified Negation in Differentiable ILP

Giri P Krishnan*
Department of Medicine
University of California, San Diego
San Diego, CA
gkrishnan@ucsd.edu

Frederick Maier*
Institute for Artificial Intelligence
The University of Georgia
Athens, GA 30602
fmaier@uga.edu

Ramyaa Ramyaa*
Department of Computer Science
New Mexico Tech
Socorro, NM 87801
ramyaa.ramyaa@nmt.edu

*Equal Contribution

Abstract

Differentiable methods to learn first order rules (logic programs) have the potential to integrate the interpretability, transferability and low data requirements of inductive logic programming with the noise tolerance of non-symbolic learning. Negation is an essential component of reasoning, but incorporating it into logic programming frameworks poses several problems (hence its central place in the logic programming and nonmonotonic reasoning communities). Current implementations of differentiable rule learners do not learn rules with negations. Here, we introduce stratified negation into a differentiable inductive logic programming framework, and we demonstrate that the resulting system can learn recursive programs with inventive predicates in which negation plays a central role. We include examples from multiple domains, e.g., arithmetic, graph, sets and lists.

1 Introduction

Learning logic programs using gradient descent-based methods has the potential to integrate the interpretability, transferability, and low data requirements of inductive logic programming (ILP) with the noise tolerance of non-symbolic learning. In recent years, several differentiable rule learning systems have been introduced (discussed below). Negation is a critical component of reasoning (expressing some concepts, e.g., *prime* and *unconnected*, require it) and it plays a central role in formal logic, knowledge representation, automated theorem proving, and other fields. But existing implementations either explicitly exclude negation, do not mention it, or state their systems can be extended to include negations.

Incorporating negation into differential ILP poses some challenges. The use of unrestricted negation may result in inconsistent reasoning. Care must be taken to ensure that negation is added using some reasoned methodology (such as stratification) and that the resulting system allows interesting programs to be learned via gradient descent.

In this work, using the differentiable inductive logic programming system ∂ILP [5] as the underlying framework, we add stratified, safe negation (i.e., as used in Datalog [3]), and we demonstrate that our system can learn recursive programs that fundamentally rely on negation.

35th Conference on Neural Information Processing Systems (aiplans 2021), Sydney, Australia.

Our **main contributions** are the following:

1. We modify ∂ILP to incorporate stratified negation [1]. This requires extending both the syntax and the semantics for clauses and modifying the forward chaining deduction process to ensure that consequences are correctly computed *wrt* stratified negation.
2. We present ILP problems of varying complexity requiring negation from several domains (arithmetic, graphs, sets and lists). We use these to demonstrate that our system is expressive and that our modifications work well with gradient descent-based learning of programs.
3. In order to reduce ambiguity in final programs without sacrificing the space of learnable programs, we introduce 'entropy of rule weights' as a regularizer.[1]

Related Works: The ∂ILP [5] system which we extend uses forward-chaining inference and allows predicate invention and recursion in learned rules. To reduce complexity, ∂ILP restricts the search space of rules using rule templates. *Differentiable Neural Logic-ILP* [11] also uses forward-chaining but does not require restrictive rule templates, and it allows Boolean functions in rule bodies. The examples presented, however, do not involve negation, and it is unclear (to us) how the system, particularly since it utilizes forward-chaining with a maximum number of steps, performs on learning relations depending on negation. *Neural Logic Inductive Learning* [16] also allows unrestricted rule bodies, but the examples presented again do not contain negation. In *Neural logic machines* [4], rules are encoded as weights of a network, and so extracting symbolic rules would be nontrivial.

Neural Theorem Provers (NTPs) [12, 13] use differentiable backward-chaining to reason over distributed representations of constants and predicates as well as generalized rule structures. Negation (which requires care in classical backward chaining) is not addressed. Campero et al. [2] describe a forward-chaining framework similar to NTPs (again using vector representations of predicates and constants). The authors compare it to ∂ILP and indicate that generates correct results more frequently. Negation is not addressed, however. Regarding systems based on semantics for logic programs with negation, Nickles [10] presents *differentiable Satisfiability and Answer Set Programming*. This system learns sets of models rather than rules, however, and so it has a different focus.

2 Methods

Inductive logic programming (*ILP*) [8, 9] is a machine learning framework in which the learned hypothesis is a logic program. A background knowledge base B and sets of positive and negative examples (P, N) are used as inputs, and the objective is to generate a program Π such that $\Pi \cup B \models a$ for all $a \in P$ and $\Pi \cup B \not\models a$ for all $a \in N$. Here, $\Pi \models a$ means that a is a consequence of Π.

The *ILP* framework is agnostic to the specific logic programming syntax and semantics used. In ∂ILP, the hypotheses are *definite logic programs*, i.e., collections of first order rules (clauses) r of the form $a \ :- \ b_1, b_2, \ldots, b_n$, where a and each b_i are *atomic formulas* (*atoms*) and $head(r) = a$ and $body(r) = \{b_1, \ldots, b_n\}$. An atom, e.g., $p(d, X)$, is a predicate (p) followed by constant (d) and variable (X) *terms*.[2] In ∂ILP, B, P, and N are sets of *ground* atoms (i.e, no variables appear), while no constants appear in rules. This is essentially positive *Datalog* [3].

If Π is definite, the ground atoms a such that $\Pi \cup B \models a$ can be computed via the *immediate consequence operator* T_Π which maps interpretations to interpretations. An interpretation $\mathcal{I}$ is a set of ground atoms; atom a is *true* in $\mathcal{I}$ iff $a \in \mathcal{I}$, and it *false* is otherwise. Below, $G(\Pi)$ is the set of ground rules made from all of the the rules of Π.

$$T_\Pi(\mathcal{I}) = \{head(r) | \text{ rule } r \in G(\Pi) \text{ and each member of } body(r) \text{ is true according to } \mathcal{I}\}$$

Intuitively, we start with $\mathcal{I} = \varnothing$ and repeatedly apply T_Π to perform forward-chaining inference. If Π is definite, a monotonically nondecreasing sequence of interpretations is created which eventually reaches a fixpoint.[3] The fixpoint is precisely the set of all ground atomic consequences of Π [7].

[1] ∂ILP defines 'entropy of rule weights' as a measure of the number of programs with non-zero weights learned. We use this as a regularizer in the loss term as explained in the implementation.

[2] As in ∂ILP, we disallow function expressions as terms, so the ground instantiations are finite.

[3] This claim assumes that the ground instantiation of Π is finite, which in this context is true.

∂ILP: One technique for finding solutions in ILP is a top-down generate-and-test approach in which all possible programs (from a given template) are generated and evaluated on the training data. This problem can be posed as an instance of SAT by assigning Boolean flags to the candidate clauses and stipulating that the resulting programs must be consistent with the training data. ∂ILP ([5]) is a differentiable relaxation of this 'ILP as SAT' approach. Programs are generated with trainable continuous weights on collections of clauses. Forward chaining, which is performed to a preset limit, utilizes real-valued evaluations of ground atoms and differentiable operators: $\wedge$ is replaced by multiplication, and $\vee$ is replaced by maximum. The loss value is given by the cross entropy between values computed for ground instances of the target predicate and the training examples. Gradient descent is used to train weights, and the program with the highest weight is taken as output. Thus, solving the ILP problem becomes an optimization problem of the form $min_W L(Q, C, W)$ where Q is the ILP problem, C is the set of clauses generated using a program template, W is the set of weights, and L is the loss function measuring the accuracy of a program (i.e., its predictions).

2.1 Adding Negation

∂ILP restricts hypotheses to be sets of definite clauses, and while the semantics is straightforward, the restriction is significant. *Normal logic programs* allow negated atoms $\neg b$ in rule bodies. One difficulty is that the forward chaining process used for definite programs cannot be directly applied.

In the following example, $\mathcal{I}$ is an interpretation, we will use $\mathcal{I}(a \wedge b) = \mathcal{I}(a) * \mathcal{I}(b)$ and $\mathcal{I}(\neg a) = 1 - \mathcal{I}(a)$. For a clause c, the truth value of the clause head is the truth value of the body. That is, $\mathcal{I}(head(c)) = \mathcal{I}(body(c))$.

T_Π is defined in essentially the same way as in the classical case: In $T_\Pi(\mathcal{I})$, the truth value of each atom a is the maximum truth value (according to $\mathcal{I}$) of the clauses having a as head. Note that this is equivalent to combining the rules for a given atom into a single rule, using $\vee$ to join the original rule bodies together. Here $\mathcal{I}(b_1 \vee \ldots \vee b_n) = max(\mathcal{I}(b_1), \ldots, \mathcal{I}(b_n))$.

Let Π be the following program. We assume `true` always has the value 1.

```
a :- true.      c :- b.      d :- b.
b :- ¬a.        c :- d.      d :- c.
```

In the sequence below, $\mathcal{I}_2$ is a fixpoint. Neither c nor d should be derivable, but they are.

$$\mathcal{I}_0 = \{a = 0, b = 0, c = 0, d = 0\} \qquad \mathcal{I}_2 = T_\Pi(\mathcal{I}_1) = \{a = 1, b = 0, c = 1, d = 1\}$$
$$\mathcal{I}_1 = T_\Pi(\mathcal{I}_0) = \{a = 1, b = 1, c = 0, d = 0\} \qquad \mathcal{I}_3 = T_\Pi(\mathcal{I}_2) = \{a = 1, b = 0, c = 1, d = 1\}$$

This result is obtained using simple stepwise forward chaining. If we amalgamate the interpretations using either $\mathcal{I}_{t+1} = max(\mathcal{I}_t, T_\Pi(\mathcal{I}_t))$ or probabilistic sum $\mathcal{I}_{t+1} = \mathcal{I}_t + T_\Pi(\mathcal{I}_t) - \mathcal{I}_t * T_\Pi(\mathcal{I}_t))$ as [5] suggest, then b would also be derivable, which is arguably a worse result. In the absence of stratification, because iterating T_Π from $\varnothing$ is no longer guaranteed to be monotonic, amalgamating interpretations as ∂ILP does will sometimes lead to irrational inferences.

Stratified Negation One widely known semantics for normal logic programs is based on *stratified negation* [1]. It is only defined for *stratified* programs, i.e., programs Π which can be partitioned into subsets $\Pi_1, \ldots, \Pi_k$ such that for any Π_i and any rule $r \in \Pi_i$, if $b \in body(r)$, the rules defining b are contained in $\bigcup_{j \leq i} \Pi_j$, and if $\neg b \in body(r)$, the rules defining b are contained in $\bigcup_{j < i} \Pi_j$. The idea is to use T_Π to compute the fixpoint for Π_1 and use the result to provide truth values for the negative literals in Π_2. The consequences of Π_2 can then be computed. A new sequence of interpretations is defined which terminates in what is sometimes called the *standard model* of Π.

Semantics for stratified normal logic program is discussed further in Appendix A.

Implementation of safe stratified negation in ∂ILP: The following are the major modifications that we made to ∂ILP to implement safe, stratified negation. Our clause generation process enforces stratification. The strata themselves are specified as part of the ∂ILP program template, along with names and arities of predicates to be learned. ∂ILP requires clauses to be *safe*, i.e., every variable in the head of a rule appears in the body of the rule (i.e., `p(X) :- q(Y)` is not safe).[4]

[4] User-given strata is only for convenience. Given a program template the maximum strata needed would be number of predicates specified. So, it would be direct to run the ∂ILP with every combination of strata.

Table 1: Programs learned and convergence in ∂ILP with stratified negation. All predicates other than the target and background predicates are invented (and renamed to increase readability). Also, | is used to indicate a disjunction of clauses.

Name	Program Found	Convergence
Birds fly	`fly(X):- birds(X), ¬penguin(X).`	100 %
A and not B	`target(X) :- A(X), ¬B(X).`	100 %
Prime	`prime(X) :- ¬factor(X).` `factor(X) :- div(X,Y), ¬eq(X,Y).`	100 %
Relative Prime	`rprime(X,Y) :- ¬comFactor(X, Y).` `comFactor(X,Y) :- div(X,Z), div(Y,Z).`	50 %
Not Element	`notElement(X,Y) :- ¬element(X,Y).`	100 %
Not Subset	`notSubset(X,Y) :- ele(X,Z), notEle(Y,Z).` `notEle(X,Y):- type(X,Y), ¬ele(X,Y).`	10 %
Not in List	`ListNotEle(X,Y) :- ¬ListEle(X,Y).` `ListEle(X,Y):- head(X,Y) | tail(X,Z), ListEle(Z,Y).`	100 %
Unconnected	`unconn(X,Y):- ¬path(X, Y).` `path(X,Y):- edge(X,Y) | edge(X,Z),path(Z,Y).`	93 %

Monopoly	`m(X,Y):- gpath(X,Y),¬rpath(X, Y).`
	`rpath(X,Y):- redge(X,Y) | redge(X,Z),rpath(Z,Y).`
	`gpath(X,Y):- gedge(X,Y) | gedge(X,Z),gpath(Z,Y).`
		85 %

We generalize safety to normal clauses by requiring that all variables of a rule appear in positive (non-negated) atoms in the body. Our clause generation process allows negation and respects stratification and safety. An example is included in Appendix C. The negation of an atom a is evaluated as $value(\neg a) = 1 - value(a)$, and forward chaining is done on a stratum-by-stratum basis. This is implemented using binary masks with 1 used when the predicate belongs in current stratum, 0 otherwise. The total number of forward chaining steps were equally divided between the strata.

To prevent high entropy of the rule weights, we introduced average entropy of the rule weights as a regularization term in loss as $L = 0.5 * H(P) + 0.5 * H(N) + \gamma * avgH(R)$ where L is the loss, H is the cross entropy, P and N are positive and negative examples respectively, γ is regularization scaling factor and R is the rule weights. We used a small γ, so the regularization does not prevent trajectory from leaving local minima. We varied the number of steps of forward chaining (5 to 15 steps per strata), the learning rate (from 0.0001 and 0.05) and the regularization coefficient (from 10^{-4} to 10^{-2}). To prevent class imbalance on positive and negative examples, we used equal weighted cross-entropy. We noted that when a run converges with low loss but with high entropy, more than 1 program has nonzero weight, and the correct program (with respect to the training data) may not be weighted the highest. but, we observed that the 3 highest weighted programs included correct program.

3 Experiments and Results

We tested our framework[5] on problems requiring negation with varying levels of complexity across different domains (nonmonotonic logic, arithmetic, graph, sets and lists). Table 1 summarizes our results.[6] Here we outline the problems considered. For details of the ILP problem, see Appendix D.

We first considered negation in programs with a single clause using a classical example from nonmonotonic logic ("all birds fly except penguins"). This problem can be expressed as a disjunction of all birds with flight, but has a simpler program with negation. We then considered $(A \wedge \neg B)$, which is the simplest program requiring negation. In the domain of sets, the problem of learning $\notin$ given

A similar argument can be made of user-given program templates (which is used in the original ∂ILP). Here the maximum parameters depend on the training data size.

[5]An implementation is available at https://github.com/girip/dilp-stratified-negation

[6]We give one learned program for each problem, and we removed the guards for safety e.g., a learned program might be `prime(X):- ¬factor(X), X=X`.

elements of a set is similar, but more complex as it has two types of constants—sets and elements—which are undifferentiated. We then examined problems requiring invented predicates and negation (such as prime and relative-prime, given divisibility as background), and $\not\subset$ (given sets and elements as background).[7] We also examined graph problems requiring negation and recursive invented predicates. These include unconnected vertices (given the edges of a directed graph as background) and *monopoly* i.e., the existence of a green-path but not a red-path in an edge coloured graph (given the edge-colours of a directed, edge coloured graph as background). We also considered 'not an element of' a list, which requires recursion as a list is coded using $head$ and $tail$ predicates.

Our system was able to learn correct programs from very small training sets (loss across experiments are given in Appendix B). The domain of sets includes two types of objects (sets and elements). This generated many type-incorrect ground atoms, which in turn caused the majority of learned programs to only check for type instead of the correct programs.

For selected examples (less than, predecessor, family relationships), we ensured that our system reproduces results from ∂ILP [5]. Furthermore, we verified that the problems from Table 1 cannot be solved by ∂ILP without negation, and that unsafe or unstratified negation often leads to higher loss value (Appendix B), and we examined the noise tolerance in the "birds fly" problem, where the system was able to identify the correct program even with high (up to 70%) noise (negative examples incorrectly labeled as positive in training data. More in Appendix B). We expect these results to generalize to other problems.

The learned program is taken as the clauses with the highest weights after training. However, there are conditions under which weights become distributed across multiple clauses even when the loss converges. To prevent these conditions, we used the average entropy of weights across clauses as a regularizer. This significantly reduced average rule entropy, specifically from 0.12 to 10^{-4} for the prime problem. This is discussed in Appendix B.

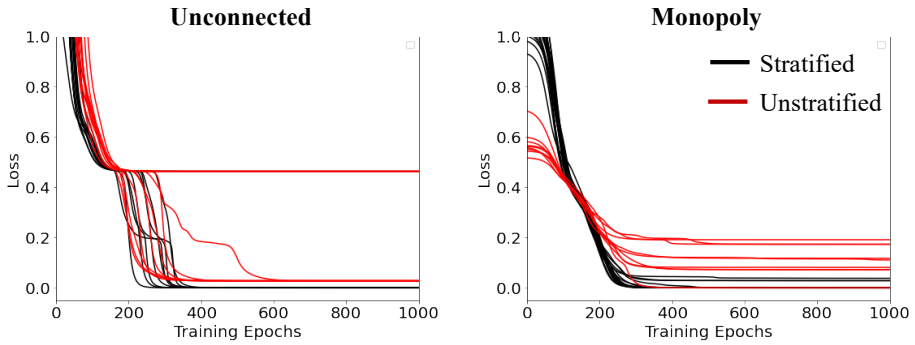

Figure 1: Trials with and without stratification for Monopoly and Unconnected.

4 Discussion and Conclusion

We modified the ∂ILP framework to incorporate safe, stratified negation. Our modifications worked well with gradient descent to learn programs that require invented predicates, recursion and negation across several domains (arithmetic, graphs, sets and lists). We also identified entropy of rule weights as a useful regularizer.

Our method is motivated by well-studied theoretical considerations on how to include negation. So, it can be generalized to other differentiable implementations of ILP that use forward chaining deduction.

As such, any application of differentiable inductive logic programming, such as Reinforcement Learning [15, 6] can benefit from our work, since negation is useful in many places. In future works, we plan to use this work to hierarchical reinforcement learning with planning. Another direction of future works is allowing non-monotonic reasoning.

[7] Here, $\subset$ is expressed with an implication. So it is directly expressed as the negation of $\not\subset$.

References

[1] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[2] A. Campero, A. Pareja, T. Klinger, J. Tenenbaum, and S. Riedel. Logical rule induction and theory learning using neural theorem proving. *CoRR*, abs/1809.02193, 2018.

[3] S. Ceri, G. Gottlob, and L. Tanca. What you always wanted to know about datalog (and never dared to ask). *IEEE Transactions on Knowledge and Data Engineering*, 1(1):146–166, 1989.

[4] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou. Neural logic machines. In *International Conference on Learning Representations*, 2019.

[5] R. Evans and E. Grefenstette. Learning Explanatory Rules from Noisy Data. *Journal of Artificial Intelligence Research*, 61:1–64, Jan. 2018.

[6] Z. Jiang and S. Luo. Neural Logic Reinforcement Learning. In *International Conference on Machine Learning*, pages 3110–3119. PMLR, May 2019.

[7] J. W. Lloyd. *Foundations of Logic Programming, 2nd Edition*. Springer, 1987.

[8] S. Muggleton and L. de Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19-20:629–679, 1994. Special Issue: Ten Years of Logic Programming.

[9] S. Muggleton, L. D. Raedt, D. Poole, I. Bratko, P. A. Flach, K. Inoue, and A. Srinivasan. ILP turns 20 - biography and future challenges. *Machine Learning*, 86(1):3–23, 2012.

[10] M. Nickles. Differentiable SAT/ASP. In E. Bellodi and T. Schrijvers, editors, *Proc. of the 5th International Workshop on Probabilistic Logic Programming (PLP 2018), Ferrara, Italy, Sept. 1, 2018*, volume 2219 of *CEUR Workshop Proceedings*, pages 62–74. CEUR-WS.org, 2018.

[11] A. Payani. *Differentiable neural logic networks and their application onto inductive logic programming*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 2020.

[12] T. Rocktäschel and S. Riedel. Learning knowledge base inference with neural theorem provers. In J. Pujara, T. Rocktäschel, D. Chen, and S. Singh, editors, *Proceedings of the 5th Workshop on Automated Knowledge Base Construction, AKBC@NAACL-HLT 2016, San Diego, CA, USA, June 17, 2016*, pages 45–50. The Association for Computer Linguistics, 2016.

[13] T. Rocktäschel and S. Riedel. End-to-end differentiable proving. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, pages 3788–3800, 2017.

[14] J. D. Ullman. Assigning an appropriate meaning to database logic with negation. In H. Yamada, Y. Kambayashi, and S. Ohta, editors, *Computers as Our Better Partners*, pages 216–225. World Scientific Press, March 1994.

[15] D. Xu and F. Fekri. Interpretable Model-based Hierarchical Reinforcement Learning using Inductive Logic Programming. *arXiv:2106.11417 [cs]*, June 2021.

[16] Y. Yang and L. Song. Learn to explain efficiently via neural logic inductive learning. In *International Conference on Learning Representations (ICLR 2020)*, 2020.

A Semantics for Stratified Normal Logic Programs

The following account is based in part on Apt et al. [1]. Here, Π is a stratified normal logic program.

Given a fixed vocabulary of constants, function symbols, and predicate symbols for Π, the set of all ground terms definable over that vocabulary is the Herbrand universe H_Π of Π, and the set of all ground atoms is the Herbrand base B_Π. An Herbrand interpretation $\mathcal{I}$ of Π is any subset of B_Π. $\mathcal{I}$ is a *model* of a ground rule r iff $head(r)$ is true in $\mathcal{I}$ or some element of $body(r)$ is false in $\mathcal{I}$. $\mathcal{I}$ is a *model* of a non-ground rule r iff it is a model of every rule in $G(r)$, where $G(r)$ is the ground instantiation of r. Similarly, $\mathcal{I}$ is a *model* of Π if it is a model of every rule of Π.

Herbrand interpretations can be ordered using subset inclusion ($\mathcal{I}_1 \leq \mathcal{I}_2$ iff $\mathcal{I}_1 \subseteq \mathcal{I}_2$). If Π is definite, there is a unique minimal model $MM(\Pi)$ of Π that coincides with the ground atoms entailed by Π and which can be computed via T_Π. We define a sequence of interpretations using some $\mathcal{I}$ as basis.

- $T_\Pi \uparrow 0(\mathcal{I}) = \mathcal{I}$
- $T_\Pi \uparrow \alpha(\mathcal{I}) = T_\Pi(T_\Pi \uparrow \alpha - 1(\mathcal{I})) \cup T_\Pi \uparrow \alpha - 1(\mathcal{I})$ for all successor ordinals α
- $T_\Pi \uparrow \beta(\mathcal{I}) = \bigcup_{\alpha<\beta} T_\Pi \uparrow \alpha(\mathcal{I})$ for all limit ordinals β.

For definite programs, if $\mathcal{I} = \varnothing$, the sequence is nondecreasing and reaches $MM(\Pi)$ at $T_\Pi \uparrow \omega(\varnothing)$, a fixpoint. If Π is normal, there might not be a unique minimal model, but a model with with desirable properties can be obtained by defining a sequence of interpretations based on a stratification $\Pi_1, \ldots, \Pi_k$ of Π.

- $M_1 = T_{\Pi_1} \uparrow \omega(\varnothing)$
- $M_i = T_{\Pi_i} \uparrow \omega(M_{i-1})$ for all $1 < i \leq k$

The final interpretation M_k is taken as the canonical model of the program and as defining the ground atomic consequences of Π. It is a minimal model of Π [1], and it is supported (each $a \in M_k$ is either a fact or the head of a ground rule whose body is true in the model). Importantly, the model is also independent of the specific stratification; all stratifications of Π yield the same model.

B Additional details on experimental results

Figure 2 shows the loss plotted against training epochs for various experiments. We ran 20 trials for each problem, and each plot shows the convergence behaviour of each problem. Number of runs converged defined as loss $\leq 10^{-3}$ were identified as convergent. Loss is computed as incorrect classifications of training examples. As mentioned in the Methods section, we introduced entropy as a regularizer in the loss. We examined the distribution of weights after convergence (Figure 3) and found that including a regularization term reduced entropy and resulted in only one of the clauses having high weight for each of the learned predicates. In contrast, when regularization was not used, even in cases when the loss converged, many clauses contributed to the output used in predictions and increased the ambiguity of the program learned by the framework.

We then tested whether stratification was required for solving problems with negation. For Unconnected and Monopoly (which required 2 and 3 strata, respectively) we tested convergence and examined the programs learned when stratification was enforced and also when it was not (in the latter case, all clauses were placed in a single stratum). We found that the loss was higher than 10^{-3} when stratification was not enforced (Figure 1) and also that the programs identified were incorrect. Finally, we varied the noise for the birds-fly problem. Results for that are presented in Figure 4.

C Clause generation

This section discusses the generation of safe and stratified clauses.

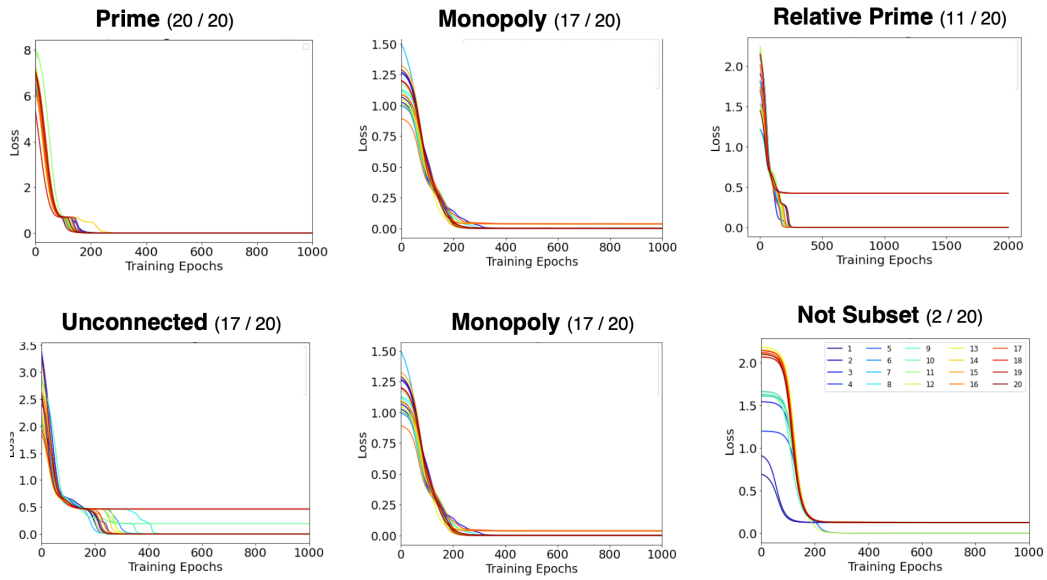

Figure 2: Convergence across different experiments. Each plot shows the loss value across training epochs. Number of runs converged defined as loss $\leq 10^{-3}$ were identified as convergent.

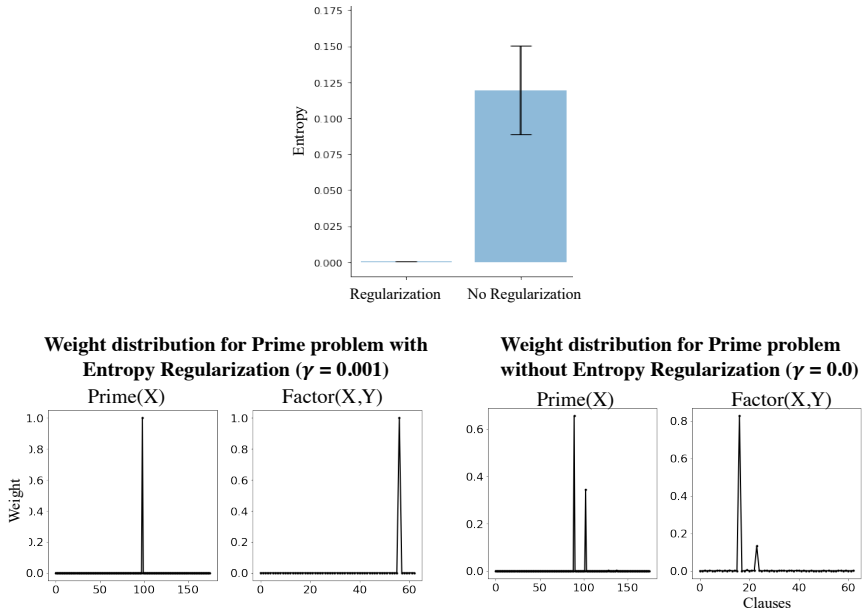

Figure 3: Effect of regularization on entropy in the Prime problem. Top. Average entropy of rule weights for experiments on Prime problem with and without regularization, taken after training (1000 steps). Bottom. Example distribution of the rule weight for one trial with regularization (and without), showing several rules had high weights for the run without regularization.

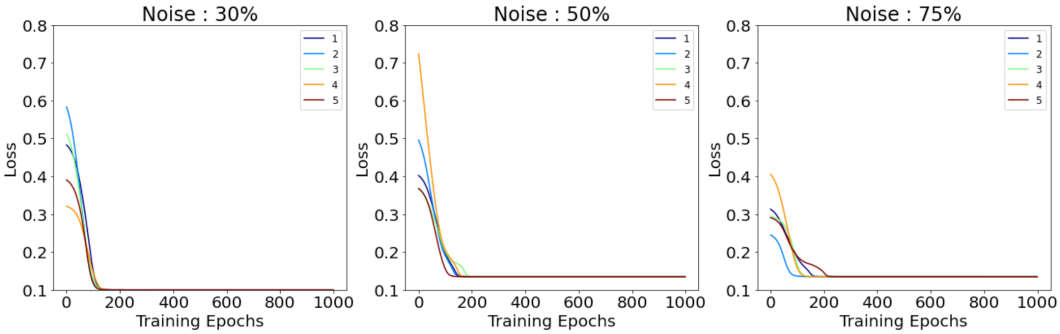

Figure 4: Effect of noise in the Birds fly problem. Noise was introduced by changing the percentage of negative examples incorrectly labeled as positive for training. In all these cases, the program `fly(X):- birds(X), ¬penguin(X)` was correctly identified, even in case of 75% noise. Loss was higher for higher noise levels, since the learned program did not predict the correct response for the dataset. Higher noise levels led to non-convergence.

C.1 Stratification

Stratification involves assigning predicates to strata and is a well understood method to prevent inconsistencies when negation is introduced. Below we demonstrate how stratification is introduced in different programs. We start with assigning lowest strata to background, positive and negative. Then, the invented predicates are assigned the subsequent strata and final output or target predicate is given the highest strata. Below, we illustrate the reason for this assignment using examples.

The below program can be stratified by assigning `edge` to the lowest stratum (1), `path` to stratum 1 (or $n \geq 1$) and `unconn` to stratum 2 (or $m > n$).

```
unconn(X,Y) :- ¬path(X,Y).
path(X,Y) :- edge(X,Y).
path(X,Y) :- edge(X,Z), path(Z,Y).
```

The predicate `path` can call itself recursively, since stratification allows predicates of the same stratum as head to be used positively in the body. However, `unconn` uses path negatively and so `unconn` must be of a higher stratum than path.

The following program is also stratified; `p` and `q` can be assigned to the same stratum.

```
p(X) :- q(X).
q(X) :- p(X).
```

However, neither the program {`p(X) :- ¬p(X)`} nor {`p(X):- q(X)`, `q(X):- ¬p(X)`} is stratified, as there is no way to assign the body to a lower stratum than the head of any rule.

C.2 Safety

Safety in ∂ILP: A rule is safe if the set of variables in the head is a subset of the variables in the body. For instance, `p(X,Y) :- q(X,Z), p(X,Z).` is not safe as `Y` is present in the head but not the body.

Safety with negation: Here, a rule is safe if each variable in a rule is also present in a positive body literal the rule. Observe that this is a generalization of the rule above. E.g., `p(X,Y) :- q(X,Y), ¬r(X,X)` is safe, whereas `p(X,Y) :- q(X,Y), ¬r(X,Z).` is not.

C.3 Some consequences

Given the safety conditions, and the restriction from ∂ILP that each clause contain 2 literals in the body, there can be at most one negated literal in the body of any clause. Since each program can have at most 2 clauses, a recursive program has one base clause and one recursive clause. Further,

if the recursive call does not contain a variable not in the head, then a terminating recursion can be unfolded to a fixed finite number of steps. This latter consequence happens in ∂ILP as well.

D ILP problems

D.1 Birds fly, except for penguins

The task here is to learn the program `fly(X):- birds(X),¬penguin(X)` from background knowledge consisting of instances of birds, given as `bird(X)`, $\forall$`X` $\in C$ (where C is a set of constants). Each individual bird belongs to a particular type $type$`(X)`, where $type \in$ {`eagle`, `sparrow`, `owl`, `parrot`, `hawk`, `penguin`}. The positive and negative examples indicate whether the bird can fly (`fly(X)`). This is false when `X` is a penguin and true otherwise. Domain is the set of names of birds of different types. Each bird type had 5 instances.

D.2 A and not B

The task is to learn `target(X) :- a(X), ¬b(X)` from background knowledge consisting of instances of ground atoms of the form `a(X)` and `b(X)`. Positive examples are in `a` but not `b`.

D.3 Prime

The task here is to learn a program for primes. The constants forming the domain are $\{2, 3, 4, 5, 6, 7, 8, 9\}$; we exclude 1 since it is neither prime nor composite. In this setting, a prime number has only itself as a factor. The background knowledge consists of ground atoms encoding equality (`eq(X,X)` for each `X` in the domain) and divisibility (`div(X,Y)` indicates that `X` is a multiple of `Y`). The positive and negative examples are the prime and composite numbers from the domain.

Other programs found include the following. Here, `pred` is an invented predicate.

```
prime(X) :- ¬pred(X), div(X,X).
pred(X) :- div(X,Y), ¬eq(X,Y).
```

D.4 Relatively Prime

The task here is to learn a program for relative primes from background knowledge consisting of the binary predicate `div` and `eq`. We also need a guard-predicate that includes all pairs of numbers to meet our safety requirements. Program: {`relPrime(X,Y):- ¬commonFactor(X,Y). commonFactor(X,Y):- div(X,Z), div(Y,Z)`}.

D.5 Unconnected

The background knowledge here encodes a directed graph with the constants naming vertices and a binary predicate defining directed edges. The positive examples include the pairs (x, y) of unconnected vertices. The negative examples are the connected ones. The intended program requires an invented recursive predicate, e.g., `unConn(X,Y):- ¬path(X,Y), guard(X,Y)`. The base case for `path` is `path(X,Y) :- edge(X,Y)` and the recursive case is `path(X,Y) :- edge(X,Z), path(Z,Y)`.

Instead of giving an external guard predicate, we made our examples such that given any pair of vertices x and y, either there was a path from x to y or there was a path from y to x. The system was able to find this and use it as a guard, i.e., `unConn(X,Y) :- ¬path(X,Y), path(Y,X)`.

The learned predicate for the path varied with different runs, e.g., the recursive case could be `path(X,Y):- path(X,Z), path(Z,Y)`.

D.6 Monopoly

Given a directed, edge coloured graph, `mon(X,Y)` holds iff there is a green path from `X` to `Y` but no red path from `X` to `Y`.[8] The constants are vertices of a graph, and the background binary predicates

[8] This example appears to be due to Ullman [14].

`redge` and `gedge` encode red and green edges. The positive examples included pairs of vertices satisfying the property `mon` above.

Even when the system had difficulty finding the correct target programs, it was able to learn the correct auxiliary predicates for red and green paths.

D.7 Not element

The task here is to learn a program for $\notin$. The constants include $\{p, q, r, 1, 2, 3, 4\}$ with p, q and r intended to represent sets and 1, 2, 3 and 4 representing elements. The background knowledge includes binary predicates $\in$ and a `guard`. This problem is similar to $A \wedge \neg B$, but has the added complexity of two types and multiple tokens of each type.

D.8 Not Subset

The task here is to learn the program $\not\subseteq$. The constants and background knowledge are as above. Positive examples are pairs of sets X and Y such that $X \not\subseteq Y$, i.e., $\exists Z.Z \in X \wedge Z \notin Y$. The intended program uses the background predicate $\in$ and a learned predicate $\notin$.

The relation $\subseteq$ is defined using implication, i.e. $X \subseteq Y$ holds iff $\forall Z.Z \in X \rightarrow Z \in Y$. The program for strict superset would be `SS(X,Y) :- ele(Z,X), ¬ele(Z,Y).` Then, subset can be defined as `sub(X,Y) :- ¬SS(X,Y)`. Thus, $\not\subset$ is more basic than $\subset$ in this setting.

D.9 Not Element of a list

The task here is to learn a program identifying when an item is not a member of a list. Th constants are $\{p_1, p_2, q_1, q_2, 0, 1, 2, 3, \}$ where p_i, q_i and 0 are lists and 1, 2, 3 are elements of the list (0 represents the empty list). The background predicates are `head` and `tail` (e.g., the list [1,2] would be `head(`p_1`, 1)`, `tail(`p_1`, `p_2`)`, `head(`p_2`,2)` and `tail(`p_2`,0)`. Thus, to find if a given element is a member of a list, a recursive predicate is needed. Identifying list membership was investigated in ∂`ILP`[5]. In our counterpart problem, positive examples are non-members, i.e., pairs X and Y where Y is *not* a member of the list X.