Expressing unknown instance relations in Program Synthesis using neurosymbolic methods

Sondre Bolland

September 28, 2021

Abstract

Program synthesis is the task of automatically constructing a program given a high level specification. An instance of this is Inductive Logic Programming (ILP) were discrete methods are used to construct a logic program which satisfies the specification. A limitation of a traditional ILP system is its inability to handle noise, faultering at a single mislabelled data point. A system which mediates this weakness is Differentiable Inductive Logic Programming (δILP) [5], where instead of satisfying a strict requirement the task is to minimize a loss.

One limitation of δILP is that it does not allow for the use of negation in the construction of its programs. Negation as failure in logic programming is a desired tool to write programs that express unknown knowledge. By extending the system with negation we increase the expressiveness of δILP, allowing us to construct programs that are often easier to devise, write and analyse.

We propose such an extension: Stratified Negation as Failure in Differentiable Inductive Logic Programming (SNAFδILP). This system is able to learn moderately complex programs with unary and binary predicates, negation and predicate invention. The system is fairly robust to mislabelled data, in most cases satisfying the specification with up to 10 % mislabelled data.

Contents

1 Introduction

Program Synthesis is the task of automatically constructing a program which satisfies a high level specification. The problem, as stated by Alonzo Church [7], given a logical relation

$$\phi(x, y)$$

of input x and output y, there should exist a function f which maps x to y

$$\exists f. \forall x \, \phi(x, f(x)).$$

Further, Church posed that it should be possible to algorithmically construct this function.

Program synthesis has been considered the holy grail of Computer Science since the conception of Artificial Intelligence in the 1950s. An instance of this is Inductive Logic Programming (ILP). ILP is a collection of techniques for constructing a logic program from a set of examples. Given a set of positive examples and a set of negative examples an ILP system constructs a logic program which entails all positive and does not entail any of the negative. From a machine learning perspective it can be considered as implementing a rule-based binary classifier.

ILP systems have a set of appealing features, in which neural networks faulter. First, the constructed programs are explicit symbolic structures that can be inspected, understood and validated, while a neural network is a large tensor of floating point numbers, which is not inspectable or human-readable. Second, ILP tends to be impressively data efficient, being able to generalize on only a small set of examples. Neural networks are notorious in their dependency on large data sets. Third, ILP supports transferred learning. A correctly learned program can simply be copied and pasted into a knowledge base, leaving the system ready for more learning. With neural networks transferred learning is possible in cases where the intended function closely relates to what has been learned prior. Still, neural networks require some further engineering to make use of this.

While key strengths of ILP are weaknesses in neural networks, key strengths in neural networks are weaknesses in ILP. The main issue with ILP systems is their inability to handle noise. ILP uses techniques that depend on a strict formal requirement. If only a single data point is mislabelled ILP will fail to construct the intended program. Robustness to noise is a touted strength of neural networks. Even with a sizable proportion of the data being mislabelled, neural networks are often able to generalize.

To combine the strengths of ILP and neural networks one can reimplement ILP in a robust connectionist framework. Such a system is proposed by [5]: **Differentiable Inductive Logic Programming** (δILP). δILP is reimplementation of ILP in an end-to-end differentiable architecture. It attempts to combine the advantages of ILP with the advantages of neural networks: a data-efficient induction system that can learn explicit human-readable symbolic rules, that is robust to noise and ambiguous data, and that does not deteriorate when applied on unseen test data. The central component of this system is a differentiable implementation of deduction through forward chaining on definite clauses. Instead of satisfying a strict requirement, δILP reinterprets the task as a binary classification problem, minimizing cross-entropy loss with regard to ground-truth boolean labels during training.

δILP is able to learn moderately complex programs requiring recursion and predicate invention. It is also able to perform reasonably well with up to 20 % mislabelled data.

In [5] δILP is restricted to only learning definite programs, i.e. logic programs without the use of negation. Expressing unknown knowledge is a highly desired property of logic programming. Normal programs, i.e. programs with the use of negation, are often easier to devise, write and analyse, than definite programs. By use of negation negative knowledge can be expressed by what is already known.

This paper proposes an extended δILP system which allows the construction of programs with the use of negation, more specifically, a system which uses Negation as Failure to construct Stratified Logic Programs. We have aptly named this system SNAFδILP for **Stratified Negation as Failure in Differentiable Inductive Logic Programming**. In [5] the extension of negation as failure is mentioned as further work. In addition, another neural based ILP system: *!!!CAN'T FIND THE PAPER!!!* has this extension planned as a future implementation.

Find this paper

The key difference between δILP and SNAFδILP lies in the ability to construct logic programs with negation by applying a differentiable implementation of deduction through forward chaining using stratified fixpoint semantics instead of definite fixpoint semantics.

SNAFδILP is able to learn moderately complex programs with unary and binary predicates using negation and predicate invention. Unlike traditional ILP systems, SNAFδILP is shown to be moderately robust to mislabelled data, in most cases learning the intended program with up to 10 % mislabelled training data.

The structure of this paper is as follows. Section 2 gives an overview of logic programming, describing different types of logic programs and the semantics used to derive information from them, as well as an overview of the existing δILP system. Section 3 describes our contribution: the extension of δILP with stratified negation as failure. First describing the extension in a traditional ILP system and then a continuous implementation of the same extension. In Section 4 we evaluate our system on a set of learning tasks, testing which implementation of fuzzy negation as failure yields the best result, and how robust our system is to mislabelled data. We complete the paper by discussing improvements, future experiments and further implementations in section

5, and offering our conclusion in section 6.

2 Background

2.1 Logic Programming

Logic programming is a family of languages in which the central component is an If-then rule, known as a clause. A logic Program P is a finite set of such clauses, of which there are two types. The ground unit clauses are the extensional components of the program. They provide us with a set of instances of the program relations. The remaining clauses constitute the intensional component. They are the general rules of the program. The general rules and the explicit data are to be used in deductive retrieval of information [3].

We distinguish between types of logic programs by the construction and order of their clauses. We will consider three types. A **Definite logic program** is a set of definite clauses. A **definite clause** is a disjunction of literals where *exactly* one literal is positive (the rest negative). A **Normal logic program** is a set of normal clauses (often referred to as extended definite clauses). A **normal clause** is a disjunction of literals where at *least* one is positive (the rest negative). The third type we will consider is **Stratified logic programs**. Stratified logic programs are a subset of Normal Logic programs, with a restriction on the clause order and the use of negation. Section 2.3 discusses stratified programs in more detail.

2.1.1 Basic Concepts

Clauses in logic programming are typically written in the form

$$A \leftarrow L_1, ..., L_n \quad n \geq 0$$

read as "If $L_1, \ldots, L_m$ then A". A is a positive literal and each L_i is a literal. A **literal** is an atom (a "positive" literal) or of the form *not B* where B is an atom (*not B* is a "negative" literal). An **atom** α is a tuple $p(t_1, \ldots, t_n)$ where p is a n-ary predicate and $t_1, \ldots, t_n$ are terms, either variables or constants (we will not consider functions). The atom A is the **head** of the clause; the literals $L_1, \ldots, L_n$ are the **body** of the clause. An atom is **ground** if it contains no variables, e.g.:

parent(paul, bob)

Consider a program defining the ancestor relation and the parent relation

parent(paul, bob)

parent(molly, paul)

ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y)

The *parent* relation is defined **extensionaly**, using only ground atoms. The *ancestor* relation is defined **intensionaly,** using a set of non-atomic clauses (a set of one clause in this case).

Variables that appear in the head of a clause are universally quantified, while variables that only appear in the body of the clause are existentially quantified. In classical logic the *ancestor* clause would be written as

$\forall x \forall y (\exists z$ (parent(x,z) ∧ ancestor(z,y))) → ancestor(x,y)

The set of all ground atoms is called the **Herbrand base** $\mathcal{G}$. A **ground rule** is a clause in which all variables have been substituted by constants, e.g.:

ancestor(john,mary) ← parent(john,paul), ancestor(paul,mary) is a ground rule generated by applying the substitution $\theta = \{john/X, mary/Y, paul/Z\}$.

In first order logic an **interpretation** I specifies referents for the elements of our domain (objects and relations among them). I maps constant symbols to objects in the domain, predicate symbols to relations over objects in the domain and function symbols to functional relations over objects in the domain. A **Herbrand Interpretation** is an interpretation in which every constant is interpreted as itself. In a Herbrand interpretation predicate symbols are defined as denoting a subset of the Herbrand base, effectively specifying which ground atoms are true in the interpretation. All ground atoms which are not elements of the Herbrand interpretation are interpreted as false. A **model** in first order logic of a sentence S is an interpretation M such that S is true in M. A **Herbrand Model** is a Herbrand interpretation which is a model [12].

Consider the following program P

$p(a) \leftarrow$

$q(b) \leftarrow$

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

Both

$$I_1 = \{p(a), p(b), q(a)\}$$

and

$$I_2 = \{p(a), q(a), q(b)\}$$

are Herbrand interpretations of P, while only I_2 is a Herbrand model of P. In a Herbrand Model M only ground atoms γ which are elements of M are satisifed.

Definition 2.1 ($\vDash$)**.** *A program P satisfies a ground atom γ if and only if a Herbrand model M of P contains γ:*

$$P \vDash \gamma \textit{ iff } \gamma \in M$$

Definition 2.2 (Supported Model)**.** *A model M of a program P is **supported** if and only if $\forall A \in M$ there is a clause in P of the form*

$$A \leftarrow L_1, ..., L_m,$$

such that $M \vDash L_1, ..., L_m$.

Clauses without a body:

$$A \leftarrow$$

are supported since a model M will always satisfy an empty set of sentences.

Definition 2.3 (Minimal Model)**.** *A model M of a program P is minimal if and only if it has no proper subset that is also a model of P.*

Definition 2.4 (Least Model)**.** *A least model of P is a unique minimal model of P.*

2.1.2 Semantics of Definite Logic Programs

In logic programming we wish to determine entailment of our programs. One way of doing this is using a bottom-up technique called **forward chaining**. Using the clauses of our program P we derive the set of all consequences $con(P)$. The set $con(P)$ is a Herbrand model of P. To determine whether a ground atom γ is entailed by P, $P \vDash \gamma$, we check if $\gamma \in con(P)$. In the case of definite programs the set of consequences will end up being a least Herbrand model, denoted as $con_D(P)$. We properly define this set later in this section. In the case for stratified programs (which will be discussed in Section 2.3) the set of consequences will end up being a minimal and supported Herbrand model, denoted as $con_S(P)$.

To illustrate why we want minimality, consider the program

$$p \leftarrow p.$$

This program has two models $\{p\}$ and the empty set $\varnothing$. $\{p\}$ is not minimal. We rule it out since we cannot prove p using the rules of the program [1].

Support is not important in the case of definite programs, but will be in the case of normal programs (and stratified programs since they are a subset of normal programs). To illustrate why we want supported models, consider the program

$$p \leftarrow \text{not } q.$$

The program has minimal models $\{p\}$ and $\{q\}$, but only $\{p\}$ is supported. We dismiss $\{q\}$ since there is no way of proving q given this program [1].

Let us consider inference in Definite Programs (Normal and Stratified programs will be considered later).

Theorem 2.5. *[12] Let P be a definite program. Then:*

- *The Herbrand base $\mathcal{G}$ is always a model of P*
- *If M_1 is a model of P and M_2 is a model of P, then $M_1 \cap M_2$ is a model of P*
- *P has a model*
- *P has a minimal Herbrand model*
- *P has a least Herbrand model, denoted M_P*
- *M_P = the intersection of all Herbrand models of P*

If we can construct a least Herbrand model of P: M_P, we can determine entailment of a ground atom γ by checking $\gamma \in M_P$.

We can construct M_P by using the immediate consequence operator T_P. T_P maps Herbrand interpretations of P to Herbrand interpretations of P:

$$T_P : I \rightarrow I.$$

Or if you prefer, T_P maps sets of ground atoms of P to sets of ground atoms of P.

Definition 2.6 (T_P)**.** *Let P be a definite logic program and I a set of ground atoms of P.*

> $T_P(I) = \{A \mid A \leftarrow L_1, ..., L_m (m \geq 0)$ *is a ground instance of a clause in P and* $\{L_1, ..., L_m\} \subseteq I\}$

Intuitively, the operation T_P is the immediate consequence of one step of forward inference on clauses in P.

Theorem 2.7. *[12] Let P be a definite program. Then:*

- *T_P is monotonic:*

$$I_1 \subseteq I_2 \textit{ implies } T_P(I_1) \subseteq T_P(I_2)$$

- *The interpretation of I is a model of P if and only if*

$$T_P(I) \subseteq I$$

- *The interpretation I of P is supported if and only if*

$$I \subseteq T_P(I)$$

- *A fixpoint I of T_P*

$$T_P(I) = I$$

  will be a supported model of P. A least such fixpoint will be a unique minimal supported model of P.

Definition 2.8 (Powers of T_P)**.** *Let P be a definite program. The powers of T_P are defined as*

$$T_P^0(I) = I$$
$$T_P^{n+1}(I) = T_P(T_P^n(I))$$
$$T_P^\omega(I) = \bigcup_{n=0}^{\infty} T_P^n(I)$$

Definition 2.9 (Least Fixpoint of Definite Program)**.** *Let P be a definite program. The least fixpoint of P: lfp(P) is defined as the ω power of T_P*

$$lfp(P) = T_P^\omega(\emptyset)$$

Theorem 2.10. *[12] Let P be a definite program. Its unique minimal Herbrand Model M_P is given by*

$$M_P = lfp(P)$$

Definition 2.11 ($con_D(P)$)**.** *Let P be a definite program and M_P be a model of P defined by theorem 2.10.*

$$con_D(P) = M_P$$

2.2 Negation As Failure

A Normal Logic Program is a set of clauses which allows the use of negation in the body of the clauses. More formally, it is a set of clauses where each has at *least one* positive literal. In logic programming a common form of negation is **Negation as failure** (NAF) [3]. NAF is a non-monotonic inference rule, used to derive *not p* (i.e. that *p* is assumed not to hold) from a failure to derive *p*. Such an inference rule allows us to extend a logic program to include not only information about true instances of relations, but also instances which are false, increasing the expressive power of our language.

To assume that a relation instance is false if it is not implied, is to assume that the program P gives *complete* information about the true instances of the relation. More precisely, it is the assumption that a relation instance is true if and only if it is given explicitly (as a ground atom) or is implied by one the general rules of the program (an intensional clause) [3].

Let us consider a program P representing knowledge about university mathematics courses

mathCourse(mat111)

mathCourse(mat121)

mat111 and mat121 are the only math courses available at our university. For any other course c different from mat111 and mat121, mathCourse(c) is not provable. Since there are no instances explicitly given apart from mathCourse(mat111) and mathCourse(mat121), and there are no general rules in our program which implies mathCourse(c), we can infer

> ¬ mathCourse(c)

Let us consider a second program representing father relations

> father(bob, paul)
>
> father(bob, alice)
>
> fatherless(X) ← *not* father(Y,X)

The *father* relation father(X,Y) represents that X is the father of Y. The *fatherless* relation represents that X does not have a father. *fatherless* is defined by a general rule where we make use of negation as failure. *not* father(Y,X) is satisfied if an exhaustive effort to prove father(Y,X) fails. If we query our program with fatherless(bob) we will search for a substitution θ such that father(Y, bob)θ is equal to one of our ground atoms. In the case of our program this query will fail and fatherless(bob) is inferred.

2.2.1 Recursion and Negation

Another tool to increase the expressive power of our language is recursion. Recursion in logic programming is when a relation is defined in terms of itself. Our first example, the ancestor relation

> ancestor(X,Y) ← parent(X,Z), ancestor(Z,Y)

uses recursion as the relation being defined is found in the body of the clause. Recursion functions as a loop over our relations. Without it we would not be able to express the ancestor relation without explicitly stating every ancestor, increasing the size of our program drastically.

However, when both recursion and negation as failure are used we can encounter a problem. Consider the following program P:

> $q(a)$
>
> $q(b)$
>
> $p(X) \leftarrow q(X), not\, p(X)$

We wish to know whether $p(a)$ is entailed by our program P: $P \vDash p(a)$. To prove $p(a)$ we need to satisfy our subgoals $q(a)$ and *not* $p(a)$. $q(a)$ is easy enough. We have the ground atom as an extensional component of our program. To satisfy our last subgoal *not* $p(a)$ we need an exhaustive proof of $p(a)$ to fail. There are no explicit instance relations about p, hence we perform a recursive call of the clause $p(X) \leftarrow q(X), not\ p(X)$. This will lead to an infinite loop of recursive calls on the same clause.

We want to have the expressive power of both recursion and negation, but, as we have just observed, they do not necessarily mix well. To avoid this problem we consider stratified logic programs.

2.3 Stratified Programs

A stratified logic program is a partitioning of a normal logic program with restriction on the order of its clauses and the use of negation [1]. The key feature of a stratified program is that it forbids *recursion inside negation*, which occurred in the example in Section 2.2.1. Recursion inside negation is again shown in the following program, where p is defined by a recursive call inside a negation:

$p \leftarrow not\ q$

$q \leftarrow p$

Definition 2.12 (Stratified Program)**.** *A program P is stratified when there is a partition into a set of strata*

$$P = P_1 \cup P_2 \cup \ldots \cup P_n \qquad (P_i \text{ and } P_j \text{ disjoint for all } i \neq j)$$

such that for every predicate p

- *The definition of p (all clauses with p in the head) is contained in one of the partitions/strata P_i*

and, for each $1 \leq i \leq n$

- *If a predicate occurs positively in a clause P_i then its definition is contained within*

$$\bigcup j \leq i \ \ P_j$$

- *If a predicate occurs negatively in a clause P_i then its definition is contained within*

$$\bigcup j < i \ \ P_j$$

Note that a program is stratified if there is any such partition. Consider the program P:

$p(X) \leftarrow$ q(X), not r(X)

$r(X) \leftarrow$ s(X), not t(X)

t(a) $\leftarrow$

s(a) $\leftarrow$

s(b) $\leftarrow$

q(a) $\leftarrow$

A possible stratification of P is:

$$\begin{aligned} P = \{p(X) \leftarrow q(X),\ \text{not } r(X)\} \cup \qquad & (P_2) \\ \{r(X) \leftarrow s(X),\ \text{not } t(X)\} \cup \qquad & (P_1) \\ \{t(a) \leftarrow, s(a) \leftarrow, s(b) \leftarrow, q(a) \leftarrow\} \qquad & (P_0) \end{aligned}$$

2.3.1 Determining Stratification

We can determine if a program P can be stratified by constructing the *dependency graph* of P and inspect whether it contains a cycle with a negative edge.

We say that a relation p *refers to* the relation q if there is a clause in P with p in its head and q in its body.

Definition 2.13 (Dependency Graph)**.** *The dependency graph of a program P is a directed graph representing the relation* ***refers to*** *between the relation symbols of P. For any pair of relation symbols p, q there is at most one edge (p, q) in the dependency graph of P. Although, there may be that p refers to q in several clauses in P. An edge (p, q) is positive [negative] iff there is a clause C in P in which p is the relation symbol in the head of C, and q is the relation symbol of a positive [negative] literal in the body of C. Note that an edge may be both positive and negative.*

Theorem 2.14. *[1] A program P can be stratified iff in its dependency graph there are no cycles containing a negative edge.*

Consider the program P:

$p(X) \leftarrow$ q(X), not r(X)

$r(X) \leftarrow$ s(X), not t(X)

t(a) $\leftarrow$

s(a) $\leftarrow$

s(b) $\leftarrow$

q(a) $\leftarrow$

We construct the dependency graph of P. Full lines represent positive edges and dotted lines represent negative edges. We omit ground facts since they do not have any edges and therefore do not contribute to any possible cycles.

$$\begin{tikzcd}
r(X) \arrow[r] \arrow[dd, dashed] & s(X) \\
 & q(X) \\
t(X) & p(X) \arrow[u] \arrow[uul, dashed]
\end{tikzcd}$$

We have seen that the program P can indeed be stratified, and observe again this fact as there are no cycles in its dependency graph that contains a negative edge.

If we instead consider the program which we used to illustrate recursion inside negation:

$p \leftarrow$ *not* q

$q \leftarrow p$

We observe that its dependency graph does contain a cycle with a negative edge

$$\begin{tikzcd}
p \arrow[r, dashed, bend right] & q \arrow[l, bend right]
\end{tikzcd}$$

and it therefore cannot be stratified.

2.3.2 Semantics of Stratified Programs

As described in section 2.1.2 we can determine entailment of a ground atom γ by a program P: $P \vDash \gamma$, by use of forward chaining. This technique also holds for Stratified programs. The difference is that by the introduction of negation as failure to our program P, our logic system is no longer monotonic.

We observe that with negation we lose certain properties. A Definite program P has, among others, the following properties:

1. T_P is monotonic
2. If M_1 is a model of P and M_2 is a model of P, then $M_1 \cap M_2$ is a model of P
3. P has a least Herbrand model

If P is a program with negation, i.e. P is normal, we have

1. T_P does not need to be monotonic

2. If M_1 is a model of P and M_2 is a model of P, then $M_1 \cap M_2$ is not necessarily a model of P

3. P may have no least Herbrand model

Consider the program:

$$A \leftarrow \text{not } B$$

1. T_P is monotonic, i.e. $I_1 \subseteq I_2$ implies $T_P(I_1) \subseteq T_P(I_2)$. Let $I_1 = \varnothing$ and $I_2 = \{B\}$. Then $T_P(I_1) = \{A\}$ and $T_P(I_2) = \varnothing$. Thus $I_1 \subseteq I_2$, but not $T_P(I_1) \subseteq T_P(I_2)$.

2. Both $\{A\}$ and $\{B\}$ are models of P, but their intersection is not.

3. P has two different minimal models $\{A\}$ and $\{B\}$. Then P does not have a least Herbrand model.

However, we keep the following properties:

- The interpretation of I is a *model* of P if and only if

$$T_P(I) \subseteq I$$

- The interpretation I of P is supported if and only if

$$I \subseteq T_P(I)$$

Hence, a fixpoint of T_P, $I = T_P(I)$, is a supported model of P. We also want minimality, and it is therefore natural to look for minimal fixed points of the non-monotonic operator T_P [1]. The construction of such a minimal fixed point can be done using stratification. We describe the process:

We differentiate between the immediate consequence operator for definite programs: T_P, and the non-monotonic version for stratified programs: S_P.

Definition 2.15 (S_P)**.** *Let P be a stratified logic program and I a set of ground atoms of P.*

> $S_P(I) = \{A \mid A \leftarrow \rho_1, ..., \rho_p,$ *not* $\eta_1, ...,$ *not* $\eta_n\}$ $(p, n \geq 0)$ *is a ground instance of a clause in P where* $\{\rho_1, ..., \rho_p\} \subseteq I$ *and* $\{\eta_1, ..., \eta_n\} \cap I = \varnothing\}$

Definition 2.16. *Let I be a set of ground atoms of a stratified program P. The powers of the operator S_P are defined as:*

$$\begin{aligned} S_P^0(I) &= I \\ S_P^{(n+1)}(I) &= S_P(S_P^n(I)) \cup S_P^n(I) \\ S_P^\omega(I) &= \bigcup_{n=0}^{\infty} S_P^n(I) \end{aligned}$$

Theorem 2.17. *[1] Let P be a program stratified by*

$$P = P_1 \cup P_2 \cup ... \cup P_n \qquad (P_i \text{ and } P_j \text{ disjoint for all } i \neq j)$$

The interpretation M_P, constructed by

$$\begin{aligned} M_1 &= S_{P_1}^\omega(\varnothing) \\ M_2 &= S_{P_2}^\omega(M_1) \\ &\dots \\ M_n &= S_{P_n}^\omega(M_{n-1}) \end{aligned}$$

where $M_P = M_n$, is a minimal and supported model of P.

M_P is our final set of all consequences of P, which we denote as $con_S(P)$ for stratified programs.

Definition 2.18 ($con_S(P)$)**.** *Let P be a stratified program and M_P be a model of P defined by theorem 2.17.*

$$con_S(P) = M_P.$$

Let us illustrate the construction of a model M_P for a logic program P using stratification. Consider the program P:

$p(X) \leftarrow$ q(X), not r(X)

$r(X) \leftarrow$ s(X), not t(X)

t(a) $\leftarrow$

s(a) $\leftarrow$

s(b) $\leftarrow$

q(a) $\leftarrow$

A possible stratification, as we have seen, is

$$\begin{aligned} P = \{p(X) \leftarrow q(X),\ \text{not } r(X)\} \cup & \qquad (P_2)\\ \{r(X) \leftarrow s(X),\ \text{not } t(X)\} \cup & \qquad (P_1)\\ \{t(a) \leftarrow, s(a) \leftarrow, s(b) \leftarrow, q(a) \leftarrow\} & \qquad (P_0)\end{aligned}$$

Using the immediate consequence operator we construct interpretations

$$\begin{aligned} M_1 &= S^{\omega}_{P_1}(\varnothing) = \{t(a), s(a), s(b), q(a)\}\\ M_2 &= S^{\omega}_{P_2}(M_1) = \{r(b), t(a), s(a), s(b), q(a)\}\\ M_3 &= S^{\omega}_{P_3}(M_2) = \{p(a), r(b), t(a), s(a), s(b), q(a)\}\end{aligned}$$

Finally, we have $M_P = M_3$ which is a minimal and supported model of P.

2.4 Inference in Logic Programming

For both definite and stratified programs we can use forward chaining to determine entailment, by constructing $con_D(P)$ and $con_S(P)$ respectively. The Inductive Logic Programming systems we will consider in the following sections are restricted to programs without functions. This means that the Herbrand universe is finite. As the programs are also finite we only have finite models. This allows us to easily check whether a ground atom γ is an element of the set of consequences, determining entailment.

Theorem 2.19. *Let P be a stratified program. For all ground atoms γ*

$P \vDash \gamma$ *iff* $\gamma \in con_S(P)$

$P \nvDash \gamma$ *iff* $\gamma \notin con_S(P)$

2.5 Inductive Logic Programming

Inductive Logic Programming (ILP) is a collection of techniques for constructing logic programs from examples. Given a set of positive examples and a set of negative examples an ILP system constructs a logic program which entails all positive examples but does not entail any of the negative examples. ILP has several appealing features, performing an induction task, which more standard machine learning techniques, for instance Neural Networks, lack. These features include data efficiency, verifiability, are often human readable and support of transfer learning.

There are many different approaches to ILP. In this section we will describe the approach to ILP given in [5]: ILP as a satisfiability problem. We will cover the broader ideas and the necessary details for our extension of the system. For a thorough overview into the original system we refer to [5].

A Inductive Logic Programming (ILP) system seeks to construct a logic program satisfying a set of positive examples and not satisfy a set of negative examples. An ILP problem is a tuple $(\mathcal{B}, \mathcal{P}, \mathcal{N})$ of finite sets of ground atoms: background knowledge $\mathcal{B}$, positive instances $\mathcal{P}$ of the target predicate, and negative instances $\mathcal{N}$ of the target predicate.

Given an ILP problem $(\mathcal{B}, \mathcal{P}, \mathcal{N})$, a solution is a set R of clauses, i.e a logic program R, such that

$\mathcal{B} \cup R \models \gamma$ for all $\gamma \in \mathcal{P}$

$\mathcal{B} \cup R \not\models \gamma$ for all $\gamma \in \mathcal{N}$

Consider the task of learning which natural numbers are even. A minimal description of the natural numbers is given as the background knowledge $\mathcal{B}$:

$$\mathcal{B} = \{zero(0), succ(0,1), succ(1,2), succ(2,3)..., succ(19,20)\}$$

The positive and negative examples of the even predicate are:

$$\mathcal{P} = \{even(0), even(2), even(4), even(6), ..., even(20)\}$$
$$\mathcal{N} = \{even(1), even(3), even(5), even(7), ..., even(19)\}$$

A possible solution is the program R:

$$even(X) \leftarrow zero(X)$$
$$even(X) \leftarrow even(Y), succ2(Y, X)$$
$$succ2(X, Y) \leftarrow succ(X, Z), succ(Z, Y)$$

This solution requires both recursion and predicate invention ($succ2$).

2.5.1 ILP as a Satisfiability Problem

In [5] the induction task of ILP is transformed into a satisfiability problem. This is done by using a top-down approach, where a set of clauses are generated from a language definition and tested against the positive and negative examples. Each generated clause is assigned a Boolean flag indicating whether it is on or off. Now the induction problem becomes a satisfiability problem: choose an assignment to the Boolean flags such that the turned-on clauses together with the background knowledge entail the positive examples and do not entail the negative examples.

2.5.2 Basic Concepts

Definition 2.20 (Language Frame). *A language frame $\mathcal{L}$ is a tuple*

$$(target, P_e, arity_e, C)$$

- *$target$ is the target predicate, the intensional predicate we are trying to learn*
- *P_e is a set of extensional predicates*
- *$arity_e$ is a map $P_e \cup target \rightarrow \mathbb{N}$, specifying the arity of the predicate*

- *C is a set of constants*

Definition 2.21 (ILP Problem)**.** *An ILP problem (for this specific top-down approach) is a tuple*

$$(\mathcal{L}, \mathcal{B}, \mathcal{P}, \mathcal{N})$$

- *$\mathcal{L}$ is a language frame*
- *$\mathcal{B}$ is a set of background assumptions, ground atoms formed from the predicates in P_e and the constants in C*
- *$\mathcal{P}$ is the set of positive examples, ground atoms formed from the target predicate and the constants in C*
- *$\mathcal{N}$ is the set of negative examples, ground atoms formed from the target predicate and the constants in C*

Definition 2.22 (Rule Template)**.** *A rule template τ describes a range of clauses that can be generated. It is a pair*

$$(v, int)$$

- *$v \in \mathbb{N}$ specifies the number of existentially quantified variables allowed in the clause*
- *$int \in \{0, 1\}$ specifies whether the atoms in the body of the clause can use intensional predicates $(int = 1)$ or only extensional predicates $(int = 0)$*

In this approach to ILP each predicate is defined by at most two clauses. A rule template generates a set of possible clauses: $cl(\tau)$, where one is selected to be one of two clauses defining the predicate. Hence, each predicate has two rule templates. In addition, we restrict the clauses to have at most two literals in the body. These restrictions are done without loss of generality (see Apendix A).

Of the clauses specified by a rule template we omit certain clauses in a pruning stage. The clauses omitted are:

- All variables that appear in the head must appear in the body. Hence, such clauses are omitted:

  $$target(X) \leftarrow pred(Y), pred(Z)$$

- Predicates which are specified to use intensional predicates in their definition must have at least one atom in the body of an intensional predicate. Let *target* be defined with intensional atoms in the body and *succ* be defined extensionally. Then the following clause is kept:

  $$target(X, Y) \leftarrow target(Z, Y), succ(X, Y)$$

  While this clause is omitted:

  $$target(X, Y) \leftarrow succ(X, Y)$$

- No circular clauses:

  $$target(X) \leftarrow target(X)$$

  However, if the variables differ then the clause is kept:

  $$target(X) \leftarrow target(Y)$$

- No duplicate clauses. If we have the clause:

  $$target(X) \leftarrow succ(X,Y), succ(Y,Z)$$

  we omit its equivalent clause with the body atoms swapped:

  $$target(X) \leftarrow succ(Y,Z), succ(X,Y)$$

The predicates are restricted to be nullary, unary or binary. No constants are allowed in the clauses, but a constant can be represented using a nullary predicate.

Definition 2.23 (Program Template)**.** *A program template* Π *describes a range of programs that can be generated. It is a tuple*

$$(P_a, arity_a, rules, T)$$

- P_a *is a set of auxiliary (intensional) predicates; these are the additional invented predicates used to help define the target predicate*
- $arity_a$ *is a map* $P_a \rightarrow \mathbb{N}$ *specifying the arity of each auxiliary predicate*
- *rules maps each intensional predicate* p *to a pair of rule templates* (τ_p^1, τ_p^2)
- $T \in \mathbb{N}$ *specifies the max number of steps of forward chaining inference*

Definition 2.24 (Language)**.** *A language is a combination of the extensional predicates of the language frame* $\mathcal{L}$ *and the intensional predicates of the program template* Π*:*

$$(P_e, P_i, arity, C)$$

- P_e *is a set of extensional predicates*
- $P_i = P_a \cup target$
- $arity = arity_e \cup arity_a$
- C *is a set of constants*

Let P be the complete set of predicates:

$$P = P_i \cup P_e$$

A language determines the set of all ground atoms G:

$$\begin{aligned}
&G = \\
&\{p() \mid p \in P, arity(p) = 0\} \cup \\
&\{p(k) \mid p \in P, arity(p) = 1, k \in C\} \cup \\
&\{p(k_1, k_2) \mid p \in P, arity(p) = 2, k_1, k_2 \in C\} \cup \\
&\{\bot\}
\end{aligned}$$

Note that G includes the falsum atom $\bot$, the atom that is always false.

2.5.3 Reducing Induction to Satisfiability

Given a rule template Π, let τ_p^i be the i'th rule template for a predicate p. Let $C_p^{i,j}$ be the j'th clause in $cl(\tau_p^i)$: the set of all clauses generated by τ_p^i. Using a set Φ of Boolean variables, indicating which of the clauses in $C_p^{i,j}$ are to be used in the final program to define the predicate p, the induction problem is turned into a satisfiability problem. Now a SAT solver can find a truth assignment to the propositions in Φ. By extracting a subset of the clauses which Φ has set to true, the final program is constructed [5]. This method motivates the implementation of the following differentiable ILP system.

2.6 Differentiable Inductive Logic Programming

Differentiable Inductive Logic Programming (δILP) is a reimplementation of ILP in an end-to-end differentiable architecture. δILP seeks to combine the advantages of ILP with the advantages of neural network-based systems: a data-efficient induction system that can learn explicit human-readable symbolic rules, that is robust to noisy and ambiguous data, and that does not deteriorate when applied to unseen test data. δILP reinterprets the ILP task as a binary classification problem, minimizing cross-entropy loss with regard to ground-truth Boolean labels during training. Instead of mapping ground atoms to discrete values $\{False, True\}$ we map them to continuous values[1] $[0, 1]$. Instead of using Boolean flags to choose a discrete subset of clauses we now use a set W of continues weights to determine a probability distribution over clauses.

This section will summarize δILP with focus on the details necessary to implement Stratified Negation As Failure. For a complete and more detailed explanation see [5].

2.6.1 Valuations

Given a set G of n ground atoms, a valuation is a vector $[0, 1]^n$ mapping each atom $\gamma_i \in G$ to a real unit interval. Consider the following example.

Given a language frame $\mathcal{L} = (P_e, P_i, arity, C)$

$$P_e = \{r/2\} \;\; P_i = \{p/0, q/1\} \;\; C = \{a, b\}$$

A possible valuation of the ground atoms in G of $\mathcal{L}$ is

$$\begin{aligned} \bot \mapsto 0.0 \quad p() \mapsto 0.0 \quad q(a) \mapsto 0.1 \quad r(a,a) \mapsto 0.7 \\ r(a,b) \mapsto 0.1 \quad r(b,a) \mapsto 0.4 \quad r(b,b) \mapsto 0.2 \end{aligned}$$

The valuation of $\bot$ is always 0.0. The process determining the valuation of a given ground atom will be explained in a later section.

2.6.2 Induction by Gradient Descent

Given the sets $\mathcal{P}$ and $\mathcal{N}$ of positive and negative instances of the target predicate we form a set Λ of atom-label pairs:

$$\Lambda = \{(\gamma, 1) \mid \gamma \in \mathcal{P}\} \cup \{(\gamma, 0) \mid \gamma \in \mathcal{N}\}$$

This is our dataset for a binary classifier, pairs of input and label.

Now given an ILP problem $(\mathcal{L}, \mathcal{B}, \mathcal{P}, \mathcal{N})$, a program template Π and a set of clause weights W, a differentiable model is constructed that computes the conditional probability λ of a ground atom α:

$$p(\lambda \mid \alpha, W, \Pi, \mathcal{L}, \mathcal{B})$$

The desired outcome is for our predicted label $p(\lambda \mid \alpha, W, \Pi, \mathcal{L}, \mathcal{B})$ to match the actual label λ in the pair (α, λ) we sample from Λ. To manage this the expected negative log likelihood needs to be minimized when sampling uniformly (α, λ) pairs from Λ:

$$loss = -\mathop{\mathbb{E}}_{(\alpha,\lambda)\sim\Lambda}[\lambda \cdot \log(p(\lambda \mid \alpha, W, \Pi, \mathcal{L}, \mathcal{B})) + (1-\lambda) \cdot \log(1 - p(\lambda \mid \alpha, W, \Pi, \mathcal{L}, \mathcal{B}))]$$

The probability of label λ is calculated given the atom α by inferring the consequences of applying rules to the background facts. Using fixpoint semantics we applied the immediate consequence operator until no more consequences could be derived. In δILP forward chaining can always infer

[1]The values in [0, 1] are interpreted as probabilities rather than fuzzy "degrees of truth".

new consequences, i.e. valuations. Hence, the system is restricted to only perform a finite number T (time steps) of forward chaining. The probability of label λ is given by:

$$p(\lambda \mid \alpha, W, \Pi, \mathcal{L}, \mathcal{B}) = f_{extract}(f_{infer}(f_{convert}(\mathcal{B}), f_{generate}(\Pi, \mathcal{L}), W, T), \alpha)$$

These functions have the following roles.

$$f_{extract} : [0,1]^n \times G \to [0,1]$$

$f_{extract}$ takes a valuation x and an atom γ and extracts the value of the atom:

$$f_{extract}(x, \gamma) = x[index(\gamma)]$$

where $index : G \to \mathbb{N}$ is a function that assigns each ground atom a unique integer index.

$$f_{convert} : 2^G \to [0,1]^n$$

$f_{convert}$ takes a set of atoms and converts it into a valuation mapping the elements of $\mathcal{B}$ to 1 and all other elements of G to 0.

$$f_{convert}(\mathcal{B}) = y \text{ where } y[i] = \begin{cases} 1, & \text{if } \gamma_i \in \mathcal{B} \\ 0, & \text{otherwise} \end{cases}$$

$$f_{generate}(\Pi, \mathcal{L}) = \{cl(\tau_p^i) \mid p \in P, i \in \{1,2\}\}$$

$f_{generate}$ produces a set of clauses from a program template Π and a language frame $\mathcal{L}$.

$$f_{infer} : [0,1]^n \times C \times W \times \mathbb{N} \to [0,1]^n$$

f_{infer} performs T steps of forward-chaining inference using the generated clauses, amalgamating the conclusions together using the clause weights W (described in more detail below).

2.6.3 Rule Weights

The weights W are a set $W_1, \ldots, W_{|P_i|}$ of matrices. One matrix for each intensional predicate $p \in P_i$. The matrix W_p for predicate p is of shape $(|cl(\tau_p^1)|, |cl(\tau_p^2)|)$. The various matrices W_p are not necessarily the same size because the different rule templates generate a different number of clauses defining the different intensional predicates. The weight $W_p[j,k]$ represents how strongly the system believes that the pair of clauses $(C_p^{1,j}, C_p^{2,k})$ is the right way to define the intensional predicate p. The weight matrix $W_p \in R^{|cl(\tau_p^1)| \times |cl(\tau_p^2)|)}$ is a matrix of real numbers. It is transformed into a probability distribution $W_p^* \in [0,1]^{|cl(\tau_p^1)| \times |cl(\tau_p^2)|)}$ using softmax:

$$\mathbf{W}_p^*[j,k] = \frac{e^{\mathbf{W}_p[j,k]}}{\sum_{j',k'} e^{\mathbf{W}_p[j',k']}}$$

$W_p^*[j,k]$ represents the probability that the pair of clauses $(C_p^{1,j}, C_p^{2,k})$ is the right way to define the intensional predicate p.

2.6.4 Inference

Given an initial evaluation a_0 of our ground atoms G

$$a_0[x] = \begin{cases} 1, & \text{if } \gamma_x \in \mathcal{B} \\ 0, & \text{otherwise} \end{cases}$$

and a set of generated clauses the consequences of our background knowledge can be inferred using the differentiable evaluation function $\mathcal{F}_c$. An evaluation is calculated and then adjusted by the clause weights (how much the truths we calculated are believed). After T time steps of forward inference an valuation of the ground atoms are extracted and compared to our data set (positive and negative instances provided in the problem specification). The cross entropy loss is calculated so that the loss can be propagated backwards through the system to adjust the clause weights.

Each clause c induces a function $\mathcal{F}_c : [0,1]^n \to [0,1]^n$. Consider the clause

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

The set G of ground atoms derived from this clause (with constants $\{a, b\}$) is

$$G = \{p(a), p(b), q(a), q(b), \perp\}$$

The evaluation which $\mathcal{F}_c$ outputs can be seen as the (un-weighted) consequences of our clause, for instance:

G	a_0	$\mathcal{F}_c(a_0)$	a_1	$\mathcal{F}_c(a_1)$
p(a)	0.0	0.1	0.2	0.7
p(b)	0.0	0.3	0.9	0.4
q(a)	0.1	0.0	0.7	0.0
q(b)	0.3	0.0	0.4	0.0
$\perp$	0.0	0.0	0.0	0.0

The set C contains all generated clauses, where $C_p^{i,j}$ is the j'th clause of the i'th rule template for the intensional predicate p. A corresponding set $\mathcal{F}$ is defined where $\mathcal{F}_p^{i,j}$ is the valuation function corresponding to the clause $C_p^{i,j}$. Another set of functions is defined; $\mathcal{G}_p^{i,j}$, that combines the application of two functions $\mathcal{F}_p^{1,j}$ and $\mathcal{F}_p^{2,k}$. $\mathcal{G}_p^{i,j}$ is the result of applying both clauses $C_p^{1,j}$ and $C_p^{2,k}$ and taking the element-wise max:

$$\mathcal{G}_p^{i,j}(a) = x \quad \text{where} \quad x[i] = max(\mathcal{F}_p^{1,j}(a)[i], \mathcal{F}_p^{2,k}(a)[i])$$

Next, a time series of valuations is defined. Each such valuation a_t represents the conclusions after t time-steps of inference. The initial value a_0 when $t = 0$ is based on the initial set $\mathcal{B} \subseteq G$ of background axioms:

$$a_0[x] = \begin{cases} 1, & \text{if } \gamma_x \in \mathcal{B}. \\ 0, & \text{otherwise.} \end{cases}$$

$c_t^{p,j,k}$ is defined as

$$c_t^{p,j,k} = G_p^{j,k}(a_t).$$

Intuitively, $c_t^{p,j,k}$ is the result of applying one step of forward chaining inference to a_t using clauses $C_p^{1,j}$ and $C_p^{2,k}$. The weighted average of $c_t^{p,j,k}$ is defined using softmax:

$$b_t^p = \sum_{j,k} c_t^{p,j,k} \cdot \frac{e^{\mathbf{W}_p[j,k]}}{\sum_{j',k'} e^{\mathbf{W}_p[j',k']}}.$$

Intuitively, b_t^p is the result of applying possible pairs of clauses that can jointly define predicate p, and weighting the result by the weights W_p.

From this the successor a_{t+1} of a_t is defined:

$$a_{t+1} = f_{amalgamate}(a_t, \sum_{p \in P_i} b_t^p),$$

where

$$f_{amalgamate}(x, y) = x + y - x \cdot y.$$

The successor depends on the previous valuation a_t and a weighted mixture of the clauses defining the other intensional predicates.

2.6.5 Computing the $\mathcal{F}_c$ functions

The $\mathcal{F}_c$ function, in short, calculates the product (fuzzy conjunction) of each pair of ground atoms (the body) which leads to the truth of the clause head. After finding all products which lead to the truth of the clause head, the maximum value is selected.

$$\mathcal{F}_c : [0,1]^n \to [0,1]^n$$

$\mathcal{F}_c$ maps a vector of valuations to a vector of valuations. A function $\mathcal{F}_c$ is induced for every clause c constructed by the rule templates. Each function can be computed as follows. Let $X_c = \{x_k\}_{k=1}^n$ be a set of sets of pairs of indices of ground atoms of clause c. Each x_k contains all the pairs of atoms that justify atom γ_k according to the current clause:

$$x_k = \{(a,b) \mid satisfies_c(\gamma_a, \gamma_b) \wedge head_c(\gamma_a, \gamma_b) = \gamma_k\}.$$

Here, $satisfies_c(\gamma_1, \gamma_2)$ if the pair of ground atoms (γ_1, γ_2) satisfies the body of clause c. If $c = \alpha \leftarrow \alpha_1, \alpha_2$, then $satisfies_c(\gamma_1, \gamma_2)$ is true if there is a substitution θ such that $\alpha_1[\theta] = \gamma_1$ and $\alpha_2[\theta] = \gamma_2$.

Also, $head_c(\gamma_1, \gamma_2)$ is the head atom produced when applying clause c to the pair of atoms (γ_1, γ_2). If $c = \alpha \leftarrow \alpha_1, \alpha_2$ and $\alpha_1[\theta] = \gamma_1$ and $\alpha_2[\theta] = \gamma_2$ then

$$head_c(\gamma_1, \gamma_2) = \alpha[\theta].$$

For example, suppose $P = \{p, q, r\}$ and $C = \{a, b\}$. Then the ground atoms G are

k	0	1	2	3	4	5	6	7	8
γ_k	$\bot$	p(a,a)	p(a,b)	p(b,a)	p(b,b)	q(a,a)	q(a,b)	q(b,a)	q(b,b)
k	9	10	11	12					
γ_k	r(a,a)	r(a,b)	r(b,a)	r(b,b)					

Suppose clause c is:

$$r(X,Y) \leftarrow p(X,Z), q(Z,Y).$$

Then $X_c = \{x_k\}_{k=1}^n$ is:

k	γ_k	x_k
0	$\bot$	{}
1	p(a,a)	{}
2	p(a,b)	{}
3	p(b,a)	{}
4	p(b,b)	{}

k	γ_k	x_k
5	q(a,a)	{}
6	q(a,b)	{}
7	q(b,a)	{}
8	q(b,b)	{}

k	γ_k	x_k
9	r(a,a)	{(1,5),(2,7)}
10	r(a,b)	{(1,6),(2,8)}
11	r(b,a)	{(3,5),(4,7)}
12	r(b,b)	{(3,6),(4,8)}

Focusing on a particular row, the reason why (2, 7) is in x_9 is that $\gamma_2 = p(a,b)$, $\gamma_7 = q(b,a)$, the pair of atoms $(p(a,b), q(b,a))$ satisfies the body of clause c, and the head of the clause c (for this pair of atoms) is $r(a,a)$ which is γ_9.

X_c, a set of pairs, is transformed into a three dimensionel tensor: $X \in \mathbb{N}^{n \times w \times 2}$. Here, w is the maximum number of pairs for any k in $1...n$. The width w depends on the number of existensially quantified variables v in the rule template. Each existentially quantified variable can take $|C|$ values, so $w = |C|^v$. X is constructed from X_c, filling unused space with $(0,0)$ pairs that point to the pair of atoms $(\bot, \bot)$:

$$X[k,m] = \begin{cases} x_k[m], & \text{if } m < |x_k|. \\ (0,0), & \text{otherwise.} \end{cases}$$

This is why the falsum atom $\bot$ needs to be included in G, so that the null pairs have some atom to point to. In the running example, this yields:

k	γ_k	$X[k]$
0	$\bot$	[(0,0),(0,0)]
1	p(a,a)	[(0,0),(0,0)]
2	p(a,b)	[(0,0),(0,0)]
3	p(b,a)	[(0,0),(0,0)]
4	p(b,b)	[(0,0),(0,0)]

k	γ_k	$X[k]$
5	q(a,a)	[(0,0),(0,0)]
6	q(a,b)	[(0,0),(0,0)]
7	q(b,a)	[(0,0),(0,0)]
8	q(b,b)	[(0,0),(0,0)]

k	γ_k	$X[k]$
9	r(a,a)	[(1,5),(2,7)]
10	r(a,b)	[(1,6),(2,8)]
11	r(b,a)	[(3,5),(4,7)]
12	r(b,b)	[(3,6),(4,8)]

Let $X_1, X_2 \in \mathbb{N}^{n \times w}$ be two slices of X, taking the first and second elements of each pair:

$$X_1 = X[:,:,0] \qquad X_2 = X[:,:,1].$$

$gather_2 : \mathbb{R}^a \times \mathbb{N}^{b \times c} \to \mathbb{R}^{b \times c}$ is defined as:

$$gather_2(x,y)[i,j] = x[y[i,j]].$$

Finally, $\mathcal{F}_c(a)$ can be defined. Let $Y_1, Y_2 \in [0,1]^{n \times w}$ be the result of assembling the elements of a according to the matrix of indices in X_1 and X_2:

$$Y_1 = gather_2(a, X_1) \qquad Y_2 = gather_2(a, X_2).$$

Now let $Z \in [0,1]^{n \times w}$ contain the results from element-wise multiplying the elements of Y_1 and Y_2:

$$Z = Y_1 \odot Y_2.$$

Here, $Z[k,:]$ is the vector of fuzzy conjunctions of all the pairs of atoms that contribute to the truth of γ_k, according to the current clause. $\mathcal{F}_c(a)$ is defined by taking the max fuzzy truth values in Z. Let $\mathcal{F}_c(a) = a'$ where $a'[k] = max(Z[k,:])$.

The following table shows the calculation of $\mathcal{F}_c(a)$ for a particular valuation a, using the running example $c = r(X,Y) \leftarrow p(X,Z), q(Z,Y)$. Here, since there is one existentially quantified variable Z, $v = 1$, and $w = |\{a,b\}|^v = 2$.

k	γ_k	$a[k]$	$X_1[k]$	$X_2[k]$	$Y_1[k]$	$Y_2[k]$	$Z[k]$	$\mathcal{F}_c(a)[k]$
0	$\perp$	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
1	p(a,a)	1.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
2	p(a,b)	0.9	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
3	p(b,a)	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
4	p(b,b)	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
5	q(a,a)	0.1	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
6	q(a,b)	0.0	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
7	q(b,a)	0.2	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
8	q(b,b)	0.8	[0 0]	[0 0]	[0 0]	[0 0]	[0 0]	0.00
9	r(a,a)	0.0	[1 2]	[5 7]	[1.0 0.9]	[0.1 0.2]	[0.1 0.18]	0.18
10	r(a,b)	0.0	[1 2]	[6 8]	[1.0 0.9]	[0 0.8]	[0 0.72]	0.72
11	r(b,a)	0.0	[3 4]	[5 7]	[0 0]	[0.1 0.2]	[0 0]	0.00
12	r(b,b)	0.0	[3 4]	[6 8]	[0 0]	[0 0.8]	[0 0]	0.00

More simply, $\mathcal{F}_c$ for a single valuation for a ground atom is calculated as follows: We have our clause

$$r(X,Y) \leftarrow p(X,Z), q(Z,Y).$$

Considering the ground atom $r(a,a)$, atoms which contribute to its truth are found. These are the ground atom pairs $p(a,a) \wedge q(a,a)$ and $p(a,b) \wedge q(b,a)$. These ground atoms pairs are indexed by $X_1[k]$ and $X_2[k]$. The product of the pairs of valuations is calculated. $p(a,a)$ has valuation 1.0 and $q(a,a)$ has valuation 0.1 Hence, the "truth value" of the ground atom $r(a,a)$ as a consequence of $p(a,a)$ and $q(a,a)$ is $1.0 \cdot 0.1 = 0.1$. The same is done for other pairs of ground atoms which contribute to the truth of $r(a,a)$ and get $0.9 \cdot 0.2 = 0.18$. Finally, the maximum is selected, as a fuzzy conjunction, of these two calculated values as the new valuation for $r(a,a)$, 0.18.

$$\mathcal{F}_c(a)[9] = 0.18$$

2.6.6 Extracting our Program

Before training, rule weights are initialised randomly by sampling from a $\mathcal{N}(0,1)$ distribution. Each learning task is trained for 200 - 500 steps depending on the complexity of the program, to reduce computational cost. In training the rule weights are adjusted to minimise cross entropy loss as described above.

For each training step a mini-batch is sampled from the positive and negative examples, instead of using the whole dataset. Selecting a random subset of the training data gives the process a stochastic element which helps to escape local minima.

For each training step we sample a mini-batch from the positive and negative examples. Note that, instead of using the whole set of positive and negative examples each training step, we just take a random subset. This mini-batching gives the process a stochastic element and helps to escape local minima.

After the training steps, δILP produces a set of rule weights for each rule template. To validate this program, we run the model on a test dataset not used in training. This is testing the system's ability to generalise to unseen data. We compute validation error as the sum of mean-squared difference between the actual label λ and the predicted label $\hat{\lambda}$:

$$loss = \sum_{i=1}^{k} (\lambda - \hat{\lambda})^2$$

Once δILP has finished training, we extract the rule weights, and take the soft-max. Interpreting the soft-maxed weights as a probability distribution over clauses, we measure the "fuzziness" of the solution by calculating the entropy of the distribution. On the discrete error-free tasks, δILP finds solutions with zero entropy, as long as it does not get stuck in a local minimum. To extract a logic program from the learned clause weights all clauses whose probability is over some constant threshold are selected to define the clauses of the predicates of our program. This threshold is arbitrarily set to 0.1.

3 Stratified Negation as Failure

As of now δILP is restricted to reasoning using definite clauses. Learning clauses without the use of negation in the body limits the expressiveness of our language. Normal logic programs are widely used because they are much easier to devise, write and analyse, than definite programs. Normal logic programs are shorter and clearer than definite programs because negative knowledge can be expressed through what is already known. Consider, for example, a program to compute intersection, returning the intersection of input lists X and Y. Such a program must check whether an element occurring in X also occurs in Y, or not. To this end, two subprograms *member* and *not_member* are needed. If negation is allowed, we just have to device a program for *member*, and then set:

$$not\_member(X,Y) \leftarrow not\ member(X,Y)$$

If negation is not allowed, then the two subprograms must be treated as independent concepts, and a program for *not_member* must be developed as well. Since negation can make programs sensibly shorter, this may have a positive influence on their learnability, as the difficulty of learning a given logic program is very much related to its length [2]. Hence, we want to introduce a form of negation. We propose Stratified Negation as Failure to fill this role.

Stratified Negation as Failure (SNAF) is the use of the negation as failure inference rule *not* in Stratified programs. We will re-implement δILP in such a way that all programs constructed are stratified and can include the use of negation as failure. First, we re-implement the ILP system described above with SNAF. Then we "neurolize" the process, implementing SNAF in δILP. To distinguish between the extended system with SNAF and the system described in [5] we will refer to the original system as δILP and the extended system as SNAFδILP: Stratified Negation as Failure in Differentiable Inductive Logic Programming.

3.1 Stratified Negation as Failure in Inductive Logic Programming

In [5] definite programs were constructed by selecting a set of clauses which satisfy the positive examples and does not satisfy the negative ones. Using forward chaining we determine entailment by constructing a set of all consequences of our program.

When extending the ILP system to include negation as failure we use the same approach, but alter it to handle the non-monotonicity of negation as failure. This is done in three main steps

1. Add negation as failure into the construction of clauses
2. Ensure a selection of clauses such that the program is stratified
3. Forward chaining is altered to handle the non-monotonicity of negation as failure

Step 1 is fairly obvious. To construct programs that contain negation as failure we need to construct clauses that use negation as failure.

In step 2 we want to ensure stratification. We require programs to be stratified to be able to perform forward chaining and construct our set of consequences.

In step 3 we construct $con_S(P)$ for stratified programs, instead of $con_D(P)$ for definite programs.

Apart from the alterations in the next sections all parts of the ILP system remains the same as described in Section 2.5.

3.1.1 Adding Negation As failure

To have our set of generated clauses include clauses with negation as failure we extend the rule template (Definition 2.22).

Definition 3.1 (Rule Template with Negation as Failure)**.** *A **rule template** τ describes a range of clauses that can be generated. It is a tuple*

$$(v, int, neg)$$

- $v \in \mathbb{N}$ *specifies the number of existentially quantified variables allowed in the clause*
- $int \in \{0,1\}$ *specifies whether the atoms in the body of the clause can use intensional predicates* $(int = 1)$ *or only extensional predicates* $(int = 0)$
- $neg \in \{0,1\}$ *specifies whether the atoms in the body of the clause can be negated* $(neg = 1)$ *or not* $(neg = 0)$

Now the clauses of the generated program can be normal clauses, instead of just definite clauses. We will be using this definition of the rule template τ for the rest of this paper.

To illustrate the clause generation with negation we consider an ILP task with language frame $\mathcal{L}$

$$\mathcal{L} = (target, P_e, arity_e, C)$$

- target $= q/2$
- $P_e = \{p/2\}$
- $C = \{a, b, c, d\}$

The target predicate q will have two rule templates (τ_q^1, τ_q^2) to generate which clauses can be in the definition of q. Let the first rule template τ_q^1 be

$$\tau_q^1 = (v = 0, int = 0, neg = 1)$$

specifying no existentially quantified variables and disallowing intensional predicates, but allowing negative literals. The generated clauses from the rule template τ_q^1 will include the clause:

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

in addition to three extra clauses with all possible combinations of negation:

1. $q(X,Y) \leftarrow not\, p(X,X),\ p(X,Y)$
2. $q(X,Y) \leftarrow p(X,X),\ not\, p(X,Y)$
3. $q(X,Y) \leftarrow not\, p(X,X),\ not\, p(X,Y)$

This increases the number of generated clauses by four times the number of clauses generated by the original rule template, i.e. without the neg parameter. However, this number is lowered by pruning. In addition to the pruning performed in δILP clauses which use negation are discarded in these cases:

- The atoms of the body are equal, but only one is negated:
  $target(X) \leftarrow pred(X), not\ pred(X)$
- The head of the clause appears negated in the body:
  $target(X) \leftarrow not\ target(X)$

3.1.2 Selecting Clauses for Stratified Programs

In ILP as a satisfiability problem a subset P of all generated clauses Φ was selected to be our final program. To ensure that our selected clauses form a stratified program we will add a restriction to the propositions in Φ, determining which clauses can be selected for our set $P \subseteq \Phi$.

We device an algorithm for determining if a program P can be stratified. For the clauses in P we construct the dependency graph (definition 2.13) and then, using a Depth First Search, search for a negative cycle. By theorem 2.13 we know that if such a cycle exists the program cannot be stratified. The algorithm returns true if the program P can be stratified and returns false if the program P cannot be stratified.

```
NoNegativeCycle
Input Graph: G
1: for $v \in G$ do
2:     $visited \leftarrow \{\}$
3:     $recursionStack \leftarrow \{\}$
4:     if negativeCycleUtil(G, v, visited, recursionStack, False) then
5:         return False
6:     end if
7: end for
8: return True
```

```
negativeCycleUtil
Input Graph: G, Vertix: v, Set: visited, Set: recursionStack, Boolean: negativeEdge
1: if $v \in recursionStack$ then
2:     return negativeEdge
3: end if
4: if $v \in visited$ then
5:     return False
6: end if
7: $visited \leftarrow visited \cup \{v\}$
8: $recursionStack \leftarrow recursionStack \cup \{v\}$
9: for $u \in v.children$ do
10:     if negativeEdge OR $(v, u)$ is negative then
11:         return negativeCycleUtil(G, v, visited, recursionStack, True)
12:     else
13:         return negativeCycleUtil(G, v, visited, recursionStack, False)
14:     end if
15: end for
16: $recursionStack \leftarrow recursionStack \setminus \{v\}$
17: return False
```

For subsets of Φ we run the test of stratification. If it comes out true we check if this subset, i.e. our program P, satisfy our positive examples and do not satisfy our negative examples. If the test comes out false we select a different subset and retry the process, until either we find a subset which satisfies our constraints or every subset of Φ has been searched.

3.1.3 Non-monotonic Forward Chaining

For stratified programs we can construct a minimal and supported Herbrand model M_P of P by theorem 2.17. This model is our set of consequences $con_S(P)$ of P. We want to construct a program P such that our clauses and background knowledge $\mathcal{B}$ satisfy all positive examples $\mathcal{P}$ and do not satisfy any negative examples $\mathcal{N}$

$\mathcal{B} \cup P \vDash \gamma$ for all $\gamma \in \mathcal{P}$

$\mathcal{B} \cup P \nvDash \gamma$ for all $\gamma \in \mathcal{N}$

Using theorem 2.19 we can determine entailment of a stratified program P by construction of $con_S(P)$. Then we simply have to check if our positive examples $\mathcal{P}$ are elements of this set and that our negative examples $\mathcal{N}$ are not.

3.2 Stratified Negation as Failure in Differentiable Inductive Logic Programming

After introducing stratified negation as failure to Inductive Logic Programming we wish to implement it in a differentiable manner, introducing SNAF to Differentiable Inductive Logic Programming.

In δILP we map ground atoms to a real unit interval $[0,1]$ instead of the discrete values $\{True, False\}$ as in the original ILP system. Consequences of our clauses is now derived using the function $\mathcal{F}_c$ instead of the immediate consequence operator T_P/S_P[2]. The clauses of our final program are selected by retrieving the clauses whose clause weight is above a set threshold after training.

The changes to ILP as a satisfiability problem when introducing SNAF was

1. Add negation as failure into the construction of clauses
2. Ensure a selection of clauses such that the program is stratifiable
3. Forward chaining is altered to handle the non-monotonicity of negation as failure

1 was handled by extending the rule template. 2 was handled by constructing the dependency graph for our subset of clauses and checking for a cycle with a negative edge. 3 was handled by constructing $con_S(P)$ instead of $con_D(P)$ as in the original system.

Step number 1 is already handled since the process of clause construction does not change from ILP to δILP. Number 2 and 3 are the steps which we need to re-implement to fit the stratified programs. In addition, we need to define how to valuate literals with negation as failure.

3.2.1 Fuzzy Negation as Failure

To decide a fuzzy implementation of negation as failure we will consider its meaning. Negation as failure infers a negated literal *not p* by failing to prove *p*. In our valuations of ground atoms on an interval $[0,1]$ p is considered to have a probability to its truth. If p were to have the valuation

$$p \mapsto 0.6$$

then we consider p to have a 60 % probability of being true. Meaning that we cannot say with certainty that p is not true. Our proof of p does not fail (in a sense), and *not p* cannot be inferred. Only if we are certain of the falsity of p, i.e.

$$p \mapsto 0.0$$

can we infer *not p*.

An implementation which expresses this rationale is [3]

$$a_t(not\ \gamma) = \begin{cases} 1.0, & \text{if } a_t(\gamma) = 0.0. \\ 0.0, & \text{otherwise.} \end{cases}$$

This is know as *weak negation* [10].

A version of this, which allows us to decide at what point we consider something to be true, is *weak negation with threshold* [10]. Imposing that every atom which does not have a valuation of 1.0 has its negation valuated to 0.0 is rather strict. We can instead introduce a threshold $\theta \in [0,1]$ where if the valuation of γ is above θ then it is considered to be true. Conversely, if it is below θ it is considered to be false. We introduce a threshold θ to weak negation[4]:

$$a_t(not\ \gamma) = \begin{cases} 1.0, & \text{if } a_t(\gamma) > \theta \\ 0.0, & \text{otherwise.} \end{cases}$$

[2] T_P for definite programs and S_P for stratified programs

[3] Note that a_t is an indexed set of valuations at time step T, not a function. The equation mereley exists to illustrate valuation of negative literals. It would be more correct to give the indexes of *not* γ and γ.

[4] See footnote 3

However, our system requires gradient to flow through its network, meaning that having negated literals only take on the valuations 0.0 and 1.0 would not be beneficial. A continuous implementation of negation, which is most widely used, is *strong negation* [5]

$$a_t(not\,\gamma) = 1 - a_t(\gamma)$$

where the valuation of a negative literal is the complement of its positive form [10]. Now we allow our system to valuate *not p* to any value in the range $[0,1]$.

Each of the different NAF implementations is tested in section 4.3.

3.2.2 Stratification of δILP

In δILP, instead of using Boolean flags to choose a discrete subset of clauses, we use continuous weights to determine a probability distribution over clauses. To extract a human-readable logic program, we just take all clauses whose probability is over some constant threshold after training.

We do not want to perform forward chaining inference, i.e. apply $\mathcal{F}_c$ to our clauses, if the set of clauses does not form a stratified program. Hence, we need to enforce stratification before our inference step. After generating our set of all possible clauses to define our intensional predicates, we partition all clauses into a set of programs $\mathfrak{P}$

$$\mathfrak{P} = P_1 \cup P_2 \cup ... \cup P_n$$

where each P_i is a set of clauses which form a stratified program. I.e. each P_i can be partitioned

$$P_i = P_{i,1} \cup P_{i,2} \cup ... \cup P_{i,m} \qquad (P_{i,j} \text{ and } P_{i,k} \text{ disjoint for all } j \neq k)$$

and the clauses of these partitions satisfy the conditions of stratified programs from definition 2.12. Each P_i will contain several clauses which define the predicates of our program. The remaining task left to the system is to do a selection out of these clauses which will be our final program.

To partition the set of all clauses into stratified programs we place all intensional predicate into strata and, by following the definition of stratified programs (definition 2.12), we restrict how each predicate can be defined.

Consider a program which is defined by the following predicates: *target*, *pred1*, *pred2*, ..., *predN*, as well as a set of extensional relations. By placing each predicate in strata we ensure stratification. Extensional predicates we always place in the first stratum since they cannot by definition cause recursion through negation. The target predicate *target* is what we want to learn, while the auxiliary predicates *pred1*, *pred2*, ... *predN* will be used to define *target*. Hence, *target* will always be placed in the last stratum so that it can reference the auxiliary predicates.

What remains to be placed into strata are the auxiliary predicates which the system will invent to define the target predicate. For each auxiliary predicate the risk of recursion through negation is dependent on whether it is defined intensionally. Consider the following arbitrary program where the target predicate allows use of negation:

$$\begin{aligned} someRelation(a,b) &\leftarrow \\ someRelation(b,c) &\leftarrow \\ someRelation(a,c) &\leftarrow \\ target(X) &\leftarrow not\,pred(X), someRelation(X,X) \\ pred(X) &\leftarrow ... \end{aligned}$$

If *pred* allows intensional predicates in its body then the constructed clauses will include the following possible definition of *target* and *pred*:

$$\begin{aligned} target(X) &\leftarrow not\,pred(X),\ ... \\ pred(X) &\leftarrow target(X),\ ... \end{aligned}$$

[5]See footnote 3

Which is not a stratified program.

We want to avoid such clause combinations, but do not want to omit these clauses entirely. Then we run the risk of removing the correct clauses to define the target predicate. Instead we construct several clause sets which the system will run independently. In this case we construct one clause set which includes:

$$target(X) \leftarrow not\, pred(X)$$

and excludes

$$pred(X) \leftarrow target(X),$$

while another clause set will contain the second and not the first. This way we ensure that all clause combinations results in a stratified program and we do not omit any of the potentially necessary clauses.

This is done by constructing a program for each possible strata placements. For a program with the target predicate as well as two auxiliary intensional predicates we have the following combinations:

$\boldsymbol{P_{1,1}}$: [Extensional predicates]
$\boldsymbol{P_{1,2}}$: [target, pred1, pred2]

$\boldsymbol{P_{2,1}}$: [Extensional predicates]
$\boldsymbol{P_{2,2}}$: [pred1, pred2]
$\boldsymbol{P_{2,3}}$: [target]

$\boldsymbol{P_{3,1}}$: [Extensional predicates]
$\boldsymbol{P_{3,2}}$: [pred1]
$\boldsymbol{P_{3,3}}$: [target, pred2]

$\boldsymbol{P_{4,1}}$: [Extensional predicates]
$\boldsymbol{P_{4,2}}$: [pred2]
$\boldsymbol{P_{4,3}}$: [target, pred1]

$\boldsymbol{P_{5,1}}$: [Extensional predicates]
$\boldsymbol{P_{5,2}}$: [pred1]
$\boldsymbol{P_{5,3}}$: [pred2]
$\boldsymbol{P_{5,4}}$: [target]

$\boldsymbol{P_{6,1}}$: [Extensional predicates]
$\boldsymbol{P_{6,2}}$: [pred2]
$\boldsymbol{P_{6,3}}$: [pred1]
$\boldsymbol{P_{6,4}}$: [target]

For instance, in P_3 *target* can be defined by clauses with negative literals of *pred1* in the body, but can only use *pred2* positively. In P_4 it is the other way around.

Now every possible clause combination ensures a stratified program. In addition, we do not omit any clauses which might be the correct way to define our predicates.

Now we can perform inference on the clauses of each P_i separately without encountering the problems of negation and recursion. We iteratively select a program P_i and perform the remaining steps of the system: training our network until the weights of our clauses determining the probability that the clause is the correct way to define the intensional predicate, have reached our desired threshold. If the threshold is not reached, we select P_{i+1} and repeat the process.

The cardinality of $\mathfrak{P}$ is dependent on the number of intensional auxiliary predicates. For n auxiliary predicates:

$$|\mathfrak{P}| = (n-1)!$$

3.2.3 Non-Monotonic Differentiable Inference

The differentiable inference process in δILP is a reimplementation of the inference process in ILP. To adhere to the restrictions of stratified programs we will alter the inference operations in δILP to emulate the construction of $con_S(P)$, rather than $con_D(P)$ as in the original δILP system. $con_S(P)$ and $con_D(P)$ are represented by a final valuation set $a_T \in [0,1]^n$ for stratified and definite programs respectively. The construction of a_T was originally done in three steps

1. The application of $\mathcal{F}_c$ to the previous valuation.
2. Calculating the weighted valuation b_t based on the current clause weights. Step 1 and 2 represents the application of our immediate consequence operator T_P
3. Amalgamate the the previous valuations a_{t-1} with our current weighted valuation b_t. This represents the powers of T_P.

Repeat these steps T times as specified by the Program template (Definition 2.23) and we have our final valuation a_T.

Step 1 and 2 will remain the same as the definition of S_P is no different from the definition of T_P (only the powers of these operators differ) and the weighting of valuations is not a part of the construction of $con(P)$. We alter the amalgamation of valuations to fit the powers of S_P, and add a fourth step to join the consequences of each stratum of our program P.

3.2.3.1 Amalgamation of Consequences

The joining of consequences in the construction $con_D(P)$ is done by

$$T^0(I) = I$$
$$T^{(n+1)}(I) = T(T^n(I))$$

and is represented in δILP by the probabilistic sum of new and old valuations

$$a_{t+1} = a_t + b_t - a_t \cdot b_t$$

In this, $con_D(P)$ and $con_S(P)$ differ. Hence, we need to alter the calculation of a_{t+1}. $S^{(n+1)}$ differs from its monotonic counterpart $T^{(n+1)}$ by adding the union of $S^n(I)$ to $S(S^n(I))$.

$$T^{(n+1)}(I) = T(T^n(I))$$
$$S^{(n+1)}(I) = S(S^n(I)) \cup S^n(I)$$

$S(S^n(I))$ is the consequences of the previous consequences. It is represented by the probabilistic sum of the previous valuation and the weighted average of the next valuation. Hence, $S^n(I)$ is the previous valuation a_t

$$S(S^n(I)) \mapsto a_t + b_t - a_t \cdot b_t$$
$$S^n(I) \mapsto a_t$$

To take the union of these valuations we again take the probabilistic sum:

$$c_t \coloneqq a_t + b_t - a_t \cdot b_t$$
$$S(S^n(I)) \cup S^n(I) \mapsto c_t + a_t - c_t \cdot a_t$$

Another representation is:

$$S(S^n(I)) \cup S^n(I) \mapsto max(a_t + b_t - a_t, a_t)$$

as union is often represented by MAX in fuzzy set theory [11]. However, this implementation would never valuate ground atoms to a lower value than their initial valuation. This is not desired.

3.2.3.2 Joining Consequences of each Stratum

The joining of consequences for each stratum is done by

$$\begin{aligned}M_1 &= S^{\omega}_{P_1}(\varnothing)\\ M_2 &= S^{\omega}_{P_2}(M_1)\\ &\dots\\ M_n &= S^{\omega}_{P_n}(M_{n-1})\end{aligned}$$

As described in Section 3.2.2, we have partitioned our clauses into stratified programs. When deriving the consequences of clauses this will be done at different times. We sort the intensional predicates by strata and perform forward chaining for each predicate in order of which stratum they are partitioned into. Start with stratum 1 and proceed to the last. For each new stratum our initial valuation will be the valuation calculated for the previous stratum. We denote the valuation for stratum P_i at final time step T as a_{P_i}. Our final valuation after n strata and T time steps of forward chaining inference for each stratum is

$$\begin{aligned}a_{P_1} &= (a_0)_T && (a_0 \text{ defined by } \mathcal{B})\\ a_{P_2} &= (a_{P_1})_T\\ &\dots\\ a_{P_n} &= (a_{P_{n-1}})_T\end{aligned}$$

4 Experiments

4.1 ILP tasks with Negation

To test SNAFδILP we run it on a set of ILP tasks which naturally would use negation.

4.1.1 Learn innocent/1

As a simple negation test the system is given the task of determining who is innocent by knowing who is guilty. The constants is a set of people:

$$C = \{Paul, Randy, Rachel, Bob, Alice, Stan, Kyle, Peter, Tony, Monica\}$$

The background knowledge is a set of facts defining who is guilty using the *guilty* predicate:

$$\begin{aligned}\mathcal{B} = \{&guilty(Bob), guilty(Randy), guilty(Peter),\\ &guilty(Alice), guilty(Monica)\}\end{aligned}$$

The positive examples $\mathcal{P}$ are:

$$\begin{aligned}\mathcal{P} = \{&target(Paul), target(Rachel), target(Kyle),\\ &target(Tony), target(Stan)\}\end{aligned}$$

In all these examples, *target* is the target predicate we are trying to learn. In this case *target* = *innocent*. The negative examples $\mathcal{N}$ is the set containing all constants not found in the positive examples $\mathcal{P}$:

$$\begin{aligned}\mathcal{N} = \{&target(Bob), target(Randy), target(Peter),\\ &target(Alice), target(Monica)\}\end{aligned}$$

One possible language template for this task is:

$$\begin{aligned}&P_e : \{guilty/1\}\\ &P_i : \{target/1\}\end{aligned}$$

One suitable program template for this task is:

$$\begin{aligned}&\tau^1_{target} = (h = target, n_\exists = 0, int = False, neg = True)\\ &\tau^2_{target} = null\end{aligned}$$

This template specifies one clause for *target*. The architecture assumes that every predicate is defined by exactly two clauses specified by two rule templates. Here, we only need one clause to define *target*, so the second rule template is null. SNAFδILP can reliably solve this task. The solution found is:

$$target(X) \leftarrow not\ guilty(X)$$

4.1.2 Learn can_fly/1

The task is to learn which animal is able to fly. An animal is considered to be able to fly if it is a bird, but is not abnormal. Abnormal birds such as penguins cannot fly. The constants form a set of animals (different breeds of bird, dog and cat):

$$\begin{aligned}C = \{&blue\ bird, penguin, ostrich, blackbird, robin, sparrow, starling, chicken, kiwi,\\ &steamer\ duck, kakapo, cassowary, takahe, weka, pigeon, swan, duck, gold\ finch,\\ &woodpecker, blue\ tit, great\ tit, puddle, forrest\ cat, pitbull, golden\ retriever,\\ &perser, bengal, siamese, sphynx, ragdoll, savannah, sibirian\ cat, greyhound,\\ &malteser, dobermann, rottweiler, boston\ terrier, scottish\ fold, exotic, russian\ blue\}\end{aligned}$$

The background knowledge is a set of facts defining *is_bird* and *abnormal* relation on animals.

$$\begin{aligned}\mathcal{B} = \{&is\_bird(woodpecker), is\_bird(bluebird), is\_bird(penguin),\\ &is\_bird(blue\,tit), is\_bird(great\,tit), is\_bird(ostrich),\\ &is\_bird(black\,bird), is\_bird(robin), is\_bird(sparrow),\\ &is\_bird(starling), is\_bird(chicken), is\_bird(kiwi),\\ &is\_bird(steamer\,duck), is\_bird(kakapo), is\_bird(takahe),\\ &is\_bird(weka), is\_bird(pigeon), is\_bird(swan),\\ &is\_bird(duck), is\_bird(gold\,finch), is\_bird(bluebird)\}\\ &\cup\\ \{&abnormal(penguin), abnormal(ostrich), abnormal(chicken),\\ &abnormal(kiwi), abnormal(steamer\,duck), abnormal(kakapo),\\ &abnormal(takahe), abnormal(weka)\}\end{aligned}$$

The positive examples $\mathcal{P}$ are:

$$\begin{aligned}\mathcal{P} = \{&target(blue\,bird), target(black\,bird), target(robin),\\ &target(sparrow), target(starling), target(pigeon),\\ &target(swan), target(duck), target(gold\,finch),\\ &target(wood\,pecker), target(blue\,tit), target(great\,tit)\}\end{aligned}$$

In this task *target* = *can_fly*. The negative examples $\mathcal{N}$ are:

$$\begin{aligned}\mathcal{N} = \{&target(penguin), target(chicken), target(kiwi),\\ &target(steamer\,duck), target(takahe), target(puddle),\\ &target(siamese), target(forrest\,cat), target(pitbull),\\ &target(sphynx), target(sibirian\,cat), target(greyhound),\\ &target(russian\,blue)\}\end{aligned}$$

One possible language template for this task is:

$$\begin{aligned}&P_e : \{is\_bird/1, abnormal/1\}\\ &P_i : \{target/1\}\end{aligned}$$

One suitable program template for this task is:

$$\begin{aligned}&\tau^1_{target} = (h = target, n_\exists = 0, int = False, neg = True)\\ &\tau^2_{target} = null\end{aligned}$$

This template specifies one clause for *target*.

SNAFδILP can reliably solve this task. The solution found is:

$$target(X) \leftarrow not\ abnormal(X), is\_bird(X)$$

4.1.3 Learn even/1

The original δILP system of [5] was able to learn the *even* predicate over natural numbers:

$$\begin{aligned}target(X) &\leftarrow zero(X)\\ target(X) &\leftarrow target(Y), pred(Y, X)\\ pred(X, Y) &\leftarrow succ(X, Z), succ(Z, Y),\end{aligned}$$

where $target = even$.

To see if our extended system with negation is still able to learn programs that do not require negation, we ran the system with the same problem specification as before, but allowing the use of negation.

The background knowledge is the set of basic arithmetic facts defining the *zero* predicate and *succ* relation on numbers up to 20:

$$\mathcal{B} = \{zero(0), succ(0,1), succ(1,2), succ(2,3), ..., succ(19,20)\}$$

The positive examples $\mathcal{P}$ are:

$$\mathcal{P} = \{target(0), target(2), target(4), ..., target(20)\}$$

In this case, *target* = *even*. The negative examples $\mathcal{N}$ is the set containing all constants not found in the positive examples $\mathcal{P}$:

$$\mathcal{N} = \{target(1), target(3), target(5), ..., target(19)\}$$

The original language template for this task was:

$$\begin{aligned} P_e &: \{zero/1, succ/2\} \\ P_i &: \{target/1, pred/2\} \end{aligned}$$

Here, *pred* is an auxiliary binary predicate.

The program template used in the original system was:

$$\begin{aligned} \tau^1_{target} &= (h = target, n_\exists = 0, int = False) \\ \tau^2_{target} &= (h = target, n_\exists = 1, int = True) \\ \tau^1_{pred} &= (h = pred, n_\exists = 1, int = False) \\ \tau^2_{pred} &= null \end{aligned}$$

This template specifies two clauses for *target*, and one clause for the auxiliary predicate *pred*.

In our extended system the rule template has a third parameter *neg* (True/False) which allows the use of negation in our clauses. We alter the program template to include the use of negation:

$$\begin{aligned} \tau^1_{target} &= (h = target, n_\exists = 0, int = False, Neg = True) \\ \tau^2_{target} &= (h = target, n_\exists = 1, int = True, Neg = False) \end{aligned}$$

$$\begin{aligned} \tau^1_{pred} &= (h = pred, n_\exists = 1, int = False, Neg = False) \\ \tau^2_{pred} &= null \end{aligned}$$

Here δILP has to either find a way to use negation or ignore the clauses which use it. Note that if *neg* is set to $False$ the system functions the same as the original δILP system.

In some cases SNAFδILP learns the same program as provided when not using negation:

$$\begin{aligned} target(X) &\leftarrow zero(X) \\ target(X) &\leftarrow target(Y), pred(Y,X) \\ pred(X,Y) &\leftarrow succ(X,Z), succ(Z,Y) \end{aligned}$$

Other times it learns:

$$\begin{aligned} target(X) &\leftarrow zero(X), not\ succ(X,X) \\ target(X) &\leftarrow target(Y), pred(Y,X) \\ pred(X,Y) &\leftarrow succ(X,Z), succ(Z,Y) \end{aligned}$$

not succ(X, X) will never be true since no number is the successor of itself. The two programs are equivalent.

SNAFδILP successfully manages to either use negation correctly, but redundantly, or ignores it, resulting in a working program.

4.1.4 Learn has_roommate/1

In this task we wish to learn who has a roommate. We consider someone to have a roommate iff they are married, but their partner is not a researcher. Researchers are busy people, often on expeditions to Antarctica. The constants form a set of people:

$$\begin{aligned}C = \{&Paul, Randy, Rachel, Bob, Alice, Steve,\\ &Frank, Julia, Stan, Kyle, Peter, Tony, Monica,\\ &Carl, Dolores, Tommy, Pedro, Will, Sophie,\\ &Eric, Jon, Robert, Sansa, Arya, Tormund, Mance\}\end{aligned}$$

The background knowledge is a set of facts defining who is married by the *married* relation, and who is a researcher by the *researcher* predicate. *married* is a symmetrical relation, hence if *married(Bob, Rachel)* then we also have *married(Rachel, Bob)*. We omit these ground atoms in the following set:

$$\begin{aligned}\mathcal{B} = \{&married(Bob, Rachel), married(Randy, Alice),\\ &married(Steve, Julia), married(Frank, Monica),\\ &married(Stan, Dolores), married(Will, Sophie),\\ &married(Eric, Arya)\}\\ &\cup\\ \{&researcher(Robert), researcher(Eric),\\ &researcher(Bob), researcher(Frank),\\ &researcher(Monica), researcher(Mance),\\ &researcher(Jon), researcher(Tormund),\\ &researcher(Sansa)\}\end{aligned}$$

The positive examples $\mathcal{P}$ are:

$$\begin{aligned}\mathcal{P} = \{&target(Randy), target(Alice), target(Steve),\\ &target(Julia), target(Dolores), target(Stan),\\ &target(Will), target(Sophie)\}\end{aligned}$$

The negative examples $\mathcal{N}$ is the set containing all constants not found in the positive examples $\mathcal{P}$:

$$\begin{aligned}\mathcal{N} = \{&target(Bob), target(Rachel), target(Frank),\\ &target(Monica), target(Kyle), target(Peter),\\ &target(Tony), target(Carl), target(Tommy),\\ &target(Eric), target(Arya), target(Jon),\\ &target(Robert), target(Mance), target(Tormund)\}\end{aligned}$$

One possible language template for this task is:

$$\begin{aligned}P_e &: \{married/2, researcher/1\}\\ P_i &: \{pred/2, target/1\}\end{aligned}$$

Here *pred* is an auxiliary predicate that the program will have to invent.

One suitable program template for this task is:

$$\tau^1_{target} = (h = target, n_\exists = 1, int = True, neg = False)$$
$$\tau^2_{target} = null$$

$$\tau^1_{pred} = (h = pred, n_\exists = 0, int = False, neg = True)$$
$$\tau^2_{pred} = null$$

This template specifies one clause for *target*, allowing use of one existentially quantified variable and disallowing negation. *pred* is specified to be defined using one clause, with no existentially quantified variables and negation is allowed. One solution found is:

$$\begin{aligned} target(X) &\leftarrow married(X,Y), pred(X,Y) \\ pred(X,Y) &\leftarrow not\ researcher(X), not\ researcher(Y) \end{aligned}$$

Another solution found is:

$$\begin{aligned} target(X) &\leftarrow pred(X,Y), pred(Y,X) \\ pred(X,Y) &\leftarrow married(Y,X), not\ researcher(Y) \end{aligned}$$

The significance of this result is that our extended δILP system has managed to learn a program which uses binary predicate, in addition to predicate invention using negation. The complexity of these programs exceed the complexity of the programs in the previous experiments in this section.

4.1.5 Learn two_children/1

The task is to learn the predicate *has at least two children*. In a directed graph learn which node has at least two children. As background knowledge we specify a directed graph:

$$\begin{tikzcd}
 & & f \arrow[dl] \arrow[dr] & & \\
h & a \arrow[rr] \arrow[drr] & & b \arrow[d] & g \arrow[dl] \arrow[dlll] \\
 & e & & c \arrow[ll] \arrow[dl] & \\
 & & d \arrow[ul] & &
\end{tikzcd}$$

using the *edge* relation. To distinguish between equal nodes we also have the *equals* relation.

$$\begin{aligned} \mathcal{B} = \{&edge(a,b), edge(a,c), edge(b,c), edge(c,e), edge(c,d), \\ &edge(d,e), edge(f,a), edge(f,b), edge(g,c), edge(g,e)\} \\ &\cup \\ &\{equals(X,Y) \mid X = Y\} \end{aligned}$$

The constants are the nodes of the graph:

$$C = \{a, b, c, d, e, f, g, h\}$$

The positive examples $\mathcal{P}$ are:

$$\mathcal{P} = \{target(a), target(c), target(f), target(g)\}$$

The negative examples $\mathcal{N}$ is the set containing all constants not found in the positive examples $\mathcal{P}$:

$$\mathcal{N} = \{target(b), target(d), target(e), target(h)\}$$

One possible language template for this task is:

$$P_e : \{edge/2, equals/2\}$$
$$P_i : \{pred/2, target/1\}$$

Here *pred* is an auxiliary predicate that the program will have to invent.

One suitable program template for this task is:

$$\begin{aligned}\tau^1_{target} &= (h = target, n_\exists = 1, int = True, neg = False)\\ \tau^2_{target} &= null\end{aligned}$$

$$\begin{aligned}\tau^1_{pred} &= (h = pred, n_\exists = 1, int = False, neg = True)\\ \tau^2_{pred} &= null\end{aligned}$$

This template specifies one clause for *target*, allowing use of one existentially quantified variable, allowing use of intensional predicates in the body and disallowing negation. *pred* is specified to be defined using one clause, with one existentially quantified variable and negation is allowed. A valid program that SNAFδILP finds is:

$$target(X) \leftarrow edge(X,Y), pred(X,Y)$$
$$pred(X,Y) \leftarrow edge(X,Z), not\ equals(Y,Z)$$

4.1.6 Learn not_grandparent/2

The task is to learn the predicate *not_grandparent(X, Y)* which is true if X is *not* the grandparent of Y. This task is an extension of a program learnt by the original system: *grandparent(X, Y)*. We have a set of constants representing people:

$$C = \{a, b, c, d, e, f, g, h, i\}$$

As background knowledge we specify a set of *mother* and *father* relations:

$$\begin{aligned}\mathcal{B} = \{&mother(i,a), father(a,b), father(a,c), father(b,d),\\ &father(b,e), mother(c,f), mother(c,g), mother(f,h)\}\end{aligned}$$

The *negative* examples are:

$$\mathcal{N} = \{target(i,b), target(i,c), target(a,d), target(a,e), target(a,f), target(a,g), target(c,h)\}$$

The positive examples is the set of pairings that do not appear in the negative examples, i.e. all those who do not have a grandparent. This set considerably larger than the negative set: $|\mathcal{N}| = 7, \ \ |\mathcal{P}| = 74$. To avoid data imbalance $\mathcal{P}$ is reduced to the same cardinality by random sampling.

One possible language template for this task is:

$$P_e : \{mother/2, father/2\}$$
$$P_i : \{pred_1/2, pred_2/2, target/2\}$$

Here $pred_1$ and $pred_2$ are auxiliary predicates the system will have to invent.

One suitable program template for this task is:

$$\tau^1_{target} = (h = target, n_\exists = 0, int = True, neg = True)$$
$$\tau^2_{target} = null$$

$$\tau^1_{pred_1} = (h = pred_1, n_\exists = 1, int = True, neg = False)$$
$$\tau^2_{pred_1} = null$$

$$\tau^1_{pred_2} = (h = pred_2, n_\exists = 0, int = False, neg = False)$$
$$\tau^2_{pred_2} = null$$

A solution found is:

$$target(X,Y) \leftarrow not\ pred_1(X,Y)$$
$$pred_1(X,Y) \leftarrow pred_2(X,Z), pred_2(Z,Y)$$
$$pred_2(X,Y) \leftarrow mother(X,Y), father(X,Y)$$

This program has again exceeded the complexity of previously learned predicates. *not_grandparent* requires two invented auxiliary, binary predicates.

4.2 Failed Learning Tasks

For each learning task in section 4.1 the complexity of the programs required was increased. With the increase in complexity we see a steady drop off in the systems ability to correctly learn a working program with respect to the examples (see section 4.3). Two programs were omitted from section 4.1 due to SNAFδILPs inability to solve the task. The first task was to learn the target predicate *no_negative_cycles*. Given a graph with positive and negative edges[6] the program would say if a given node is part of a cycle which includes a negative edge. For example:

$$\begin{tikzcd}
& f \arrow[ld] \arrow[rd] & & \\
a \arrow[rr] \arrow[rrdd, dashed] & & b \arrow[dd] & \\
& & & g \arrow[llld, bend right] \arrow[ld] \\
e \arrow[uu] \arrow[rr, bend right] & & c \arrow[ll, dashed, bend right] \arrow[ld] & \\
h \arrow[r, bend right] & d \arrow[l, bend right] & &
\end{tikzcd}$$

Dotted edges are negative and whole edges are positive.

A suitable program for the system to learn was:

$$target(X) \leftarrow pred(X,Y)$$
$$pred(X,Y) \leftarrow edge(X,Y), not\ negative(X,Y)$$
$$pred(X,Y) \leftarrow pred(X,Z), pred(Z,Y)$$

This program requires predicate invention, recursion and use of negation. Unfortunately, SNAFδILP failed this task. This failure was most likely due to getting stuck in local optima, an issue the original δILP also had.

[6]The positive and negative edges are merely labels, not numeric values.

The other learning task omitted was *greater_or_equal*. Given a set of numbers (0-9) the task is to correctly say whether a number is greater than another. The intention for this task was for the system to learn *less_than*, as described in [5] and then negate this program to get *greater_or_equal*:

$$\begin{aligned} target(X,Y) &\leftarrow not\, pred(X,Y) \\ pred(X,Y) &\leftarrow succ(X,Y) \\ pred(X,Y) &\leftarrow pred(X,Z), pred(Z,Y) \end{aligned}$$

This task also requires predicate invention, recursion and negation, being more complex than previously learned programs.

Again, SNAFδILP failed to learn a program that valued the examples correctly.

4.3 Valuation of Negation as Failure

In section 3.2.1 we discussed the different possible implementations of a fuzzy negation as failure. These were *weak negation*[7]:

$$a_t(not\,\gamma) = \begin{cases} 1.0, & \text{if } a_t(\gamma) = 0.0 \\ 0.0, & \text{otherwise,} \end{cases}$$

weak negation with threshold[8]:

$$a_t(not\,\gamma) = \begin{cases} 1.0, & \text{if } a_t(\gamma) > \theta \\ 0.0, & \text{otherwise,} \end{cases}$$

and *strong negation*[9]:

$$a_t(not\,\gamma) = 1 - a_t(\gamma).$$

To see which implementation yielded the best result, i.e. the lowest loss, each learning task in section 4.1 was performed with the different implementations. The threshold θ selected for weak negation with threshold was 0.6 (chosen arbitrarily). The results are shown in the following table:

Task	Strong Negation	Weak Negation	Weak Negation with Threshold
Innocent	100.0	0.0	96.0
Can Fly	100.0	0.0	100.0
Even	90.0	0.0	90.0
Has Roommate	50.0	0.0	80.0
Two Children	20.0	0.0	80.0
Not Grandparent	56.0	0.0	20.0

The percentage of runs that achieve less than 1e-2 mean squared test error.

As suspected weak negation performed poorly, not even managing a single learning task in any of the runs of the system.

Also as expected strong negation performed well. This can be attributed to the fact that strong negation is a continuous implementation which allows gradient to flow through the network.

A surprising result is how well weak negation with threshold performed, yielding as good or better than strong negation in certain learning tasks.

[7]Note that a_t is an indexed set of valuations at time step T, not a function. The equation merely exists to illustrate valuation of negative literals. It would be more correct to give the indexes of *not* γ and γ.

[8]See footnote 7

[9]See footnote 7

Concluding which implementation of negation as failure to use in SNAFδILP is not clear cut given this experiment. Weak negation we can safely discard as an option, but which of strong negation and weak negation with threshold to choose requires further testing. In the simpler tasks: Innocent and Can Fly, both implementation constructed a valid program for nearly each run of the system. Strong negation managed every time, while weak negation with threshold failed Innocent a few times. In the intermediate tasks: Even, Has Roommate and Two Children, weak negation with threshold outperformed strong negation by quiet a large margin. However, in the difficult task of Not Grandparent weak negation with threshold underperformed. In the learning tasks No Negative Cycles and Greater Or Equal from Failed Learning Tasks (Section 4.2) the mean squared validation error reached a lower value when using strong negation than when using weak negation with threshold. This may not bare any significance since either implementation managed to learn the intended program for those tasks.

As stated, the threshold θ for weak negation with threshold was selected arbitrarily to 0.6. Other values could potentially yield a better result, solidifying weak negation with threshold as the better implementation of negation as failure in SNAFδILP. This is further discussed in Section 5.

4.4 Dealing with Mislabelled Data

The discrete standard ILP system finds a set R such that:

$$\mathcal{B}, R \vDash \forall \gamma \in \mathcal{P}$$
$$\mathcal{B}, R \nvDash \forall \gamma \in \mathcal{N}$$

This strict requirement does not allow for any mislabelled data. With only a single wrongly labeled element this ILP system will not be able to find the intended program. Consider the task of learning the *even* predicate with the following positive and negative examples:

$$\mathcal{P} = \{0, 1, 2, 4, 6, 8, 10\}$$
$$\mathcal{N} = \{1, 3, 5, 7, 9, 11\}$$

There is no program which can satisfy 1 and not satisfy 1 at the same time. If we omit 1 from $\mathcal{N}$ then the program we learn will have to add clauses for such edge cases. Something that is not feasible for greater data errors.

This is the key property of δILP: handling erroneous and noisy data, as it is minimizing a loss rather than trying to satisfy a strict requirement. As with the δILP, SNAFδILP was tested on its ability to correctly learn the intended logic program when given partially mislabelled data. Each learning task described in Section 4.1 was given a parameter $\rho \in [0, 1]$ specifying what percentage of the positive and negative examples were mislabelled. Given proportions ρ a random sample of $\mathcal{P}$ and $\mathcal{N}$ were transferred to the other group, and the system tries to learn as before. The implementation of fuzzy negation as failure for this experiment was strong negation. The results are shown in the following table:

$\boldsymbol{\rho}$	Innocent	Can Fly	Even	Has Roommate	Two Children	Not Grandparent
0.05	0.00	0,00	0,21	0,07	0,09	0,05
0.10	0.00	0.08	0,45	0,15	0,07	0,02
0.15	0.00	0.12	0,45	0,20	0,11	0,50
0.20	1.30	0.18	0,68	0,24	0,04	0,50
0.30	1.26	0.28	0,85	0,47	0,55	1,12
0.50	2.55	0.70	1,07	0,82	1,07	2,29

Average mean squared test error for each proportion of mislabelled examples.

The result shows that SNAFδILP is somewhat robust to mislabelled data. As the proportion ρ increases the mean squared test error degrades gracefully for most tasks. Innocent and Not

Grandparent are the exceptions. Innocent has a very small dataset:

$$|\mathcal{P}| = 5 = |\mathcal{N}|.$$

This means that the proportion ρ will have a larger effect on the systems ability to learn the correct program since the data correctly labeling the examples is so small. Not Grandparent is the most complex program that SNAFδILP learned. The more complex the program is the more correct data the system will require. The loss is then more prone to deteriorate when the system is given mislabeled data for the more complex tasks.

5 Further Work

Due to computational and time limitations of this project there are a number of planned implementations and experiments that was not executed.

5.1 Different thresholds for weak negation with threshold

In our implementations of fuzzy negation as failure the one that performed quiet well was weak negation with threshold. The chosen threshold θ of 0.6 was an arbitrary choice. A further experiment of this implementations is to test the system using a larger range of threshold values $\theta \in [0, 1]$. If any of these thresholds yield an even better result we would be able to conclude that weak negation with threshold is the better implementation of fuzzy negation as failure.

5.2 Other learning tasks

In [5] the original δILP system was given a set of benchmark learning tasks taken from four domains: arithmetic, lists, group-theory, and family tree relations. Some of the arithmetic examples appeared in the work of Cropper and Muggleton [4]. The list examples are used by Feser, Chaudhuri, and Dillig [6]. The family tree dataset comes from Wang, Mazaitis, and Cohen [13]. When introducing negation to this system we needed tasks in which it was natural to use negation. No such existing benchmark tests were found, and instead a set of learning tasks were invented. A future experiment would be to create different learning tasks which would further test the systems ability to learn. Different tasks from a selection of domains, establishing a benchmark set of tasks for negation with varying degrees of complexity.

5.3 Learning Parameters

Like δILP, SNAFδILP uses a set of parameters to solve the given learning task. These parameters include:

- a time step parameter T of how many steps of forward chaining to execute for each clause per training step,
- number of training steps for the learning task,
- and language and rule templates used in the problem specification.

In ILP deriving the consequences of a clause c is done until no more consequences can be inferred. In SNAFδILP (and δILP) consequences are represented as valuations. These valuations do not have a final state, as in we cannot infer anymore. The inference method will keep on calculating new valuations for as long as the system is run. Instead, we restrict the inference process to perform T steps of forward chaining. SNAFδILP has T set to 10, regardless of which learning task is being solved. Which values of T is best for specific learning tasks, and in general, is an experiment left to future work.

As described in Section 2.6.6 each learning task was trained for 200 - 500 steps depending on the complexity of the program to reduce computational cost. How many training steps was executed for the given learning task was chosen based on computational capacity and whether it seemed to have gotten stuck in a local optimum. Less complex programs such as Innocent and Can Fly needed only around 200 training steps to reach a satisfactory loss. If they did not reach a low loss after 200 training steps we considered the system to have gotten stuck in a local optimum. However, this is not always the case. In certain cases the system simply needs more training steps to get out of the local optimum and potentially reach a global optimum. Hence, all learning tasks should be run with a large number of training steps. This was not possible due to computational limitations.

As discussed in [5] δILP, and ILP systems in general, has a language bias. In order to have the system find the correct clauses to define our program we reduce the problem space using a language and program template, which specifies the possible clauses for the problem. The language and program template is hand engineered by a human with the intent of the system learning a specific set of clauses. However, while a human can invent a solution to these learning tasks there are often other solutions which we do not necessarily think of. By hand engineering a language and program template we might be omitting clauses of a possible solution to the problem. Giving the system a variety of language and program templates might lead to the system finding another solution which it learns more consistently than the solution the human intended for the system to find.

6 Conclusion

Stratified Negation as Failure in Differentiable Inductive Logic Programming (SNAFδILP) is an extension of the Differentiable Inductive Logic Programming system (δILP) from [5], which constructs stratified logic programs given a high level specification, using neurosymbolic methods.

This system is able to learn moderately complex programs with unary and binary predicates using negation and predicate invention. Unlike traditional ILP systems, SNAFδILP is shown to be moderately robust to mislabelled data, in most cases learning the intended program with up to 10 % mislabelled training data. Unlike neural networks, SNAFδILP manages to solve the learning tasks with great data efficiency. We evaluated the system on six symbolic ILP tasks, and showed that it can consistently solve problems with moderate complexity, while also solving more complex tasks with modest consistency.

SNAFδILP was implemented with the intent to extend the original δILP system with negation as failure by constructing stratified programs to avoid the problem of recursion and negation. While managing to learn programs using negation and programs using recursion, SNAFδILP failed to learn programs using both negation and recursion. These tasks (discussed in Section 4.2) might have had too high complexity for our system to learn, but other learning tasks where both recursion and negation is necessary might be solvable with this system.

A No Loss of Generality

In δILP and SNAFδILP the following two restrictions are set upon the construction of possible clauses:

- each predicate is defined by at most two clauses,
- each clause has at most two literals in its body.

With these restrictions we can still create programs which are equivalent to programs that do not have these restrictions. We consider programs to be equivalent if for every query they answer the same.

Definition A.1. *A program P_1, with language L_{P_1}, and P_2, with language L_{P_2}, are equivalent with respect to the language L_{P_1} if for every query in L_{P_1}, they answer the same.*

$$P_1 \equiv P_2 \quad if \quad \forall \gamma \in L_{P_1} \quad P_1?\gamma = P_2?\gamma$$

When constructing equivalent programs that adhere to these restrictions auxiliary predicates are introduced. These auxiliary predicates are not part of the language of the original program, hence they will only exist as subgoals in the new definitions of the relations of our original program and will never be queried directly. Therefore, we are content with equivalence with respect to our original language.

Theorem A.2. *Logic programs P_1 and P_2 will, for every query, have the same query answer if they have the same Herbrand model.*

Proof. Since a Herbrand model specifies which ground atoms are true and false, this directly follows. □

A.1 No Loss of Generality for Stratified Programs

In SNAFδILP we have extended the system to construct stratified logic programs. To create stratified programs with the two restrictions which are equivalent to programs without these restrictions we use the following schemes.

Scheme A.1 (Predicates are defined by at most two clauses)**.** *Let P_1 be a stratified program with a finite set C of constants and a predicate p defined by $m > 2$ clauses:*

$$\begin{aligned} p(X) &\leftarrow \beta_1(Y_1) \\ p(X) &\leftarrow \beta_2(Y_2) \\ &\dots \\ p(X) &\leftarrow \beta_m(Y_m) \end{aligned}$$

where $X, Y_1, ..., Y_m$ are sequences of variables. These sequences can contain shared variables, no variables or identical variables. All variables that appear in the head of the clause must appear in the body of the clause, i.e.:

$$\forall i : X \subseteq Y_i$$

Each β_i denote either a positive literal: $\rho_i(Y_i)$ or a negative literal: not $\eta_i(Y_i)$. In addition to the clauses defining p, P_1 will contain extensional or intensional definitions for each β_i. These predicates are arbitrary, hence we do not write the clauses of these predicates.

As P_1 is a stratified program it will be partitioned into strata. All clauses defining a predicate must be in the same stratum. Hence, all clauses for the predicate p are in the same stratum. All predicates used to define p, i.e. predicates in the body of the clauses of p, must be defined in the same or an earlier stratum if it is a positive literal, or in an earlier stratum if it is a negative literal.

From P_1 we construct a stratified program P_2 where each predicate is defined by exactly two clauses. Given the clauses of p we create a set of auxiliary predicates which are used to define p.

For each second clause of p define a new auxiliary predicate q_i expressing the clauses of p:

$$\begin{aligned} p(X) &\leftarrow \beta_1(Y_1) \\ p(X) &\leftarrow q_1(X) \\ q_1(X) &\leftarrow \beta_2(Y_2) \\ q_1(X) &\leftarrow q_2(X) \\ &\ldots \\ q_{n-2}(X) &\leftarrow \beta_{m-1}(Y_{m-1}) \\ q_{n-2}(X) &\leftarrow \beta_m(Y_{m-1}) \end{aligned}$$

As before, each β_i denote either a positive literal: $\rho_i(Y_i)$ or a negative literal: not $\eta_i(Y_i)$. X is a sequence of variables that originally appear in p. $Y_1, ..., Y_m$ are sequences of variables that appear in the respective β_i.

As with P_1, P_2 will in addition to these clauses contain clauses for each β_i. The set C of constants remains the same in P_2 as in P_1.

In P_2 p and all β_i is partitioned into strata exactly the same as in P_1. All auxiliary predicates q_i are placed in the same stratum as p.

Scheme A.2 (Each clause has at most two literals in the body)**.** *Let P_1 be a stratified program with a finite set C of constants and a clause c defining a predicate p with $m > 2$ literals in the body:*

$$p(X) \leftarrow \rho_1(Y_1), ..., \rho_p(Y_p), \textit{ not } \eta_1(Z_1), ..., \textit{not } \eta_n(Z_n) \qquad \text{(n+p = m)}$$

where $X, Y_1, ..., Y_p, Z_1, ..., Z_n$ are sequences of variables. These sequences can contain shared variables, no variables or identical variables. All variables that appear in the head of the clause must appear in the body of the clause, i.e.:

$$X \subseteq Y_1 \cup ... \cup Y_p \cup Z_1 \cup ... \cup Z_n.$$

$\rho_i(Y_i)$ denotes positive literals and $\eta_i(Y_i)$ denotes negative literals. In addition to the clause c, P_1 will contain extensional or intensional definitions for each β_i. These predicates are arbitrary, hence we do not write the clauses of these predicates.

As P_1 is a stratified program it will be partitioned into strata. All clauses defining a predicate must be in the same stratum. Hence, all clauses for the predicate p are in the same stratum. All predicates used to define p, i.e. predicates in the body of the clauses of p, must be defined in the same or an earlier stratum if it is a positive literal, or in an earlier stratum if it is a negative literal.

From P_1 we construct a stratified program P_2 where each clause has exactly two literals in the body. Given the clause c we create a set R of normal clauses with each having exactly two literals in the body. The clause c can be equivalently expressed by adding an auxiliary predicate q_i for every second literal in the body of c. Clause set R:

$$\begin{aligned} p(X) &\leftarrow \beta_1(Y_1), q_1(W_1) \\ q_1(W_1) &\leftarrow \beta_2(Y_2), q_2(W_2) \\ &\ldots \\ q_{n-2}(W_{n-2}) &\leftarrow \beta_{n-1}(Y_{p-1}), \beta_n(Y_p). \end{aligned}$$

β_i is either a positive or negative literal. X is a sequence of variables that appear in p in the clause c. $Y_1, ..., Y_m$ are sequences of variables that appear in the respective ρ_i and η_j in c. $W_1, ..., W_{n-2}$ are sequences of variables used in the auxiliary predicates q_i, where

$$W_1 = X \cup \{y \mid y \in Y_1 \textit{and } y \textit{ existentially quantified}\}$$
$$W_i = W_{i-1} \cup \{y \mid y \in Y_i \textit{and } y \textit{ existentially quantified}\}.$$

We include each existentially quantified variable in the auxiliary predicates to have them denote the same variable throughout the set of clauses, as in c.

As with P_1, P_2 will in addition to these clauses contain clauses for each β_i. The set C of constants remains the same in P_2 as in P_1.

In P_2 p and all β_i is partitioned into strata exactly the same as in P_1. All auxiliary predicates q_i are placed in the same stratum as p.

These schemes preserve the expressiveness of our language, having no loss of generality when restricting our clauses to adhere to the restrictions. This is shown in the following two theorems.

Theorem A.3. *For any stratified program P_1 there exists an equivalent program P_2 where predicate is defined by at most two clauses.*

Proof. □

Write this proof. Will do after the proof for Theorem A.4 is good.

Theorem A.4. *For any stratified program P_1 there exists an equivalent program P_2 where each clause has at most two literals in the body.*

Proof. **Proof by induction on number of strata.**
Let P_1 be a stratified program with an arbitrary amount of clauses and specifically a clause c with n literals in the body:

$$c = p(X) \leftarrow \beta_1(Y_1), ..., \beta_n(Y_n)$$

where each β_i is a predicate defined by other clauses in P_1.
Base Case
Let P_1 be stratified into one stratum (following Definition 2.12):

$$P_1 = P_{1,1}$$

Hence, all literals in the body of the clauses of P_1 are positive. P_1 has a minimal Herbrand model $con_S(P_1)$ constructed by Definition 2.18.

We construct a program P_2 by Scheme A.2, where clause c is transformed into a set of clauses R:

$$\begin{aligned} p(X) &\leftarrow \beta_1(Y_1), q_1(Z_1) \\ q_2(Z_1) &\leftarrow \beta_2(Y_2), q_2(Z_2) \\ &... \\ q_{n-2}(Z_{n-2}) &\leftarrow \beta_{n-1}(Y_{n-1}), \beta_n(Y_n), \end{aligned}$$

while the other clauses of P_1 remain the same:

$$P_1/\{c\} = P_2/R.$$

Show that

$$\forall \alpha \in L_{P_1} : \alpha \in con_S(P_1) \text{ iff } \alpha \in con_S(P_2).$$

Case 1: If $\alpha \in con_S(P_1)$ then $\alpha \in con_S(P_2)$
When deriving consequences of a stratified program we continually apply the immediate consequence operator S_P ω times (Definition 2.16), for each stratum. Let $p(t_1, ..., t_k)$ be a ground atom derived from the clause c in P_1 at the i'th application of S_P:

$$p(t_1, ..., t_k) \in S^i_{P_{1,1}}(\varnothing)$$

where $t_1, ..., t_k$ is a sequence of terms that substituted the sequence of variables X in c. Then we know that

$$\beta_1(T_1), ..., \beta_n(T_n) \in S^{i-1}_{P_{1,1}}(\varnothing)$$

where $T_1, ..., T_n$ are sequences of terms substituted for the sequences of variables $Y_1, ..., Y_n$. Since P_1 and P_2 are the same a part from the clause c and the set of clauses R we know that

$$\exists j \in \mathbb{N} : \beta_1(T_1), ..., \beta_n(T_n) \in S^{j}_{P_{2,1}}(\varnothing).$$

Then we get

$$q_{n-2}(T) \in S^{j+1}_{P_{2,1}}(\varnothing)$$

from the last clause in R:

$$q_{n-2}(Z_{n-2}) \leftarrow \beta_{n-1}(Y_{n-1}), \beta_n(Y_n),$$

where T is a sequence of terms substituted for the sequence of variables Z_{n-2} in q_{n-2}. Continually applying the immediate consequence operator $j + (n-2)$ times (going through all clauses of R) we get

$$p(t_1, ..., t_k) \in S^{j+(n-2)}_{P_{2,1}}(\varnothing).$$

Hence, if $p(t_1, ..., t_k) \in con_S(P_1)$ then $p(t_1, ..., t_k) \in con_S(P_2)$.

Case 2: If $\alpha \in con_S(P_2)$ then $\alpha \in con_S(P_1)$
Let $p(t_1, ..., t_k)$ be a ground atom derived from the first clause in R:

$$p(X) \leftarrow \beta_1(Y_1), q_1(Z_1)$$

in P_2 at the i'th application of S_P:

$$p(t_1, ..., t_k) \in S^{i}_{P_{2,1}}(\varnothing)$$

where $t_1, ..., t_k$ is a sequence of terms that substituted the sequence of variables X. Then we know that

$$\beta_1(T_1), q_1(T_2) \in S^{i-1}_{P_{2,1}}(\varnothing)$$

where T_1 and T_2 are sequences of terms substituted for the sequences of variables Y_1 and Z_1. Since each q_i can only be satisfied if q_{i+1} and each β_{i+1} is satisfied we know that

$$\beta_1(T_1), ..., \beta_n(T_n), q_1(T_{n+1}), ...q_n(T_{2n}) \in S^{i-1}_{P_{2,1}}(\varnothing).$$

Since each β_i is defined outside of the set of clauses R, and P_1 and P_2 have the exact same clauses outside of c and R we know that

$$\exists j \in \mathbb{N} : \beta_1(T_1), ..., \beta_n(T_n) \in S^{j}_{P_{1,1}}(\varnothing),$$

and therefore

$$p(t_1, ..., t_k) \in S^{j+1}_{P_{1,1}}(\varnothing)$$

Hence, if $p(t_1, ..., t_k) \in con_S(P_2)$ then $p(t_1, ..., t_k) \in con_S(P_1)$.

From case 1 and case 2 it follows that

$$\forall \alpha \in L_{P_1} : \alpha \in con_S(P_1) \text{ iff } \alpha \in con_S(P_2).$$

Induction Step
Let P_1 be a stratified program with S strata:

$$P_1 = P_{1,1}, P_{1,2}, ..., P_{1,S},$$

and P_2 a stratified program constructed from P_1 by Scheme A.2. We define:

$$(M_{P_i})_0 \coloneqq S^{\omega}_{P_{i,0}}(\varnothing)$$
$$(M_{P_i})_s \coloneqq S^{\omega}_{P_{i,s}}((M_{P_i})_{s-1}),$$

Induction Hypothesis: For some stratum s:

$$\forall \alpha \in L_{P_1} : \alpha \in (M_{P_1})_s \text{ iff } \alpha \in (M_{P_2})_s$$

Show that for stratum $s+1$

$$\forall \alpha \in L_{P_1} : \alpha \in (M_{P_1})_{s+1} \text{ iff } \alpha \in (M_{P_2})_{s+1}$$

Case 1: If $\alpha \in (M_{P_1})_{s+1}$ then $\alpha \in (M_{P_2})_{s+1}$

$$\forall \alpha \in (M_{P_1})_{s+1}$$

there exists an $i \leq \omega$ such that:

Is it $i \leq \omega$ or $i \in \omega$?

$$\alpha \in S^i_{P_1}((M_{P_1})_s).$$

As P_2 is constructed from P_1 using Scheme A.2 it will contain a larger number of clauses, and a ground atom α may require more applications of S_{P_2} to be added to the set of consequences.

For a ground atom $\alpha \in S^i_{P_1}((M_{P_1})_s)$ which is defined in $P_{1,s+1}$ there is a clause in $P_{1,s+1}$:

$$\alpha \leftarrow \rho_1, ..., \rho_p, \textit{ not } \eta_1, ..., \textit{ not } \eta_n$$

where each ρ_i is a positive literal and each η_i is a negative literal. If $\alpha \in S^i_{P_1}((M_{P_1})_s)$ then

$$\rho_1, ..., \rho_n \in S^{i-1}_{P_1}((M_{P_1})_s)$$
$$\eta_1, ..., \eta_n \notin S^{i-1}_{P_1}((M_{P_1})_s).$$

Since each η_i occurs as a negative literal in the clause they must be defined in an earlier stratum than $s+1$. Hence, all consequences of these predicates will have already been derived. Therefore:

$$\eta_1, ..., \eta_n \notin (M_{P_1})_s.$$

By the induction hypothesis we know that

$$\eta_1, ..., \eta_n \notin (M_{P_2})_s.$$

For the positive literals in the body of the clause: $\rho_1, ..., \rho_p$, if a ρ_i is defined in an earlier stratum, the same argument using the induction hypothesis tells us that

$$\rho_i \in (M_{P_2})_s.$$

For any ρ_j defined in stratum $s+1$ they only occur as positive literals in the clauses of stratum $s+1$. Another valid stratification of P_1 would be to place the clauses of each ρ_j, i.e. their definition, in

Is this argument okay? Or do I need

an earlier stratum. Hence,

$$\rho_j \in (M_{P_1})_s.$$

Then by the induction hypothesis

$$\rho_j \in (M_{P_2})_s.$$

Hence, we know that

$$\begin{aligned} \exists k \in \mathbb{N} : \rho_1, ..., \rho_n &\in S^k_{P_2}((M_{P_2})_s) \\ \eta_1, ..., \eta_n &\notin S^k_{P_2}((M_{P_2})_s). \end{aligned}$$

Clauses that are transformed using Scheme A.2 into a set of clauses are then trivially satisfied by further application of the immediate consequence operator, and we can conclude with:

$$\alpha \in (M_{P_2})_{s+1}$$

Case 2: If $\alpha \in (M_{P_2})_{s+1}$ then $\alpha \in (M_{P_1})_{s+1}$

$$\forall \alpha \in (M_{P_2})_{s+1}$$

there exists an $i \leq \omega$ such that:

$$\alpha \in S^i_{P_2}((M_{P_2})_s).$$

For a ground atom α with its definiton in stratum $s+1$, derived from a clause which originally in P_1 had $m > 2$ literals in the body, there is a set of clauses in stratum $s+1$ in P_2:

$$\begin{aligned} \alpha &\leftarrow \beta_1, q_1 \\ q_1 &\leftarrow \beta_2, q_2 \\ &... \\ q_{m-2} &\leftarrow \beta_{m-1}, \beta_m \end{aligned}$$

where each β_i is a positive literal: ρ_i or a negative literal: *not* η_i:

$$\beta_1, ...\beta_m = \rho_1, ..., \rho_p, \ \textit{not} \ \eta_1, ..., \ \textit{not} \ \eta_n,$$

and each q_i are auxiliary predicates constructed by Scheme A.2.

Since $\alpha \in S^i_{P_2}((M_{P_2})_s)$ we know that:

$$\begin{aligned} \rho_1, ..., \rho_p, q_1, ..., q_{m-2} &\in S^{i-1}_{P_2}((M_{P_2})_s) \\ \eta_1, ..., \eta_n &\notin S^{i-1}_{P_2}((M_{P_2})_s). \end{aligned}$$

Since $\eta_1, ..., \eta_n$ occurs negatively in the body of the clauses, the definition for each η_i must be in an earlier stratum. Hence:

$$\eta_1, ..., \eta_n \notin (M_{P_2})_s.$$

By the induction hypothesis we know that

$$\eta_1, ..., \eta_n \notin (M_{P_1})_s.$$

For any ρ_i that is defined in an earlier stratum than $s+1$ we have already derived the consequences of those clauses, and therefore any ground atom derived from any of the ρ_i will be in $(M_{P_2})_s$. By the induction hypothesis:

$$\rho_i \in (M_{P_1})_s.$$

For any ρ_j defined in stratum $s+1$ we know that any occurrence of ρ_j in the body of clauses in $s+1$ must be as a positive literal. Another valid stratification of P_2 would be to place ρ_j in an earlier stratum. Then the consequences of ρ_j would already be derived:

$$\rho_j \in (M_{P_2})_s.$$

Then, by the induction hypothesis:

$$\rho_j \in (M_{P_1})_s.$$

We now know that:

$$\begin{aligned}\exists k \in \mathbb{N} : \rho_1, ..., \rho_p, &\in S^k_{P_1}((M_{P_1})_s)\\ \eta_1, ..., \eta_n &\notin S^k_{P_1}((M_{P_1})_s).\end{aligned}$$

The literals in the body of the original clause in P_1:

$$\alpha \leftarrow \rho_1, ..., \rho_p, \ \textit{not}\ \eta_1, ..., \ \textit{not}\ \eta_n$$

are now satisfied and α can be derived:

$$\alpha \in S^{k+1}_{P_1}((M_{P_1})_s).$$

We have then shown that:

$$\alpha \in \text{If } (M_{P_2})_{s+1} \text{ then } \alpha \in (M_{P_1})_{s+1}.$$

From case 1 and case 2 we conclude with:

$$\forall \alpha \in L_{P_1} : \alpha \in (M_{P_1})_{s+1} \text{ iff } \alpha \in (M_{P_2})_{s+1}$$

□

References

[1] K. Apt, H. Blair, and Adrian Walker. Towards a theory of declarative knowledge. In Foundations of Deductive Databases and Logic Programming., 1988.

[2] F. Bergadano, D. Gunetti, M. Nicosia, and G. Ruffo. Learning logic programs with negation as failure. 1996.

[3] Keith L. Clark. Negation as Failure, pages 293–322. Springer US, Boston, MA, 1978.

[4] Andrew Cropper and S. Muggleton. Learning higher-order logic programs through abstraction and invention. In IJCAI, 2016.

[5] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data, 2018.

[6] John Feser, Swarat Chaudhuri, and Isil Dillig. Synthesizing data structure transformations from input-output examples. ACM SIGPLAN Notices, 50:229–239, 06 2015.

[7] Joyce Friedman. Alonzo church. application of recursive arithmetic to the problem of circuit synthesissummaries of talks presented at the summer institute for symbolic logic cornell university, 1957, 2nd edn., communications research division, institute for defense analyses, princeton, n. j., 1960, pp. 3–50. 3a-45a. Journal of Symbolic Logic, 28(4):289–290, 1963.

[8] Vladimir Lifschitz, David Pearce, and Agustín Valverde. Strongly equivalent logic programs. ACM Trans. Comput. Logic, 2(4):526–541, October 2001.

[9] W. Marek and V.S. Subrahmanian. The relationship between stable, supported, default and autoepistemic semantics for general logic programs. Theoretical Computer Science, 103(2):365 – 386, 1992.

[10] G. Metcalfe. Fundamentals of fuzzy logics.

[11] Himanshu Singh and Yunis Ahmad Lone. Introduction to Fuzzy Set Theory, pages 1–34. Apress, Berkeley, CA, 2020.

[12] M. H. Van Emden and R. A. Kowalski. The semantics of predicate logic as a programming language. J. ACM, 23(4):733–742, October 1976.

[13] William Yang Wang, Kathryn Mazaitis, and William W. Cohen. A soft version of predicate invention based on structured sparsity. In Proceedings of the 24th International Conference on Artificial Intelligence, IJCAI'15, page 3918–3924. AAAI Press, 2015.

[14] L.A. Zadeh. Fuzzy sets. Information and Control, 8(3):338 – 353, 1965.