dSILP: A Differentiable Inductive Logic Programming Framework for Non-Monotonic Reasoning

Author: Christopher Hawkes

Supervisors:
Prof. Alessandra Russo
Dr. Krysia Broda

Second Marker:
Dr Matthew Williams

June 19, 2019
Abstract

ILP systems are increasingly popular ways of developing interpretable solutions to machine learning problems. They develop an inductive hypotheses described as a logic program which has the appealing property that it can accommodate any relational background knowledge. Unfortunately most systems only deal with symbolic representations of data and are unable to handle large datasets which severely limits the practical use and widespread adoptability of such systems. Recently, a move towards integrating the hugely successful optimisation methods used by neural networks in ILP have been put forward; these methods are able to handle subsymbolic representations of data and can deal with noisy data out of the box. In this work, we improve upon the state of the art in differentiable ILP systems by introducing the capability of handling non-monotonic reasoning and reducing the memory requirements in our system, thus opening the door for larger datasets to be tackled with more complex hypotheses.

Practical evaluations of the proposed systems will be presented, both in a qualitative and quantitative manner, before looking at possible extensions to further its capability.
Acknowledgements

To my supervisors, Prof. Alessandra Russo and Dr. Kryia Broda, I would like to express my deepest gratitude and appreciation for all of their insightful comments, remarkable energy, and ceaseless waves of encouragement I was so lucky to receive.

To my family, I send my greatest love and indebtedness for everything they do.

To Mark Law, the author of ILASP, I send my thankfulness for catalysing this area of research.

To Rachel, who proved to be a wonderful source of procrastination.
Contents

1 Introduction ............................................. 4
   1.1 Motivation ......................................... 4
   1.2 Contributions .................................... 5
   1.3 Outline ........................................... 6

2 Background .............................................. 7
   2.1 Inductive Logic Programming ......................... 7
   2.2 ILASP ............................................. 9
      2.2.1 Answer Set Programming .......................... 9
      2.2.2 Violating Hypotheses and Violating Reasons .... 10
   2.3 Probabilistic Inductive Logic Programming ............ 11
      2.3.1 Problog ....................................... 11
      2.3.2 DeepProblog ................................... 12
   2.4 Differentiable Logic Constraints ....................... 14
      2.4.1 Semantic Loss .................................. 14
      2.4.2 DL2 .......................................... 15
   2.5 Symbolic Reinforcement Learning ....................... 16
   2.6 Markov Logic Networks ................................ 17

3 Related Work ............................................ 18
   3.1 Differentiable Approaches ......................... 18
      3.1.1 δILP ......................................... 18
      3.1.2 Neural Theorem Proving (NTP) ................... 19
   3.2 Non-Differentiable Approaches ....................... 19
      3.2.1 ILASP3 ....................................... 19
      3.2.2 nFOIL ....................................... 19
      3.2.3 CILP++ ...................................... 20

4 Critical Analysis of Related Work ........................ 21
   4.1 ILASP3 ............................................ 21
   4.2 δILP .................................................. 21
      4.2.1 Forward chaining Over T Time Steps .......... 22
      4.2.2 Probabilistic Interpretation ................... 23
   4.3 Overview of dSILP .................................. 24

5 Groundings and Templates ................................ 25
   5.1 Weighted ASP Programs ............................... 25
   5.2 Relevant Grounding .................................. 27
## CONTENTS

5.3 Rule and Program Templates ................................. 28

6 **Supported ILP (dSILP)** ........................................ 30
   6.1 Continuous relaxation of Supported model semantics .......... 30
   6.2 Differentiable Supported ILP Task ........................... 32
      6.2.1 Training .................................................. 34
      6.2.2 Memory Requirements .................................. 35

7 **Supported dILP Evaluation** .................................. 36
   7.1 Grandparent .................................................... 36
   7.2 Adjacent to Red ............................................... 37
   7.3 Even-Odd ....................................................... 37
   7.4 Graph Colouring ............................................... 38
   7.5 Noise .......................................................... 38

8 **Stable ILP** ..................................................... 41
   8.1 Pairwise Scoring for Stable Models ........................... 41
   8.2 Adversarially Training Unfounded Sets ........................ 43
   8.3 Removing Solutions with Violating Hypotheses ............... 43

9 **Discussion of Future Work** .................................. 44
   9.1 Probabilistic Interpretation .................................. 44
   9.2 Lazy Grounding ............................................... 46
   9.3 Hard Constraints ............................................. 47

10 **Conclusion** .................................................. 48

A **Learned Hypotheses** ......................................... 49
Chapter 1

Introduction

1.1 Motivation

Prediction is an integral part of intelligence and the problem of prediction has troubled many of the great minds over the last century. The problem is inherently appealing as anyone who can grasp a better prediction of the future can set themselves up for success over their peers. Alongside the individual motivations of the field one can note that such a tool that extrapolates from data effectively can help catalyze medical discoveries and bring about a much smarter wave of digital assistants, such as Siri and Alexa, that are already being integrated into many of our lives.

Predicting patterns from data has become a large part of almost any company currently rising in the world today. Deep learning has seen remarkable success with this task and it is aggressively being integrated into many existing companies today. Due to the increasing capacity of computing power available it has also enjoyed widespread adoption outside large firms and has created an immense amount of potential even for the individual. The tool of choice in deep learning has been Artificial Neural Networks (ANNs), which have seen a general trend in becoming deeper and generally more complex so that their expressivity is capable of modelling almost any function. However this has also caused their variance to grow accordingly where the common solution has been to apply intense regularisation and prepare a large amount of training and test data, although the latter may be impractical or extremely expensive in the general case. Additionally, as these models grow in size, it has caused them to be less and less interpretable; they become a byzantine combination of millions of parameters. This has become an increasingly important issue in lieu of the recent General Data Protection Regulation (GDPR[Reg16]) which requires actions taken as a result of a prediction from a model to be justified. In particular, the actions need to be shown not to be motivated by any of the following: racial or ethnic origin, political opinions, and religious or philosophical beliefs. Understanding the justifications behind a model’s reasoning can not only help us meet these new requirements but also improve the model itself and comprehend the problem better ourselves.

It is clear that we need to understand more clearly the opportunities Artificial Intelligence (AI) brings to society and their associated risk. For example, an AI system can identify security attacks much quicker than humans can, although the associated risks with mass surveillance are the possibilities that it encroaches our freedom of information and
anonymity. The ethical questions surrounding the subject are complex, but I argue that we will enjoy more of the benefits AI can bring when we can understand and interact with it. It is crucial that we address ethical issues now while the field is still maturing so that we can continue harnessing the full potential of AI safely.

Today, one of the most challenging open problems in Artificial Intelligence is finding an architecture that is as capable as ANNs and still has an interpretable solution. Recently there have been an increasing number of efforts from the deep learning community to combine Inductive Logic Programming (ILP) techniques with those of ANNs. ILP has a number of advantages over ANNs: they have an explicit symbolic structure that can be understood and verified; they tend to generalise well from only a handful of examples; and they are able to easily transfer knowledge between tasks by simply incorporating learned inductive programs in one task as background knowledge in the next. This last point is known as a subfield of AI called transfer learning. Techniques for ANNs that have been developed to achieve transfer learning normally involve fine-tuning an existing network or manually designing a network architecture that enforces the prior background knowledge. Examples include: training a network that was trained on cheaply available data of the same domain as the required task and then training on the ‘real’ dataset; training Convolutional Neural Networks whose convolutional layers enforce local spatial priors on the features.

The adoption of ILP systems has been much lower however as they have not been able to scale well and are also not easily ubiquitously applicable to all domains (for example raw pixels) like ANNs. Further, ILP systems generally rely on a given hypothesis space in which to look for solutions that need to be crafted for each individual task; this in contrast to ANNs which don’t require that expertise and can effectively search a much larger function space. It has also been argued that many of these algorithms need to ‘bake in’ much of the solution required by greatly constricting the hypothesis space which nullifies the purpose of learning the solution. This has been the case in many of the researched approaches to ILP, which have been unable to cope with large hypothesis spaces. However, as the learning power of these systems increases, the capacity of the hypothesis space accessible will become sufficiently general that domain expertise should not be an issue. Clearly a combination of both approaches that can successfully bring their respective advantages forwards would be the next step in the machine learning revolution we are currently experiencing.

1.2 Contributions

In this report we consider the available systems which pursue a combination of the approaches as described in the motivation section and critically analyse the most prominent restrictions each possess. Having done this we formulate a new system which builds upon the successes of its predecessors and addresses some of the limitations that were not previously handled. We list the main contributions of the work here:

- A critical review of related work, exposing flaws in state of the art approaches.
- The construction and subsequent evaluation of dSILP, a non-monotonic ILP learner that addresses some of the flaws exposed in the critical review section.
- A consideration of the system under two non-monotonic semantics: stable, and sup-
ported models.

- A discussion of the limitations of dSILP and proposals of future extensions.

1.3 Outline

In chapter 2 we present a survey of some of the existing literature behind Answer Set Programming and some of the probabilistic extensions which are able to handle noisy data. Then in chapter 3 we explore in greater depth the available systems that attempt to harmoniously combine deep learning and logic based approaches, whose pitfalls are afterwards exposed in chapter 4. You will then find a collection of definitions required to introduce the task dSILP solves in chapter 5. We subsequently define the learning task dSILP solves in chapter 6 alongside a high level overview of the algorithm used to solve the learning task. An evaluation of the system is then undertaken under two forms of non-monotonic semantics with the hope of addressing some of the disadvantages which have been found so far in chapter 7. Proposals on how to increase the quality of the learned solution are discussed in chapter 8. You will dutifully find a final discussion of the presented approach in chapter 9 and the report’s corresponding conclusion in chapter 10.
Chapter 2

Background

The report assumes that the reader is familiar with First Order logic and some Logic Programming. As and when required, more terminology will be introduced throughout the report.

2.1 Inductive Logic Programming

In its most general form, Inductive Logic Programming defines the following machine learning task: given some background knowledge and a set of positive and negative examples, find a hypothesis that together with the background knowledge covers the positive examples and none of the negative examples. Generally, the language of the background knowledge and the hypothesis is based on first-order logic and the coverage of the hypothesis together with the background knowledge is based on classical logical entailment. One can immediately see that ILP has a clear way of incorporating background knowledge and provides a human interpretable representation of the solution. We have the following formal definition for a fairly general ILP task.

**Definition 2.1.** Given a set of positive examples $E^+$, a set of negative examples $E^-$, some background knowledge $B$, a hypothesis space $S_M$ and a coverage criterion. An ILP theory $\mathcal{H} \subseteq S_M$ is a solution iff

1. $\forall \ e \in E^+ \ . \ covers(B, \mathcal{H}, e)$
2. $\forall \ e \in E^- \ . \ \neg covers(B, \mathcal{H}, e)$

**Example 2.2.** The following example task describes an ILP theory which can be hoped to be learned when logical entailment is used as the the coverage criterion:

\[
B = \left\{ \begin{array}{l}
\text{son(chris, nick).} \\
\text{daughter(nicole, nick).}
\text{father(nick, chris).}
\text{mother(sofia, chris)} \\
\text{brother(jeremy, nick).}
\text{daughter(jeremy, tessa).}
\text{son(jeremy, luke).}
\end{array} \right\} \quad \quad (2.1)
\]
\( S_M = \left\{ \begin{array}{l} h_1 : niece(X,Y) : - brother(Z,Y), daughter(X,Z) \\ h_2 : niece(X,Y) : - sister(Z,Y), son(X,Z). \end{array} \right\} \) \hspace{1cm} (2.2)

\( E^+ = \{ niece(tessa,nick). \} \) \hspace{1cm} (2.3)

\( E^- = \{ niece(luke,nick). \} \) \hspace{1cm} (2.4)

When classical logical entailment is used as the coverage criterion the task is referred to as Learning from Entailment [Law18, p. 48] where it is denoted as \( ILP_{LFE} \). We say that for a task \( T = (B,S_M,(E^+,E^-)) \), \( H \) is an ILP theory of the task if \( H \in ILP_{LFE}(T) \). With this notation we note that:

- \( \emptyset \notin ILP_{LFE}(T) \) under learning via entailment as it does not entail the positive example.

- \( \{ h_2 \} \notin ILP_{LFE}(T) \) entails the negative example given the background knowledge and is therefore not a consistent hypotheses and not an ILP theory of the task.

- \( \{ h_1 \} \in ILP_{LFE}(T) \) is the only set \( \subseteq S_M \) that both entails the positive example and is consistent with the negative examples under the given background knowledge.

There has been a large body of work in this area that has brought about many algorithms with different approaches to computing the solution of an ILP task. Generally they can be grouped into a few different categories: Bottom-up, Top-down (also known as generate and test), and Meta-Level.

A Top-down approach will start with a strong conjecture, from which too general clauses are removed and replaced by specializations of the removed clauses until a consistent theory is found. Specializations of removed clauses are found via a downward refinement operator such as the one proposed by Shapiro[Sha81] which is represented by \( \rho_o \). By contrast, upward refinement operators have also been proposed, which have led to a Bottom-up approach that begins with a weak conjecture (a more specific one) and searches for more general clauses that are computed by the upward refinement operator. Refinement operators induce a lattice structure on the search space as can be seen in Figure 2.1.

![Figure 2.1: A partial expansion of the lattice induced by Shapiro’s \( \rho_o \) refinement operator](image)

Meta-level approaches are relatively new approaches to the ILP task that aim to encode the task itself as a logic program. The logic program is then solved by delegating the task to a SAT solver or by the usual means of backwards or forwards chaining. Meta-Interpretive Learning (MIL) is an example of a recent meta-level technique that adapts the Prolog
meta-interpreter by fetching higher order meta-rules that unify with a given goal; the resulting meta-rules form the learned program. A powerful and novel aspect of MIL is that when learning a predicate definition it ‘automatically introduces sub-definitions, allowing decomposition into a hierarchy of reusable parts’[GHOK+15]. Another example is the set of ILASP algorithms, which induce programs in the Answer Set Programming language and are the current state of the art in symbolic ILP. These programs generate constraints which are solved by delegating them to clingo[GKKS14], a state of the art SAT solver.

The following summary table of algorithms is adapted from Inductive Learning of Answer Set Programs by Mark Law (2018), which the reader is encouraged to review for an in depth understanding of the Answer Set semantics behind the ILASP algorithms. In the next section I discuss the intuition behind some of the ILASP algorithms where some of the notions introduced there are put into practice in chapter 8 as a method to search the hypothesis space.

Definition 2.3. A **Horn clause** is a disjunction of literals with at most one positive literal.

Definition 2.4. A **Definite clause** is a Horn clause with exactly one positive literal. It is represented as a rule in the following form where $a$ and $b_i$ are positive literals.

\[
a \leftarrow b_1, \ldots, b_n
\]

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Language of $B \cup H$</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Progol [Mug95]</td>
<td>Definite Clauses</td>
<td>Bottom-up</td>
</tr>
<tr>
<td>Metagol [LDE+14]</td>
<td>Definite Clauses</td>
<td>Meta-level</td>
</tr>
<tr>
<td>TAL [CRL10]</td>
<td>Normal Clauses</td>
<td>Metal-level</td>
</tr>
<tr>
<td>HAIL [RBR03]</td>
<td>Definite Clauses</td>
<td>Bottom-up</td>
</tr>
<tr>
<td>FOIL [Qui90]</td>
<td>Definite Clauses</td>
<td>Top-down</td>
</tr>
<tr>
<td>XHAIL [BR15]</td>
<td>Normal ASP programs</td>
<td>Bottom-up</td>
</tr>
<tr>
<td>ILASP [LRB18b]</td>
<td>Answer Set Programs(^1)</td>
<td>Meta-level</td>
</tr>
</tbody>
</table>

Table 2.1: A Summary of some of the main algorithms of ILP

2.2 ILASP

2.2.1 Answer Set Programming

Answer Set Programming is a powerful language for knowledge representation and reasoning that is capable of non-monotonic reasoning. Under the answer set programming semantics there are two different interpretations of Learning from Entailment (ILP\(_{LFE}\)) as introduced earlier. These are brave and cautious induction. In [LRB18a] it is shown that the framework coined Context-dependent Learning from Ordered Answer Sets (ILP\(_{LOAS}\)) is more general than both of these interpretations and maintains the same complexity as

\(^1\)The restricted class of ASP programs containing only normal rules, choice rules and hard and weak constraints.
that of cautious induction. Given an ASP program formed from only normal rules, choice rules, and hard and weak constraints the solutions of an ASP program $P$ form a subset of the Herbrand models of $P$, called the answer sets of $P$ and denoted as $AS(P)$.

In the following definitions (adapted from [Law18, p. 32–34]) it is assumed that $h_i$ and $b_i$ are atoms and $l$ and $u$ are integers. It is further assumed that $not$ is interpreted in the ‘negation as failure’ sense which was introduced by Clark [Cla78].

**Definition 2.5. Normal rules.** Are of the form $h : ¬b_1, \ldots, b_m, not b_{m+1}, \ldots, not b_n$.

**Definition 2.6. Hard Constraints.** Are of the form $¬b_1, \ldots, b_m, not b_{m+1}, \ldots, not b_n$.

**Definition 2.7. Choice Rules.** These represent the condition that between $l$ and $u$ of the atoms $h_1$ to $h_k$ are true when the body is true and are of the form $l\{h_1, \ldots, h_k\}u : ¬b_1, \ldots, b_m, not b_{m+1}, \ldots, not b_n$.

**Definition 2.8. Weak Constraints.** These express a preference ordering over the answer sets of a program and are of the form $¬b_1, \ldots, b_m, not b_{m+1}, \ldots, not b_n. [wt@lev; t_1, \ldots, t_k]$ where $wt$ is the weight of the constraint, $lev$ is the priority level, and $t_i$ are terms which specify which ground weak constraints should be considered unique.

**Example 2.9.** The following program expresses the fact that a coin is either heads or tails.

\[
\text{coin}(1).
\]

\[
1\{\text{heads}(C), \text{tails}(C)\}1 : ¬\text{coin}(C).
\]

Its answer sets are $\{\text{coin}(1), \text{heads}(1)\}$ and $\{\text{coin}(1), \text{tails}(1)\}$.

**Example 2.10.** The following program builds intuition behind the non-monotonic semantics of ASP. Intuitively, negated atoms are a source of ‘contradiction’ or ‘unstability’ and an interpretation will be an answer set if it is not self-contradicting.

\[
\text{married}(X) : ¬\text{person}(X), not \text{single}(X).
\]

\[
\text{single}(X) : ¬\text{person}(X), not \text{married}(X).
\]

\[
\text{person}(\text{chris}).
\]

Its answer sets are $\{\text{person}(\text{chris}), \text{single}(\text{chris})\}$ and $\{\text{person}(\text{chris}), \text{married}(\text{chris})\}$.

### 2.2.2 Violating Hypotheses and Violating Reasons

**Definition 2.11.** A positive hypothesis is one which covers all of the positive examples.

**Definition 2.12.** A violating hypothesis is one which covers some negative example.

It is proved in [Law18] that the inductive solutions to any $ILP^\text{context}_{\text{LOAS}}$ task are exactly the positive hypotheses that are not violating hypothesis. Hence ILASP1 proceeds by constructing a meta-level answer set program that iteratively searches for a hypothesis by first computing all of the violating hypotheses of a given length. Then, it computes all of the positive hypotheses that are not violating hypothesis, and if this set is not empty, it
will return the answer set that encodes the solution. This algorithm is shown to be sound and complete and return a hypothesis of optimal length. Hence, in contrast to ANNs, we can already see it is not sensitive to its initialisation like many deep learning algorithms.

The reason ILASP1 has to iteratively search for solutions is because we cannot write an ASP using intuitive rules that solve ILP_{\text{LOAS}} tasks since these tasks are $\Sigma_2^P$-complete and deciding the satisfiability of these programs is only $\Sigma_1^P$-complete (unless the polynomial hierarchy collapses). If disjunction was allowed in ASP then it would be possible to solve the task in one go, however encoding the problems in this more expressive language can be unintuitive, and are often only understandable by experts in ASP\cite{GKS11}. This would defeat the much touted benefit of logic programming solutions being interpretable by the layman.

ILASP2 augments ILASP1 with the notion of violating reasons, which enable ruling out many violating hypotheses which are all violating for the same ‘reason’. This helps combat some of the scalability issues of ILASP1 because there may be an extremely large amount of violating hypotheses that need to be ruled out before one can find an optimal inductive solution. ILASP2 still suffers when the problem domain is large because this affects the size of the relevant grounding that needs to be encoded. Further it still enumerates the hypothesis space in full. ILASP2i then makes further progress in scalability by finding a set of relevant examples which is in general much smaller than the full set of examples. Finally ILASP3 solves an extension of the first learning framework that handles noisy data, it is introduced in more detail in the related work chapter as this closely relates to the learning task investigated in this report.

2.3 Probabilistic Inductive Logic Programming

Whilst in the Learning from Answer Sets framework the aim is to learn programs whose answer sets capture the set of possibilities, Probabilistic Inductive Logic Programming (PILP) aims to learn a probability distribution over the set of possibilities. Generally, a possibility is still an interpretation whose elements are $\in \{0, 1\}$. For prediction and inference tasks that deal with uncertainty this is a necessary extension to the ILP setting. To introduce DeepProblog which is the first example of combining symbolic and subsymbolic representations in this section we first describe Problog.

2.3.1 Problog

Problog\cite{DKM+15} is a probabilistic implementation of the Prolog programming language where one can query the marginal probabilities over a subset of a program’s atoms as opposed to just the success or failure of that subset. To achieve this rules in Problog are represented as Annotated Disjunctions defined below. Problog also uses Sentential Decision Diagrams (SDDs) which are efficient representations of a program and supports parameter learning over the probabilities in Annotated Disjunctions from partial interpretations (incomplete information about all of the atoms in the Herbrand Base).

**Definition 2.13. Annotated Disjunction.** These are interpreted as saying ‘If the body is true then the head atom $h_i$ is true with probability $p_i$.’ If $\sum_i p_i < 1$ then with probability
(1 − \sum_{i}^{n} p_i) none of the atoms in the head are true.

p_1 :: h_1; \cdots; p_n :: h_n : - body, \ text{ where } \sum_{i}^{n} p_i \leq 1

Given a set of partial interpretations (called the evidence), which in Problog is a set of atoms which are labelled true or false, Problog finds the probabilities p_i with maximum likelihood over this evidence. This is achieved through random initialisation of the probabilities and the use of expectation maximisation (EM) to iteratively revise them until convergence. What allows Problog to do efficient parameter learning is the compilation of the program (a set of Annotated Disjunctions) into SDDs. However this is a large obstacle for rule induction as it would be intractable to compile all of the possible programs and then pick the model with the highest likelihood of explaining the data. Hence we find that Problog does not support rule induction but it does give an intuitive model for statistical relational learning. Problog programs are a set of annotated disjunctions.

Example 2.14. The following Problog program\[smo\] infers the probabilities we annotate as t(\_).

\[
\begin{align*}
t(\_) &:: stress(X) : -person(X). \tag{1} \\
t(\_) &:: influences(X,Y) : -person(X),person(Y). \\
smokes(X) &:: -stress(X). \\
smokes(X) &:: -friend(X,Y),influences(Y,X),smokes(Y). \\
\text{person(1).person(2).person(3).person(4).} \\
\text{friend(1,2).friend(2,1).friend(2,4).friend(3,2).friend(4,2).} \\
\text{Given the following evidence (here just one partial interpretation)} \\
evidence(smokes(2),false). \\
evidence(smokes(4),true). \\
evidence(influences(1,2),false). \\
evidence(influences(4,2),false). \\
evidence(influences(2,3),true). \\
evidence(stress(1),true). \\
\end{align*}
\]

It returns the following after 15 iterations

0.333 :: influences(X,Y) : -person(X),person(Y). 0.666 :: stress(X) : -person(X).

In this example we can see that our posterior probabilities reflect that any person has a \( \frac{2}{3} \) chance of being stressed and there is a \( \frac{1}{3} \) chance that any person influences another.

2.3.2 DeepProblog

DeepProbLog\[MDK^+18\] is an extension to Problog that allows the combination of both symbolic and subsymbolic representations in both querying marginal distributions and program induction. During program induction DeepProblog trains (possibly many) neural networks with a fixed number of outputs that are treated as a distribution (and so needs to be normalized) over different possible scenarios (like in a classification setting). Interestingly, DeepProblog still manages to also jointly learn the weights over facts alongside the training of the neural predicates. A DeepProbLog program is a Problog program that is extended with a set of ground neural Annotated Disjunctions.
Definition 2.15. (Neural Annotated Disjunction). Are of the form

\[ nn(m_q, t, u) :: q(t, u_1); \cdots; nn(m_q, t, u) :: q(t, u_n) : \neg \text{body} \]

Where the body is a conjunction of atoms, \( t \) a vector of ground terms that are the input to the network model \( m_q \) for predicate \( q \). This model specifies a probability distribution over its output values \( u_1 \ldots u_n \) given input \( t \).

Although the semantics and inference methods remain the same as that of Problog, the authors build on an algebraic extension of Problog (based on semirings) that supports automatic differentiation. This allows training of the whole model through gradient-descent based optimization, as opposed to EM. Given a set of training pairs \((q,p)\) where \( q \) is a query and \( p \) is its desired success, they minimize the sum of some loss function \( L \) between the result of the query and \( p \) over the parameters of the model. With this extension DeepProblog has achieved the sought after combination of ANNs and logic programming. Prior knowledge is now more easily expressible and a task now has the option of being decomposed into smaller, easier tasks whilst still being trained in an end to end fashion.

Example 2.16. In this example we present the program that DeepProblog would use to train a network model \( m_{\text{digit}} \) to recognize handwritten digits where the training examples are pairs of digits and we expect the output to be their sum.

\[ nn(m_{\text{digit}}, X, [0]) :: \text{digit}(X, 0); \cdots; nn(m_{\text{digit}}, X, [9]) :: \text{digit}(X, 9). \]

\[ \text{addition}(X, Y, Z) :: \neg\text{digit}(X, X^2), \text{digit}(Y, Y^2), Z \text{ is } X^2 + Y^2. \]

Example 2.17. DeepProblog would use the following program to perform program induction by training the neural network model \( m_{\text{swap}} \) to fill in the ‘hole’ in the program. This allows the program to perform bubblesort on handwritten images.

\[ nn(m_{\text{swap}}, [X, Y], [0, 1]) :: \text{swap}(X, Y, 0); nn(m_{\text{swap}}, [X, Y], [0, 1]) :: \text{swap}(X, Y, 1). \]

\[ \text{hole}(X, Y, X, Y) :: \neg\text{swap}(X, Y, 0). \]

\[ \text{hole}(X, Y, Y, X) :: \neg\text{swap}(X, Y, 1). \]

\[ \text{bubble}([X], [], X). \]

\[ \text{bubble}([H1, H2|T], [X1|T1], X) :: \neg\text{hole}(H1, H2, X1, X2), \]

\[ \text{bubble}([X2|T], T1, X). \]

\[ \text{bubblesort}([], L, L). \]

\[ \text{bubblesort}(L, L3, \text{Sorted}) :: \neg\text{bubble}(L, L2, X), \]

\[ \text{bubblesort}(L2, [X|L3], \text{Sorted}). \]

\[ \text{sort}(L, L2) :: \neg\text{bubblesort}(L, [], L2). \]

From example 2.17 we can see that the claim of program induction by DeepProblog is weak, as the structure of the program has to be provided and only holes left in the program are filled by neural network predicates. Essentially it is learning the less than relation in the context of bubblesort. Further we note that the templates for the neural predicates are rigid and fully defined before training, thus the interface between the symbolic and subsymbolic deduction requires expert knowledge about the task at hand. Hence, although DeepProblog has retained the full power of both logical reasoning, probabilistic reasoning and deep learning, I argue it is not capable of program induction.
2.4 Differentiable Logic Constraints

A number of different differentiable logic constraints have recently been proposed to address the problem of including background knowledge into the training of ANNs. Logical constraints are not limited to just providing an input method for a priori knowledge however, as it has been shown [FBDC+19] that they can be used to query ANNs to find inputs that serve as counter examples to the query which is of use in generative models and in verification. Further, by imposing constraints such as the Lipshitz condition (definition 2.23), ANNs can be made more stable.

The constraints, normally expressed in some subset of first order logic, can also express desirable properties we normally wish an ANN to have. Some examples of expressible constraints are shown below for clarity. In chapter 6 I propose an architecture that takes inspiration from the following differentiable logic constraints.

2.4.1 Semantic Loss

All of the results and definitions in this section come from [XZF+17]. Semantic loss is a differentiable loss defined over probabilistic logic where the intuition is that it is proportional to the negative log of the probability of generating a state that satisfies the constraint, when sampling Bernoulli values according to a given vector of probabilities. Thus if the constraint is more likely to be satisfied it will result in a lower loss. One can note here the similarity between this loss and the probability of success of an atom in Problog. The constraint is represented as a sentence $\alpha$ in propositional logic and the loss is defined with respect to a vector $p$ where each element $p_i$ corresponds to the predicted probability of a variable $X_i$ from a state vector $X$. It is required that the variables in $\alpha$ are contained in $X$.

**Definition 2.18. Semantic Loss.** Let $p$ be a vector of probabilities, one for each variable in $X$, and let $\alpha$ be a sentence over $X$. The semantic loss between $\alpha$ and $p$ is

$$L^s(\alpha, p) \propto -\log \sum_{x=\alpha} \prod_{i: x = X_i} p_i \prod_{i: x = \neg X_i} (1 - p_i)$$

Incorporating this loss into the training of a neural network is easily achieved by a weighted addition of the semantic loss and the existing loss. Semantic loss was proved to have a number of important features desirable in any differentiable logic constraint. In particular it was shown that the following hold:

**Theorem 2.19. Monotonicity.** If $\alpha \models \beta$ then $L^s(\alpha, p) \geq L^s(\beta, p)$ for any vector $p$.

**Theorem 2.20. Semantic Equivalence.** If $\alpha \equiv \beta$ then $L^s(\alpha, p) = L^s(\beta, p)$ for any vector $p$.

**Theorem 2.21. Satisfication.** If $x = \alpha$ then $L^s(\alpha, x) = 0$.

Monotonicity is an important constraint as together with a refinement operator as I argue it can be used to efficiently search some hypothesis space of constraints. Semantic Equivalence guarantees that the loss is a semantic property of the sentence and does not depend on the syntax. A consequence of Satisfication and Monotonicity is that the loss is always non-negative and we can interpret the value 0 as the constraint being satisfied.
Example 2.22. If we let the output values of a neural network be the vector \( p \), so that each node in its output represents a variable in the state, then we can define the sentence \( \alpha \) over its output variables \( X_i \) below, where this sentence represents the fact that we want at most one of the outputs to be true\([XZF^{+}17]\).

\[
\bigvee_{i \neq j} \neg X_i \lor \neg X_j
\]

The fact that these constraints are only expressible in propositional logic means that the prior knowledge expressible in this language is limited. Further, the authors also note that the given definition of Semantic Loss it is computationally demanding and so we look at other proposed differentiable losses.

### 2.4.2 DL2

DL2\([FBDC^{+}19]\) is another differentiable loss that can capture a richer class of constraints on inputs, outputs and internals of models. However some properties of this loss have not been proved such as Monotonicity which we had for Semantic Loss.

Their language consists of quantifier-free constraints which can be formed with conjunction (\( \land \)), disjunction (\( \lor \)) and negation (\( \neg \)) over terms of their language. Terms in this language are variables or constants, which represent real-valued vectors, or functions over these terms. Functions in this language are allowed so long as they are differentiable (almost everywhere). Atoms are comparisons \( \ast \) of terms where \( \ast \in \{=, \neq, <, \leq, >, \geq\} \). An example constraint that can be expressed in this language is the Lipshitz Condition:

**Definition 2.23. Lipshitz Condition.**

\[
\forall z_1 \in L_\infty(x_1, \epsilon), z_2 \in L_\infty(x_2, \epsilon). \ |p_\theta(z_1) - p_\theta(z_2)|_2 < L |z_1 - z_2|_2
\]

This constraint expresses that for two inputs from the training set \( (x_1, x_2) \) and any two points in their \( \epsilon \)-neighborhood \( (z_1, z_2) \) the difference between the predicted probabilities for \( (z_1, z_2) \) are bounded by a constant \( L \) times the difference between the inputs \( z_1 \) and \( z_2 \). We interpret \( p_\theta(z_1) \) as the vector output from a neural network with weights \( \theta \) and input \( z_1 \).

The loss \( \mathcal{L}(\phi) \) is defined recursively on the structure of a given formula \( \phi \) of this language. For a given assignment \( x \) into the variables of the formula, it has been shown that the corresponding loss \( \mathcal{L}(\phi)(x) \) is non-negative and that Satisfication holds. The proposed method of training neural networks with these constraints differs from what we saw with Semantic Loss, as it involves minimizing the expectation of the maximal violation of the constraints. Gradient descent is still used in most cases but when working with constraints such as the Lipshitz Condition where the set of variables \( z \) outside of the training set is convex (in this case \( L_\infty \) balls) then the author’s have shown that some constraints have a closed-form analytical solution and so they use Projected Gradient Descent (PGD) to make the optimization tractable in these cases. I outline below some of the recursive definition of the DL2 loss \( \mathcal{L}(\phi) \). In what follows, \( d(.,.) \) is a metric which is differentiable almost everywhere (for example the absolute value function \( |.| \) or L2-norm \( ||.||_2 \)) and \( \xi \) is a pre-defined constant.
L(φ ∨ ψ) := L(φ)L(ψ), \quad L(φ ∧ ψ) := L(φ) + L(ψ)

L(t_1 = t_2) := d(t_1, t_2), \quad L(t_1 < t_2) := L(t_1 < t_2 ∨ t_1 > t_2),

L(t_1 ≠ t_2) := L(t_1 < t_2 ∧ t_1 > t_2), \quad L(t_1 < t_2) := L(t_1 + ξ ≤ t_2)

Negations in this language are dealt with by using pushing them down with De Morgan’s laws until they are next to atoms, and then the comparison operator of the atom is flipped (for example > would become ≤). DL2 loss has shown experimental success on the CIFAR-100 dataset[KH09], improving ANN performance by including sensible priors that group sensible classes together such as the class of ‘people’, not actually present in the dataset but defined as \( p_{people} = p_{baby} + p_{boy} + p_{girl} + p_{man} + p_{woman} \). The constraint is then defined to express the fact that the probability of \( p_{people} \) should either be very small or very large:

\[ \forall x. p^θ_{people}(x) < ϵ ∨ p^θ_{people}(x) > 1 - ϵ \]

A differentiable loss inspired by the above differentiable logic constraints are applied in this project to guide the search for stable and supported models in chapter 6 and chapter 8. Using a differentiable loss will allow us to handle noisy labels directly and the evaluation chapters will aim to test this behaviour.

2.5 Symbolic Reinforcement Learning

Reinforcement learning has seen great success in game playing such as in learning the Atari video games, and in learning games with large state spaces such as Alpha Zero when playing Go [SHS+18]. However akin to their deep learning counterparts RL solutions are opaque to the human interpreter and so they suffer similar disadvantages when verifiability is important. To tackle this issue [GAS16] proposed a reinforcement learning architecture comprising of a neural back end and a symbolic front end. They tested their model on a game based on three simple shapes within an image. The neural back end is an autoencoder network and the front end takes in symbolic representations of the features from the middle layer of the autoencoder. After training the autoencoder on frames of the game the front end learns to extract these symbolic representations and Q-learning is used to pick the action the agent should take next. Figure 2.2 shows the structure of the system.

Although this proposed architecture did not fair so well in an easy variant of a game as the DQN[MKS+15] baseline it was pitted against, it was shown that it had a better capacity for transfer learning between the different game variants tested. It was also able to handle a more complex version of the game better than the baseline. I argue that although this architecture takes a step forward in the interpretability of the agent’s representation of a scene, analysing tables of Q-function values to justify any decision it makes is still laborious and would be improved if decisions were logical implications from a set of learned rules and a frame of the game. As the author’s themselves note, a further elaboration to their architecture would be to allow logic-based planning methods in the decision phase (as opposed to choosing the action with the maximum Q value).
2.6 Markov Logic Networks

A Markov Logic Network [RD06] provides another way of combining first order logic and probability. In a sense, first order logic is ‘embedded’ in MLNs by weighting formulas and having the probability of an interpretation be proportional to the exponential sum of the weights of the formulas which are true given that interpretation. This induces a probability distribution over the discrete interpretations which has modes at the satisfying assignments of the formulas with the largest weights. This formulation allows useful results to be drawn even in the presence of contradictions in the knowledge base and is therefore useful in real world applications and has seen widespread application.

However, MLN’s have the drawback that they cannot express (non-ground) inductive definitions [LW15] because these are not expressible in classical models. \( L_{\text{PMLN}} \) is a more powerful model which subsumes MLN’s by combining MLN’s with stable model semantics. It is shown that they can correctly handle the transitive closure of inductive definitions and probabilistic transition systems with their new semantics. The formulation in chapter 9 of the likelihood of interpretations over weighted programs can be seen as having a similar aim to \( L_{\text{PMLN}} \)’s but defined over continuous interpretations as opposed to discrete ones.
Chapter 3

Related Work

3.1 Differentiable Approaches

Due to the incredible success of deep learning approaches that rely on using numerical optimisation methods that themselves require differentiable objectives, attempts have been made at turning rule induction into a differentiable mechanism. Once this has been achieved the task can then be handed to an off the shelf numerical optimiser that finds a locally optimal solution. Here the methods rely on the fact that most locally optimal solutions have proven to be sufficiently good in practice. Some of the main approaches to this task can be classified into trying to model logical inference itself with an ANN like in [CR18] or actually turning the logical inference procedure into a differentiable procedure. The advantage of the latter being that we still end up with a set of interpretable rules and hence we focus on those approaches here.

3.1.1 $\delta$ILP

$\delta$ILP [EG17] is an ILP system where the forward-chaining inference method from rules in the logic program are end-to-end differentiable and therefore rules can be connected to neural networks so that the resulting structure can support rule induction via gradient descent. Thus this system can now support a combination of symbolic and sub-symbolic predicates and also learn with some robustness to noise. The claim made by Evans et al. that purely logical ILP systems are not applicable to noisy settings is refuted by the authors of [LRB18b] and a direct comparison is made of the performance between $\delta$ILP and ILASP3. This comparison is extended in this report in chapter 7 to dSILP when considering learning a ‘member of’ relation over lists.

The method for rule induction presented here seems to differentiate across the whole relevant grounded hypothesis space to perform one iteration of gradient descent and hence consumes a large amount of memory. Because of its memory consumption, they mention that they currently restrict themselves to nullary, unary, and binary predicates. However $\delta$ILP does achieve true rule induction as opposed to DeepProblog, which just fills in holes in a program with black box ANNs, and so it retains the possibility of being fully interpretable.
3.1.2 Neural Theorem Proving (NTP)

Rocktäschel and Riedel (2016) proposed an end-to-end differentiable prover\cite{RR17} that is similar in spirit to $\delta$ILP but implements a backward chaining inference instead (inspired by Prolog's backward chaining algorithm). The theorem prover relies on learning dense vector representations of symbols and function-free first-order rules of predefined structure. These vector embeddings are learnt simultaneously along with the induction of the rules. Thus they achieve ILP by gradient descent instead of a combinatorial search over the space of a set of predefined hypotheses.

The NTP is built out of the differentiable ‘modules’ UNIFY, AND, and OR. For example the unification module of the inference procedure is based on the distance between the vector embeddings. Each module takes as inputs discrete objects (atoms and rules) and a proof state, and returns a list of new proof states. They define the overall success score of proving a goal $G$ given a Knowledge Base and a predefined maximum proof depth as the maximum (most likely) proof path. It is noted that since the optimized loss is based on this overall success score of training queries the gradient updates for the respective subsymbolic representations will only be along the maximum proof path; this means it can take a long time until NTPs learn to place similar symbols close to each other in the vector space and to make effective use of rules. Additionally, it was noted that this method suffers from severe computational limitations as the NTP is representing all possible proofs up to some predefined depth. I also expect that the dense vector representations also have the possibilities of experiencing saturation problems when the number of symbols in the program increase.

3.2 Non-Differentiable Approaches

3.2.1 ILASP3

Here the examples are given penalties that are summed to represent the cost of not covering them. The optimal solution of this framework is one which minimises the cost over all examples alongside a cost for the size of the solution found, therefore preferring smaller solutions. The intuition behind ILASP3 is to iteratively improve an approximation of the cost of a hypothesis by computing coverage constraints from the given examples. ILASP3 has been evaluated against $\delta$ILP on monotonic tasks that $\delta$ILP can handle in \cite{LRB18b}. It was shown that $\delta$ILP was unable to correctly learn the inverse hypothesis when the proportion of noise was far greater than $\frac{1}{2}$, where indeed ILASP3 succeeded, which was hypothesised to be due to the restrictive nature of the hypothesis space in $\delta$ILP.

3.2.2 nFOIL

nFoil\cite{LKDR05} performs a covering loop over the examples given, essentially learning one feature (a clause) one after another, until it cannot find further improvements in its search heuristic. The search heuristic is based on conditional likelihoods where the Naive Bayes assumption is made which essentially reduces to assuming that clauses are independent. nFoil does not allow for negative literals as its coverage loop approach makes induction
efficient but also unable to learn recursive clauses. Learning non-monotonic ASP programs with negations as failure requires a different approach[Ray09].

3.2.3 CILP++

As an initial step, CILP++[FZG14] maps each example into a bottom clause via BCP (Bottom Clause Propositionalisation) and each target literal into a numerical vector that a neural network can use as an input. Then it builds a neural network by mapping for each bottom clause generated, for each body literal in the bottom clause an input neuron to an output neuron representing the head of the clause. It can then be shown that the constructed network computes the intended meaning of the background knowledge. CILP++ is also unable to learn recursive clauses, due to the incompleteness of Bottom Clause Generalisation.
Chapter 4

Critical Analysis of Related Work

In the background section we noted that ILASP2i innovated mechanisms that help tackle some of the scalability issues in searching for an optimal solution given a hypothesis space. Then we also saw that differentiable logical constraints have been successfully used to make neural networks more robust to adversarial attacks and also augment their performance via the addition of a priori background knowledge constraints. Further, in the related work section, we have seen a differentiable implementation of the forward chaining mechanism which allows end-to-end training of neural network predicates and rule induction, as well as symbolic methods which achieve the same task under the presence of noise. This chapter provides us with a deeper dive into the underlying issues present in two of the related systems, with a heavy insight into the latter. This will serve to provide us with an understanding between the transformations taken to create the dSILP system.

4.1 ILASP3

ILASP3 is the current baseline for the state of the art in symbolic inductive learning tasks. However it still struggles with the grounding of some logic programs and searching a large search space of rules. Therefore many tasks are still out of scope for this evolving and promising algorithm with its current approach. It has been mentioned\[LRB18b\] that future work on this algorithm includes avoiding computing the hypothesis space in full, which limits the feasible size of the hypothesis space the system can consider. Also, it is not yet clear whether it is possible to integrate subsymbolic learning into the ILASP algorithms, which is arguably its biggest downfall as this limits its application into new domains. It does have some very appealing properties however, as it can boast soundness and completeness of the system, along with a rich subset of the ASP language which handles non-monotonic logic. Since this work focuses on a differentiable approach that may jointly train its symbolic and subsymbolic counterparts, only some of the notions introduced by the ILASP algorithms will be carried forward but not its core approach.

4.2 δILP

We begin by reviewing the forward chaining mechanism used in δILP and criticize the justification of their choice of amalgamation function as a means to motivate a new system.
CHAPTER 4. CRITICAL ANALYSIS OF RELATED WORK

After the introduction of a few more basic concepts, this will be followed by a formal description of the dSILP system that aims to solve the ILP as a re-formulated optimisation problem under non-monotonic logic semantics.

4.2.1 Forward chaining Over T Time Steps

$\delta$ILP relies on gradients flowing backwards through repeated application of forward chaining inference and therefore it is susceptible to slower training times and vanishing gradients. The author’s chose to use the probabilistic sum for their $f_{\text{amalgamate}}$ function which is responsible for calculating the valuation of the next time step given the valuation at the previous time step and its consequences which is defined as follows:

$$f_{\text{amalgamate}}(x, y) = x + y - x \ast y$$

(4.1)

Although this helped $\delta$ILP overcome the adverse effect that the $\text{max}$ function had on their gradient flows, it can now also be seen from the table below that just through repeated application of their forward chaining mechanism the fuzzy valuation of some atoms may be made arbitrarily close to 1. This would not in general be desirable as the increased confidence is not justified. This may be seen from the following program with only one clause that defines the intensional predicate $q$.

Example 4.1.

$$q(X) \leftarrow p(X)$$

(4.2)

<table>
<thead>
<tr>
<th>$T$</th>
<th>$p(1)$</th>
<th>$q(1)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0.3</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>0.3</td>
<td>0.3</td>
</tr>
<tr>
<td>2</td>
<td>0.3</td>
<td>0.51</td>
</tr>
<tr>
<td>3</td>
<td>0.3</td>
<td>0.657</td>
</tr>
</tbody>
</table>

Table 4.1: Increasing confidence in $q(1)$ over time

From this example we can see that using the above specified $f_{\text{amalgamate}}$ function results in a tight coupling between the weights of the learned clauses in $\delta$ILP and the number of steps used in its forward chaining inference procedure $T$, so that performance will vary with $T$; this is an unwanted effect as we should expect the performance to remain invariant for high values of $T$. This also means that the hyperparameter $T$ defined at training should be considered when trying to use the system in transfer learning tasks that use a pre-trained set of weights on which to continue training. Of course if the output is a single un-weighted hypothesis then for a single learning task, and large enough $T$ to form all of the consequences available in the training set but not too large to have adverse effects, then $\delta$ILP will perform as expected.

Recently, a new differentiable ILP architecture termed as a Differentiable Recurrent Logic Machine (DRLM)[JL19], which is an improved version of the $\delta$ILP system, was proposed. The system works with valuation vectors which are made up of real numbers in
and has a more flexible program template that is carried over into this work as they do not restrict the number of clauses that may define a predicate to 2 as is done in δILP. Also, it requires a vastly reduced amount of memory as weights are not stored for each pair of program templates but instead they are stored as individual candidates for each possible clause of the hypothesis. This enables linear, as opposed to a quadratic, scaling of the number of weights in the program with respect to the number of candidate rules given the number of clauses in a hypothesis. A detailed discussion of the memory requirements of dSILP compared to the memory requirements of these systems stated below will be found in the following chapters.

Similarly to δILP, a DRLM makes repeated single step deductions of all the possible clauses weighted by their confidences. However, DRLM’s do not amalgamate the valuation of deductions at a particular time step with the previous valuation but instead only with the first valuation. In this way they are not susceptible to an unjustified increase in the degree of truth of an atom over time in the same way as δILP, but instead a similar effect still occurs due to the use of the same function being applied between clauses on repeated forward deductions. δILP and DRLMs could remove this effect by sticking to using max for their amalgamate function, however their evaluation showed their system coupled with the max function was unable to learn even some simple programs. Even with these adjustments, both systems suffer problems with large forward chains required for some inference problems as the time and memory required will grow linearly with the number of time steps specified by the user for each gradient update.

Supported model semantics allow us to drop the requirement for repeated applications of a forward chaining inference procedure and instead look for solutions which are fixed points of a one-step consequence operator. A system based on these semantics should then hope to exhibit lower memory usage and advantages in problems requiring large chains of forward inference whilst also keeping its robustness against noise/uncertainty and ability to deal with fuzzy data.

4.2.2 Probabilistic Interpretation

Here we address the claim that the deductions from the system can correspond to the degree of belief that an atom is true. We show that the system is not suitable for probabilistic reasoning and therefore the hypotheses that it produces should be discrete and the weights δILP learns may only be treated as confidence values in the learned hypotheses.

Example 4.2. Choosing Coloured Balls. In this example we consider three balls $x_1, x_2, x_3$, two of which are red. The following program describes that the $i^{th}$ ball is picked, independently of the rest, with probability $p_i$ and that you win if you pick a ball that is red.

\[
\begin{align*}
red(x_1), & \quad red(x_2), \quad green(x_3), \\
p_1 \text{ pick}(x_1), & \quad p_2 \text{ pick}(x_2), \quad p_3 \text{ pick}(x_3), \\
\text{win} & \leftarrow \text{pick}(X), \text{red}(X).
\end{align*}
\]

The question we consider is what is the probability of winning?

After one forward pass using the method as described by δILP, assuming that the probabilities $p_{1,3} = \frac{1}{3}$ and that the initial valuation has 1’s only at the facts $red(x_1), red(x_2)$, and $green(x_3)$,
then the value associated with \textit{win} would then become \( \frac{1}{3} \). However, we would expect the correct probability of picking at least one red ball to be \( 1 - \left( \frac{2}{3} \cdot \frac{2}{3} \right) = \frac{5}{9} \).

The probabilistic interpretation limitation is not directly addressed in the following chapters and instead this line of discussion is resurfaced when we talk about future work in \textit{chapter 9}.

### 4.3 Overview of dSILP

We will aim to address the limitations of the above systems by introducing non-monotonic reasoning into a differentiable system; attempting to reduce the memory requirements of our system; introducing a new training mechanism enabling the incorporation of negative examples in the learning task; and thirdly to improve upon the probabilistic interpretation of the models of the learning task. The exploration of these aims will follow the above order in the following chapters.
Chapter 5

Groundings and Templates

One method available to extend a propositional system to first order predicate logic is to simply ground the corresponding program. Thus the first step involved in any method that wishes to solve a propositionalised version of a first order problem will be to ground it. This chapter will introduce the syntax that will be used to ground programs and corresponding rule templates in the following chapters. The rule templates will produce hypothesis spaces for the ILP learning tasks that will also follow in the next chapter.

Although many state of the art grounders are freely available such as gringo [GKKS11], I found it necessary during this project to develop a bespoke rudimentary grounder to enable an attribution process linking the grounded rules to the first order rules that generated them. The loss of structure when grounding a first order program, such as that found by applying any state of the art grounder, will also be briefly discussed again in the following chapter as it will then be made clear what choices are available when attributing consequences of a weighted program to certain rules. This becomes important during the gradient updates of weighted rules to help avoiding getting stuck in local optimums. In order to define a grounded program, we will first define a restricted class of weighted ASP programs below.

5.1 Weighted ASP Programs

The weighted program as defined below takes direct inspiration from the DRLM, where it consists of N clause sets and the $i^{th}$ set has a $M_i$ clauses. Each set defines a probability distribution over its $M_i$ clauses where each clause probability represents the degree of belief that that clause should be the representative member of that set. Taking all of the sets of clauses together creates a weighted program by defining the one-step consequences of the program to be the application of a probabilistic sum (definition 5.2) over the consequences of each set. This is formalised in definitions 5.1 and 5.3. Using the probabilistic sum here is equivalent to assuming that clauses in one set are independent of the clauses in another set. Although this interpretation may not always be desired, it provides a class of programs over which one-step deductions are fast.

**Definition 5.1.** A Weighted Program $P_W$ is defined as a set of sets of clauses, where each clause is either a normal rule or a hard constraint, and a set of weight vectors $W$. Each
set of clauses $s_i$ has a corresponding weight vector $w_i \in W$ with the following restrictions so that it defines a distribution over the clauses in the $i^{th}$ set:

1. $\sum_j w_{i,j} \leq 1$
2. $\forall j w_{i,j} \geq 0$

The weight $w_{i,j}$ represents the probability that the the $j^{th}$ clause is the right definition for the $i^{th}$ set of clauses. The weight vectors may not necessarily be of the same size and a vector of real numbers may be transformed to satisfy the restriction by applying softmax. When $\sum_j w_{i,j} < 1$ then a probability of $1 - \sum_j w_{i,j}$ is attributed to the empty clause specifying our belief that this clause set does not have any specific member.

Given a weighted program $P_W$, we would like to consider all of the consequences it produces, given a current interpretation, by applying all of its rules in proportion to their weights under the assumption of independence between clause sets. In the definition below, we limit ourselves to finding the values after only allowing each rule to be applied once. By only ever applying the rules once, we will avoid coupling our program with the unnecessary hyperparameters and avoid the unjustified increase of probabilities when applying a probabilistic sum multiple times as shown in subsection 4.2.2. Our definition of applying a weighted program once is as follows:

**Definition 5.2.** The **probabilistic sum** of $x$ and $y$ is defined by

$$x \oplus y = x + y - x \ast y$$  \hspace{1cm} (5.1)

**Definition 5.3.** A **one-step deduction** mapping $g : [0, 1]^G \rightarrow [0, 1]^G$ of an interpretation $I \in [0, 1]^G$ given a weighted program $P_W$ is:

$$g(I)_k = \sum_i \sum_j w_{i,j} \max_{r_j^k} h_{i,j}(I, r_j^k)$$  \hspace{1cm} (5.2)

Where $G$ is the size of the relevant grounding, $w_{i,j}$ is the weight of the $j^{th}$ possible rule of the $i^{th}$ clause, $r_j^k$ is a grounded instance of this rule whose head unifies with the $k^{th}$ atom of the interpretation $I$, and $h_{i,j}(I)$ is the one step deduction using only that rule defined by:

$$h_{i,j}(I, r_j^k) = \prod_{b^+ \in r_j^k} v(b^+) \prod_{b^- \in r_j^k} (1 - v(b^-))$$  \hspace{1cm} (5.3)

Where $v(b^+)$ is the value of the grounded positive body atom in the interpretation $I$ and similarly for the negative body atoms $v(b^-)$. Constraints are assumed to always unify with a unique **bottom** atom denoted by $\bot$.

The one step deduction of a rule, $h_{i,j}(I, r_j^k)$, as defined above can be interpreted as the fuzzy conjunction of the degrees of truth of its body atoms under the **Product t-norm**. This satisfies the conditions of a t-norm[EGM01] which forms the basis of one interpretation of fuzzy logic:
1. **commutativity**: \( x \ast y = y \ast x \)

2. **associativity**: \((x \ast y) \ast z = x \ast (y \ast z)\)

3. **monotonicity**:
   - (a) \( x \leq y \implies x \ast z \leq y \ast z \)
   - (b) \( x \leq y \implies z \ast y \leq z \ast x \)

The Product t-norm was chosen here as experimental evaluation on the \(\delta\)ILP system concluded it carries improved gradient information over other available t-norms such as Godel’s and Lukasiewicz’s. We also note here that applying a probabilistic sum over the consequences of each set of clauses creates a smoother surface over which we traverse by gradient descent into local optima as opposed to using a max function. Using a probabilistic sum, as noted before, places independence assumptions on clauses between sets but also takes further strides away from the canonical definition of a one-step deduction operator in discrete space. However, the stark contrast between the surface induced by each definition is clear when you plot the derivative of a rule’s weight against the value of the weights of two rules who’s heads unify with the same atom. The simplest example we can consider is when we have two clause sets which have identical rules with empty bodies:

\[
\{ w_{1,1} \ a. \} \quad \{ w_{2,1} \ a. \}
\]

\[ (a) \ \frac{\partial}{\partial w_{1,1}} \text{ when taking max over clause sets} \]
\[ (b) \ \frac{\partial}{\partial w_{1,1}} \text{ when taking } \oplus \text{ over clause sets} \]

### 5.2 Relevant Grounding

Existing ILP systems such as ILASP use an equivalent subset (in terms of the answer sets it produces) of the ground program as opposed to the complete grounding, which is denoted by \(\text{ground}(P_W)\) and is the program consisting of all the ground instances of all rules in \(P_W\). This is because although in some cases the complete grounding may be infinite, the relevant grounding is finite [Law18]. This allows more programs to become tractable for ILP systems which use the relevant grounding in the propositionalisation step.

**Definition 5.4.** The **Herbrand Universe** of a weighted program \(P_W\) is the set of all terms that can be constructed from the function and constant symbols in \(P_W\). This will be denoted by \(\text{HU}_{P_W}\).
Definition 5.5. The Herbrand Base of a weighted program $P_W$ is the set of all atoms that can be constructed from the predicate symbols in $P$ whose arguments are terms from $HUP_W$. This will be denoted by $HB_{P_W}$.

Definition 5.6. Let $P_W$ be any weighted program. $f_{P_W} : \mathcal{P}(HB_{P_W}) \rightarrow \mathcal{P}(HB_{P_W})$ is the function $f_{P_W}(I) = I \cup \{h | r \in \text{ground}(P_W), \text{weight}(r) > 0, \text{body}^+(r) \text{ is satisfied by } I, \text{ and } h \in \text{head}(r)\}$

Definition 5.7. Given a weighted program $P_W$, $f_{P_W}$ has a unique least fixpoint. We call this the relevant Herbrand Base of $P_W$ and denote it by $HB_{P_W}^{rel}$.

5.3 Rule and Program Templates

A common way of constructing a learning task is to first define a search space in which to look for solutions of that task. This section describes the templates that are used in this report which define the search space for the learning tasks solved by dSILP. Although first order logic allows its terms to be generated by a set of constants, variables and a set of function symbols, we will restrict our attention here to first order languages whose set of function symbols is empty. This restriction aids us in maintaining a finite Herbrand Universe so that we are able to exhaustively produce the relevant grounding of a program. We will also make the distinction between two ways of defining predicates which form part of a program:

Definition 5.8. An extensional predicate is a predicate that is wholly defined by a set of ground atoms.

Definition 5.9. An intensional predicate is defined by a set of clauses.

Definition 5.10. A Rule Template $\tau$ is a tuple $\langle p_h, P_b, max_{pos}, max_{neg}, max_{tot} \rangle$. It describes the set of rules which can be generated such that:

- $p_h$ describes the head predicate of the rule
- $P_b$ is a set of predicates that are allowed in the body
- $max_{pos}$ is the maximum number of positive body predicates allowed in the body
- $max_{neg}$ is the maximum number of negative body predicates allowed in the body
- $max_{tot}$ is the maximum total number of body predicates

Definition 5.11. A Program Template $\Pi$ is a tuple $\langle R, B \rangle$ which constructs a search space in the following way:

- $R$ is a set of sets of rule templates such that $|R|$ is the maximum number of clauses in the learned program.
- Each set of rule templates $\tau_i \in R$ defines the possible clauses that may be the representative member of the $i^{th}$ clause.
- $B$ defines the background knowledge of the learning task.
Predicates that form part of a rule template may also optionally be ‘typed’ by naming their arguments with extensional predicates that must hold. For example $\text{succ}/2$ defines a predicate of arity 2 whilst $\text{succ}(\text{number}/1, \text{number}/1)$ defines a predicate of arity 2 whose arguments must both satisfy the extensional predicate $\text{number}/1$.

**Example 5.12.** Here we will construct a program template that is later used in the *even* learning task.

- $\tau_1 = <\text{even}/1, \{\text{succ}/2, \text{zero}/1, \text{helper}/1\}, 2, 2, 2>$
- $\tau_2 = <\text{helper}/1, \{\text{succ}/2, \text{zero}/1, \text{even}/1\}, 2, 2, 2>$
- $R = \{\{\tau_1, \tau_2\}, \{\tau_1, \tau_2\}\}$, $B = \{\text{zero}(0), \text{succ}(1, 0), \text{succ}(2, 1), \cdots, \text{succ}(10, 9)\}$
- $\Pi = <R, B>$

A sample of the rules generated for the above hypothesis space would be:

- $\text{even}(A) \leftarrow \text{succ}(B, A)$.
- $\text{even}(A) \leftarrow \text{succ}(A, B)$.
- $\text{even}(A) \leftarrow \text{zero}(A), \text{helper}(A)$.
- $\text{helper}(A) \leftarrow \text{succ}(A, B), \text{even}(B)$.
- $\text{helper}(A) \leftarrow \text{succ}(A, A), \text{even}(A)$.

We note that as the number of predicates allowed in the body grow, as well as their arity, we are faced with a combinatorial explosion of possible rules and enumerating the search space in full becomes a computational bottleneck. This issue is raised again in chapter 9.
Chapter 6

Supported ILP (dSILP)

In this chapter, we will recall the definition of a Learning from Answer Sets (ILP$_{LAS}$) task \cite{Law18} and reformulate it as an optimisation problem by re-defining what it means to be an inductive solution of a similar task in a new setting considering weighted programs. The core algorithms behind the dSILP system will be introduced and subsequently evaluated in the following chapter. Some specific details about the training procedure are laid bare and design decisions corresponding to the optimised loss function are discussed.

6.1 Continuous relaxation of Supported model semantics

Stable models have a 1-1 correspondence with the extensions of default logic\cite{MS92}. The stable model semantics defines a popular mode of non-monotonic reasoning however these models can be highly intractable to compute. In this section we focus instead on supported model semantics which still possess most properties an ideally introspective agent would exhibit. This would allow the agent to reflect upon their own knowledge in a way that is currently not available in state of the art differentiable ILP systems such as δILP, DRLM, and NTP. The trade off we will make here will be reduced computational complexity by relaxing our correspondence with only the weak extensions of default logic. These weak extensions may not always be ideal due to the fact that atoms in the model may only be explained due to circular arguments. An example of this can be seen from the following program where the supported models are $\{p, q\}$ and $\emptyset$ however the only stable model is $\emptyset$. One may argue that given no external observation the only model we should be interested in is the $\emptyset$.

Example 6.1.

\begin{align}
& p \leftarrow q \quad \text{(6.1)} \\
& q \leftarrow p \quad \text{(6.2)}
\end{align}

However, by adopting the supported model semantics we are also able to leverage a continuous relaxation of the same notion in an intuitive way using fixed point operators. In what follows, the continuous relaxation of ‘supportedness’ will be used to construct a new ILP system capable of reasoning with interpretations $I \in [0, 1]^n$ where the task is reformulated.
as an optimisation problem.

**Definition 6.2. Supported Model.** Let $P_W$ be a normal program and $I \in [0,1]^n$ be an interpretation. Then using the one-step deduction mapping $g$ for program $P$ we have that $g(I) = I$ if and only if $I$ is a supported model.

**Definition 6.3. Supported Model Loss $SupM_{P_W}(.)$.** Given a program $P$, the fuzzy measure of a supported model is defined as follows. We say that $I$ is supported iff $SupM_{P_W}(I) = 0$.

\[
SupM_P(M) : [0,1]^n \rightarrow \mathbb{R} \\
SupM_P(M) = ||I - g(I)||^2
\]

The above notion of the ‘supportedness’ of an interpretation under a given weighted program can intuitively be thought of as having a stable degree of belief in every atom such that there is no change in your belief of any atom after applying the rules of the program which describe the different interactions of the atoms. The supported model loss is then a monotonically increasing loss that also has the property of Satisfaction from theorem 2.21.

**Example 6.4.** All supported models of the program in Example 6.1 assign the same value to both $p$ and $q$. Clearly, any valuation $v$ of $p$ and $q$ is a supported model of this program iff $v(p) = v(q)$ as our degree of belief in the atoms does not change.

Unfortunately extending the continuous relaxation to stable model semantics is not so clear, as from some of its known equivalent definitions given below a naive solution for a continuous relaxation would require checking the minimality of a model, finding a well-founded ordering over the grounded terms, or proving the interpretation does not contain an atom from an unfounded set. The first of these options does not easily lend itself to a differentiable implementation. However, finding a well-founded ordering over the grounded terms could be posed as a machine-learned ranking (MLR) problem for which there are a number of evaluation measures that are commonly used to reformulate the problem as an optimisation problem. Additionally, the final option could be posed as an adversarial training problem. Both of these directions are pursued in chapter 8 of this report.

**Definition 6.5. Stable Model.** Given a program $P$, an interpretation $I$ is a stable model iff any of the following holds:

1. $I = LH M(P^I)$.

2. $I$ is a supported model and there exists a well-founded ordering $\prec$ over the grounding such that: If $h \in I$, there exists a rule $r \in P$ such that $body^+(r) \subseteq I$, $body^-(r) \cap I = \emptyset$; and for every $bi \in body^+(r)$ we have $bi \prec h$.

3. $I$ is a model of $P$ and $I$ is unfounded-free. An interpretation $I$ is called unfounded-free if $I \cap U = \emptyset$ for the greatest unfounded set $U$ of $P$ relative to $I$.[EIK09]
6.2 Differentiable Supported ILP Task

Most of the previous work on learning logic programs can be captured by describing them as either brave - where examples should be explained by at least one answer set – or cautious where examples should be explained by all answer sets. Systems such as δILP, DRLM, and CILP++ come under brave learning tasks and as such it is sufficient to provide a ‘dataset’ of labels $\in \{0, 1\}$ for some ground atoms that describe whether they are or aren’t desired consequences of an inductive solution to the learning task. Labels may be grouped into examples which form a partial interpretation alongside some further ‘contextual’ information which is combined with the background knowledge in order to deduce the labelled atoms. For the case of δILP, this is then paired with a cross-entropy loss between the labels and the deductions of the system, thereby reducing it to a classification problem. The cross-entropy function applied in this way specifies the granularity of the loss at the level of ground atoms within each example, and thus by maximising the expected likelihood that the labels are predicted correctly the system is capable of dealing with noise.

Tasks posed as either brave or cautious frameworks are unable to learn hard or weak constraints[Law18]. Thus in the formulation of our dSILP task we will leverage the more general ILP\textsubscript{LAS} task introduced by Mark Law that is able to handle the learning of such rules. Note that the notion of what constitutes a positive and negative example is different here compared to brave and cautious frameworks. We will make use of the following definitions from [Law18] that have been mildly simplified so as to not include ordered examples which are not considered in this report:

\begin{definition}
A partial interpretation $e$ is a pair of sets of atoms $(e^{inc}, e^{exc})$, we refer to $e^{inc}$ and $e^{exc}$ as the inclusions and exclusions respectively. An interpretation $I$ is said to extend $e$ if and only if $e^{inc} \subseteq I$ and $e^{exc} \cap I = \emptyset$.
\end{definition}

\begin{definition}
A context-dependent partial interpretation (CDPI) $e$ is a pair $(e_{pi}, e_{ctx})$, where $e_{pi}$ is a partial interpretation and $e_{ctx}$ is a program called a context. Given a program $P$, an interpretation $I$ is said to be an accepting answer set of $e$ wrt $P$ if and only if $I \in AS(P \cup e_{ctx})$ and $I$ extends $e_{pi}$. $P$ is said to accept $e$ if there is at least one accepting answer set of $e$ wrt $P$.
\end{definition}

\begin{definition}
A Context-dependent Learning from Answer Sets (ILP\textsubscript{context} LAS) task is a tuple $T = \langle B, S_M, \langle E^+, E^-\rangle \rangle$ where $B$ is an ASP program, $S_M$ is a set of ASP rules, and $E^+$ and $E^-$ are finite sets of CDPIs. A hypothesis $H \subseteq S_M$ is an inductive solution of $T$ if and only if:
\begin{enumerate}
\item $\forall e^+ \in E^+, B \cup H$ accepts $e$ \\
\item $\forall e^- \in E^-, B \cup H$ does not accept $e$
\end{enumerate}

In our Supported dILP task defined below we relax the notion of accepting a partial interpretation with our fuzzy measure of ‘supportedness’ and therefore enabling numerical optimisation methods to look for optimal solutions of the task with only one-step deductions from a trainable weighted program.

\begin{definition}
A Supported dILP Task is a tuple $T = \langle B, \Pi, E^+, E^-\rangle$. Where we have that:
\end{definition}
- $B$ is a weighted program
- $\Pi$ is a weighted program template
- $E^+$ is a set of context dependent partial interpretations
- $E^-$ is a set of context dependent partial interpretations

We then define a weighted hypothesis $H \subseteq \Pi$ to be an inductive solution of $T$ if and only if $\neg \exists H' \subseteq \Pi$ such that:

$$\sum_{e^+ \in E^+} l_{H'}(e^+) + \sum_{e^- \in E^-} l_{H'}(e^-) < \sum_{e^+ \in E^+} l_H(e^+) + \sum_{e^- \in E^-} l_H(e^-)$$  \hfill (6.5)

Where the loss for a particular example is defined as:

$$l_P(e) = \begin{cases} \min_{I} SupM_P(I) + extends(I,e), & e \in E^+ \\ \max(0, 1 - \min_{I} SupM_P(I) + exists(I,e)), & e \in E^- \end{cases} \hfill (6.6)$$

$$\text{extends}(I,e) = ||v^T (g(I) - e_{vec})||^2 + ||v^T (e_{vec} - I)||^2 \hfill (6.7)$$

$$v_i = \begin{cases} 1, & i^{th} \text{ground atom is } \in e^{inc} \text{ or } \in e^{exc} \\
0, & \text{otherwise} \end{cases} \hfill (6.8)$$

$$(e_{vec})_i = \begin{cases} 1, & i^{th} \text{ground atom is } \in e^{inc} \\
0, & \text{otherwise} \end{cases} \hfill (6.9)$$

The loss for a positive example incorporates both a fuzzy measure of supportedness and another fuzzy measure of ‘extension’ for an example given an interpretation. Note that the minimisation which occurs in the above definitions is over a continuous space. The definition of $v_i$ helps us to accrue a loss for extension only at the indices which are included in the example, and $e_{vec}$ is a dense vector representation of the partial interpretation $e$. In other words, minimising a positive example loss reflects looking for the existence of an interpretation which extends it and is supported. A similar approach is taken with the negative examples however the loss is capped at 1 as this reflects the need of only one atom being unsupported. This definition is such that it coincides with the $ILP^{context}_{LAS}$ when only considering discrete interpretations and the loss becomes 0. However it will also accept ‘noisy’ solutions to tasks at the granularity level of the ground atoms.

Comparing the above definition to ILASP’s task with noise, it is noteworthy that the losses here can be seen as an extension of their penalty system in a setting with continuous valuations. In the above definition it is assumed that the penalties for all of the examples are equal, however per example penalties may well be provided by a user when specifying the hyperparameters of the training procedure. This may be of use when examples are particularly imbalanced in the training set. The optimised loss for an example $l_P(e)$ as described above was originally as defined below which we call $l'_P(e)$, which omits a fuzzy definition for ‘extension’. This proved faster in finding solutions to tasks without any noise however it proved less successful in finding a solution at all when noise was introduced to the learning tasks.

$$l'_P_W(e) = \begin{cases} \min_{I \text{ extends } e^+} SupM_P(I), & e \in E^+ \\ \max(0, 1 - \min_{I \text{ extends } e^-} SupM_P(I)), & e \in E^- \end{cases} \hfill (6.10)$$
6.2.1 Training

The dSILP learning framework was implemented in tensorflow\cite{AAB15} in order to leverage the vast amounts of improvements the machine learning community has made in their numerical optimisation software. The evaluation of the dSILP learning framework has been limited in this report to considering only positive examples and this has followed the form of the first algorithm presented below. In the second formulation, where negative examples are introduced, we require adversarial training of the negative examples such as is found when training the minimax of a two player adversarial game in Generative Adversarial Networks\cite{GPAM14}. Generally speaking, GAN’s are much harder to train and require more fine-tuning of the hyperparameters involved in a training task such that the ‘Generator’ and the ‘Discriminator’ don’t overpower each other. The algorithm to deal with negative examples that has been listed below is yet to be evaluated. Note that in both algorithms, the grounding of the examples happens in parallel, creating a trainable interpretation for each example and then following a similar format to the Coordinate ascent variational inference (CAVI)\cite{BKM17} algorithm which first updates the local parameters (which here are the interpretations for each example), and then the global parameters (here the weights of the weighted program). The relevant grounding is computed for each example by finding the fixed point of the operator defined in definition 5.6 using the background knowledge $B$ and the example context $e_{ctx}$. Also the weights for both the weighted program and the interpretations for each example are initialised using a uniform distribution over $[0, 1]$ and the values are capped to stay within the same interval after gradient updates.

Algorithm 1 Supported Model Induction with no Negative Examples

1: \textbf{procedure} supportedILP$(< B, \Pi, E^+, \emptyset >)$
2: \hspace{1cm} $W \leftarrow \text{randomInitialisation}()$
3: \hspace{1cm} \textbf{for} $e^+ \in E^+$ \textbf{do}
4: \hspace{2cm} $HB_{e^+}^{rel} \leftarrow \text{relevantGrounding}(B, e^+)$
5: \hspace{2cm} $I_{e^+} \leftarrow \text{randomInitialisation}()$
6: \hspace{1cm} $t \leftarrow 0$
7: \hspace{1cm} \textbf{while} $t < EPOCHS$ \textbf{do}
8: \hspace{2cm} \textbf{for} $e^+ \in E^+$ \textbf{do}
9: \hspace{3cm} $I_{e^+} \leftarrow I_{e^+} - \eta_1^t \nabla I_{e^+} l_{PW}(e^+))$
10: \hspace{2cm} $W \leftarrow W - \eta_2^t \nabla_W \left( \sum_{e^+ \in E^+} l_{PW}(e^+)) \right)$
11: \hspace{1cm} $t \leftarrow t + 1$

In the above formulation of the procedure, all of the examples are considered before updating the global program weights $W$. However, this is not necessary and stochastic updates could be made by uniformly sampling from the set of examples within the body of the while loop. To support training of negative examples the algorithm will need to add further auxiliary variables into the above procedure for each negative example and train these by applying gradient ascent on their loss. The hyperparameters $\eta_1^t, \eta_2^t$, and $EPOCHS$ define the learning rates for the positive interpretations, the program weights, and the number of steps to run over the training dataset respectively.
Algorithm 2 Supported Model Induction with Negative Examples

1: procedure SUPPORTEDILP(<B, Π, E⁺, E⁻>)
2: W ← randomInitialisation()
3: for e⁺ ∈ E⁺ do
4:     HB⁺el ← relevantGrounding(B, e⁺)
5:     Ie⁺ ← randomInitialisation()
6: for e⁻ ∈ E⁻ do
7:     HB⁻el ← relevantGrounding(B, e⁻)
8:     Ie⁻ ← randomInitialisation()
9: t ← 0
10: while t < EPOCHS do
11:     for e⁺ ∈ E⁺ do
12:         Ie⁺ ← Ie⁺ − η₁⁻∇₁IpW(e⁺))
13:     for e⁻ ∈ E⁻ do
14:         Ie⁻ ← Ie⁻ + η₂⁻∇₂IpW(e⁻))
15:     W ← W − η₃⁻∇₃W (∑e⁺∈E⁺ IpW(e⁺) + ∑e⁻∈E⁻ IpW(e⁻))
16:     t ← t + 1

For the experiments that will follow, which evaluate the dSILP learning algorithm, we employ the ADAM[KB14] optimiser for the gradient updates η in the above algorithms and keep a fixed learning rate of 5e-2. The number of EPOCHS is also fixed at 350. Furthermore, as a first defensive counter-measure to learning supported models which are not stable, we add an Elastic-Net regularisation[ZH05] (which is a linear combination of both ridge and lasso penalties) on the learned interpretation for each example. We also add the same regularisation on the weights of the learned program to encourage finding a binary solution and prefer hypotheses with ‘empty’ clauses inspired by the type of hypotheses ILASP prefers.

6.2.2 Memory Requirements

The memory requirements for the above dSILP algorithm is (where |Rᵢ| denotes the number of ground instances generated by the 𝑖ᵗʰ set of clause templates) the following number of floats:

$$O\left(\sum_{Rᵢ∈Π}|Rᵢ| + \sum_{e⁺∈E⁺} HB⁺el + \sum_{e⁻∈E⁻} HB⁻el\right)$$  \hspace{1cm} (6.11)

This can be seen as a vast improvement over the memory requirements of δILP (for reasonable numbers of examples) which is[EG17]:

$$O\left(T \cdot |HB⁺el| \cdot \sum_{i} |cl(τⁱ¹)| \cdot |cl(τⁱ²)|\right)$$  \hspace{1cm} (6.12)

Where T represents the number of forward chaining steps performed, |cl(τⁱ¹)| and |cl(τⁱ²)| represent the number of clauses generated by the corresponding template for the first and second clause of the 𝑖ᵗℎ predicate respectively, and |Pᵢ| denotes the number of intensional predicates in used.
Chapter 7

Supported dILP Evaluation

The tasks that dSILP was evaluated on are presented here alongside the rule templates used to learn the corresponding solutions. The learned solutions were always taken as the clauses with the highest weights in each clause set such that the sum of the clause weights from the chosen clauses in each set exceeded 0.9. This crude method for sampling a discrete hypothesis from the learned weighted program works surprisingly well in all of the examples that follow. We leave the formulation of a model selection criterion from a given learned weighted program to future work.

7.1 Grandparent

The task here is to learn the non-recursive grandparent relation given the notion of mother and father relations. The exclusions in the single positive example of this task consist of all ground target relations which were not in the inclusions. The context for the example is also empty.

- $e_{inc}^1 = \{ \text{target}(i, b), \text{target}(i, c), \text{target}(a, d), \text{target}(a, e), \text{target}(a, f), \text{target}(a, g), \text{target}(c, h) \}$
- $E^+ = \{ \langle e_{inc}^1, e^{exc}_1 \rangle, e^{dir}_1 \}, E^- = \emptyset$
- $\tau_1 = \langle \text{target}/1, \{ \text{mother}/2, \text{father}/2 \}, 2, 0, 2 \rangle$
- $R = \{ \{ \tau_1 \}, \{ \tau_1 \}, \{ \tau_1 \}, \{ \tau_1 \} \}$
- $B = \{ \text{mother}(i, a), \text{father}(a, b), \text{father}(a, c), \text{father}(b, d), \text{father}(b, e), \text{mother}(c, f), \text{mother}(c, g), \text{mother}(f, h) \}$
- $\Pi = \langle R, B \rangle$

A learned solution is:

\[
\begin{align*}
\text{target}(A, B) &\leftarrow \text{mother}(A, C), \text{mother}(C, B). \\
\text{target}(A, B) &\leftarrow \text{mother}(A, C), \text{father}(C, B). \\
\text{target}(A, B) &\leftarrow \text{mother}(C, B), \text{father}(A, C). \\
\text{target}(A, B) &\leftarrow \text{father}(A, C), \text{father}(C, B).
\end{align*}
\]
CHAPTER 7. SUPPORTED DILP EVALUATION

7.2 Adjacent to Red

This task aims to learn the predicate is adjacent to a red node in a graph with red and green nodes through the invention of a new predicate helper. Again, the exclusions for each positive example consist of the ground target atoms which are not in the inclusions.

- $e_{1\text{inc}}^{\text{inc}} = \{\text{target}(b), \text{target}(c)\}$
- $e_{1\text{exc}}^{\text{exc}} = \{\text{edge}(a,b), \text{edge}(b,a), \text{edge}(c,d), \text{edge}(c,e), \text{colour}(a,\text{red}), \text{colour}(b,\text{green}), \text{colour}(c,\text{red}), \text{colour}(d,\text{red}), \text{colour}(e, \text{green})\}$
- $e_{2\text{inc}}^{\text{inc}} = \{\text{target}(b), \text{target}(d)\}$
- $e_{2\text{exc}}^{\text{exc}} = \{\text{edge}(b,c), \text{edge}(d,c), \text{edge}(c,d), \text{edge}(c,e), \text{colour}(a,\text{red}), \text{colour}(b,\text{green}), \text{colour}(c,\text{red}), \text{colour}(d,\text{red}), \text{colour}(e, \text{green})\}$
- $E^{+} = \{\langle\langle e_{1\text{inc}}^{\text{inc}}, e_{1\text{exc}}^{\text{exc}}\rangle, e_{2\text{exc}}^{\text{exc}}, e_{2\text{exc}}^{\text{exc}}\rangle,\}, E^{-} = \emptyset$
- $\tau_{1} = \langle\text{target}/1, \{\text{edge}/2, \text{colour}/2, \text{green}/1, \text{red}/1, \text{helper}/1\} \rangle, 2, 0, 2\rangle$
- $\tau_{2} = \langle\text{helper}/1, \{\text{edge}/2, \text{colour}/2, \text{green}/1, \text{red}/1\} \rangle, 2, 0, 2\rangle$
- $R = \{\{\tau_{1}, \tau_{2}\}, \{\tau_{1}, \tau_{2}\}\}$, $B = \{\text{red}(\text{red}), \text{green}(\text{green})\}$
- $\Pi = \langle R, B \rangle$

A learned solution is:

\[
\text{target}(A) \leftarrow \text{edge}(A,B), \text{helper}(B).
\]
\[
\text{helper}(A) \leftarrow \text{colour}(A,B), \text{red}(B).
\]

7.3 Even-Odd

This task aims to show the capability of dSILP on tasks where $\delta$ILP would have required a large number of forward chaining steps $T$. The task as stated below has a larger domain in comparison to the one used by $\delta$ILP. The example context is empty. To avoid learning the trivial supported model solution the helper predicate was not allowed to contain the same variable as the target predicate thereby removing rules such as:

\[
\text{target}(A) \leftarrow \text{helper}(A).
\]
\[
\text{helper}(A) \leftarrow \text{target}(A).
\]

- $e_{1\text{inc}}^{\text{inc}} = \{\text{target}(0), \text{target}(2), \text{target}(8), \text{target}(10), \text{target}(12), \text{target}(18), \text{target}(30), \text{target}(34), \text{target}(38)\}$
- $e_{1\text{exc}}^{\text{exc}} = \{\text{target}(1), \text{target}(3), \text{target}(9), \text{target}(13), \text{target}(17), \text{target}(23), \text{target}(33), \text{target}(37), \text{target}(39)\}$
- $E^{+} = \{\langle\langle e_{1\text{inc}}^{\text{inc}}, e_{1\text{exc}}^{\text{exc}}\rangle, e_{1\text{exc}}^{\text{exc}}, e_{1\text{exc}}^{\text{exc}}\rangle,\}, E^{-} = \emptyset$
- $\tau_{1} = \langle\text{target}/1, \{\text{succ}/2, \text{zero}/1, \text{helper}/1\} \rangle, 2, 0, 2\rangle$
- $\tau_{2} = \langle\text{helper}/1, \{\text{succ}/2, \text{zero}/1, \text{target}/1\} \rangle, 2, 0, 2\rangle$
- $R = \{\{\tau_{1}, \tau_{2}\}, \{\tau_{1}, \tau_{2}\}\}$, $B = \{\text{succ}(1, 0), \ldots \text{succ}(40, 39), \text{zero}(0)\}$
• $\Pi = \langle \text{R, B} \rangle$

A learned solution is:

$$\text{helper}(A) \leftarrow \text{succ}(A, B), \text{target}(B). \quad (7.3)$$

$$\text{target}(A) \leftarrow \text{succ}(B, A), \text{helper}(B). \quad (7.4)$$

$$\text{helper}(A) \leftarrow \text{zero}(B), \text{succ}(A, B). \quad (7.5)$$

### 7.4 Graph Colouring

This task aims at identifying nodes which are adjacent to a node of the same colour. The example targets and contextual background knowledge is as shown in [EG17]. The helper predicate in this task has an arity of 2 which means more rules would be required to be removed from the hypothesis in order to avoid learning trivial supported model solutions such as:

$$\text{target}(A) \leftarrow \text{edge}(A, B), \text{helper}(B, A).$$

$$\text{helper}(A, B) \leftarrow \text{edge}(B, A), \text{target}(B).$$

We have scored dSILP with a zero for this task in the table below as although it can solve this task by allowing more predicates in the body of the rules in the hypothesis space, the naive rule templates lead to the naive hypothesis which is not acceptable even though its solution produces supported interpretations which extend the examples. Outlining the methods available in order to mitigate this effect is the subject of chapter 8. Using the method of violating hypotheses allows dSILP to successfully run on all of the following tasks (where we note that dSILP is the only system in this list which allows for negation as failure in its invented predicates):

<table>
<thead>
<tr>
<th>Domain/Task</th>
<th>dILP</th>
<th>Metagol</th>
<th>Supported dILP</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arithmetic/Predecessor</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Arithmetic/Even</td>
<td>100</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Arithmetic/Even-Odd</td>
<td>49</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Graph/Adjacent to Red</td>
<td>51</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>Graph Colouring</td>
<td>95</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 7.1: Percentage of runs which achieved less than $1\text{e}^{-4}$ MSE

### 7.5 Noise

In this section we compare how well dSILP learns the member predicate - under the presence of noise - which describes the ‘membership’ concept of a list. We use the following program template and leave the rest unchanged from the task as defined in [EG17]. Note that we allow negative body literals in this task.

• $\tau_1 = \langle \text{member}/1, \{ \text{head}/2, \text{tail}/1, \text{helper}/1 \} , 2, 1, 2 \rangle$
• $\tau_2 = \langle \text{helper}/1, \{ \text{head}/2, \text{tail}/1 \}, 2, 1, 2 \rangle$

• $R = \{ \{\tau_1, \tau_2\}, \{\tau_1, \tau_2\}, \{\tau_1, \tau_2\} \}$

Figure 7.1: MSE of predicted labels as a function of Noise

In [LRB18b] it is argued that when the noisy examples outnumber the correctly labelled examples, the learner should start learning the negation of the target hypothesis; and therefore we should see the MSE stay near 0 until the 50% mark and then the MSE should become 1. Such an ideal outcome would be possible only by preferring hypotheses which are in some ways ‘better’ than others by applying a pre-agreed upon method of determining what constitutes a ‘better’ hypothesis such as Occam’s Razor. ILASP3 directly encodes this into its search for a solution by including the size of the hypothesis in the sum of the penalties of its uncovered examples. dSILP has yet to consider penalties for larger hypothesis however these could be included with a simple amendment to its loss function. Which hypothesis is ‘best’ is the subject of active research and indeed we note here that in fact yet another method available to distinguish candidate hypotheses could be to prefer hypotheses which contain rules which follow common formats that we expect to see by writing metarules such as those used by Metagol.

By the above criterion, dSILP appears to have a very good MSE as a function of the proportion of noise in the examples, even without the consideration of penalties for larger hypotheses or preferred rule formats. Instead what has happened here is that dSILP has been fortunate and found a local optimum that corresponds to the intended relation we were looking to find and has been unable to fit to the noise. Conversely ILASP3, an arguably more powerful system, has been able to overfit to the noise and thus boasts a more linear MSE as a function of the noise. In the above experiment ILASP3 weights the examples in one class (positive or negative) by the proportion of the whole set of examples which are in the other class times a user specified constant. They chose ‘100’ for this constant and this is what I believe results in such a linear function as this leads to preferring coverage of the examples far more than the hypothesis length. dSILP currently finds and returns the first local optimum it finds which can be seen as a greedy approach.
for choosing a preferred hypothesis, bypassing the need to explicitly compare hypotheses, whereas ILASP3 directly makes a comparison between two competing hypotheses. I expect that by employing multiple random restarts and using a model selection criterion which chooses the hypotheses which reduces dSILP’s loss function would see its function align closer to that exhibited by ILASP3.

Notably, δILP is unable to breach an MSE much higher than 0.2 even when all of the examples are labelled in the opposite sense. Again, this was noted in [LRB18b] as an artifact of their system being unable to express negation as failure. dSILP is able to express negation as failure and, as can be seen by some of the hypotheses learned for some of the levels of noise in Appendix A, the learned hypotheses leverage this ability in order to produce the concept of not being a member of a list.

Below we show the entropy of the learned rules as a function of the proportion of noise in the examples. Entropy can be seen as a measure of the fuzziness of the learned rules or a measure in the uncertainty of the learned hypotheses provided by the system. We can see that it takes the largely expected shape of being most confident when there is least or most noise. The entropy of the learned hypothesis is found by summing over the entropy of the distribution for each clause set which is usually denoted by $H$ and is defined as:

$$
\text{Entropy } H(X) = - \sum p(X) \log p(X)
$$

Figure 7.2: Entropy of the rule weights as a function of Noise
Chapter 8

Stable ILP

This chapter will continue the underlying methodology of identifying a collection of preferred models in a relaxed continuous setting. The preferred models we seek to find in this chapter will be stable.

The structure will follow three orthogonal directions taken in an attempt to meet this end: the first will formulate an objective function based on the equivalence between stable models and well-ordered supported models leveraging learning-to-rank techniques developed by the machine learning community; the second will again build a new objective function, this time based on the equivalence with unfounded-free models; and finally, without resorting to a new objective function, a method inspired by the violating hypotheses notion introduced by Mark Law[Law18] will be pursued.

8.1 Pairwise Scoring for Stable Models

Taking definition 6.5 as our starting point, we can see that the rules in a hypothesis can be used to construct a partial ordering over the grounded atoms in Herbrand Universe and thus this lends itself to a supervised learning-to-rank setting. By leveraging a pairwise loss for each positive body in a rule alongside a pointwise scoring function we will therefore be able to determine a new differentiable satisfiability score for a weighted hypothesis that will only become zero for stable hypotheses.

I chose to adapt the logistic loss of a GSF (Group Scoring function)[AWG+18] as presented in the pairwise setting that is also very similar to the one used in LambdaMART[Bur10]. Note that there are a number of different pairwise loss functions such as ‘hinge loss’ which could also be easily adapted to the following setting.

**Definition 8.1. Pairwise Logistic Loss.** Given a grounded head \( h \) and a positive body literal \( b^+ \) where \( s_h \) and \( s_{b^+} \) represent their respective pointwise relative scores then we define the pairwise logistic loss for \( h \prec b^+ \) as:

\[
pl(h, b^+) = \log(1 + e^{-(s_h - s_{b^+})})
\]  

(8.1)

This pairwise logistic loss can then be coupled with the supported model loss such that we gain a new satisfiability score for weighted hypotheses. The only difficulty that arises
when constructing this new score function is the requirement that for each atom in the model we only need the existence of a rule in the hypothesis that entails it and satisfies the well-ordering relation by definition 6.5. I now show how this can be formulated using the pairwise logistic loss from the machine learning-to-rank literature.

**Definition 8.2. Stable Model Loss.** Given a model \( I \in [0,1]^n \), a weighted program \( P_W \), and a scoring function \( f \) which maps ground atoms to a score in \( \mathbb{R} \), we define the stable model loss of \( I \) with respect to \( P_W \) as:

\[
sm_{PL}(I) = \sum_{m_i \in I} \prod_{w : r \in P_W} \left( |m_i - o_{i,r}|^2 \oplus (m_i \cdot \sum_{b^+ \in r} pl(h, b^+)) \right) \tag{8.2}
\]

Where \( o_{i,r} \) is defined as the one step deduction for the weighted program which contains only the grounded rule \( r \), \( w : r \in P_W \) is taken to mean any weighted rule in any of the sets of clauses of \( P_W \), and \( \oplus \) is defined by the probabilistic sum.

**Theorem 8.3.** A model \( I \in \{0,1\}^n \) is stable iff there exists a scoring function \( f \) such that the sum \( sm_{PL}(I) + SupMP_W(I) = 0 \).

**Proof.**

\( \Rightarrow \) The forward direction is trivial by setting the scoring function to one that satisfies the well-ordering which exists by definition 6.5 and then taking the limit as the difference between the scores of the atoms tends to infinity.

\( \Leftarrow \) All of the elements of the sum are non-negative and therefore they must all be equal to zero. This means that for each \( m_i \in I \) there must exist at least one rule \( r \) in \( P_W \) such that both elements of the probabilistic sum are equal to zero. The first element being zero implies that the rule supports the belief of that atom in the model. The second element being zero implies that either \( m_i \) is 0 or that \( m_i \) is 1 and all of the positive body elements satisfy the well ordering \( \prec \) by using the scoring function \( f \). By definition 6.5, I must therefore be a stable model.

From the above theorem we could now formulate a new optimisation problem by adding some appropriate scoring function for the grounded atoms. The approach taken here would be to store a further trainable variable that will hold the score for each trainable grounded atom in the Herbrand Base. The scoring function will fetch the score of a trainable grounded atom from the appropriate index and will assume a score of \(-\infty\) for every extensional predicate. In the simplified setting when the background only contains extensional predicates this will be sufficient for creating a scoring function that satisfies a well-ordering over the rules in a program if a well-ordering exists. Note also that the inductive solutions that minimise the sum of the supported model loss and the stable model loss correspond to inductive solutions of an ILASP when the task is satisfiable in the hypothesis space and there is no noise.
8.2 Adversarially Training Unfounded Sets

In this section we focus our attention to the third point in definition 6.5. Intuitively, taking the GAN perspective, the problem will be modelled by an unfounded set generator and a stable model discriminator. The generator will train the unfounded set such that it minimises the probability of the stable model and similarly the discriminator will maximise the probability that the model is supported and does not intersect with the unfounded set. Given a greatest unfounded set $U$ we will define the probability that $I$ is stable as follows:

$$P(I \text{ stable}) = P(I \text{ supported}) \ast P(|I \cap U| = \emptyset)$$  \hspace{1cm} (8.3)

$$= P(I \text{ supported}) \ast P(\text{no element in both})$$  \hspace{1cm} (8.4)

$$= P(I \text{ supported}) \ast \prod_i (1 - P(m_i \in U)P(m_i \in I))$$  \hspace{1cm} (8.5)

The above definition would require a method for turning the supported model loss of dSILP into a probability distribution over the interpretations. One way of doing this is shown at the start of chapter 10.

8.3 Removing Solutions with Violating Hypotheses

Arguably the simplest solution to the task of removing supported models which are not stable is to allow the search over all supported models as in chapter 6 and then check the solutions validity under the stable model semantics. If it is not valid, remove the hypothesis from the search space by adding a constraint to the model and continue the search. This method was inspired by the notion of violating hypotheses used in ILASP1 as it also uses a solver that is suited for a simpler task and then continually filters out solutions which are not valid.

It should be noted that this method is not as desirable as the aforementioned methods of modifying the supported model task to the stable model semantics when considering the use of neural predicates. This is because this method will traverse the search space in largely varying directions when it is forced to add a constraint to the model, meaning that jointly trained neural predicates may have to be largely retrained on each re-start following a failed solution. This is as opposed to the previous methods which would take a more incremental approach. However, this also comes with some positive side effects such as the ability to recover from local optima.
Chapter 9

Discussion of Future Work

9.1 Probabilistic Interpretation

The unsuitability of a probabilistic interpretation of the forward chaining inference mechanism as described by δILP has already been exemplified in subsection 4.2.2. For a probabilistic interpretation of the consequences of a program to be valid we may look towards Causal Probabilistic (CP) logic\cite{VDB09} for a new definition of the forward chaining procedure so that 4.2 produces the expected answer in future work.

In order to be able to perform bayesian model selection over candidate hypotheses we would also require a probabilistic interpretation of the likelihood of the data under a given hypothesis, which in our case would be a weighted program. Following a similar construction to that found in Markov Logic Networks, we may weight an interpretation by our loss function for supported models by:

\[
P(I|P_W) = \frac{1}{Z} \exp(-\alpha \text{SupM}_{P_W}(I)) \]

\[
\propto \exp(-\alpha \text{SupM}_{P_W}(I))
\]

(9.1)

(9.2)

Note that we can recover the discrete definition of a supported model by taking the limit as \( \alpha \to \infty \). Now in order to train weighted programs in order to find the MLE hypothesis for the data we require finding the derivative of an interpretation under this definition:

\[
\frac{d}{dw_i} \log(P(I|w)) = \frac{d}{dw_i} \log(\prod_I \frac{1}{Z} \exp(-\text{SupM}_w(I)))
\]

\[
= - \frac{d}{dw_i} \text{SupM}_w(I) - \frac{d}{dw_i} \log(Z)
\]

\[
= - \frac{d}{dw_i} \text{SupM}_w(I) - \frac{d}{dw_i} \frac{Z}{Z}
\]

\[
= - \frac{d}{dw_i} \sum_{e^+} \text{SupM}_w(I) + \mathbb{E}_I[\frac{d}{dw_i} \text{SupM}_w(I)]
\]

(9.3)

(9.4)

(9.5)

(9.6)

In the Markov Logic literature, \( Z \) is known as the ‘partition function’, which would require calculating a high-dimensional integral in this case. Instead of directly attempting to
calculate Z, Markov Logic Network weight learners employ a few different methods to try and estimate the gradient. The simplest would be to estimate $\frac{d}{dw} \log(Z)$ by using a few samples from the distribution of I using MCMC. This is normally known as contrastive divergence and produces relatively good results for even a small number of samples. We can also combine this probability with the probability that an interpretation intersects with the greatest unfounded set $U$ as shown previously to define a probability distribution for continuous interpretations under the stable model semantics.

To gain intuition as to how the discrete definition of stable and supported model semantics map into this probabilistic interpretation we can look at two simple (unweighted) propositional programs and the contours of the values proportional to their associated probability distributions induced by the definitions above. We see that the modes of the distribution are where the discrete version would be supported or stable.

$$
\begin{align*}
  a &\leftarrow b \\
  b &\leftarrow a
\end{align*}
$$

$$
\begin{align*}
  a &\leftarrow \neg b \\
  b &\leftarrow \neg a
\end{align*}
$$

(a) $P(I|w)$ for program (9.8) 
(b) $P(I|w)$ for program (9.9)

Figure 9.2: $P(I \text{ stable})$ for program (9.8) given the greatest unfounded set $U = \{a, b\}$
CHAPTER 9. DISCUSSION OF FUTURE WORK

9.2 Lazy Grounding

In this section we will describe a modification of the dSILP training procedure which will allow for the algorithm to avoid storing the whole of the relevant grounding in memory. In order to achieve this, we will first look at a recently proposed method which drastically reduces the training time need for deep neural networks on CPU’s. Locally Sensitive Hashes are used to scale training in the Sub-LInear Deep learning Engine (SLIDE) [CMS19]. The intuition behind their success is the exploitation of the sparseness of ‘active’ neurons in each layer of a neural network, especially when activation functions such as RELU are used. The LSH can be used to query only the ‘active’ neurons in a neural network layer and thus bypassing the need for large dense matrix multiplication required when training the networks. The tables are only then updated periodically which allows their method to reap the LSH functions benefit without paying too high a price. They show that with this new training procedure they achieve the same accuracy on a dataset training the same network on a CPU in a fraction of the time it took on a GPU with more commonplace methods.

Inspecting the one-step deduction procedure of the weighted program defined in this report, we can see that \( \max \) is used over all of the groundings of a given rule that unify to the same head. This can arguably be the sparsest activation procedure possible and therefore keeping only some of the rules with the largest one-step deduction values for each ground head could reduce memory significantly. In this way we may also avoid computing the relevant grounding of a weighted program upfront and instead rely on only constructing a uniform sampler over the groundings of rules in a weighted program. This would allow computing the solution for tasks with large groundings, such as the ‘member’ task considered earlier in this report, when the list length is large. We note that the size of the grounded rules increases at an alarming rate with the length of the list in this task as it is considering all of the permutations of the lists:

<table>
<thead>
<tr>
<th>List Length</th>
<th>Rules</th>
<th>Grounded Rules</th>
<th>Trainable Herbrand Base</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>929</td>
<td>275</td>
<td>4</td>
</tr>
<tr>
<td>2</td>
<td>929</td>
<td>2,170</td>
<td>20</td>
</tr>
<tr>
<td>3</td>
<td>929</td>
<td>15,309</td>
<td>96</td>
</tr>
<tr>
<td>4</td>
<td>929</td>
<td>147,620</td>
<td>520</td>
</tr>
<tr>
<td>5</td>
<td>929</td>
<td>2,525,215</td>
<td>3,260</td>
</tr>
</tbody>
</table>

Figure 9.3: Increasing Memory Requirements for ‘Member’ Task

An example trainable Herbrand Base for the target predicate when the list length is 2 is shown below. The trainable Herbrand base also includes similar instances for the helper predicate and thus the full Trainable Herbrand Base is double what is shown here:

\[
\begin{align*}
\text{target}(0, ()) & \quad \text{target}(0, ()) \\
\text{target}(0, (0,)) & \quad \text{target}(0, (0,)) \\
\text{target}(0, (1,)) & \quad \text{target}(0, (1,)) \\
\text{target}(0, (0,1)) & \quad \text{target}(0, (0,1)) \\
\text{target}(0, (1,0)) & \quad \text{target}(0, (1,0))
\end{align*}
\]
In the context of this simpler task it is not so clear *why* we may be so interested in increasing the capability of the system to larger domains directly. One may assume that solving the task in the smaller domain and then extrapolating that hypothesis to the larger domain would be sufficient. However, this leads to bias in the proposed solutions such that they may not generalize correctly, much like overfitting a neural network decimates their usefulness. An example biased hypothesis for ‘member’ when lists are of length 2 is shown below:

\[
\text{target}(A, B) \leftarrow \text{head}(A, B), \text{num}(A), \text{list}(B).
\]
\[
\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(C, B), \text{num}(A), \text{list}(B).
\]

Thus we would like to train over larger domains if possible in order to reduce the likelihood of bias in our induced solutions. Also, if we were to try and consider an iterative method which incrementally increases the domain size, then this would be the same as employing a coverage loop approach over the smaller examples first. Coverage loop approaches do not work for non-monotonic learning and therefore we must revert back to considering all of the examples jointly.

### 9.3 Hard Constraints

Most of the experiments in this work have been concerned with only positive examples, which map onto a *brave* learning task’s semantics - that a learned program should have at least one interpretation that covers the example. The results put forward by Law et al. in [Law18] about the generality of learning frameworks prove that no matter the examples given, brave induction systems cannot learn hard (or weak) constraints. A detailed evaluation of how well dSILP learns constraints using negative examples will be the subject of future work.
Chapter 10

Conclusion

The main contribution of this work has been to create a differentiable implementation of non-monotonic Inductive Logic Programming coupled with a corresponding well-defined learning task. The initial background research lead to critical insights about existing ILP systems and in an attempt to mitigate the shortcomings of these systems we were able to create a more powerful differentiable system with lower memory requirements.

We showed that dSILP was capable of reasoning in the presence of noise and also capable of learning hypotheses containing negative body literals. However, the selection criterion for the final solution to the system that was used was brittle and thus further evaluation of this method is required on larger numbers of tasks. Additionally, we showed that supported model semantics sometimes lead to lower quality hypotheses and presented a mixture of regularisation heuristics and three other methods in order to tackle the more favourable stable model semantics.

We noted that a current bottleneck of dSILP is the computation of the consequences from all of the relevant groundings of each rule and proposed a stochastic variation of the algorithm that keeps only the ‘active’ rules for each ground head to potentially mitigate this problem. In future work, we would also like to consider a continuous relaxation of the hypothesis space so that the combinatorial explosion created when using large rule templates does not cause another computational bottleneck.

There is large scope for further research in this area and in particular in further extensive evaluation of the dSILP system as presented here with the addition of any (or many) of the proposed amendments to the system that aim to increase the quality of the solutions of the system or the capability of the system under larger datasets and probabilistic tasks.
Appendix A

Learned Hypotheses

Here we present most of the hypotheses learned by dSILP for varying amounts of noise annotated with the probabilities of each rule belonging to each clause. We choose to show all of the rules for which the algorithm finds rules which have a probability of greater than 0.01. We note that some of the hypotheses for mostly noisy examples describe that the target predicate is true when there exists a bigger list with the the element in it. This was true in the tested domain, as all of the permutations of the lists of length 4 were used, without repeats.

0% Noise:

<table>
<thead>
<tr>
<th>Clause0</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>$target(A, B) \leftarrow head(C, B), helper(A, B), num(A), list(B)$</td>
<td>0.97</td>
</tr>
<tr>
<td>$helper(A, B) \leftarrow tail(C, B), helper(A, C), num(A), list(B)$</td>
<td>0.99</td>
</tr>
</tbody>
</table>

5% Noise:

<table>
<thead>
<tr>
<th>Clause0</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>$target(A, B) \leftarrow tail(C, B), not helper(A, B), num(A), list(B)$</td>
<td>0.9836247</td>
</tr>
<tr>
<td>$helper(A, B) \leftarrow tail(C, B), helper(A, C), num(A), list(B)$</td>
<td>0.7687759</td>
</tr>
<tr>
<td>$helper(A, B) \leftarrow head(A, C), tail(C, B), num(A), list(B)$</td>
<td>0.1165921</td>
</tr>
<tr>
<td>$target(A, B) \leftarrow head(C, B), not helper(A, B), num(A), list(B)$</td>
<td>0.7688644</td>
</tr>
<tr>
<td>$helper(A, B) \leftarrow head(A, C), tail(B, C), num(A), list(B)$</td>
<td>0.22401236</td>
</tr>
</tbody>
</table>
10% Noise:

\[ \begin{align*}
\text{Clause0 :} & \quad 0.6074169 \ target(A, B) \leftarrow \ head(A, B), target(C, B), num(A), list(B). \\
& \quad 0.3619868 \ target(A, B) \leftarrow \ tail(C, B), target(A, C), num(A), list(B). \\
\text{Clause1 :} & \quad 0.49559215 \ target(A, B) \leftarrow \ head(A, C), tail(C, B), num(A), list(B). \\
& \quad 0.37249914 \ target(A, B) \leftarrow \ tail(C, B), target(A, C), num(A), list(B). \\
\text{Clause2 :} & \quad 0.3728277 \ target(A, B) \leftarrow \ tail(C, B), target(A, C), num(A), list(B). \\
& \quad 0.35643253 \ target(A, B) \leftarrow \ head(A, B), tail(B, C), num(A), list(B). \\
\end{align*} \]

15% Noise:

\[ \begin{align*}
\text{Clause0 :} & \quad 0.69248027, target(A, B) \leftarrow \ tail(C, B), target(A, C), num(A), list(B). \\
& \quad 0.23997092, target(A, B) \leftarrow \ head(A, C), tail(C, B), num(A), list(B). \\
\text{Clause1 :} & \quad 0.29156426, helper(A, B) \leftarrow \ head(A, C), empty(B), num(A), list(B). \\
& \quad 0.2240775, target(A, B) \leftarrow \ head(A, B), tail(B, C), num(A), list(B). \\
& \quad 0.053254392, helper(A, B) \leftarrow \ head(A, C), tail(B, C), num(A), list(B). \\
\text{Clause2 :} & \quad 0.9259534, helper(A, B) \leftarrow \ tail(C, B), helper(A, C), num(A), list(B). \\
\end{align*} \]

25% Noise:

\[ \begin{align*}
\text{Clause0 :} & \quad 0.3478714, target(A, B) \leftarrow \ head(A, B), tail(B, C), num(A), list(B). \\
& \quad 0.084411785, target(A, B) \leftarrow \ tail(C, B), target(A, C), num(A), list(B). \\
& \quad 0.030091405, helper(A, B) \leftarrow \ head(A, C), tail(C, B), num(A), list(B). \\
\text{Clause1 :} & \quad 0.44112766, helper(A, B) \leftarrow \ head(A, C), empty(B), num(A), list(B). \\
& \quad 0.43746066, target(A, B) \leftarrow \ tail(C, B), target(A, C), num(A), list(B). \\
\text{Clause2 :} & \quad 0.9078862, helper(A, B) \leftarrow \ tail(C, B), helper(A, C), num(A), list(B). \\
& \quad 0.012129607, helper(A, B) \leftarrow \ head(A, C), tail(C, B), num(A), list(B). \\
\end{align*} \]
40% Noise:

Clause0:

- 0.30517012  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.2653618  \(\text{target}(A, B) \leftarrow \text{tail}(C, B), \text{not target}(A, C), \text{num}(A), \text{list}(B)\).
- 0.12434368  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)\).
- 0.06616606  \(\text{target}(A, B) \leftarrow \text{tail}(B, C), \text{not target}(A, C), \text{num}(A), \text{list}(B)\).
- 0.05126519  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{tail}(C, B), \text{num}(A), \text{list}(B)\).

Clause1:

- 0.32988453  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.2664807  \(\text{target}(A, B) \leftarrow \text{tail}(C, B), \text{not target}(A, C), \text{num}(A), \text{list}(B)\).
- 0.12575358  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)\).
- 0.054924466  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{tail}(C, B), \text{num}(A), \text{list}(B)\).

Clause2:

- 0.31405988  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.26550597  \(\text{target}(A, B) \leftarrow \text{tail}(C, B), \text{not target}(A, C), \text{num}(A), \text{list}(B)\).
- 0.13254526  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)\).
- 0.052161776  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{tail}(C, B), \text{num}(A), \text{list}(B)\).

50% Noise:

Clause0:

- 0.36651373  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.3509676  \(\text{target}(A, B) \leftarrow \text{head}(A, B), \text{target}(C, B), \text{num}(A), \text{list}(B)\).
- 0.19958892  \(\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)\).

Clause1:

- 0.49078137  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)\).
- 0.35995543  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).

Clause2:

- 0.53672636  \(\text{target}(A, B) \leftarrow \text{tail}(C, B), \text{not target}(A, C), \text{num}(A), \text{list}(B)\).
- 0.36350352  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).

75% Noise:

Clause0:

- 0.3176094  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.27159706  \(\text{target}(A, B) \leftarrow \text{tail}(C, B), \text{not target}(A, C), \text{num}(A), \text{list}(B)\).
- 0.23632918  \(\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)\).
- 0.058113694  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)\).

Clause1:

- 0.3148569  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.23966828  \(\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)\).
- 0.09506649  \(\text{helper}(A, B) \leftarrow \text{tail}(B, C), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.08890039  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{tail}(C, B), \text{num}(A), \text{list}(B)\).
- 0.06608643  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)\).
- 0.057594515  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)\).

Clause2:

- 0.31309444  \(\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)\).
- 0.26551005  \(\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)\).
- 0.23757716  \(\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)\).
APPENDIX A. LEARNED HYPOTHESES

85% Noise:

Clause0:
0.6675493 $\text{target}(A, B) \leftarrow \text{not helper}(A, B), \text{num}(A), \text{list}(B)$.
0.30093768 $\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)$.

Clause1:
0.9186152 $\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)$.
0.02662083 $\text{helper}(A, B) \leftarrow \text{head}(A, B), \text{tail}(B, C), \text{num}(A), \text{list}(B)$.
0.011491848 $\text{helper}(A, B) \leftarrow \text{head}(A, B), \text{head}(A, C), \text{num}(A), \text{list}(B)$.

Clause2:
0.5495823 $\text{target}(A, B) \leftarrow \text{not helper}(A, B), \text{num}(A), \text{list}(B)$.
0.41875938 $\text{helper}(A, B) \leftarrow \text{head}(A, B), \text{head}(C, B), \text{num}(A), \text{list}(B)$.
0.017307522 $\text{helper}(A, B) \leftarrow \text{head}(A, B), \text{helper}(C, B), \text{num}(A), \text{list}(B)$.

90% Noise:

Clause0:
0.52559495 $\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{empty}(B), \text{num}(A), \text{list}(B)$.
0.123159476 $\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)$.
0.08676846 $\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)$.
0.08011859 $\text{helper}(A, B) \leftarrow \text{tail}(B, C), \text{helper}(A, C), \text{num}(A), \text{list}(B)$.
0.054775406 $\text{helper}(A, B) \leftarrow \text{head}(A, C), \text{tail}(C, B), \text{num}(A), \text{list}(B)$.

Clause1:
0.75306976 $\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)$.
0.08632283 $\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)$.
0.07405825 $\text{target}(A, B) \leftarrow \text{tail}(B, C), \text{target}(A, C), \text{num}(A), \text{list}(B)$.

Clause2:
0.7418968 $\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)$.
0.12239885 $\text{target}(A, B) \leftarrow \text{head}(A, C), \text{tail}(B, C), \text{num}(A), \text{list}(B)$.

100% Noise:

Clause0:
0.9614976 $\text{helper}(A, B) \leftarrow \text{head}(A, B), \text{head}(A, C), \text{num}(A), \text{list}(B)$.

Clause1:
0.9920433 $\text{target}(A, B) \leftarrow \text{empty}(C), \text{not helper}(A, B), \text{num}(A), \text{list}(B)$.

Clause2:
0.99022853 $\text{helper}(A, B) \leftarrow \text{tail}(C, B), \text{helper}(A, C), \text{num}(A), \text{list}(B)$.
Bibliography


