# Stratified Negation as Failure in Differentiable Inductive Logic Programming

Sondre Bolland

January 29, 2021

**Abstract**

This paper seeks to increase the expressiveness of Differentiable Inductive Logic Programming ($\delta$ILP) by introducing stratified negation as failure.

## Contents

**1 Introduction** **2**

**2 Background** **2**
2.1 Logic Programming . . . . . 2
2.1.1 Basic Concepts . . . . . 3
2.1.2 Semantics of Definite Logic Programs . . . . . 4
2.2 Negation As Failure . . . . . 7
2.2.1 Recursion and Negation . . . . . 7
2.3 Stratified Programs . . . . . 8
2.3.1 Determining Stratification . . . . . 9
2.3.2 Semantics of Stratified Programs . . . . . 10
2.3.3 Terminating Stratified Programs . . . . . 13
2.4 Inference in Logic Programming . . . . . 13
2.5 Inductive Logic Programming . . . . . 14
2.5.1 ILP as a Satisfiability Problem . . . . . 15
2.5.2 Basic Concepts . . . . . 15
2.5.3 Reducing Induction to Satisfiability . . . . . 18
2.5.4 Limitations of ILP . . . . . 18
2.6 Differentiable Inductive Logic Programming . . . . . 18
2.6.1 Valuations . . . . . 19
2.6.2 Induction by Gradient Descent . . . . . 19
2.6.3 Rule Weights . . . . . 20
2.6.4 Inference . . . . . 21
2.6.5 Computing the $\mathcal{F}_c$ functions . . . . . 22
2.6.6 Extracting our Program . . . . . 25

**3 Stratified Negation as Failure** **27**
3.1 Stratified Negation as Failure in Inductive Logic Programming . 27
3.1.1 Adding Negation As failure . . . . . 28
3.1.2 Selecting Clauses for t-Stratified Programs . . . . . 29
3.1.3 Non-monotonic Forward Chaining . . . . . 30
3.1.4 No Loss of Generality . . . . . 30
3.1.5 Decidability of Stratified Programs in ILP and $\delta$ILP . . . 34
3.2 Stratified Negation as Failure in Differentiable Inductive Logic Programming . . . . . 34
3.2.1 Fuzzy Negation as Failure . . . . . 35
3.2.2 Stratification of $\delta$ILP . . . . . 36
3.2.3 Non-Monotonic Differentiable Inference . . . . . 37

**4 Experiments** **39**

**5 Discussion** **39**

**6 Conclusion** **39**

**References** **39**

# 1 Introduction

**This will come later**

# 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 clauses. 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 [4].

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. We will discuss stratified programs in more detail later.

### 2.1.1 Basic Concepts

Clauses in logic programming are typically written on 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** $\alpha$ 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**, only using ground atoms. The *ancestor* relation is defined **intensionaly,** using a set of clauses (a set of one 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) $\wedge$ ancestor(z,y) $\rightarrow$ 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 your 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 an element of the Herbrand interpretation is interpreted as false. A **model** in first order logic of a sentence $\alpha$ is an interpretation $M$ such that $\alpha$ is true in $M$. A **Herbrand Model** is a Herbrand interpretation which is a model [8].

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

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

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

$$P \vDash \gamma \text{ if } \gamma \in M$$
$$P \nvDash \gamma \text{ if } \gamma \notin M$$

**Definition 2.1** (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 $M \vDash L_1, ..., L_m$.*

**Definition 2.2** (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 2.3** (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 $con(P)$. To determine whether a ground atom $\gamma$ 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 later) 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 2.4.** *[8] 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 $\gamma$ 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 2.5** ($T_P$)**.** *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 2.6.** *[8] Let $P$ be a definite program. Then:*

- *$T_P$ is monotonic:*

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

- *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 2.7** (Powers of $T_P$)**.** *Let $P$ be a definite program. The powers of $T_P$ is defined as such*

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

**Definition 2.8** (Least Fixpoint of Definite Program)**.** *Let $P$ be a definite program. The least fixpoint of $P$, lfp(P) is defined as the $\omega$ power of $T_P$*

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

**Theorem 2.9.** *[8] Let $P$ be a definite program. Its unique minimal Herbrand Model $M_P$ is given by*

$$M_P = lfp(P)$$

**Definition 2.10.** *Let $P$ be a definite program and $M_P$ be a model of $P$ defined by theorem 2.9.*

$$con_D(P) = M_P$$

**Maybe add an example here**

## 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). 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 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

$\neg$ mathCourse($C$)

Let us consider a second program representing father relations

father(bob, paul)

father(bob, alice)

fatherless(X) $\leftarrow$ *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(X,Y) fails. If we query our program with fatherless(bob) we will search for a substitution $\theta$ such that father(Y, bob)$\theta$ 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 $P$, $P \models p(a)$. To prove $p(a)$ we need to satisfy our subgoals $q(X)$ and $not\ p(X)$. $q(X)$ is easy enough. We have a choice of both $q(a)$ and $q(b)$. We select $q(a)$, i.e. we have the substitution $\theta = \{a/X\}$. 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. The key feature of a stratified program is that it forbids recursion "inside negation". Recursion inside negation is shown in the following program, where $p$ is defined by a recursive call inside a negation:

$p \leftarrow not\ q$

$q \leftarrow p$

**Definition 2.11.** *A program $P$ is stratified when there is a partition into a set of strata*

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

*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 withing*

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

Note that a program is stratified if there is any such partition [1]. 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:

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

### 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 2.12** (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 2.13.** *[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)

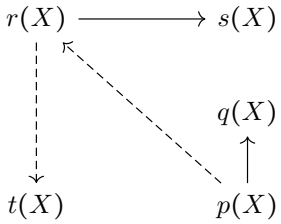
t(a) $\leftarrow$

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.




We have seen that the program $P$ can indeed be stratified, and observe again this fact seeing 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


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 $\gamma$ 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

$$I \subseteq T_P(I)$$

Meaning that 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 2.14** ($S_P$)**.** *Let $P$ be a stratified logic program and $I$ a set of ground atoms of $P$.*

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

$T_P$ and $S_P$ have the same definition. The two operators differ by their powers.

**Definition 2.15.** *Let $I$ be a set of ground atoms of a stratified program $P$. The powers of the operator $S_P$ are defined as:*

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

**Theorem 2.16.** *[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*

$$M_0 = S_{P_0}^\omega(\varnothing)$$
$$M_1 = S_{P_1}^\omega(M_0)$$
$$...$$
$$M_n = S_{P_n}^\omega(M_{n-1})$$

*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 2.17.** *Let $P$ be a stratified program and $M_P$ be a model of $P$ defined by theorem 2.16.*

$$con_S(P) = M_P$$

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

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

Using the immediate consequence operator we construct interpretations

$$M_0 = S^{\omega}_{P_0}(\varnothing) = \{t(a), s(a), s(b), q(a)\}$$
$$M_1 = S^{\omega}_{P_1}(M_0) = \{r(b), t(a), s(a), s(b), q(a)\}$$
$$M_2 = S^{\omega}_{P_2}(M_1) = \{p(a), r(b), t(a), s(a), s(b), q(a)\}$$

Finally, we have $M_P = M_2$ which is a minimal and supported model of $P$.

### 2.3.3 Terminating Stratified Programs

For the purposes of the Inductive Logic Programming system of this paper we define a new type of logic programs.

**Definition 2.18** (t-stratified program)**.** *A logic program $P$ is t-stratified, Terminating Stratified Program, if it is stratified and contains no cycles.*

In Stratified Negation as Failure in Differentiable Inductive Logic Programming we restrict all programs to be t-stratified. This is to avoid infinite loops in our inference.

Consider the following program $P$:

$p \leftarrow$ q

$q \leftarrow$ p

$P$ is stratified as we can partition both clauses in the same stratum, but when querying the program it will result in an infinite loop. Querying $P$ with $p$ we need to satisfy the subgoal of $q$. $q$ is satisifed if subgoal $p$ is satisfied. Hence, the infinite loop.

To check for cycles in our programs we simply construct the dependency graph and do a graph search for cycles.

## 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 we restrict our programs to not include functions. This means that our Herbrand universe is finite. As our programs are also finite we will only have finite models. This allows us to easily check whether a ground atom $\gamma$ is an element of our set of consequences, determining entailment.

**Theorem 2.19.** *Let $P$ be a stratified program. For all ground atoms $\gamma$*

$P \vDash \gamma$ *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 [4], ILP as a satisfyability problem. We will cover the broader ideas and the necessary details for our extension of the system. For a thorough look into the original system we recommend reading [4] (its a good read).

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 ground atoms: Background knowledge $\mathcal{B}$[1], 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 $P$ of clauses, i.e a logic program $P$, such that

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

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

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

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

The positive and negative examples of the even predicate are:

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

A possible solution is the set $P$:

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

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

[1] $\mathcal{B}$ is assumed to be a set of ground atoms, but not all ILP systems make this restriction. In some ILP systems, the background assumptions are clauses, not atoms

### 2.5.1 ILP as a Satisfiability Problem

In [4] the induction task of ILP is transformed into a satisfyability 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 together 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 $P_e \cup target \to \mathbb{N}$, 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 $\tau$ 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 intentional predicates $(int = 1)$ or only extensional predicates $(int = 0)$*

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

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

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

In this approach to ILP each predicate is defined by exactly two clauses. Hence, the pair of rule templates. This is done without loss of generality by the following theorem.

**The following two theorems are an addition on my part. The original paper states these claims in footnotes. The second theorem is shown in the footnote by an example, but not a proper proof. I am not sure how to deal with these theorems. They are not 100 % my work, but nor are they part of the original paper. I do not have to have them here. My original thought was to use these theorems to extend the "non-loss of generality" to stratified programs later, but those later theorems are fine by them selves.**

**Theorem 2.24.** *For any predicate p defined by* $n > 2$ *definite clauses there exists an equivalent set of clauses where each predicate is defined by exactly two clauses.*

*Proof.* Let $p$ be a predicate defined by $n > 2$ definite clauses.

$$\begin{aligned} p &\leftarrow \beta_1 \\ p &\leftarrow \beta_2 \\ &... \\ p &\leftarrow \beta_n \end{aligned}$$

Let $R$ be a set of definite clauses where for each second clause of $p$ we define a new auxiliary predicate $q_i$ expressing the clauses of $p$

$$\begin{aligned} p &\leftarrow \beta_1 \\ p &\leftarrow q_1 \\ q_1 &\leftarrow \beta_2 \\ q_1 &\leftarrow q_2 \\ &... \\ q_{n-1} &\leftarrow \beta_n \end{aligned}$$

□

In some cases, the clauses defining a predicate are the same. Then only one is written for clarity.

This system also restricts each clause to have exactly two atoms in the body. Again, this can be done without loss of generality, by the following theorem.

**Theorem 2.25.** *For any definite clause with $n > 2$ atoms in the body there exists an equivalent set of clauses with each having exactly two atoms in the body.*

*Proof.* Let $c$ be a definite clause defining the predicate $p$ with $n > 2$ atoms in the body

$$p \leftarrow \beta_1, \beta_2, ..., \beta_n$$

Let $R$ be a set of definite clauses with each having exactly two atoms in the body. The clause $c$ can be equivalently expressed by adding an auxiliary predicate $q_i$ for every second atom in the body of $c$

$$\begin{aligned} p &\leftarrow \beta_1, q_1 \\ q_1 &\leftarrow \beta_2, q_2 \\ &... \\ q_{n-2} &\leftarrow \beta_{n-1}, \beta_n \end{aligned}$$

□

The predicates are restricted to be nullary, unary and binary. No constants are allowed in the clauses, but a constant can be represented using a nullary predicate. Unsafe clauses (clauses with a variable not seen in the body), circular clauses (the head appears in the body), duplicated clauses (equivalent to another clause with the body atoms permuted) are disallowed.

**Definition 2.26** (Language)**.** *A language is a combination of the extensional predicates of the language frame $\mathcal{L}$ and the intensional predicates of the program template $\Pi$:*

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

- *$P_e$ 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 $G$:

$$G = \{\gamma_i\}_{i=1}^{n} =$$

$$\{p() \mid p \in P, arity(p) = 0\} \cup$$

$$\{p(k) \mid p \in P, arity(p) = 1, k \in C\} \cup$$

$$\{p(k_1, k_2) \mid p \in P, arity(p) = 2, k_1, k_2 \in C\} \cup$$

$$\{\bot\}$$

### 2.5.3 Reducing Induction to Satisfiability

Given a rule template Π, let $\tau_p^i$ 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 generate by $\tau_p^i$. Using a set $\Phi$ 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 $\Phi$. By extracting a subset of the clauses which $\Phi$ has set to true, the final program is constructed [4].

### 2.5.4 Limitations of ILP

ILP has several appealing features 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 [4]. Neural Networks are rarely data efficient, are not verifiable, arguably almost never human readable, and can only support transfer learning at certain times [**Need some source backing this up**].

The main disadvantages of traditional ILP systems is their inability to handle noisy, erroneous, or ambiguous data. These ILP disadvantages are key strengths of Neural Networks. Hence, one way to overcome the brittleness of traditional ILP systems is to reimplement them in a robust connectionist framework.

## 2.6 Differentiable Inductive Logic Programming

Differentiable Inductive Logic Programming ($\delta$ILP) is a reimplementation of ILP in an end-to-end differentiable architecture. $\delta$ILP 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. $\delta$ILP 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 $[0, 1]$ [2]. Instead of using Boolean flags to choose a discrete subset of clauses we now use continues weights to determine a probability distribution over clauses.

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

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 [4].

### 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 on what valuation a given ground atom receives 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 $\Lambda$ 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 $\Pi$ and a set of clause weights $W$, a differentiable model is constructed that computes the conditional probability of $\lambda$ for a ground atom $\alpha$:

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

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

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

The probability of label $\lambda$ is calculated given the atom $\alpha$ by inferring the consequences of applying rules to the background facts (using $T$ time steps of forward chaining). The probability of label $\lambda$ is given by:

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

Each 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 $\gamma$ 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. More plainly, it finds the valuation for the current atom from a vector of evaluations.

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

$f_{convert}$ 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 $\Pi$ 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 intentional 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 of $G$ of ground atoms derived from this clause (with constants $\{a, b\}$) is

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

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

| 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 |
| $\bot$ | 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 of $\mathcal{F}$ 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 of valuations is defined of the form $a_t$. A 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 $c$. Each function can be computed as follows. Let $X_c = \{x_k\}_{k=1}^n$ be a set of pairs of indices of ground atoms of clause $c$. Each $x_k$ contains all the pairs of atoms that justify atom $\gamma_k$ 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 $(\gamma_1, \gamma_2)$ 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 $\theta$ 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 $(\gamma_1, \gamma_2)$. 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 |
|---|---|---|---|---|---|---|---|---|---|
| $\gamma_k$ | $\bot$ | 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 | | | | | |
| $\gamma_k$ | 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 | $\gamma_k$ | $x_k$ |
|---|---|---|
| 0 | $\bot$ | {} |
| 1 | p(a,a) | {} |
| 2 | p(a,b) | {} |
| 3 | p(b,a) | {} |
| 4 | p(b,b) | {} |

| k | $\gamma_k$ | $x_k$ |
|---|---|---|
| 5 | q(a,a) | {} |
| 6 | q(a,b) | {} |
| 7 | q(b,a) | {} |
| 8 | q(b,b) | {} |

| k | $\gamma_k$ | $x_k$ |
|---|---|---|
| 9 | r(a,a) | {} |
| 10 | r(a,b) | {} |
| 11 | r(b,a) | {} |
| 12 | r(b,b) | {} |

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 $(p(a,b), q(b,a))$ satisfy the body of clause $c$, and the head of the clause $c$ (for this pair of atoms) is $r(a,a)$ which is $\gamma_9$.

$X_c$, 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 tbhe running example, this yields:

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

| k | $\gamma_k$ | $X[k]$ |
|---|---|---|
| 5 | q(a,a) | [(0,0)] |
| 6 | q(a,b) | [(0,0)] |
| 7 | q(b,a) | [(0,0)] |
| 8 | q(b,b) | [(0,0)] |

| k | $\gamma_k$ | $X[k]$ |
|---|---|---|
| 9 | r(a,a) | [(0,0)] |
| 10 | r(a,b) | [(0,0)] |
| 11 | r(b,a) | [(0,0)] |
| 12 | r(b,b) | [(0,0)] |

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 $\gamma_k$, 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 existensially quantified variable Z, $v = 1$, and $w = |\{a,b\}|^v = 2$.

| k | $\gamma_k$ | $a[k]$ | $X_1[k]$ | $X_2[k]$ | $Y_1[k]$ | $Y_2[k]$ | $Z[k]$ | $\mathcal{F}_c(a)[k]$ |
|---|---|---|---|---|---|---|---|---|
| 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 such: 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

**This section needs to be re-written. I straight up copied it because there is something I suspect I have misunderstood.**

Before training, rule weights are initialised randomly by sampling from a $\mathcal{N}(0,1)$ distribution. We train for 6000 steps, adjusting rule weights to minimise cross entropy loss as described above. During training, $\delta$ILP is given multiple $(\mathcal{B},\mathcal{P},\mathcal{N})$ triples. Each triple represents a different possible world. Each training step, $\delta$ILP first samples one of these $(\mathcal{B},\mathcal{P},\mathcal{N})$ triples, and then samples a mini-batch from $\mathcal{P} \cup \mathcal{N}$. Providing multiple alternative possible worlds helps the system to generalise correctly. It is less likely to focus on irrelevant aspects of one particular situation if it is forced to consider multiple situations.

Each 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 6000 steps, $\delta$ILP produces a set of rule weights for each rule template. To validate this program, we run the model with the learned weights on an entirely new set of background facts. 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 $\lambda$ and the predicted label $\hat{\lambda}$:

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

Once $\delta$ILP 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, $\delta$ILP finds solutions with zero entropy, as long as it does not get stuck in a local minimum. To extract a human-readable logic program, we just take all clauses whose probability is over some constant threshold (currently set, arbitrarily, to 0.1).

# 3 Stratified Negation as Failure

As of now $\delta$ILP is restricted to reasoning using definite clauses. Restricting our clauses to not have negation in the body reduces the expressiveness of our language. Normal logic programs are widely used just 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 *notmember* are needed. If negation is allowed, we just have to device a program for *member*, and then set:

notmember(X,Y) ← not member(X,Y)

If negation is not allowed, then the two subprograms must be treated as independent concepts, and a program for *notmember* must be developed too. 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 $\delta$ILP in such a way that all programs constructed are stratified and 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 $\delta$ILP.

## 3.1 Stratified Negation as Failure in Inductive Logic Programming

In [4] definite programs were constructed by selecting a set of clauses which satisfy our positive examples and does not satisfy our negative. 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 t-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. By theorem 2.16 we require programs to be stratified to be able to construct our set of consequences $con_S(P)$, which is needed for the forward chaining. We further impose the requirement that all programs are t-stratified to avoid infinite loops.

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** $\tau$ 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 intentional 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 $\tau$ 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 $(\tau_q^1, \tau_q^2)$ to generate which clauses can be in the definition of $q$. Let the first rule template $\tau_q^1$ be

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

specifying no existensially quantified variables and disallowing intensional predicates, but allowing negated atoms. The generated clauses from the rule template $\tau_q^1$ (after pruning) are:

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

   ...

32. $q(X,Y) \leftarrow not\ p(X,Y),\ not\ p(Y,Y)$

In short, each clause generated by the original rule template now has three more clauses constructed by adding *not* to the first atom, the second atom and both atoms. 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.

### 3.1.2 Selecting Clauses for t-Stratified Programs

In ILP as a satisfiability problem a subset $P$ of all generated clauses $\Phi$ 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 $\Phi$, 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.

**Algorithm 3.2.** *For the clauses in $P$ we construct the dependency graph (definition 2.12) and then, using Bellman-Ford, 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.*

To determine if our program $P$ is t-stratified we also check $P$ for cycles.

**Algorithm 3.3.** *For the clauses in $P$ we construct the dependency graph (definition 2.12). Using a Depth First Search (DFS) we construct a DFS-tree. The graph has a cycle if there is a back-edge present. To detect a back-edge we keep track of the vertices currently in the recursion stack used in the DFS. If a vertex is reached that is already in the stack, then there is a back-edge in the graph. A cycle is detected. The algorithm returns true if $P$ has a cycle and false if $P$ does not contain a cycle.*

**Here I have simply described how the algorithms works. Later I will write the pseudocode. Also, since a dependecy graph simply labels its edges positive/negative the Bellman-Ford might not be necesarry. Instead, a simple DFS can suffice. When a cycle is found using algorithm 3.3 we save which edges are in the cycle. All edges are labeled and then we just have to check if any har labeled negative. Will change this part when writing the algorithm.**

These two algorithms joint is the test for t-stratification.

For subsets of $\Phi$ we run the test of t-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 $\Phi$ has been searched.

**The selection process can be more sophisticated than this, but for now a brute force solution will suffice.**

### 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 2.16. 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 $\mathcal{N}$

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

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

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.

### 3.1.4 No Loss of Generality

In the original ILP system all predicates are defined with exactly two clauses and each clause has exactly two atoms in its body. This restriction is imposed to reduce the search space of all clauses. This was done without loss of generality [4].

In our extended system, including negation as failure, we impose the same restriction on the construction of our clauses. This is done without loss of generality as we can construct equivalent programs with these restrictions. We consider programs to be equivalent if for every query they answer the same.

**Definition 3.4.** *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 $L_{P_1}$, $P?\gamma$[3], 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 the restrictions of our system auxiliary predicates are introduced. These auxiliary predicates are not part of the language of the original program. These predicates will only exist as subgoals in the new definitions of the relations of our original program,

[3] $P$ is the logic program being queried. $\gamma$ is a ground atom. The query asks if $\gamma$ is entailed by $P$.

and will never be queried directly. Hence, we are content with equivalence with respect to our original language.

**Theorem 3.5.** *A t-stratified program $P_1$ and $P_2$ will, for every query, have the same query answer if they have the same Herbrand model.*

*Proof.* Generally, in logic programming, the answer to a query $\gamma$ will be $True$, $False$ or *loop*. For t-stratified programs have no cycles so no infinite loops are possible. Hence, we are restricted to $True$ and $False$. Let $P_1$ and $P_2$ be logic programs which have the same Herbrand model. The Herbrand model specifies which ground atoms are true in each program. All ground atoms that are not in the Herbrand model are false. Given the query of a ground atom $\gamma$ to the two programs: $P_1?\gamma$ and $P_2?\gamma$, the queried atom will either be in their model or it will not. If $\gamma$ is in the model then $P_1?\gamma$ will answer $True$, just as $P_2?\gamma$ will answer $True$. If $\gamma$ is not in the model then $P_1?\gamma$ will answer $False$, just as $P_2?\gamma$ will answer $False$. □

To construct a program $P_2$ which is equivalent to our original program $P_1$, where $P_2$ adheres to the restrictions of our systems, we will add a set of predicate symbols, expanding the language of $P_2$.

The restrictions are:

1. Each predicate is defined by exactly two clauses
2. Each clause has exactly two atoms in the body

The following two theorems show how to construct equivalent programs with these restrictions.

**Theorem 3.6.** *Let $P$ be a t-stratified program. For any program $P$ with a relation $r$ defined by $n > 2$ normal clauses there exists an equivalent program defining $r$ with exactly two clauses.*

*Proof.* Let $P_1$ be a t-stratified logic program with a relation $r$ defined by $n > 2$ normal clauses

$$r \leftarrow not\ \beta_1 \qquad \text{(clause 1)}$$
$$r \leftarrow not\ \beta_2 \qquad \text{(clause 2)}$$
$$\ldots$$
$$r \leftarrow not\ \beta_n \qquad \text{(clause n)}$$

We can construct a t-stratified program $P_2$ by introducing auxiliary predi-

cates $\gamma_i$ for every third clause defining $r$. Let $P_2$ be

$$\begin{aligned} r &\leftarrow not\ \beta_1 \\ r &\leftarrow\ \gamma_1 \\ \\ \gamma_1 &\leftarrow not\ \beta_2 \\ \gamma_1 &\leftarrow \gamma 2 \\ &... \\ \gamma_{n-2} &\leftarrow not\ \beta_{n-1} \\ \gamma_{n-2} &\leftarrow not\ \beta_n \end{aligned}$$

$P_1$ and $P_2$ are equivalent with respect to language $L_{P_1}$. By theorem 3.5 and definition 3.4 $P_1$ and $P_2$ are equivalent with respect to language $L_{P_1}$ if they have the same Herbrand model, excluding all ground atoms $\gamma$ that are not in the language of $P_1$, $\gamma \notin L_{P_1}$. This means ignoring all auxiliary predicates $\gamma_i$ added in the construction of $P_2$. We can construct $con_S(P_1)$ and $con_S(P_2)$ (Herbrand models for $P_1$ and $P_2$ respectively) and check if they contain the same elements from $L_{P_1}$.

$con_S(P_1)$:
The only stratification of $P_1$ is:

$$P_1 = \{(r \leftarrow not\ \beta_1),\ (r \leftarrow not\ \beta_2),\ ...,\ (r \leftarrow not\ \beta_n)\} \qquad (P_{1_0})$$

By theorem 2.16 we construct $M_{P_1}$

$$M_0 = S^{\omega}_{P1_0}(\varnothing) = \{r\}$$

$M_0 = M_{P_1}$. By definition 2.17

$$con_S(P_1) = \{r\}$$

$con_S(P_2)$:
A possible, and the simplest, stratification of $P_2$ is

$$\begin{aligned} P_2 = \{&(r \leftarrow not\ \beta_1),\ (r \leftarrow \gamma_1), \\ &(\gamma_1 \leftarrow not\ \beta_2),\ (\gamma_1 \leftarrow \gamma_2), \\ &..., \\ &(\gamma_{n-2} \leftarrow not\ \beta_{n-1}),\ (\gamma_{n-2} \leftarrow not\ \beta_n)\} \end{aligned} \qquad (P_{2_0})$$

By theorem 2.16 we construct $M_{P_2}$

$$M_0 = S^{\omega}_{P2_0}(\varnothing) = \{r, \gamma_1,\ ...,\ \gamma_{n-2}\}$$

$M_0 = M_{P_2}$. By definition 2.17

$$con_S(P_2) = \{r, \gamma_1, \ ..., \ \gamma_{n-2}\}$$

Lastly, we omit all $\gamma \notin L_{P_1}$ and get

$$con_S(P_1) = \{r\} = con_S(P_2)$$

By theorem 3.5 and definition 3.4 $P_1$ and $P_2$ are equivalent with respect to language $L_{P_1}$.

□

**Theorem 3.7.** *Let $P$ be a t-stratified program. For any normal clause in $P$ with more than two atoms in the body there exists an equivalent set of clauses with each having exactly two atoms in the body.*

*Proof.* Let $P_1$ be a stratified logic program with the normal clause $c$. $c$ has $n$ negative literals in its body

$$\alpha \leftarrow not\ \beta_1, not\ \beta_2, ..., not\ \beta_n$$

We can construct a t-stratified program $P_2$ by introducing an auxiliary predicate $\gamma_i$ for every second literal in the body of $c$. Let $P_2$ be

$$\begin{aligned} \alpha &\leftarrow not\ \beta_1, \gamma_1 \\ \gamma_1 &\leftarrow not\ \beta_2, \gamma_2 \\ &... \\ \gamma_{n-2} &\leftarrow not\ \beta_{n-1}, \ not\ \beta_n \end{aligned}$$

By theorem 3.5 and definition 3.4 $P_1$ and $P_2$ are equivalent with respect to language $L_{P_1}$ if they have the same Herbrand model, excluding all ground atoms $\gamma$ that are not in the language of $P_1$, $\gamma \notin L_{P_1}$. This means ignoring all auxiliary predicates $\gamma_i$ added in the construction of $P_2$. We can construct $con_S(P_1)$ and $con_S(P_2)$ (Herbrand models for $P_1$ and $P_2$ respectively) and check if they contain the same elements from $L_{P_1}$.

$con_S(P_1)$:
The only stratification of $P_1$ is:

$$P_1 = \{\alpha \leftarrow not\ \beta_1, not\ \beta_2, ..., not\ \beta_n\} \qquad (P_{1_0})$$

By theorem 2.16 we construct $M_{P_1}$

$$M_0 = S^{\omega}_{P1_0}(\varnothing) = \{\alpha\}$$

$M_0 = M_{P_1}$. By definition 2.17

$$con_S(P_1) = \{\alpha\}$$

$con_S(P_2)$:
A possible, and the simplest, stratification of $P_2$ is

$$P_2 = \{(\alpha \leftarrow not\ \beta_1, \gamma_1),\ \ (\gamma_1 \leftarrow not\ \beta_2, \gamma_2), ..., (\gamma_{n-2} \leftarrow not\ \beta_{n-1},\ not\ \beta_n)\}\ \ (P_{2_0})$$

By theorem 2.16 we construct $M_{P_2}$

$$M_0 = S^{\omega}_{P2_0}(\varnothing) = \{\alpha, \gamma_1,\ ...,\ \gamma_{n-2}\}$$

$M_0 = M_{P_2}$. By definition 2.17

$$con_S(P_2) = \{\alpha, \gamma_1,\ ...,\ \gamma_{n-2}\}$$

Lastly, we omit all $\gamma \notin L_{P_1}$ and get

$$con_S(P_1) = \{\alpha\} = con_S(P_2)$$

By theorem 3.5 and definition 3.4 $P_1$ and $P_2$ are equivalent with respect to language $L_{P_1}$. □

### 3.1.5 Decidability of Stratified Programs in ILP and $\delta$ILP

As with [4] we wish to preserve the decidability of our system. The programming language which $\delta$ILP (and the ILP system it derives from) constructs its programs is Datalog. Datalog arises from Horn Logic via two restrictions and one extension. Functions symbols are disallowed and all variables present in the head of a clause must be part of its body. With these restrictions the predicates are decidable. This holds for both definite and normal programs [].

**Need something here. This: https://www.cs.cmu.edu/ fp/courses/lp/lectures/26-datalog.pdf says so, but it is not a paper.**

## 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 $\delta$ILP 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 $T/S$[4]. 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 the ILP as a satisfiability problem when introducing SNAF was

[4] $T$ for definite programs and $S$ for stratified programs

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 $\delta$ILP. 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 can express this rational is [5]

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

This is know as *weak negation* [7].

A version of this, which allows us to decide at what point we consider something to be true, is *weak negation with threshold*. 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 $t \in [0, 1]$ where if the valuation of

[5] 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* $\gamma$ and $\gamma$.

$\gamma$ is above $t$ then it is considered to be true. Conversely, if it is below $t$ it is considered to be false. We introduce a threshold $t$ to weak negation[6]:

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

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* [7]

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

where the valuation of a negative literal is the complement of its positive form [7]. Now we allow our system to valuate *not* $p$ to any value in the range $[0, 1]$.

**Exactly which form of continuous negation is selected should perhaps be determined by experimentation. When I actually program this I can try all versions and see which yields the best result.**

### 3.2.2 Stratification of $\delta$ILP

In $\delta$ILP, instead of using Boolean flags to choose a discrete subset of clauses, we use continues 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

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

and the clauses of these partitions satisfy the conditions of stratified programs described in definition 2.11. Note that not all subsets of $\mathfrak{P}$ form a stratified program $P_i$. Each subset needs to be checked for stratification using algorithm 3.2.

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

[6] See footnote 5
[7] See footnote 5

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.

We know that a given program $P_i$ will never reach the threshold when our network has finished training on our dataset, i.e our positive and negative examples. Considering one of the features we are trying to add to the connectionist methods of neural networks is data efficiency, this dataset will be rather small. Hence, it will not be infeasible to train the network for each of the $n$ programs we construct.

**To generate $\mathfrak{P}$ I will have to run the test for stratification for all subsets of the generated clauses, i.e. the power set of all generated clauses. Performing all the subsequent steps on $\mathfrak{P}$ might be too much. I may have to look into a smarter way of generating $\mathfrak{P}$ such that its cardinality is a bit smaller. There should be certain clause combinations that should not be checked.*

*For instance, if I check a program $P$ which satisfy all positive examples $\mathcal{P}$, but also satisfy some of the negative examples $\mathcal{N}$, then there is no need to check any other programs which $P$ is a subset of. Adding more clauses will only increase the number of ground atoms which the program satisfies, not reduce it as we want.*

*I think the statement above is correct. I will have to prove some stuff here.**

### 3.2.3 Non-Monotonic Differentiable Inference

The differentiable inference process in $\delta$ILP is a reimplementation of the inference process in ILP. To adhere to the restrictions of stratified programs we will alter the inference operations in $\delta$ILP to emulate the construction of $con_S(P)$, rather than $con_D(P)$ as in the original $\delta$ILP 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 our previous valuation.
2. Calculating the weighted valuation $b_t$ based on our 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 $T$[8] times 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 $P$.

[8]Not to be confused with the immediate consequence operator $T_P$.

#### 3.2.3.1 Amalgamation of Consequences

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

$$\begin{aligned} T^0(I) &= I \\ T^{(n+1)}(I) &= T(T^n(I)) \end{aligned}$$

and is represented in $\delta$ILP 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 $S(S^n(I))$.

$$\begin{aligned} T^{(n+1)} &= T(T^n(I)) \\ S^{(n+1)} &= S(S^n(I)) \cup S^n(I) \end{aligned}$$

$S(S^n(I))$ 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, $S^n(I)$ will simply we the previous valuation $a_t$

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

$$S^n(I) \mapsto a_t$$

In Fuzzy set theory the union of a set $A$ and $B$ is defined as their max [9]. Hence

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

**This is what seemed to be correct. There are other representations of $S(S^n(I)) \cup S^n(I)$ I can use. I would think this needs some experimentation.**

#### 3.2.3.2 Joining Consequences of each Stratum

The joining of consequences for each stratum is done by

$$\begin{aligned} M_0 &= S^{\omega}_{P_0}(\varnothing) \\ M_1 &= S^{\omega}_{P_1}(M_0) \\ &\dots \\ M_n &= S^{\omega}_{P_n}(M_{n-1}) \end{aligned}$$

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 start with the first stratum $P_0$ and perform $T$ time steps of forward chaining inference, before we proceed to do the same for the remaining strata.

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 the 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

$$
\begin{aligned}
a_{P_1} &= a_T \quad && \text{using } a_0 \text{ (defined by } \mathcal{B}) \text{ as the inital valuation} \\
a_{P_2} &= a_T \quad && \text{using } a_{P_1} \text{ as the inital valuation} \\
&\ldots \\
a_{P_n} &= a_T \quad && \text{using } a_{P_{n-1}} \text{ as the inital valuation}
\end{aligned}
$$

# 4 Experiments

**This section will be written after the implementation of SNAF**

# 5 Discussion

# 6 Conclusion

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

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

[6] 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.

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

[8] 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.

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