Stable and Supported Semantics in Continuous Vector Spaces

Yaniv Aspis[1] , **Krysia Broda**[1] , **Alessandra Russo**[1] , **Jorge Lobo**[1,2,3]
[1]Imperial College London
[2]ICREA
[3]Universitat Pompeu Fabra
{yaniv.aspis17, k.broda, a.russo}@imperial.ac.uk, jorge.lobo@upf.edu

Abstract

We introduce a novel approach for the computation of stable and supported models of normal logic programs in continuous vector spaces by a gradient-based search method. Specifically, the application of the immediate consequence operator of a program reduct can be computed in a vector space. To do this, Herbrand interpretations of a propositional program are embedded as 0-1 vectors in $\mathbb{R}^N$ and program reducts are represented as matrices in $\mathbb{R}^{N \times N}$. Using these representations we prove that the underlying semantics of a normal logic program is captured through matrix multiplication and a differentiable operation. As supported and stable models of a normal logic program can now be seen as fixed points in a continuous space, non-monotonic deduction can be performed using an optimisation process such as Newton's method. We report the results of several experiments using synthetically generated programs that demonstrate the feasibility of the approach and highlight how different parameter values can affect the behaviour of the system.

1 Introduction

Stable and supported models are two widely used approaches to defining the semantics of a logic program in the presence of default negation (Marek and Subrahmanian 1992). Stable semantics is at the heart of Answer Set Programming, a declarative paradigm for knowledge representation, reasoning and learning geared towards computationally hard problems (Lifschitz 2008). Typically, one encodes a problem as a set of clauses representing background knowledge, constraints or choices. The clauses form a logic program whose models under stable semantics correspond to solutions of the original problem. To discover these stable models, an answer set solver would perform a search over a discrete space of program interpretations, typically involving trial-and-error, conflict analysis and backtracking. Supported models are a superset of stable models that are characterized through the Clark completion of a program (Clark 1978).

The recent popularity of deep learning approaches has sparked interest in performing symbolic reasoning in continuous vector spaces rather than discrete. Approaches include defining a learning task for a neural network for learning models (Minervini et al. 2019), programs (Evans and Grefenstette 2017) or the reasoning process itself (Selsam et al. 2018). These approaches are appealing as they allow for symbolic reasoning and learning to be carried out over fuzzy and noisy data in a natural manner. But they are limited to performing approximate inference in the context of classical semantics. Approximating the reasoning process risks to make the learned semantics strongly dependent on the training data rather than on the given symbolic program. Reasoning only in the context of classical semantics makes the approaches less suited to capture common-sense reasoning which often requires a non-monotonic semantics. To date, stable and supported reasoning have not been tackled in neural-symbolic literature.

Yet, working in vector spaces offers unique advantages. Recent advancements in GPU hardware allows for very efficient computation of linear algebraic operations, such as matrix and vector multiplication. Linear algebra offers a variety of algorithms such as matrix decomposition and numerical optimisation that logical inference may benefit from. In addition, it may facilitate neural-symbolic learning by allowing a systematic method for translating symbolic logic programs to vector spaces and back. One wonders, then, if it is possible to extend exact logical inference to continuous space while maintaining stable and supported semantics.

Work supporting such an approach has recently been taken by Sakama et al. (2017; 2018). The authors proposed a matrix representation of definite and normal programs that preserves their semantics with respect to matrix multiplication and application of a non-continuous operation. The resulting algorithm allows for the computation of stable models in a vector space, but the representation is still discrete. In this work, we lay the foundations for a novel method to compute supported and stable models in continuous vector spaces using differentiable operations. We extend the above approach to continuous vector space while preserving two-valued semantics, allowing gradient-based methods for computation of supported and stable models. Our main contributions are:

- Presenting a discrete matrix representation of propositional normal program reducts.
- Demonstrating how to extend this discrete representation to continuous space and proving under what conditions two-valued semantics is preserved.
- Presenting a novel algorithm for gradient-based computa-

tion of supported and stable models based on the above representation.

- Systematic evaluation of this algorithm to show the feasibility over a class of programs containing negative loops, positive loops and choice.

The novel method we present here can naturally be extended in the future to support neural-symbolic architectures of reasoning and learning in the presence of default negation, and this is our main motivation.

The structure of this paper is as follows: Section 2 covers the necessary background and definitions for logic programming under supported and stable semantics. Section 3 presents and justifies our method, including a matrix-based characterization of program reducts, its generalization into continuous vector space and a gradient-based search algorithm for the computation of a normal program's supported models. Section 4 details the results of experiments testing the ability of the algorithm to compute supported and stable models for synthetic programs containing both positive and negative loops. Section 5 covers related work and section 6 concludes the paper.

2 Background

We consider propositional programs over a finite alphabet $\Sigma = \{p_1, p_2, p_3, ...\}$. The elements of Σ are called propositional variables or atoms. A *normal rule*, or *clause*, r is of the form:

$$h^r \leftarrow b_1^r, b_2^r, ..., b_{n^r}^r, \text{not}\, c_1^r, \text{not}\, c_2^r, ..., \text{not}\, c_{m^r}^r \tag{1}$$

where h^r, b_i^r, c_j^r, for $1 \leq i \leq n^r$, $1 \leq j \leq m^r$, are atoms in Σ. h^r is referred to as the *head*, $b_1^r, b_2^r, ..., b_{n^r}^r$ (resp. $c_1^r, c_2^r, ..., c_{m^r}^r$) as (collectively) the *positive body*, B^r, (resp. the *negative body*, C^r) of the rule. We denote $|B^r| = n^r$ and $|C^r| = m^r$. A *definite clause* r is a normal clause where $m^r = 0$. A *fact* is a definite rule where $n^r = 0$. A *normal program* P is a finite set of normal rules. If all rules of a program are definite, it is referred to as a *definite program*.

The *Herbrand Base*, B_P, of a program P is the set of all propositional variables that appear in P. We denote $|B_P| = N$. A Herbrand *interpretation* I of a program P is a subset of B_P. A *model* M of a program P is an interpretation of P where for every clause $r \in P$, if $\{b_1^r, b_2^r, ..., b_{n^r}^r\} \subseteq M$ and $\{c_1^r, c_2^r, ..., c_{m^r}^r\} \cap M = \emptyset$ then $h^r \in M$. A *minimal model* M of P is a model of P such that no proper subset of M is a model of P. A definite program P has exactly one minimal model (van Emden and Kowalski 1976), called the *Least Herbrand Model* and is denoted $LHM(P)$. For example, the program in equation (2) has the minimal model $\{p, q\}$. Non-minimal models include $\{p, q, t\}$ and $\{p, q, r, s\}$.

$$\begin{aligned} &p \leftarrow \\ &q \leftarrow p \\ &r \leftarrow q, s \\ &t \leftarrow t \end{aligned} \tag{2}$$

Different (alternative) semantics have been proposed for normal logic programming and relationships between them have been investigated (Marek and Subrahmanian 1992). *Supported model* and *Stable model* semantics are among these.

A *supported model* M of a program P is a model of P where for every $p \in M$ there exists a clause $r \in P$ such that $p = h^r$, $\{b_1^r, ..., b_{n^r}^r\} \subseteq M$ and $\{c_1^r, ..., c_{m^r}^r\} \cap M = \emptyset$. This definition is equivalent to the characterization of supported models as classical models of the Clark Completion of a program (Clark 1978; Marek and Subrahmanian 1992). In the program above both $\{p, q\}$ and $\{p, q, t\}$ are supported models. So, the $LHM(P)$ of a definite program P is a minimal supported model of P. Consider now this other program:

$$\begin{aligned} &p \leftarrow q \\ &q \leftarrow p \end{aligned} \tag{3}$$

The empty set is a supported and minimal model, but $\{p, q\}$ is also a supported model. This is due to the "positive loop" arising from the set of clauses $\{p \leftarrow q, q \leftarrow p\}$, so that $\{p, q\}$ are deduced to be true only if $\{p, q\}$ is assumed. One can formalise the concept of positive loops using the concept of the *atom dependency graph* $G(P)$, defined as a pair of atoms and arcs:

$$G(P) = (B_P, \{(p, q) | \exists r \in P, h^r = q, p \in B^r \cup C^r\}) \tag{4}$$

For an edge (p, q), if $p \in B^r$ then the edge is referred to as *positive*, while if $p \in C^r$ then it is *negative*. A *positive loop* is a cycle in $G(P)$ made of only positive edges. We call a cycle with a negative edge a *negative loop*. If we consider now the following program:

$$\begin{aligned} &p \leftarrow \text{not}\, p \\ &p \leftarrow q \\ &q \leftarrow p \end{aligned} \tag{5}$$

the empty set is no longer a model, as it does not satisfy the first clause, but $\{p, q\}$ is still a supported model, and also happens to be minimal, still due to the positive loop, to which somebody might object. To capture more closely the notion of a non-monotonic, default semantics, the notion of *stable model semantics* is often used instead.

A stable model of a normal program P is defined using the notion of program reduct (Gelfond and Lifschitz 1988). Given a program P and an Herbrand interpretation $M \subseteq B_P$, the *program reduct* P^M is constructed from P by firstly removing any rule whose negative body contains an atom $c_i^r \in M$, and (2) removing the negative body from the remaining rules. P^M is a definite program and therefore has a Least Herbrand Model. If $LHM(P^M) = M$ then M is a stable model. A definite program has exactly one stable model, its Least Herbrand Model. So for the program in equation (2), {p, q} is a stable model, and for the program in equation (3) the empty set is stable model. In fact, stable models are both supported and minimal (Marek and Subrahmanian 1992). The opposite however, does not hold in general. For example, the program in equation (5) has no stable model, but $\{p, q\}$ is a minimal supported model. Programs may also have multiple stable models. For example, the program in equation (6) has two stable models: $\{p\}$, $\{q\}$.

$$\begin{aligned} &p \leftarrow \text{not}\, q \\ &q \leftarrow \text{not}\, p \end{aligned} \tag{6}$$

For programs without positive loops, the notion of supported and stable models coincide (Kaminski and Kaufmann 2012). The problem of deciding whether a propositional normal program has a stable model is NP-complete (Dantsin et al. 2001). This is also the case for supported models (Truszczynski 2011).

Given a program P, the *Immediate Consequence* operator is a transformation function T_P on Herbrand interpretations defined as follows:

$$\begin{aligned} &T_P : 2^{B_P} \to 2^{B_P} \\ &T_P(I) = \{h^r \mid r \in P, B^r \subseteq I, C^r \cap I = \emptyset\} \end{aligned} \tag{7}$$

Note that $T_{P^I}(I) = T_P(I)$ (Marek and Subrahmanian 1992). Supported models of a program P are fixed points of T_P, i.e. $T_P(M) = M$. In this work, we use this fact to search for supported models by translating the equality $T_{P^M}(M) = M$ into a dynamical system in a vector space. As all stable models are also supported models, a search for fixed points of T_P is a useful step towards finding stable models of a given program.

In this paper, similarly to Sakama et al. (2017), we assume our programs satisfy the following *Multiple Definitions (MD) condition*: for every atom $p \in B_P$, there exists at most one clause r such that $n^r > 1$ or $m^r > 1$. We refer to such as rule as *long*, otherwise it is a *short* rule. Every normal program P can be transformed in linear time into a program satisfying the MD condition in the following way. Suppose h is the head of at least two rules $h \leftarrow b_1^{r_1}, b_2^{r_1}, ..., b_{n^{r_1}}^{r_1}, \text{not}\, c_1^{r_1}, \text{not}\, c_2^{r_1}, ..., \text{not}\, c_{m^{r_1}}^{r_1}$ and $h \leftarrow b_1^{r_2}, b_2^{r_2}, ..., b_{n^{r_2}}^{r_2}, \text{not}\, c_1^{r_2}, \text{not}\, c_2^{r_2}, ..., \text{not}\, c_{m^{r_2}}^{r_2}$. Introduce a new atom h_2 and replace the second rule by two new rules $h_2 \leftarrow b_1^{r_2}, b_2^{r_2}, ..., b_{n^{r_2}}^{r_2}, \text{not}\, c_1^{r_2}, \text{not}\, c_2^{r_2}, ..., \text{not}\, c_{m^{r_2}}^{r_2}$ and $h \leftarrow h_2$. Repeat this process for all rules and atoms until the resulting program P' satisfies the MD condition. The MD program P' is semantically equivalent to P in the sense that for every model M of P there exists a model M' of P' such that $M = M' \cap B_P$. Conversely, for every model M' of P' it holds that $M = M' \cap B_P$ is a model of P. This equivalence also holds for supported and stable models.

In the rest of the paper, we will assume the Herbrand base of a program to also include two special symbols: $\bot$ and $\top$. $\bot$ indicates "False" and $\top$ indicates "True". Every interpretation of a program is assumed to contain $\top$ and not to contain $\bot$, although when writing an interpretation explicitly we omit $\top$. Therefore, the empty interpretation contains just $\top$. Also, to simplify some of the formulation in the paper, we will not allow the positive body or the negative body of any clause to be empty. Therefore, clauses with empty positive body will have $B^r = \{\top\}$ and clauses with empty negative body will have $C^r = \{\bot\}$. For technical reasons (see section 3), we also assume that every program contains the clause $\top \leftarrow \top$, which will be omitted when writing the program explicitly.

In subsequent sections we use the following notations: Vectors are represented as lower-case letters in bold ($\boldsymbol{v}$). $\boldsymbol{v}$ is a column vector and $\boldsymbol{v}^\intercal$ is its corresponding row vector. The entries of a vector are lower-case non-bold letters with the index appearing in the subscript. So the i-th entry of $\boldsymbol{v}$ is v_i. Many vectors in this paper also have a superscript to distinguish them from other vectors, such as $\boldsymbol{v}^p$. A superscript never identifies an index, it is simply a name for the vector. So $\boldsymbol{v}^p$ is a vector and v_i^p is its i-th entry. Matrices are represented as capital non-bold letters such as D. As is the case with vectors, many matrices have a superscript identifying them, and a subscript represents entry indices such as D_{ij}^P. $\mathbb{I}$ is the identity matrix. The set of real values is $\mathbb{R}$.

3 Method

Let P be a normal program satisfying the MD condition. We embed P and interpretations in a vector space. To do so, we first place an ordering over its Herbrand Base $B_P = \{p_1 = \bot, p_2 = \top, p_3, ..., p_N\}$. We embed each element p_i as a vector $\boldsymbol{v}^{p_i}$ in $\mathbb{R}^N$ using a one-hot encoding. In other words, if $\{\boldsymbol{e_i}\}_{i=1}^N$ is the standard basis of $\mathbb{R}^N$, then $\boldsymbol{v}^{p_i} = \boldsymbol{e_i}$. A set of atoms A is embedded as $\boldsymbol{v}^A$ by summing over the embedding of its elements:

$$\boldsymbol{v}^A = \sum_{p \in A} \boldsymbol{v}^p \tag{8}$$

$\boldsymbol{v}^A$ can be thought of as a binary vector where an element is set to 1 if the corresponding atom is in A, otherwise it is 0. Specifically, for interpretations, we always have $v_1^I = 0$ and $v_2^I = 1$.

Definite rules can be embedded as $N \times N$ matrices, using both the vector representation of the head and the body, as follows:

$$D^r = \frac{1}{n^r} \boldsymbol{v}^{h^r} (\boldsymbol{v}^{B^r})^\intercal \tag{9}$$

We can then embed the program by summing over the embeddings of each rule:

$$D^P = \sum_{r \in P} D^r \tag{10}$$

This definition coincides with the matrix embedding given by Sakama et al. (2017). They prove that an application of the Immediate Consequence operator for a definite program can be computed using this matrix in the following manner. Define:

$$H_1(x) = \begin{cases} 1 & x \geq 1 \\ 0 & x < 1 \end{cases} \tag{11}$$

Then $\boldsymbol{v}^{T_P(I)} = H_1(D^P \boldsymbol{v}^I)$, where H_1 is applied element-wise. To see the intuition behind this, consider the case where there is a rule r such that $I \subseteq B^r$. Then $(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I = n^r$. After dividing by n^r and applying H_1 the resulting vector has a value of 1 set for the entry of h^r as expected. Otherwise, if $I \not\subseteq B^r$ then $(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I < n^r$ and in this case H_1 will set the value to 0. Note that the MD condition is necessary to ensure the correctness of this approach as multiple long rules defining h^r may result in their sum $\sum \frac{1}{n^r} (\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I > 1$ and the value of h^r to be set to 1 incorrectly.

For a normal clause, we modify the matrix embedding given in (9) using the concept of program reduct. Given an interpretation I, the embedding of r with respect to I is:

$$D^{r,I} = \frac{1}{n^r} \boldsymbol{v}^{h^r} (\boldsymbol{v}^{B^r})^\intercal \frac{(\boldsymbol{1} - \boldsymbol{v}^I)^\intercal \boldsymbol{v}^{C^r}}{m^r} \tag{12}$$

where $\mathbf{1}$ is a vector where all entries are set to 1. One can understand the additional term in equation (12) as follows: $(\mathbf{1}-\boldsymbol{v}^{\boldsymbol{I}})^\intercal \boldsymbol{v}^{C^r}$ is a count of the number of elements in C^r that are not in I. It is equal to m^r if $C^r \cap I = \emptyset$, else it is less than m^r. This means that the extra term is equal to 1 only if r is in P^I. The matrix embedding of r in that case would be the same as if the rule were definite after having its negative body removed. If $C^r \cap I \neq \emptyset$, the extra term is less than one. Since we apply H_1 when performing one step deduction, the rule will have no effect on the result, which is as if it has been removed from the program (see example 1). Just as in the case of definite programs, the MD condition is required to ensure the correctness of the result.

As is the case for definite programs, we can define the matrix representation of P^I as:

$$D^{P,I} = \sum_{r \in P} D^{r,I} \tag{13}$$

Proposition 1. *Suppose P is a normal program satisfying the MD condition and let I, J, M be interpretations. Then $J = T_{P^M}(I)$ if and only if $\boldsymbol{v}^J = H_1(D^{P,M}\boldsymbol{v}^I)$.*

Proof. Denote $\boldsymbol{u} = H_1(D^{P,M}\boldsymbol{v}^I)$. For any $p \in B_P$ we have $(\boldsymbol{v}^p)^\intercal \boldsymbol{u} = H_1\left(\sum_{h^r=p} \frac{(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r}(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I}{n^r m^r}\right)$. Suppose $J = T_{P^M}(I)$. If $p \in J$, then $(\boldsymbol{v}^p)^\intercal \boldsymbol{v}^J = 1$ and there exists a rule $r \in P^M$ such that $B^r \subseteq I$ and $M \cap C^r = \emptyset$. Therefore $(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I = n^r$ and $(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r} = m^r$ which implies $H_1\left(\sum_{h^r=p} \frac{(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r}(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I}{n^r m^r}\right) = 1$ and therefore $(\boldsymbol{v}^p)^\intercal \boldsymbol{u} = 1$. If $p \notin J$ then $(\boldsymbol{v}^p)^\intercal \boldsymbol{v}^J = 0$. Suppose r is a rule with $h^r = p$. There are two cases: **(1)** r is the long rule with p as its head, then either $B^r \not\subseteq I$ and hence $(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I < n^r$ or $C^r \cap M \neq \emptyset$ and hence $(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r} < m^r$. Either way we have $\frac{(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r}(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I}{n^r m^r} < 1$. **(2)** r is a short rule with p as its head. Then either $B^r \not\subseteq I$ and hence $(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I = 0$ or $C^r \cap M \neq \emptyset$ and hence $(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r} = 0$. Either way we have $\frac{(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r}(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I}{n^r m^r} = 0$. Hence for both (1) and (2) we have $H_1\left(\sum_{h^r=p} \frac{(\mathbf{1}-\boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r}(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I}{n^r m^r}\right) = 0$ which means $(\boldsymbol{v}^p)^\intercal \boldsymbol{u} = 0$. All of these together imply $\boldsymbol{u} = \boldsymbol{v}^J$. Conversely, suppose $\boldsymbol{v}^J = H_1(D^{P,M}\boldsymbol{v}^I)$. We know that $\boldsymbol{v}^{T_{P^M}(I)} = H_1(D^{P,M}\boldsymbol{v}^I)$, which means $\boldsymbol{v}^J = \boldsymbol{v}^{T_{P^M}(I)}$. From the definition of the vector representation of a set of atoms, this implies $J = T_{P^M}(I)$.

□

Example 1. *Consider the following program:*

$$\begin{aligned} &p \leftarrow not\, q \\ &q \leftarrow not\, p \\ &r \leftarrow p, s, not\, q, not\, t \\ &t \leftarrow p, not\, s, not\, r \end{aligned} \tag{14}$$

And let $M = \{p, r, t\}$. Then the matrix representation of P^M is given by:

$$D^{P,M} = \begin{array}{c|ccccccc} & \bot & \top & p & q & r & s & t \\ \hline \bot & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ \top & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ p & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ q & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ r & 0 & 0 & \frac{1}{4} & 0 & 0 & \frac{1}{4} & 0 \\ s & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ t & 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 \end{array} \tag{15}$$

The entry $D^{P,M}_{22} = 1$ represents the clause $\top \leftarrow \top$ and $D^{P,M}_{32} = 1$ the fact $p \leftarrow$ which is in the program reduct P^M. The entries $D^{P,M}_{53} = D^{P,M}_{56} = \frac{1}{4}$ come from the clause $r \leftarrow p, s, not\, q, not\, t$. Since they do not sum to 1, they cannot be used to derive any atoms, so the clause has effectively been "disabled" due to $t \in M$. Similarly, the entry $D^{P,M}_{73} = \frac{1}{2}$ represents the clause $t \leftarrow p, not\, s, not\, r$. Again, the entries do not sum to 1, this time due to $r \in M$. If we take an interpretation such as $I = \{p, s\}$ we can compute $T_{P^M}(I)$ by:

$$H_1(D^{P,M}\boldsymbol{v}^I) = H_1(D^{P,M}\begin{bmatrix}0\\1\\1\\0\\0\\1\\0\end{bmatrix}) = H_1(\begin{bmatrix}0\\1\\1\\0\\\frac{1}{2}\\0\\\frac{1}{2}\end{bmatrix}) = \begin{bmatrix}0\\1\\1\\0\\0\\0\\0\end{bmatrix} \tag{16}$$

Which is equal to $\boldsymbol{v}^{\{p\}}$, and indeed $T_{P^M}(\{p,s\}) = \{p\}$. Note how the entry for $\top \leftarrow \top$ ensures the $\top$ remains true after this operation.

A corollary of proposition 1 shows that a supported model M satisfies $\boldsymbol{v}^M = H_1(D^{P,M}\boldsymbol{v}^M)$. However, it is difficult to solve this equation for $\boldsymbol{v}^M$ since H_1 is a non-continuous operation. In this paper we explore replacing H_1 with a differential approximation. H_1, being a Heaviside Step function, has several differentiable approximations (Bracewell 2000). In this work, we choose to focus on one, specifically Sigmoid, and leave the investigation of other approximations to future work. We define:

$$\sigma_{\gamma,\tau}(x) = \frac{1}{1+e^{\frac{\gamma-x}{\tau}}} \tag{17}$$

γ is a 'confidence threshold' which represents a boundary between values representing 'True' ($x > \gamma$) and 'False' ($x < \gamma$). τ is called 'temperature'. To retain exact inference when replacing H_1 with $\sigma_{\gamma,\tau}$, we must pick the values of γ and τ carefully. Suppose $p \in B_P$. We denote rules$(p) = \{r \in P | h^r = p\}$ and $R_p = |\text{rules}(p)|$. Define:

$$n_p = \max_{\text{rules}(p)} n^r \qquad m_p = \max_{\text{rules}(p)} m^r \tag{18}$$

We must pick $\gamma > \frac{n_p - 1}{n_p}$ so that partially satisfied positive bodies do not make p true. Similarly, $\gamma > \frac{m_p - 1}{m_p}$ to prevent partially satisfied negative bodies from doing the same. Now, for two interpretations M, I, consider:

$$x \triangleq (\boldsymbol{v}^p)^\intercal D^{P,M}\boldsymbol{v}^I \tag{19}$$

Substituting for $D^{P,M}$ from equation (13) we get:

$$x = \sum_{r \in P : h^r = p} \frac{(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}^I (\mathbf{1} - \boldsymbol{v}^M)^\intercal \boldsymbol{v}^{C^r}}{n^r m^r} \tag{20}$$

Suppose that the body of $r \in \text{rules}(p)$ is satisfied (so that p must be deduced). Then for each $b \in B^r$ we have $(\boldsymbol{v}^b)^\intercal \boldsymbol{v}^I > \gamma$ and for each $c \in C^r$ we have $(\boldsymbol{v}^c)^\intercal \boldsymbol{v}^M < \gamma$. Plugging these values in equation (20) we get a lower bound on x:

$$x > \gamma(1-\gamma) \tag{21}$$

Therefore, to ensure p is deduced, we would like the following condition to hold:

$$x > \gamma(1-\gamma) \Leftrightarrow \sigma_{\gamma,\tau}(x) > \gamma \tag{22}$$

Unfortunately, this cannot be satisfied in general. Even the case where $x > \gamma > \gamma(1-\gamma)$ only guarantees $\sigma_{\gamma,\tau}(x) > \frac{1}{2}$. To remedy this, we introduce two further parameters $\gamma^\perp, \gamma^\top$ such that $0 < \gamma^\perp < \gamma < \gamma^\top < 1$. $\gamma^\perp$ is an upper bound on false values that variables can take. $\gamma^\top$ is a lower bound on true values. Plugging these new parameters into equation (20) we get:

$$x > \gamma^\top(1-\gamma^\perp) \tag{23}$$

And our new condition is:

$$x > \gamma^\top(1-\gamma^\perp) \Leftrightarrow \sigma_{\gamma,\tau}(x) > \gamma^\top \tag{24}$$

This is achieved when:

$$\tau < \frac{\gamma - (1-\gamma^\perp)\gamma^\top}{\ln(\frac{1}{\gamma^\top} - 1)} \tag{25}$$

Note that in order for τ to be greater than 0 in equation (25) we also require $\gamma^\top > \frac{1}{2}$ and $\gamma < (1-\gamma^\perp)\gamma^\top$.

We now turn our attention to the case where none of the bodies of $\text{rules}(p)$ is satisfied, so that p must be assigned a value less than $\gamma^\perp$. Consider the contribution of some rule $r \in \text{rules}(p)$ to x. In the worst case, all but one of the literals in its body are true. If this literal is positive, then the contribution of r to x is less than $\frac{n_p^1 - 1 + \gamma^\perp}{n_p^1}$. If it is negative, this contribution is upper bounded by $\frac{m_p^1 - \gamma^\top}{m_p^1}$. We can therefore put an upper bound on the value of x as follows:

$$x < \sum_{r \in \text{rules}(p)} \max\{\frac{n_p^r - 1 + \gamma^\perp}{n_p^r}, \frac{m_p^r - \gamma^\top}{m_p^r}\} \tag{26}$$

Note that since at most one of the rules of $\text{rules}(p)$ is long, we have $n_p^r = 1$ and $m_p^1 = 1$ for all but (perhaps) one of the summands. Denoting the right hand side of the above inequality as $x^\perp$, we now require:

$$x < x^\perp \Leftrightarrow \sigma_{\gamma,\tau}(x) < \gamma^\perp \tag{27}$$

This is satisfied when:

$$\tau < \frac{\gamma - x^\perp}{\ln(\frac{1}{\gamma^\perp} - 1)} \tag{28}$$

To ensure $\tau > 0$ in equation (28), we require both $\gamma > x^\perp$ and $\gamma^\perp < \frac{1}{2}$. When all the above conditions are met, exact inference in continuous vector space is achieved through matrix multiplication and application of the Sigmoid operation. We formalise this idea using the following notion of semantic equivalence between vectors.

Definition 1. *Suppose $\boldsymbol{v}, \boldsymbol{u} \in [0,1]^N$ and let $0 < \gamma^\perp < \gamma^\top < 1$. We say $\boldsymbol{v}$ and $\boldsymbol{u}$ are semantically equivalent if for all $1 \le i \le N$:*

1. *$v_i > \gamma^\top$ if and only if $u_i > \gamma^\top$.*
2. *$v_i < \gamma^\perp$ if and only if $u_i < \gamma^\perp$.*

In such a case we write $\boldsymbol{v} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{u}$.

A vector $\boldsymbol{v}$ in $[0,1]^N$ can therefore be semantically equivalent to $\boldsymbol{v}^I$ for some interpretation I, essentially representing the same interpretation. Note that if for some entry of $\boldsymbol{v}$ we have $\gamma^\perp < v_i < \gamma^\top$ then it does not represent any interpretation.

Proposition 2. *Let P be a normal program and I, M be interpretations. Let $0 < \gamma^\perp < \gamma < \gamma^\top < 1$ and $\tau > 0$ such that:*

- $\gamma > \max\left\{\frac{n_p - 1}{n_p}, \frac{m_p - 1}{m_p}\right\}$
- $\gamma^\top > \frac{1}{2}$
- $\gamma^\perp < \frac{1}{2}$
- $\gamma < (1-\gamma^\perp)\gamma^\top$
- $\gamma > x^\perp$
- $\tau < \min\left\{\frac{\gamma-(1-\gamma^\perp)\gamma^\top}{\ln(\frac{1}{\gamma^\top}-1)}, \frac{\gamma - x^\perp}{\ln(\frac{1}{\gamma^\perp}-1)}\right\}$

(n_p, m_p, $x^\perp$ as defined above). Suppose that for a vector $\boldsymbol{u} \in [0,1]^N$ we have $\boldsymbol{u} \sim_{\gamma^\perp}^{\gamma^\top} \boldsymbol{v}^I$. Then $H_1\left(D^{P,M}\boldsymbol{v}^I\right) \sim_{\gamma^\perp}^{\gamma^\top} \sigma_{\gamma,\tau}\left(D^{P,M}\boldsymbol{u}\right)$.

Proof. (Sketch) The analysis above demonstrated the condition under which exact inference is maintained. If these are satisfied, the result follows immediately from proposition 1. □

We are now ready to propose a gradient-based method for computing supported models of a normal program. First, we extend the definition of a matrix representation of a program reduct to a general vector $\boldsymbol{v}$ in $\mathbb{R}^\mathbb{N}$ as follows:

$$D^{P,\boldsymbol{v}} = \sum_{r \in P} \frac{1}{n^r} \boldsymbol{v}^{h^r} (\boldsymbol{v}^{B^r})^\intercal \frac{(\mathbf{1}-\boldsymbol{v})^\intercal \boldsymbol{v}^{C^r}}{m^r} \tag{29}$$

Next, given a normal program P satisfying the MD condition, we define the following mapping:

$$F(\boldsymbol{v}) = \sigma_{\gamma,\tau}(D^{P,\boldsymbol{v}}\boldsymbol{v}) - \boldsymbol{v} \tag{30}$$

From Brouwer's fixed-point theorem, one can show that for a supported model M there exists a root of F that is semantically equivalent to $\boldsymbol{v}^M$. Therefore, we can compute supported models of P by searching for roots of F. To do so, we employ Newton's method. This is formalised in Algorithm 1.

Algorithm 1 Newton Deduction

```
Require: $\epsilon > 0$, $MaxK \geq 1$, $0 < \gamma < 1$, $\tau > 0$
  Pick an initial vector:
      $\boldsymbol{v}^0 = [0 \quad 1 \quad p_3 \quad \dots \quad p_N]^\intercal$
  repeat
    Solve $J_{\boldsymbol{v}^k}(\boldsymbol{v}^{k+1} - \boldsymbol{v}^k) = \boldsymbol{v}^k - \sigma_{\gamma,\tau}(D^{P^{\boldsymbol{v}^k}} \cdot \boldsymbol{v}^k)$
  until $||\boldsymbol{v}^{k+1} - \boldsymbol{v}^k||_2 < \epsilon$ or $k \geq MaxK$
  if $k < MaxK$ then
    return $\boldsymbol{v}^k$
  else
    return 'FAIL'
  end if
```

Newton's method for finding roots of a mapping such as $F(\boldsymbol{v})$ proceeds by first picking an initial vector $\boldsymbol{v}^0$. At each iteration k the Jacobian of F at point $\boldsymbol{v}^k$, denoted $J_{\boldsymbol{v}^k}$, is computed and used to find the next vector $\boldsymbol{v}^{k+1}$. The update rule is given by:

$$J_{\boldsymbol{v}^k}(\boldsymbol{v}^{k+1} - \boldsymbol{v}^k) = -F(\boldsymbol{v}^{\boldsymbol{k}}) \tag{31}$$

In our case, the Jacobian at a general point $\boldsymbol{v}$ is given by:

$$J_{\boldsymbol{v}} = \frac{1}{\tau}\text{diag}\big(\sigma_{\gamma,\tau}(D^{P,\boldsymbol{v}}\boldsymbol{v}) \times (\mathbf{1} - \sigma_{\gamma,\tau}(D^{P,\boldsymbol{v}}\boldsymbol{v}))\big) \cdot \left(D^{P,\boldsymbol{v}} - \sum_{r \in P} \frac{\boldsymbol{v}^{h^r}(\boldsymbol{v}^{B^r})^\intercal \boldsymbol{v}(\boldsymbol{v}^{C^r})^\intercal}{n^r m^r}\right) - \mathbb{I} \tag{32}$$

where $\times$ indicates element-wise multiplication, $\mathbf{1}$ is a vector where all entries are set to 1, and $\text{diag}(\boldsymbol{v})$ is a matrix with the entries of $\boldsymbol{v}$ along its main diagonal.

For different initial vectors $\boldsymbol{v}^0$, the algorithm will return different results. We always set $v_1 = 0$ and $v_2 = 1$ representing that $\perp$ is false and $\top$ is true, respectively. For the initial values of $v_3, ..., v_N$, we consider here two methods:

- **Uniform sampling** - The values are picked uniformly from $[0, 1]$.
- **Semantic sampling** - The values are picked uniformly from $[0, \gamma^\perp] \cup [\gamma^\top, 1]$. In other words, the initial vector is always semantically equivalent to an interpretation.

Newton's method does not guarantee convergence, which is why we limit the number of iterations to $MaxK$ before declaring non-convergence. When convergence occurs, the discovered root $\boldsymbol{v}^k$ is returned. However, this root is not necessarily semantically equivalent to an interpretation. If it is semantically equivalent to $\boldsymbol{v}^M$ for some interpretation M, it is guaranteed from proposition 2 that M is a supported model. When a root semantically equivalent to an interpretation is returned, we consider it a successful case for the algorithm. In addition, we can easily check if the discovered supported model is stable by computing the Least Herbrand Model of the program reduct.

4 Experiments

To test the proposed method, an initial implementation has been made. At present, efficiency is not a goal of the implementation, and therefore we do not test its time performance. Instead, we focus on the rate of success for computing supported and stable models under various conditions. We define the rate of success for a given program to be the number of times we applied Algorithm 1 for the program and converged to a root semantically equivalent to a supported model, divided by the total number of attempts. We begin by considering a simple negative loop:

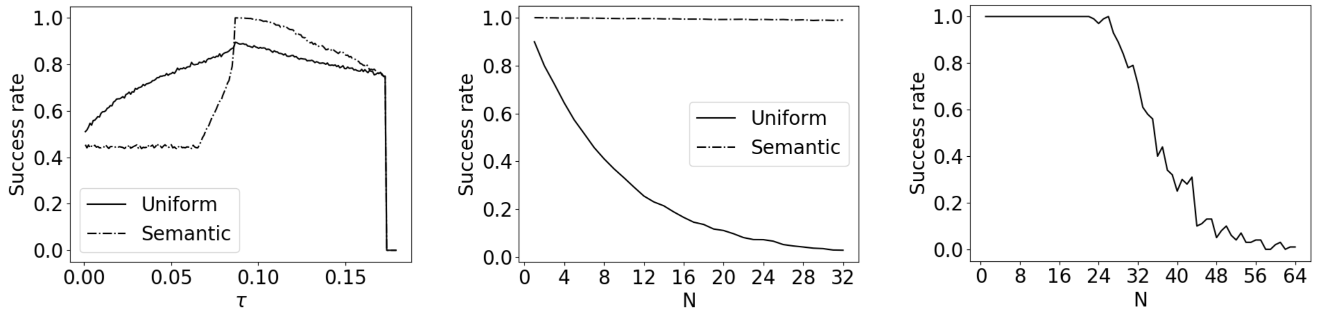
$$\begin{aligned} p &\leftarrow \text{not}\, q \\ q &\leftarrow \text{not}\, p \end{aligned} \tag{33}$$

The program has four Herbrand interpretations: $\emptyset$, $\{p\}$, $\{q\}$, $\{p, q\}$. Two of these are both supported and stable models, namely $\{p\}$ and $\{q\}$. Simply guessing an interpretation and checking if it is stable has a 50% success rate. To test the rate of success of finding a supported or stable model in this case, we repeatedly applied Algorithm 1. We tested for 200 different values of τ from just above 0 to 0.179 (which is slightly above the maximum allowed temperature value) in equal intervals. For each τ, we applied the algorithm 10,000 times, with randomly selected values of $\boldsymbol{v}^0$, and calculated the portion for which it converged to either $\{p\}$ or $\{q\}$. We repeated this experiment separately for both uniform and semantic sampling.

From Figure 1a, one can see that the rate of success approaches 50% at the limit of τ reaching zero, for both methods of sampling. This behaviour can be understood in the following way: When τ approaches zero, the Sigmoid begins behaving like a Heaviside function such as H_1. The computation is therefore changing from a continuous nature to discrete. In other words, in the limit where τ goes to zero, the Sigmoid implements T_P, and the algorithm behaves like a purely symbolic guess and check. Hence, the success rate approaches 50%.

As τ increases, the gradient information helps to direct the search and the rate of success increases, peaking at 89% for uniform sampling and near 100% for semantic sampling around $\tau = 0.087$. It is interesting that the increase is delayed for the semantic sampling case, not showing any improvement for $\tau < 0.07$. This is due to the approximation of T_P being more pronounced for vectors semantically equivalent to interpretations, and providing weaker gradient information at low temperatures.

For $\tau = 0.087$, it is interesting to ask what went wrong in the 11% of the cases where a stable model was not found, and why semantic sampling does not seem to have this problem. As it turns out, for cases where uniform sampling failed, the algorithm always converged to the same root, $[0, 1, 0.5, 0.5]$. This root is not semantically equivalent to any interpretation, hence it is still considered a failure case for the algorithm. We do not consider it in this work, but it is possible this root still holds some semantic meaning of the program. For instance, it may have a link to the Well-Founded Model of the program (Van Gelder, Ross, and Schlipf 1991). For now, however, we simply consider it an undesired output of the algorithm. The set of initial vectors that result in converging to $[0, 1, 0.5, 0.5]$ is mostly clustered around the root. Semantic sampling therefore avoids most of these points, improving the chance of success.



(a) One negative loop. (b) N negative loops. (c) N negative loops and within 1 second.

Figure 1: Success rate of converging to a supported model with negative loops. (a) A single negative loop for varying temperatures. (b) N negative loops for $\tau = 0.087$. (c) N negative loops for $\tau = 0.087$ and when given as many attempts as possible within 1 second.

The maximum value of τ for which proposition 2 holds is around 0.173. At that point we see a sudden drop in the success rate to 0. In other words, the conditions imposed on τ not only guarantee the correctness of proposition 2 but are also important to converge to a meaningful fixed point. The main takeaway from this experiment is that both the temperature and the method of sampling have a significant impact on the performance of Algorithm 1.

Next, we consider how the algorithm fares as negative loops are added to the program. Consider a program with N pairs of clauses of the form:

$$\begin{aligned} p_i &\leftarrow \text{not}\, q_i \\ q_i &\leftarrow \text{not}\, p_i \end{aligned} \tag{34}$$

For each pair, we have an 89% chance of success with uniform sampling and an optimal choice of τ. Since each pair is independent of the rest, one can think of their embedded semantics as existing in independent two-dimensional subspaces. Consequently, the chance of finding a stable model is equivalent to tossing N coins, each with 89% chance of landing on heads, and observing N heads. We confirm this intuition experimentally, as shown in Figure 1b. In contrast, for semantic sampling, the high probability of success implies a very slow drop as N increases. This is also confirmed in Figure 1b.

So far, we have only considered the chance of success when applying Algorithm 1 once. But we can also repeatedly apply the algorithm on the same program, for different initial vectors, until a supported/stable model is found. Consider, for instance, a scenario in which we are given 1 second to find a supported/stable model, and within that second we are allowed to apply Algorithm 1 as often as we can. We perform this experiment for uniform sampling on a computer with the following specifications:

- **CPU:** Intel(R) Core(TM) i7-8700 CPU @ 3.20GHz.
- **Memory:** 15.4 GB RAM.
- **Implementation Language:** Python 3.6.9.

Note that the implementation is single-threaded. As can be seen from Figure 1c, even with an inefficient implementation we can achieve a perfect success rate for up to 32 pairs. Past 32, we see a fast drop in performance. This can be understood when considering the chance of a success as N increases. For N pairs and K applications of the algorithm, the chance of success is given by $1 - (1 - 0.89^N)^K$. For a large enough K, one can retrieve the graph in Figure 1c. Therefore, as N increases one must also increase K by allowing further time.

We now turn our attention to the case where programs contain both positive and negative loops. As before, we first add N negative loop pairs of the form in equation (34) to the program. We then add M pairs of definite clauses using the following process: We select two indices $1 \le i, j \le N$ such that $i \ne j$ and add to the program a pair of clauses chosen randomly from one of the following forms:

$$\begin{array}{cccc} p_i \leftarrow p_j & p_i \leftarrow q_j & q_i \leftarrow p_j & q_i \leftarrow q_j \\ q_j \leftarrow q_i & p_j \leftarrow q_i & q_j \leftarrow p_i & p_j \leftarrow p_i \end{array} \tag{35}$$

We repeat this selection process M times. Note that each time we may select a different form for the pairs. As M becomes larger, positive loops will begin to appear more often in the program. This process will therefore generate programs containing many positive loops without enforcing any kind of structure over these loops.

We generated 100 different programs for several values of N and M. When $M < N$, many atoms only appear in negative loops. If the atoms in a negative loop do not appear anywhere else in the program, we remove the loop. For each program we applied Algorithm 1 100 times. We report the average success rate for each case in Table 1. Results for semantic sampling were very similar and are not reported here.

From the table we see the clear effect of positive loops on the success rate. Positive loops help the system discover supported models. However, they have the opposite effect on stable models. As the number of positive loops increases, the number of supported models that are not stable also increases, hence stable models become more difficult to find.

Finally, we consider the case where a choice between multiple options is required. Suppose we have N options, represented by the N variables $p_1, p_2, ..., p_N$, and we are required to select only one. This can be modelled by the

N	M	Success rate (supported)	Success rate (stable)
10	5	70%	69%
	10	84%	66%
	20	99%	7%
40	20	46%	44%
	40	68%	39%
	60	88%	13%
Overall		76%	40%

Table 1: Success rate with uniform sampling for programs with positive and negative loops.

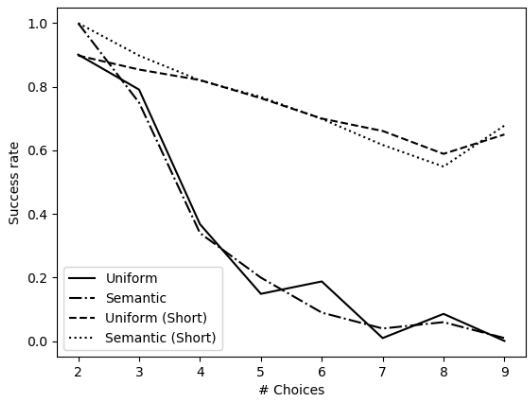


Figure 2: Success rate for N choices under various settings. Short stands for the representation of choice as a set of short clauses.

following set of N clauses:

$$\begin{aligned} p_1 &\leftarrow \text{not}\, p_2, \text{not}\, p_3, ..., \text{not}\, p_N \\ p_2 &\leftarrow \text{not}\, p_1, \text{not}\, p_3, ..., \text{not}\, p_N \\ &\vdots \\ p_N &\leftarrow \text{not}\, p_1, \text{not}\, p_2, ..., \text{not}\, p_{N-1} \end{aligned} \tag{36}$$

Then $\{p_1\}, \{p_2\}, ..., \{p_N\}$ are the supported (and stable) models of the program. When $N = 2$ we retrieve the case of a single pair.

Results for the case $N = 2$ to 9 appear in Figure 2 without the label "short". As can be seen, the success rate falls quickly as the number of choices increases. This is a result from the conditions for τ imposed by equations (25) and (28). As the length of rules increases, τ must be made smaller to guarantee exact inference holds. For a very small τ, the gradient information is insufficient, hurting the chance of success. Clearly, we must prefer representing our programs using shorter rules.

To accommodate this, we transform the problem of N choices into a sequence of choices between two atoms. We divide the variables $p_1, p_2, ..., p_N$ into pairs $(p_1, p_2), (p_3, p_4), ...$ and for each i pair we create a new atom c_i representing a choice between the two. We can

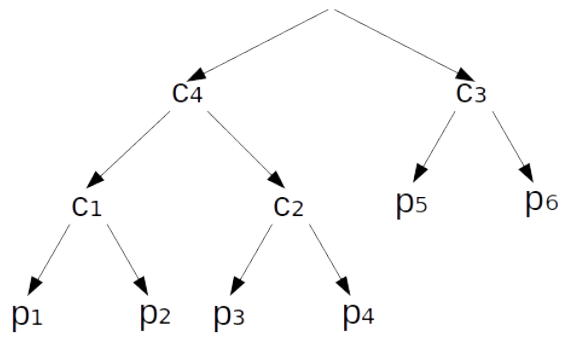

$$\begin{tikzcd}
 & & & \cdot \arrow[dl] \arrow[dr] & & & \\
 & c_4 \arrow[dl] \arrow[dr] & & & & c_3 \arrow[dl] \arrow[dr] & \\
c_1 \arrow[d] \arrow[dr] & & c_2 \arrow[d] \arrow[dr] & & p_5 & & p_6 \\
p_1 & p_2 & p_3 & p_4 & & &
\end{tikzcd}$$

Figure 3: The binary tree corresponding to the encoding of $N = 6$ choices using short clauses only.

encode these choices using each pair and its corresponding new atom as follows:

$$\begin{aligned} p_1 &\leftarrow c_1, \text{not}\, p_2 \\ p_2 &\leftarrow c_1, \text{not}\, p_1 \\ p_3 &\leftarrow c_2, \text{not}\, p_4 \\ p_4 &\leftarrow c_2, \text{not}\, p_3 \\ &\vdots \end{aligned} \tag{37}$$

Note that the number of new atoms is $\lfloor \frac{N}{2} \rfloor$. We then take the new atoms $c_1, c_2, ...$ and possibly p_N if N is odd, and encode a choice among them using the same process. By repeating this process, we are eventually left with only two atoms, for which we can encode a choice using a regular negative loop as in equation (33).

This encoding is best visualised using a binary tree structure. For example, for $N = 6$, the encoding above results in the binary tree depicted in figure 3, which corresponds to the program:

$$\begin{aligned} p_1 &\leftarrow c_1, \text{not}\, p_2 & p_2 &\leftarrow c_1, \text{not}\, p_1 \\ p_3 &\leftarrow c_2, \text{not}\, p_4 & p_4 &\leftarrow c_2, \text{not}\, p_3 \\ p_5 &\leftarrow c_3, \text{not}\, p_6 & p_6 &\leftarrow c_3, \text{not}\, p_5 \\ c_1 &\leftarrow c_4, \text{not}\, c_2 & c_2 &\leftarrow c_4, \text{not}\, c_1 \\ c_3 &\leftarrow \text{not}\, c_4 & c_4 &\leftarrow \text{not}\, c_3 \end{aligned} \tag{38}$$

In this way, we successfully encoded a choice among N variables using only short rules, and by adding only $O(N)$ new variables and clauses. It is easy to verify that the stable models of the new representation correspond to stable models of the original, once the auxiliary atoms are removed. The drawback to this representation is that it does not guarantee an equal probability of finding each of the stable models. Figure 2 shows the results with this representation, under the label "Short". As can be seen, with a representation that uses short rules, the success rate scales far better. With both representations, there does not seem to be a significant difference between uniform and semantic sampling.

5 Related Work

Answer set solvers, such as smodels (Simons 2000) and clasp (Kaminski and Kaufmann 2012) search for stable models of a normal program using SAT-solving techniques. For instance, clasp applies a variant of Conflict-Driven Clause Learning (Marques-Silva, Lynce, and Malik 2009), where conflicts during search are analyzed for their underlying causes to facilitate more efficient backtracking.

Research into differentiable inference in logic programs is usually done within the context of neural networks. Examples include relational learning in knowledge graphs (Cohen 2016; Kazemi and Poole 2017; Minervini et al. 2019), inducing logical rules from examples (Evans and Grefenstette 2017; Payani and Fekri 2019) or combining statistical machine learning with logic-based deduction (Manhaeve et al. 2018; Dai et al. 2019). Real-valued semantics for differentiable learning have been suggested (Serafini and d'Avila Garcez 2016). Unlike such approaches, we maintain two-valued semantics and only use continuous representations as a means towards converging to a desired two-valued interpretation. We do not require learning, either.

The matrix representation of logic programs was first introduced by Sakama et al. (Sakama, Inoue, and Sato 2017; Sakama et al. 2018; Nguyen et al. 2018). In this work we build upon their method for computing the semantics of a definite program by extending it in a novel way to the larger class of normal programs. While they present a method for computing stable models in their work, it involves a translation of normal to definite programs plus a non-differentiable operation, thus, excluding gradient-based search. More recently, the authors considered the computation of 3-valued models of a program's completion in vector spaces as a first step towards computing supported models (Sato, Sakama, and Inoue 2020), although the method is also non-differentiable. Further non-differentiable tensor-based characterizations of logic programming have been considered for datalog programs (Sato 2017) and abduction (Sato, Inoue, and Sakama 2018; Aspis, Broda, and Russo 2018).

Blair et al. (1999) consider an extension of two-valued logic into continuous vector space by means of a polynomial representation of a given formula. This polynomial can be found by solving a linear system of equations constructed from the formula's truth table. Given a set of normal clauses and their polynomial representation, a continuous dynamical system is constructed such that its fixed points correspond to supported models. These models are then searched for using gradient descent. In contrast, our dynamical system relies on a second-degree polynomial and a Sigmoid. The usage of a Sigmoid is critical, as it allows us to define a two-valued semantics over continuous spaces in a meaningful way, which Blair et al. cannot do.

More recently, gradient-based methods have been considered for logical inference by Sato and Kojima (2019). Their work focuses on abduction, SAT solving and probabilistic inference. Computing supported models is discussed briefly for a restricted class of Datalog programs but the approach is not fully demonstrated or proven. Our approach, on the other hand, focuses on Answer Set Programs. It is our belief that computation in vector spaces has the potential of allowing for ASP solving for large-scale programs, although this is not the focus of this paper.

6 Conclusion

We presented a new method for computing supported and stable models of a propositional normal program, using a gradient-based search algorithm in continuous vector spaces. The method, which relies on a vector representation of interpretations and a matrix representation of program reduct, was proven to maintain the semantics of the original program under appropriate conditions. Experiments demonstrated the feasibility of the approach, as well as the importance of selecting an appropriate distribution for initial vectors, depending on the type of program, to improve the chance of converging to a supported model. The importance of the problem encoding with respect to the length of clauses was also demonstrated.

The results presented in this paper open new possibilities for facilitating neural-symbolic learning and inference. More efficient solving of ASP or SAT is also an interesting goal that we do not take at present, but may be achieved by developing this method further. In particular, alternative approximations of H_1 such as $tanh$ can be considered. Experimentation aimed at establishing good choices of initial vector distributions under various conditions will improve its empirical performance. The method is easily implementable to be run on GPU hardware, allowing many attempts to be carried out in parallel. In addition, both the matrix representation of program reducts and the Jacobian tend to be highly sparse matrices. An implementation using an efficient sparse representation will therefore prove highly beneficial. Finally, less expensive optimisation algorithms such as Quasi-Newton methods may prove effective as well.

Acknowledgements

This research was sponsored by the U.S. Army Research Laboratory and the U.K. Ministry of Defence under Agreement Number W911NF-16-3-0001. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Army Research Laboratory, the U.S. Government, the U.K. Ministry of Defence or the U.K. Government. The U.S. and U.K. Governments are authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. Jorge Lobo was supported by the Spanish Ministry of Economy and Competitiveness under Grant Numbers TIN201681032P, MDM20150502, and the U.S. Army Research Office under agreement number W911NF1910432.

References

Aspis, Y.; Broda, K.; and Russo, A. 2018. Tensor-based abduction in horn propositional programs. In *Up-and-Coming and Short Papers of the 28th International Conference on Inductive Logic Programming*, volume 2206 of *CEUR Workshop Proceedings*, 68–75.

Blair, H. A.; Dushin, F.; Jakel, D. W.; Rivera, A. J.; and Sezgin, M. 1999. *Continuous Models of Computation for Logic Programs: Importing Continuous Mathematics into Logic Programming's Algorithmic Foundations*. Berlin, Heidelberg: Springer Berlin Heidelberg. 231–255.

Bracewell, R. 2000. *The Fourier Transform and Its Applications*. Electrical engineering series. McGraw Hill.

Clark, K. L. 1978. *Negation as Failure*. Springer US. 293–322.

Cohen, W. W. 2016. Tensorlog: A differentiable deductive database.

Dai, W.-Z.; Xu, Q.; Yu, Y.; and Zhou, Z.-H. 2019. Bridging machine learning and logical reasoning by abductive learning. In *Advances in Neural Information Processing Systems*, 2811–2822.

Dantsin, E.; Eiter, T.; Gottlob, G.; and Voronkov, A. 2001. Complexity and expressive power of logic programming. *ACM Computing Surveys (CSUR)* 33(3):374–425.

Evans, R., and Grefenstette, E. 2017. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research* 61.

Gelfond, M., and Lifschitz, V. 1988. The stable model semantics for logic programming. In Kowalski, R.; Bowen; and Kenneth., eds., *Proceedings of International Logic Programming Conference and Symposium*, 1070–1080. MIT Press.

Kaminski, R., and Kaufmann, B. 2012. Answer set solving in practice. *Morgan & Claypool Publishers*.

Kazemi, S. M., and Poole, D. 2017. Relnn: A deep neural model for relational learning. In *AAAI*.

Lifschitz, V. 2008. What is answer set programming? In *Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3*, AAAI'08, 1594–1597. AAAI Press.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. Deepproblog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems*, 3749–3759.

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

Marques-Silva, J.; Lynce, I.; and Malik, S. 2009. *Conflict-driven clause learning SAT solvers*. Number 1 in Frontiers in Artificial Intelligence and Applications. Netherlands: IOS Press, 1 edition. 131–153.

Minervini, P.; Bošnjak, M.; Rocktäschel, T.; Riedel, S.; and Grefenstette, E. 2019. Differentiable reasoning on large knowledge bases and natural language. *ArXiv* abs/1912.10824.

Nguyen, H. D.; Sakama, C.; Sato, T.; and Inoue, K. 2018. Computing logic programming semantics in linear algebra. In *MIWAI*.

Payani, A., and Fekri, F. 2019. Inductive logic programming via differentiable deep neural logic networks.

Sakama, C.; Nguyen, H. D.; Sato, T.; and Inoue, K. 2018. Partial evaluation of logic programs in vector spaces. *ArXiv* abs/1811.11435.

Sakama, C.; Inoue, K.; and Sato, T. 2017. Linear algebraic characterization of logic programs. In Li, G.; Ge, Y.; Zhang, Z.; Jin, Z.; and Blumenstein, M., eds., *Knowledge Science, Engineering and Management*, 520–533. Cham: Springer International Publishing.

Sato, T., and Kojima, R. 2019. Logical inference as cost minimization in vector spaces. In *Proceedings of the Fourth International Workshop on Declarative Learning Based Programming (DeLBP'19), https://delbp.github.io/.*

Sato, T.; Inoue, K.; and Sakama, C. 2018. Abducing relations in continuous spaces. In *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*, 1956–1962. International Joint Conferences on Artificial Intelligence Organization.

Sato, T.; Sakama, C.; and Inoue, K. 2020. From 3-valued semantics to supported model computation for logic programs in vector spaces. In *Proceedings of the 12th International Conference on Agents and Artificial Intelligence (ICAART 2020; Valletta, Malta, 22-24 February 2020)*, 758–765.

Sato, T. 2017. A linear algebraic approach to datalog evaluation. *Theory and Practice of Logic Programming* 17(3):244–265.

Selsam, D.; Lamm, M.; Bünz, B.; Liang, P.; de Moura, L.; and Dill, D. L. 2018. Learning a sat solver from single-bit supervision.

Serafini, L., and d'Avila Garcez, A. 2016. Logic tensor networks: Deep learning and logical reasoning from data and knowledge.

Simons, P. 2000. Extending and implementing the stable model semantics.

Truszczynski, M. 2011. Trichotomy and dichotomy results on the complexity of reasoning with disjunctive logic programs. *Theory and Practice of Logic Programming* 11(6):881–904.

van Emden, M., and Kowalski, R. 1976. The semantics of predicate logic as a programming language. *J. ACM* 23:733–742.

Van Gelder, A.; Ross, K. A.; and Schlipf, J. S. 1991. The well-founded semantics for general logic programs. *J. ACM* 38(3):619–649.