Neuro-Symbolic AI Approaches to Enhance Deep Neural Networks with Logical Reasoning and Knowledge Integration

by

Zhun Yang

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved April 2023 by the
Graduate Supervisory Committee:

Joohyung Lee, Chair
Chitta Baral
Baoxin Li
Yezhou Yang

ARIZONA STATE UNIVERSITY

August 2023

ABSTRACT

One of the challenges in Artificial Intelligence (AI) is to integrate fast, automatic, and intuitive System-1 thinking with slow, deliberate, and logical System-2 thinking. While deep learning approaches excel at perception tasks for System-1, their reasoning capabilities for System-2 are limited. Besides, deep learning approaches are usually data-hungry, hard to make use of explicit knowledge, and struggling with interpretability and justification. This dissertation presents three neuro-symbolic AI approaches that integrate neural networks (NNs) with symbolic AI methods to address these issues.

The first approach presented in this dissertation is NeurASP, which combines NNs with Answer Set Programming (ASP), a logic programming formalism. NeurASP provides an effective way to integrate sub-symbolic and symbolic computation by treating NN outputs as probability distributions over atomic facts in ASP. The explicit knowledge encoded in ASP corrects mistakes in NN outputs and allows for better training with less data.

To avoid NeurASP's bottleneck in symbolic computation, this dissertation presents a Constraint Loss via Straight-Through Estimators (CL-STE). CL-STE provides a systematic way to compile discrete logical constraints into a loss function over discretized NN outputs and scales significantly better than state-of-the-art neuro-symbolic methods. This dissertation also presents a finding when CL-STE was applied to Transformers. Transformers can be extended with recurrence to enhance its power for multi-step reasoning. Such Recurrent Transformer can straightforwardly be applied to visual constraint reasoning problems while successfully addressing the symbol grounding problem.

Lastly, this dissertation addresses the limitation of pre-trained Large Language Models (LLMs) on multi-step logical reasoning problems with a dual-process neuro-symbolic reasoning system called LLM+ASP, where an LLM (e.g., GPT-3) serves as a highly effective few-shot semantic parser that turns natural language sentences into a logical form that can be used as input to ASP. LLM+ASP achieves state-of-the-art performance on several tex-

tual reasoning benchmarks and can handle robot planning tasks that an LLM alone fails to solve.

To my beloved wife, parents, and the little angel who will soon join us, for their unwavering support and encouragement every step of the way.

ACKNOWLEDGMENTS

I am deeply grateful to many individuals who have supported me throughout my academic journey.

Firstly, I would like to express my sincere appreciation to my advisor, Joohyung Lee, for his unwavering guidance and support throughout my academic journey. I first met Joohyung when I was an MS student in his Artificial Intelligence (AI) course, which sparked my interest in AI and inspired me to pursue research in the field. Since then, he has been a remarkable mentor for my MS thesis and Ph.D. dissertation, providing countless insightful suggestions and feedback that have been instrumental in shaping my research. I am also grateful for the opportunities he has provided me to present my research at top-tier conferences, to help organize an international conference (KR 2018), and to serve as a judge representing AAAI in Intel Science and Engineering Fair 2019. I am fortunate to have had such a patient, dedicated, and generous advisor.

I would also like to thank the members of the Automated Reasoning Group (ARG) at ASU. Manjula Malaiarasan, Nikhil Loney, Samidh Talsania, Anish Pradhan, Yi Wang, Jiaxuan Pang, Jinyung Hong, Man Luo, and Adam Ishay have all been instrumental in shaping my work and helping me grow as a good researcher and programmer. Their feedback, mentorship, and support have been invaluable. I would like to offer special thanks to Yi Wang and Adam Ishay for their exceptional assistance, collaboration, and feedback throughout my academic journey.

I would also like to extend my sincere gratitude to my committee members, Chitta Baral, Baoxin Li, and Yezhou Yang, for their invaluable support and guidance throughout my research. Chitta Baral, in particular, has been an incredible mentor and advocate for my work. He has invited me to present my research in his lab and class and has provided insightful suggestions. Baoxin Li and Yezhou Yang have also been incredibly supportive and have provided valuable feedback and suggestions for both research and career path.

I feel fortunate to have had such a knowledgeable and supportive committee, and their contributions have been invaluable to my academic journey.

Finally, I would like to express my deepest gratitude to my wife, Jing Zhou, for her unwavering support, encouragement, and understanding throughout my entire Ph.D. journey. Her love and presence have been a constant source of strength and inspiration. I would also like to thank my parents, Fenlin Xia and Chunping Yang, for their unconditional love, guidance, and inspiration. They have always believed in me, and their unwavering faith in my abilities has motivated me to study abroad and pursue my dreams. And lastly, I want to thank my little angel, who is on her way to join our family, for bringing added blessings and inspiration to my life.

My work in this thesis was partially supported by the National Science Foundation under Grants IIS-1319794, IIS-1526301, IIS-1815337, and IIS-2006747.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

Chapter 1

INTRODUCTION

System 1 and system 2 thinking (Kahneman, 2011) are two distinct modes of cognitive processing: system 1 thinking is fast, automatic, and intuitive while system 2 thinking is slow, deliberate, and logical. These years, system 1 thinking is widely modeled by deep learning approaches, which excel at perception tasks such as language (Vaswani *et al.*, 2017; Zhang *et al.*, 2020; Helwe *et al.*, 2021; Li *et al.*, 2020) and vision (Dosovitskiy *et al.*, 2020; Gabeur *et al.*, 2020) with unprecedented success.

Although there has been some success to implement system 2 thinking in deep neural networks (Šourek *et al.*, 2015; Rocktäschel and Riedel, 2017; Donadello *et al.*, 2017; Kazemi and Poole, 2018; Cohen *et al.*, 2018; Palm *et al.*, 2018; Lin *et al.*, 2019), the reasoning supported in neural networks is still considered shallow (Helwe *et al.*, 2021). For example, while large language models (LLMs) are considered one of the most powerful deep neural network models for various tasks (Brown *et al.*, 2020), Nye *et al.* (2021) noted that LLMs work well for system 1 intuitive thinking but not for system 2 logical thinking. Creswell *et al.* (2022) also asserted that LLMs tend to perform poorly on multi-step logical reasoning problems. On the other hand, the latter subject has been well-studied in the area of symbolic AI, which allows for convenient representation of knowledge (Lifschitz, 2008a; Brewka *et al.*, 2011b; Lee and Wang, 2016) and complex reasoning on both certainty and uncertainty (Katzouris and Artikis, 2020; Verreet *et al.*, 2022; Lee and Yang, 2023).

Symbolic AI does not incorporate high-dimensional vector space and state-of-the-art models for perception tasks as handled in deep neural networks, which limits the applicability of symbolic AI in many practical applications. On the other hand, deep neural net-

works are usually data-hungry and hard to make use of explicit (commonsense or expert) knowledge. They also lack explainability and justification since rich amount of information are stored in vector space and hard to interpret.

Since the strengths of deep neural networks and symbolic AI approaches are complementary, neuro-symbolic AI (Besold *et al.*, 2017; Mao *et al.*, 2019; De Raedt *et al.*, 2019; Garcez *et al.*, 2019) became an active research area with the hopes of a best-of-both worlds scenario. There is no consensus on such combination and various approaches have been proposed with different strengths.

To keep sound reasoning, some recent works in neuro-symbolic AI (Manhaeve *et al.*, 2018; Tsamoura *et al.*, 2021) associate continuous parameters in neural networks (NNs) with logic languages so that logical reasoning applied to NN outputs produces "semantic loss" (Xu *et al.*, 2018). However, these logic languages do not allow many flexible Knowledge Representation (KR) constructs, such as recursive definition, cardinality constraints, and weak constraints, which are supported by answer set programming (ASP) (Lifschitz, 2008b; Brewka *et al.*, 2011a) for convenient representation of complex knowledge. To fill in the gap, we proposed a neuro-symbolic formalism `NeurASP` (Yang *et al.*, 2020), which is a simple extension of ASP where NN outputs are treated as probability distributions over atomic facts in answer set programs. `NeurASP` provides a convenient way to enforce structured knowledge in NNs. We defined the probability of each "intended model" in `NeurASP` using neural network outputs and proved the mathematical properties of its gradients. We empirically showed that `NeurASP` can improve a neural network's perception result by applying symbolic reasoning in ASP and train a neural network better with ASP rules so that a neural network not only learns from implicit correlations from the data but also from the explicit complex semantic constraints expressed by the rules. Remarkably, the same CNN for Sudoku (Park, 2018) trained by `NeurASP` achieves 66.5% accuracy with 70k unlabeled data while the baseline only achieves 23.3% accuracy with

fully-labeled data.

On the other hand, the symbolic computation is often the bottleneck of the above methods, limits the computation to CPUs only, and restricts their domain of usage to relatively small scale. To allow more time-efficient learning and larger domains, we design a semantic regularization method `CL-STE` (Yang *et al.*, 2022), which provides a systematic way to represent discrete logical constraints as a regularization function over discretized NN output. To make a discretizing function meaningfully differentiable, we turn to the idea of straight-through estimators (STE) (Courbariaux *et al.*, 2015), which were originally introduced to train binary neural networks — neural networks with binary weights and activation at run-time. The main idea of STE is to use a binarization function in forward propagation while its gradient, which is zero almost everywhere, is replaced by the gradient of a different function in backward propagation. We proved that minimizing the loss in `CL-STE` method using gradient descent via a straight-through-estimator updates the neural network's weights in the direction that the binarized outputs satisfy the logical constraints. Our experimental results on `CL-STE` show that a neural network can be trained better with less data and fewer labels when semantic constraints are given through `CL-STE`. Improvements of accuracy have been observed on fully-, semi-, and un-supervised learning tasks over different types of neural networks including multi-layer perceptron (MLP), convolutional neural networks (CNN), graph neural networks (GNN), and Transformers. More importantly, leveraging GPUs and batch training, `CL-STE` method scales significantly better on a big range of existing benchmark problems compared to state-of-the-art neuro-symbolic methods that use heavy symbolic computation as a blackbox for computing gradients.

When applying `CL-STE` to Transformers, we found that Transformers can be extended with recurrence to enhance its power for multi-step reasoning and can be a viable approach to learning to solve Constraint Satisfaction Problems (CSPs) in an end-to-end manner. We designed `Recurrent Transformer` (Yang *et al.*, 2023) and showed

that it has clear advantages over state-of-the-art methods such as Graph Neural Networks (Palm *et al.*, 2018), SATNet (Wang *et al.*, 2019), and some neuro-symbolic models (Yang *et al.*, 2020; Bai *et al.*, 2021). With the ability of Transformer to handle visual input, `Recurrent Transformer` can straightforwardly be applied to visual constraint reasoning problems while successfully addressing the symbol grounding problem (Chang *et al.*, 2020; Topan *et al.*, 2021). We also designed a variant of `CL-STE`, which is restricted to cardinality constraints only but is even more computationally efficient than `CL-STE`. We showed how to leverage deductive knowledge of discrete constraints in the Transformer's inductive learning to achieve sample-efficient learning and semi-supervised learning for CSPs.

Recently, LLMs have shown wide success on many downstream tasks, demonstrating general reasoning capability on diverse tasks without being retrained. However, when we restrict our attention to individual NLP reasoning benchmarks, they usually do not perform as well as state-of-the-art models despite various efforts to improve accuracy through prompt engineering (Wei *et al.*, 2022; Zhou *et al.*, 2022). As training LLMs is not capable, we limited our attention to inference with LLMs and designed a dual-process neuro-symbolic reasoning system `LLM+ASP`. We found that the rich semantic knowledge that LLMs possess makes them effective general-purpose few-shot semantic parsers that can convert linguistically variable natural language sentences into atomic facts that serve as input to logic programs. We also found that the fully declarative nature of answer set programs (Lifschitz, 2008a; Brewka *et al.*, 2011b) makes them a good pair with the LLM semantic parsers, providing interpretable and explainable reasoning on the parsed result of the LLMs using background knowledge. `LLM+ASP` works across multiple QA tasks without retraining for individual tasks. It requires only a few examples to direct an LLM (i.e., GPT-3) to tune to an individual task, along with ASP knowledge modules that can be reused over multiple tasks. We demonstrated that this method achieves state-of-the-art

performance on several NLP benchmarks, such as bAbI (Weston *et al.*, 2016), StepGame (Shi *et al.*, 2022), CLUTRR (Sinha *et al.*, 2019), and gSCAN (Ruis *et al.*, 2020), and also handles robot planning tasks that an LLM alone fails to solve.

The dissertation is organized as follows. Chapter 2 examines related works. Chapter 3 presents a neuro-symbolic formalism `NeurASP`, which integrates neural networks with answer set programs. Chapter 4 describes a semantic regularization method `CL-STE`. Chapter 5 introduces `Recurrent Transformer` and a semantic loss stemmed from `CL-STE`. In Chapter 6, we present `LLM+ASP`, which augments LLMs with symbolic reasoning in ASP for textual question-answering. Chapter 7 concludes.

Chapter 2

BACKGROUND

2.1 Neuro-Symbolic Methods

Recent years have observed the rising interest in combining neural and symbolic systems (Marcus, 2018; Lamb *et al.*, 2020; Sarker *et al.*, 2021).

Xu *et al.* (2018) proposed a semantic loss function to bridge neural network (NN) output and logical constraints. The method treats NN output as probabilities and computes semantic loss as the negative logarithm of the probability to generate a state satisfying the logical constraints. Their experiments show that the encoded semantic loss function guides the learner to achieve state-of-the-art results on supervised and semi-supervised learning on multi-class classification. For the efficient computation of a loss function, they encode logical constraints in Sentential Decision Diagram (SDD) (Darwiche, 2011). However, generating SDDs is computationally expensive for most practical tasks.

Several neuro-symbolic formalisms, such as Logic Tensor Network (Serafini and Garcez, 2016), `DeepProbLog` (Manhaeve *et al.*, 2018), `NeuroLog` (Tsamoura *et al.*, 2021), `DeepStochLog` (Winters *et al.*, 2021), and fuzzy logic regularizer (Roychowdhury *et al.*, 2021), have been proposed to enforce logical constraints in neural network training by appending a logic layer to an existing neural network. Many of them treat the logic component as a blackbox module (Pogancic *et al.*, 2020) to neural networks. Since discrete logical inference cannot be in general captured via a differentiable function, they use relaxation to fuzzy logic or weighted models or probability. Our work `NeurASP` (Yang *et al.*, 2020) is also of this kind where the logic layer is built upon the ASP solver CLINGO (Lifschitz, 2008b; Brewka *et al.*, 2011a; Calimeri *et al.*, 2020). While this approach provides a

systematic representation of constraints, the symbolic computation is often a bottleneck.

Another approach is to embed logic rules in neural networks by representing logical connectives by mathematical operations and allowing the value of an atom to be a real number. For example, Neural Theorem Prover (NTP) (Rocktäschel and Riedel, 2017) adopts the idea of dynamic neural module networks (Andreas *et al.*, 2016) to embed logic conjunction and disjunction in and/or-module networks. A proof-tree like end-to-end differentiable neural network is then constructed using Prolog's backward chaining algorithm with these modules. Another method that also constructs a proof-tree like neural network is TensorLog (Cohen *et al.*, 2018), which uses matrix multiplication to simulate belief propagation that is tractable under the restriction that each rule is negation-free and can be transformed into a polytree.

Other works train neural networks for learning satisfiability, such as (Wang *et al.*, 2019; Selsam *et al.*, 2019). SATNet (Wang *et al.*, 2019) builds on a line of research exploring SDP relaxations as a tool for solving MAXSAT, which produces tighter approximation guarantees than standard linear programming relaxation. Remarkably, their method learns to solve Sudoku puzzles without any hand-coded knowledge. On the other hand, our experiments show that SATNet does not extrapolate well — it does not work well for reasonably harder Sudoku instances than those that it is trained on.

Graph Neural Networks (GNNs) (Kipf and Welling, 2017; Battaglia *et al.*, 2018; Lamb *et al.*, 2020) have been widely applied. Since a graph can encode objects and relations between objects, by learning message functions between the nodes, one can perform certain relational reasoning over the objects. For example, ExpressGNN (Zhang *et al.*, 2019) constructs a graph neural network to simulate variational inference in Markov Logic Network. Recurrent Relational Network (RRN) (Palm *et al.*, 2018) is a state-of-the-art GNN for multi-step relational reasoning, achieving 96.6% accuracy for Sudoku problems. GNNs use message-passing to propagate logical constraints in neural networks, but they do not

have the mechanism to specify the logical constraints directly.

Neuro-Symbolic Concept Learner (Mao *et al.*, 2019) separates between visual perception and symbolic reasoning. It shows the data-efficiency by using only 10% of the training data and achieving the state-of-the-art 98% accuracy on CLEVR dataset. Such kind of dual-system models achieved new state-of-the-art results in visual QA (Goldman *et al.*, 2018; Sampat and Lee, 2018; Yi *et al.*, 2019; Chen *et al.*, 2020; Ding *et al.*, 2021). In the case of textual problems, to improve LLMs to generate more consistent and coherent sentences, Nye *et al.* (2021) suggest that generation be decomposed into two parts: candidate sentence generation by an LLM (system 1 thinking) and a logical pruning process (system 2 thinking) implemented via a separate symbolic module. They show that this neuro-symbolic, dual-process model requires fewer data to learn and achieves higher accuracy and better generalization.

2.2 Answer Set Programs

Answer Set Programming (ASP) (Lifschitz *et al.*, 2001), based on the stable model semantics (Gelfond and Lifschitz, 1988), is a widely-used Knowledge Representation (KR) framework that facilitates elegant and efficient representations for many problem domains that require complex reasoning.

We assume a first-order signature σ that contains no function constants of positive arity, which yields finitely many Herbrand interpretations. The syntax of formulas is defined the same as in the standard first-order logic. We say that a formula is *negative* if every occurrence of every atom in this formula is in the scope of negation.

In this dissertation, for simplicity, we mainly consider a *rule* of the form

$$A \leftarrow B \wedge N \tag{2.1}$$

where A is a disjunction of atoms, B is a conjunction of atoms, and N is a negative formula

constructed from atoms using conjunction, disjunction and negation. We identify rule (2.1) with formula $B \wedge N \rightarrow A$. We often use comma for conjunction, semi-colon for disjunction, *not* for negation, as widely used in the literature on logic programming. For example, N could be

$$\neg B_{m+1} \wedge \ldots \wedge \neg B_n \wedge \neg\neg B_{n+1} \wedge \ldots \wedge \neg\neg B_p,$$

which can be also written as

$$\textit{not } B_{m+1}, \ldots, \textit{not } B_n, \textit{not not } B_{n+1}, \ldots, \textit{not not } B_p.$$

We write $\{A_1\}^{\mathrm{ch}} \leftarrow \textit{Body}$, where A_1 is an atom, to denote the rule $A_1 \leftarrow \textit{Body} \wedge \neg\neg A_1$. This expression is called a "choice rule" in ASP.

If the head of a rule (A in (2.1)) is $\bot$, we often omit it and call such a rule *constraint*.

A logic program under the stable model semantics (a.k.a. *answer set program*) is a finite conjunction of rules. A logic program is called *ground* if it contains no variables.

We say that an Herbrand interpretation I is a *model* of a ground program Π if I satisfies all implications (2.1) in Π (as in classical logic). Such models can be divided into two groups: "stable" and "non-stable" models, which are distinguished as follows. The *reduct* of Π relative to I, denoted Π^I, consists of "$A \leftarrow B$" for all rules (2.1) in Π such that $I \models N$ ($I \models N$ denotes "I satisfies N" in classical logic). The Herbrand interpretation I is called a *(deterministic) stable model* of Π if I is a minimal Herbrand model of Π^I. (Minimality is understood in terms of set inclusion. We identify an Herbrand interpretation with the set of atoms that are true in it.)

The definition is extended to any non-ground program Π by identifying it with $gr_\sigma[\Pi]$, the ground program obtained from Π by replacing every variable with every ground term of the signature σ.

A *weak constraint* (Buccafurri *et al.*, 2000; Calimeri *et al.*, 2012) has the form

$$:\sim F \quad [Weight \ @ \ Level]$$

where F is a conjunction of literals, $Weight$ is a real number, and $Level$ is a nonnegative integer.

Let Π be a program $\Pi_1 \cup \Pi_2$, where Π_1 is an answer set program that does not contain weak constraints, and Π_2 is a set of ground weak constraints. We call I a *stable model* of Π if it is a stable model of Π_1. For every stable model I of Π and any nonnegative integer l, the *penalty* of I at level l, denoted by $Penalty_\Pi(I, l)$, is defined as

$$\sum_{:\sim\ F[w@l]\in\Pi_2,\ I\models F} w.$$

For any two stable models I and I' of Π, we say I is *dominated* by I' if

- there is some nonnegative integer l such that $Penalty_\Pi(I', l) < Penalty_\Pi(I, l)$ and
- for all integers $k > l$, $Penalty_\Pi(I', k) = Penalty_\Pi(I, k)$.

A stable model of Π is called *optimal* if it is not dominated by another stable model of Π.

Chapter 3

NEURASP

$\texttt{LP}^{\texttt{MLN}}$ (Lee and Wang, 2016) is a probabilistic logic programming language that extends answer set programs with the concept of weighted rules, whose weight scheme is adopted from that of Markov Logic (Richardson and Domingos, 2006). Since the parameter learning of $\texttt{LP}^{\texttt{MLN}}$ is already by the gradient descent method (Lee and Wang, 2018) as used in the neural network training, it's natural to combine ASP and neural networks through $\texttt{LP}^{\texttt{MLN}}$. However, there is a technical challenge: the existing parameter learning method for $\texttt{LP}^{\texttt{MLN}}$ does not scale up to be coupled with typical neural network training. For example, the simple integration of NN and $\texttt{LP}^{\texttt{MLN}}$ cannot complete the training in a day for the MNIST Addition problem proposed in Manhaeve *et al.* (2018). This motivates us to consider a fragment of $\texttt{LP}^{\texttt{MLN}}$ where the interface between neural networks and ASP is specified by a *neural atom* originally proposed in `DeepProbLog` (Manhaeve *et al.*, 2018). It turns out that this fragment is general enough to cover the ProbLog language and keeps the expressiveness of full ASP language. The efficient algorithms we designed to compute probabilities and gradients in this fragment further serve as the basis of `NeurASP`. By treating the neural network output as the probability distribution over atomic facts in ASP, `NeurASP` provides a simple and effective way to integrate sub-symbolic and symbolic computation in both neural network inference and learning.

3.1 Syntax

We assume that neural network M allows an arbitrary tensor as input whereas the output is a matrix in $\mathbb{R}^{e \times n}$, where e is the number of random events predicted by the neural network and n is the number of possible outcomes for each random event. Each row of the matrix

represents the probability distribution of the outcomes of each event. For example, if M is a neural network for MNIST digit classification, then the input is a tensor representation of a digit image, e is 1, and n is 10. If M is a neural network that outputs a Boolean value for each edge in a graph, then e is the number of edges and n is 2. Given an input tensor $\mathbf{t}$, by $M(\mathbf{t})$, we denote the output matrix of M. The value $M(\mathbf{t})[i, j]$ (where $i \in \{1, \ldots, e\}$, $j \in \{1, \ldots, n\}$) is the probability of the j-th outcome of the i-th event upon the input $\mathbf{t}$.

In `NeurASP`, the neural network M above can be represented by a *neural atom* of the form

$$nn(m(e, t), [v_1, \ldots, v_n]), \tag{3.1}$$

where (i) nn is a reserved keyword to denote a neural atom; (ii) m is an identifier (symbolic name) of the neural network M; (iii) t is a list of terms that serves as a "pointer" to an input data; related to it, there is a mapping $\mathbf{D}$ (implemented by an external Python code) that turns t into an input tensor; (iv) $v_1, \ldots, v_n$ represent all n possible outcomes of each of the e random events.

Each neural atom (3.1) introduces propositional atoms of the form $c = v$, where $c \in \{m_1(t), \ldots, m_e(t)\}$ and $v \in \{v_1, \ldots, v_n\}$. The output of the neural network provides the probabilities of the introduced atoms (defined in Section 3.2).

Example 1 *Let M_{digit} be a neural network that classifies an MNIST digit image. The input of M_{digit} is (a tensor representation of) an image and the output is a matrix in $\mathbb{R}^{1\times 10}$. The neural network can be represented by the neural atom*

$$nn(digit(1, d),\ [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]),$$

which introduces propositional atoms $digit_1(d)=0$, $digit_1(d)=1$, …, $digit_1(d)=9$.

Example 2 *Let M_{sp} be another neural network for finding the shortest path in a graph with 24 edges. The input is a tensor encoding the graph and the start/end nodes of the*

path, and the output is a matrix in $\mathbb{R}^{24\times 2}$*. This neural network can be represented by the neural atom*

$$nn(sp(24, g),\ [\text{TRUE}, \text{FALSE}]).$$

A `NeurASP` *program* Π is the union of Π^{asp} and Π^{nn}, where Π^{asp} is a set of propositional rules (standard rules as in ASP-Core 2 (Calimeri *et al.*, 2020)) and Π^{nn} is a set of neural atoms. Let σ^{nn} be the set of all atoms $m_i(t) = v_j$ that is obtained from the neural atoms in Π^{nn} as described above. We require that, in each rule *Head* $\leftarrow$ *Body* in Π^{asp}, no atoms in σ^{nn} appear in *Head*.

We could allow schematic variables into Π, which are understood in terms of grounding as in standard answer set programs. We find it convenient to use rules of the form

$$nn(m(e, t), [v_1, \ldots, v_n]) \leftarrow \textit{Body} \tag{3.2}$$

where *Body* is either identified by $\top$ or $\bot$ during grounding so that (3.2) can be viewed as an abbreviation of multiple (variable-free) neural atoms (3.1).

Example 3 *An example* `NeurASP` *program* Π_{digit} *is as follows, where* d_1 *and* d_2 *are terms representing two images. Each image is classified by neural network* M_{digit} *as one of the values in* $\{0, \ldots, 9\}$*. The addition of two digit-images is the sum of their values.*

$$\begin{aligned}
&img(d_1).\\
&img(d_2).\\
&nn(digit(1, X), [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) \leftarrow img(X).\\
&addition(A, B, N) \leftarrow digit_1(A) = N_1, digit_1(B) = N_2, N = N_1 + N_2.
\end{aligned} \tag{3.3}$$

The neural network M_{digit} *outputs 10 probabilities for each image. The addition is applied once the digits are recognized and its probability is induced from the perception as we explain in the next section.*

3.2 Semantics

For any NeurASP program $\Pi = \Pi^{asp} \cup \Pi^{nn}$, we first obtain its ASP counterpart $\Pi' = \Pi^{asp} \cup \Pi^{ch}$ where Π^{ch} consists of the following set of rules for each neural atom (3.1) in Π^{nn}

$$\{m_i(t){=}v_1;\ \ \ldots\ ; m_i(t){=}v_n\} = 1 \quad \text{for } i \in \{1, \ldots e\}.$$

The above rule (in the language of CLINGO) means to choose exactly one atom in between the set braces.[1] We define the *stable models* of Π as the stable models of Π', and define the *total choices* of Π as the stable models of Π^{ch}. For each total choice C of Π, we use $\mathit{Num}(C, \Pi)$ to denote the number of stable models of Π that satisfy C. We require a NeurASP program Π to be *coherent* such that $\mathit{Num}(C, \Pi) > 0$ for every total choice C of Π.

To define the probability of a stable model, we first define the probability of an atom $m_i(t) = v_j$ in σ^{nn}. Recall that there is an external mapping $\mathbf{D}$ that turns t into a specific input tensor of M. The probability of each atom $m_i(t){=}v_j$ is defined as $M(\mathbf{D}(t))[i, j]$:

$$P_\Pi(m_i(t){=}v_j) = M(\mathbf{D}(t))[i, j].$$

For instance, recall that the output matrix of $M_{digit}(\mathbf{D}(d))$ in Example 3 is in $\mathbb{R}^{1\times 10}$. The probability of atom $digit_1(d) = k$ is $M_{digit}(\mathbf{D}(d))[1, k{+}1]$.

Given an interpretation I, by $I|_{\sigma^{nn}}$, we denote the projection of I onto σ^{nn}. Since $I|_{\sigma^{nn}}$ is a total choice of Π, $\mathit{Num}(I|_{\sigma^{nn}}, \Pi)$ is the number of stable models of Π that agree with $I|_{\sigma^{nn}}$ on σ^{nn}.

The probability of a stable model I of Π is defined as the product of the probability of each atom $c = v$ in $I|_{\sigma^{nn}}$, divided by the number of stable models of Π that agree with

[1] In practice, each atom $m_i(t) = v$ is written as $m(i, t, v)$.

$I|_{\sigma^{nn}}$ on σ^{nn}. That is, for any interpretation I,

$$P_{\Pi}(I) = \begin{cases} \frac{\prod_{c=v \in I|_{\sigma^{nn}}} P_{\Pi}(c=v)}{\mathit{Num}(I|_{\sigma^{nn}}, \Pi)} & \text{if } I \text{ is a stable model of } \Pi; \\ 0 & \text{otherwise.} \end{cases}$$

An *observation* is a set of ASP constraints (i.e., rules of the form $\bot \leftarrow \mathit{Body}$). The probability of an observation O is defined as

$$P_{\Pi}(O) = \sum_{I \models O} P_{\Pi}(I)$$

($I \models O$ denotes that I satisfies O).

The probability of the set $\mathbf{O} = \{O_1, \ldots, O_o\}$ of observations is defined as the product of the probability of each O_i:

$$P_{\Pi}(\mathbf{O}) = \prod_{O_i \in \mathbf{O}} P_{\Pi}(O_i).$$

Example 3 Continued The ASP program Π'_{digit}, which is the ASP counterpart of Π_{digit}, is obtained from (3.3) by replacing the third rule with

$$\{digit_1(d_1){=}0; \ \ldots \ ; digit_1(d_1){=}9\} = 1.$$
$$\{digit_1(d_2){=}0; \ \ldots \ ; digit_1(d_2){=}9\} = 1.$$

The following are the stable models of Π_{digit}, i.e., the stable models of Π'_{digit}.

$$I_1 = \{digit_1(d_1){=}0, digit_1(d_2){=}0, addition(d_1, d_2, 0), \ldots\},$$
$$I_2 = \{digit_1(d_1){=}0, digit_1(d_2){=}1, addition(d_1, d_2, 1), \ldots\},$$
$$I_3 = \{digit_1(d_1){=}1, digit_1(d_2){=}0, addition(d_1, d_2, 1), \ldots\},$$
$$\ldots,$$
$$I_{100} = \{digit_1(d_1){=}9, digit_1(d_2){=}9, addition(d_1, d_2, 18), \ldots\}.$$

Their probabilities are as follows:

$$P_{\Pi}(I_1) = M_{digit}(\mathbf{D}(d_1))[1,1] \times M_{digit}(\mathbf{D}(d_2))[1,1],$$
$$\ldots,$$
$$P_{\Pi}(I_{100}) = M_{digit}(\mathbf{D}(d_1))[1,10] \times M_{digit}(\mathbf{D}(d_2))[1,10].$$

The probability of $O = \{\leftarrow \mathit{not}\ addition(d_1, d_2, 1)\}$ is

$$P_{\Pi}(O) = P_{\Pi}(I_2) + P_{\Pi}(I_3).$$

3.3 Inference with NeurASP

We implemented NeurASP by integrating PyTorch (Adam *et al.*, 2017) and CLINGO (Gebser *et al.*, 2011). PyTorch takes care of neural network processing including data loading and mapping $\mathbf{D}$ that maps pointer terms in neural atoms to input tensors. Computing the probability of a stable model is done by calling CLINGO and post-processing in Python. This section illustrates how this integration can be useful in reasoning about relations among objects recognized by neural networks.

3.3.1 Commonsense Reasoning about Image

Suppose we have a neural network M_{label} that outputs classes of objects in the bounding boxes that are already detected. The following rule asserts that the neural network M_{label} classifies the bounding box B into one of $\{car, cat, person, truck, other\}$, where B is at location (X_1, Y_1, X_2, Y_2) in image I:

$$nn(label(1, I, B), [car, cat, person, truck, other]) \leftarrow box(I, B, X_1, Y_1, X_2, Y_2).$$

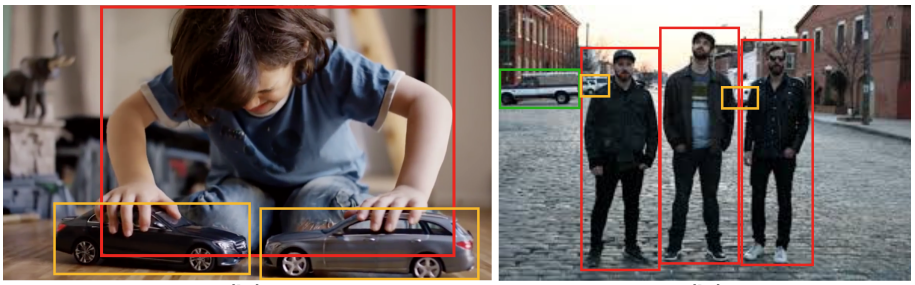

Figure 3.1: Reasoning about Relations among Perceived Objects

Consider the two images i_1 and i_2 in Figure 3.3.1. The bounding boxes can be represented by the following facts.

$$box(i_1, b_1, 100, 0, 450, 350).$$
$$box(i_1, b_2, 300, 300, 500, 400).$$
$$...$$

The first rule says that there is a bounding box b_1 (i.e., the red box with a child) in image i_1, and the coordinates of its left-top and right-bottom corners are $(100, 0)$ and $(450, 350)$.

Below we describe rules that allow for reasoning about the recognized objects. The following rules describe the general size relation between objects.

$$smaller(cat, person).$$
$$smaller(person, car).$$
$$smaller(person, truck).$$
$$smaller(X, Y) \leftarrow smaller(X, Z), smaller(Z, Y).$$

Next is the rule asserting that by default we conclude the same size relationship as above.

$$smaller(I, B_1, B_2) \leftarrow \textit{not } {\sim}smaller(I, B_1, B_2),$$
$$label_1(I, B_1) = L_1, label_1(I, B_2) = L_2, smaller(L_1, L_2).$$

(The $\sim$ symbol stands for strong negation in ASP, which asserts explicit falsity.) On the other hand, there are some exceptions, for instance,

$${\sim}smaller(I, B_2, B_1) \leftarrow box(I, B_1, X_1, Y_1, X_2, Y_2), box(I, B_2, X_1', Y_1', X_2', Y_2'),$$
$$Y_2 \geq Y_2', |X_1 - X_2| \times |Y_1 - Y_2| < |X_1' - X_2'| \times |Y_1' - Y_2'|.$$

$$smaller(I, B_1, B_2) \leftarrow {\sim}smaller(I, B_2, B_1).$$

$$toy(I, B_1) \leftarrow label_1(I, B_1) = L_1,\ label_1(I, B_2) = L_2,$$
$$smaller(I, B_1, B_2),\ smaller(L_2, L_1).$$

The first rule says that "B_2 is not smaller than B_1 if (i) B_1 and B_2 are objects in image I, (ii) B_1 is closer to the camera (i.e., B_1's bottom boundary is closer to the bottom of I), and (iii) the box in the image for B_1 is smaller than B_2." [2]

The neural network model M_{label} outputs that the red boxes are persons, the yellow boxes are cars, and the green box is a truck. Upon this input and the rules above, NeurASP allows us to derive that the two cars in image i_1 are *toy* cars, whereas the two cars in image i_2 are not: although they are surrounded by smaller boxes than those of humans, their boxes are not closer to the camera.

3.3.2 Solving Sudoku Puzzle in Image

Consider the task of solving a Sudoku puzzle given as an image. In NeurASP, we could use a neural network to recognize the digits in the given puzzle and use an ASP solver to compute the solution instead of having a single network that accounts for both perception and solving.

We use the following NeurASP program Π_{sudoku} to first identify the digits in each grid cell on the board and then find the solution by assigning digits to all empty grid cells. [3]

```
% identify the number in each of the 81 positions
nn(identify(81, img), [empty,1,2,3,4,5,6,7,8,9]).


% assign one number N to each position (R,C)
a(R,C,N) :- identify(Pos,img,N), R=Pos/9, C=Pos\9, N!=empty.
{a(R,C,N): N=1..9}=1 :- identify(Pos, img, empty), R=Pos/9, C=Pos\9.


% no number repeats in the same row
```

[2] We assume that the camera is at the same height as the objects.

[3] The expression $\{a(R,C,N) : N = 1..9\} = 1$ is a shorthand for $\{a(R,C,1); \ldots; a(R,C,9)\} = 1$ in the language of CLINGO.

```
:- a(R,C1,N), a(R,C2,N), C1!=C2.


% no number repeats in the same column
:- a(R1,C,N), a(R2,C,N), R1!=R2.


% no number repeats in the same 3*3 box
:- a(R,C,N), a(R1,C1,N), R!=R1, C!=C1, ((R/3)*3 + C/3) = ((R1/3)*3 + C1/3).
```

The neural network model $M_{identify}$ is rather simple. It is composed of 5 convolutional layers with dropout, a max pooling layer, and a 1×1 convolutional layer followed by softmax. Given a Sudoku board image (.png file), neural network $M_{identify}$ outputs a matrix in $\mathbb{R}^{81\times 10}$, which represents the probabilities of the values (empty, 1, . . . , 9) in each of the 81 grid cells. The network $M_{identify}$ is pre-trained using $\langle image, label\rangle$ pairs, where each $image$ is a Sudoku board image generated by ***OpenSky Sudoku Generator*** (http://www.opensky.ca/~jdhildeb/software/sudokugen/) and each $label$ is a vector of length 81 in which 0 is used to represent an empty cell at that position.

Let $Acc_{identify}$ denote the accuracy of identifying all empty cells and the digits on the board given as an image without making a single mistake in a grid cell. Let Acc_{sol} denote the accuracy of solving a given Sudoku board without making a single mistake in a grid cell. Let r be the following rule in Π_{sudoku}:

```
{a(R,C,N): N=1..9}=1 :- identify(Pos, img, empty), R=Pos/9, C=Pos\9.
```

Table 3.1 compares $Acc_{identify}$ of each of $M_{identify}$, `NeurASP` program $\Pi_{sudoku} \setminus r$ with $M_{identify}$, `NeurASP` program Π_{sudoku} with $M_{identify}$, as well as Acc_{sol} of Π_{sudoku} with $M_{identify}$.

Intuitively, $\Pi_{sudoku} \setminus r$ only checks whether the identified numbers (by neural network $M_{identify}$) satisfy the three constraints (the last three rules of Π_{sudoku}), while Π_{sudoku} further

Table 3.1: Sudoku: Accuracy on Test Data

Num of Train Data	$Acc_{identify}$ of $M_{identify}$	$Acc_{identify}$ of NeurASP w/ $\Pi_{sudoku} \setminus r$	$Acc_{identify}$ of NeurASP w/ Π_{sudoku}	Acc_{sol} of NeurASP w/ Π_{sudoku}
15	15%	49%	71%	71%
17	31%	62%	80%	80%
19	72%	90%	95%	95%
21	85%	95%	98%	98%
23	93%	99%	100%	100%
25	100%	100%	100%	100%

checks whether there exists a solution given the identified numbers. As shown in Table 3.1, the use of reasoning in NeurASP program $\Pi_{sudoku} \setminus r$ improves the accuracy $Acc_{identify}$ of the neural network $M_{identify}$ as explained in the introduction. The accuracy $Acc_{identify}$ is further improved by trying to solve Sudoku completely using Π_{sudoku}. Note that the solution accuracy Acc_{sol} of Π_{sudoku} is equal to the perception accuracy $Acc_{identify}$ of Π_{sudoku} since the ASP yields a 100% correct solution once the board is correctly identified.

Palm *et al.* (2018) use a Graph Neural Network to solve Sudoku but the work restricts attention to textual input of the Sudoku board, not images as we do. Their work achieves 96.6% accuracy after training with 216,000 examples. In comparison, even with the more challenging task of accepting images as input, the number of training examples we used is 15 – 25, which is much less than the number of training examples used in (Palm *et al.*, 2018). Our work takes advantage of the fact that in a problem like Sudoku, where the constraints are explicitly given, a neural network only needs to focus on perception tasks, which is simpler than learning the perception and reasoning together.

Furthermore, using the same trained perception neural network $M_{identify}$, we can solve

some elaborations of Sudoku problems by adding the following rules:

[Anti-knight Sudoku] No number repeats at a knight move

```
:- a(R1,C1,N), a(R2,C2,N), |R1-R2|+|C1-C2|=3.
```

[Sudoku-X] No number repeats at the diagonals

```
:- a(R1,C1,N), a(R2,C2,N), R1=C1, R2=C2, R1!=R2.
:- a(R1,C1,N), a(R2,C2,N), R1+C1=8, R2+C2=8, R1!=R2.
```

With neural network only approach, since the neural network needs to learn both perception and reasoning, each of the above variations would require training a complex and different model with a big dataset. However, with `NeurASP`, the neural network only needs to recognize digits on the board. Thus solving each Sudoku variation above uses the same pre-trained model for the image input and we only need to add the aforementioned rules to Π_{sudoku}.

Some Sudoku variations, such as Offset Sudoku, are in colored images. In this case, we need to increase the number of channels of $M_{identify}$ from 1 to 3, and need to retrain the neural network with the colored images. Although not completely elaboration tolerant, compared to the pure neural network approach, this is significantly simpler. For instance, the number of training data needed to get 100% perception accuracy for Offset Sudoku ($Acc_{identify}$) is 70, which is still much smaller than what the end-to-end Sudoku solver would require. Using the new network trained, we only need to add the following rule to Π_{sudoku}.

[Offset Sudoku] No number repeats at the same relative position in 3*3 boxes

```
:- a(R1,C1,N), a(R2,C2,N), R1\3 = R2\3, C1\3 = C2\3, R1 != R2, C1 != C2.
```

3.4 Learning in NeurASP

In this section, we show how the semantic constraints expressed in NeurASP can be used to train neural networks better. We denote a NeurASP program by $\Pi(\boldsymbol{\theta})$ where $\boldsymbol{\theta}$ is the set of the parameters in the neural network models associated with Π. Assume a NeurASP program $\Pi(\boldsymbol{\theta})$ and a set $\mathbf{O}$ of observations such that $P_{\Pi(\boldsymbol{\theta})}(O) > 0$ for each $O \in \mathbf{O}$. The task is to find $\hat{\boldsymbol{\theta}}$ that maximizes the log-likelihood of observations $\mathbf{O}$ under program $\Pi(\boldsymbol{\theta})$, i.e.,

$$\hat{\boldsymbol{\theta}} \in \operatorname*{argmax}_{\boldsymbol{\theta}} \ log(P_{\Pi(\boldsymbol{\theta})}(\mathbf{O})),$$

which is equivalent to

$$\hat{\boldsymbol{\theta}} \in \operatorname*{argmax}_{\boldsymbol{\theta}} \ \sum_{O \in \mathbf{O}} log(P_{\Pi(\boldsymbol{\theta})}(O)).$$

Let $\mathbf{p}$ denote the probabilities of the atoms in σ^{nn}. Since $\mathbf{p}$ is indeed the outputs of the neural networks in $\Pi(\boldsymbol{\theta})$, we can compute the gradient of $\mathbf{p}$ w.r.t. $\boldsymbol{\theta}$ through backpropagation. Then the gradient of $\sum_{O \in \mathbf{O}} log(P_{\Pi(\boldsymbol{\theta})}(O))$ w.r.t. $\boldsymbol{\theta}$ is

$$\frac{\partial \sum_{O \in \mathbf{O}} log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial \boldsymbol{\theta}} = \sum_{O \in \mathbf{O}} \frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial \mathbf{p}} \times \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$$

where $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ can be computed through the usual neural network backpropagation, while $\frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial p}$ for each $p \in \mathbf{p}$ can be computed as follows.

Proposition 1 *Let $\Pi(\boldsymbol{\theta})$ be a NeurASP program and let O be an observation such that $P_{\Pi(\boldsymbol{\theta})}(O) > 0$. Let p denote the probability of an atom $c = v$ in σ^{nn}, i.e., p denotes $P_{\Pi(\boldsymbol{\theta})}(c = v)$. We have that*[4]

$$\frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial p} = \frac{\sum_{\substack{I:\ I \models O \\ I \models c=v}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v)} - \sum_{\substack{I, v':\ I \models O \\ I \models c=v', v \neq v'}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v')}}{\sum_{I:\ I \models O} P_{\Pi(\boldsymbol{\theta})}(I)}.$$

[4] $\frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v)}$ and $\frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v')}$ are still well-defined since the denominators have common factors in $P_{\Pi(\boldsymbol{\theta})}(I)$.

Intuitively, the proposition tells us that each interpretation I that satisfies O tends to increase the value of p if $I \models c = v$, and decrease the value of p if $I \models c = v'$ such that $v' \neq v$. `NeurASP` internally calls CLINGO to find all stable models I of $\Pi(\boldsymbol{\theta})$ that satisfy O and uses PyTorch to obtain the probability of each atom $c = v$ in σ^{nn}.

3.4.1 Learning Digit Classification from Addition

All experiments in Section 3.4 were done on Ubuntu 18.04.2 LTS with two 10-cores CPU Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and four GP104 [GeForce GTX 1080].

The digit addition problem is a simple example used in (Manhaeve *et al.*, 2018) to illustrate DeepProbLog's ability for both logical reasoning and deep learning. The task is, given a pair of digit images (MNIST) and their sum as the label, to let a neural network learn the digit classification of the input images.

The problem can be represented by `NeurASP` program Π_{digit} in Example 3. For comparison, we use the same dataset and the same structure of the neural network model used in (Manhaeve *et al.*, 2018) to train the digit classifier M_{digit} in Π_{digit}. For each pair of images denoted by d_1 and d_2 and their sum n, we construct the ASP constraint $\leftarrow$ *not* $addition(d_1, d_2, n)$ as the observation O. The training target is to maximize $log(P_{\Pi_{digit}}(O))$.

Figure 3.2 shows how the forward and the backward propagations are done for `NeurASP` program Π_{digit} in Example 3. The left-to-right direction is the forward computation of the neural network extended with the rule layer, whose output is the probability of the observation O. The right-to-left direction shows how the gradient from the rule layer is backpropagated further into the neural network by the chain rule to update all neural network parameters so as to find the parameter values that maximize the probability of the given observation.

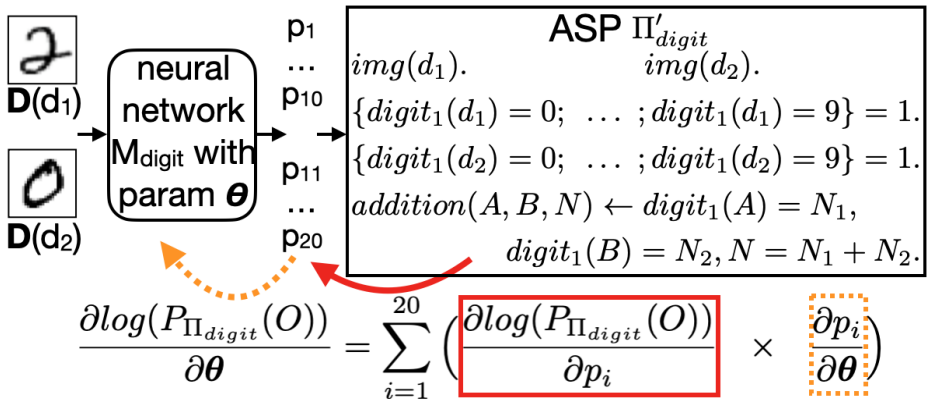

Figure 3.2: `NeurASP` Gradient Propagation

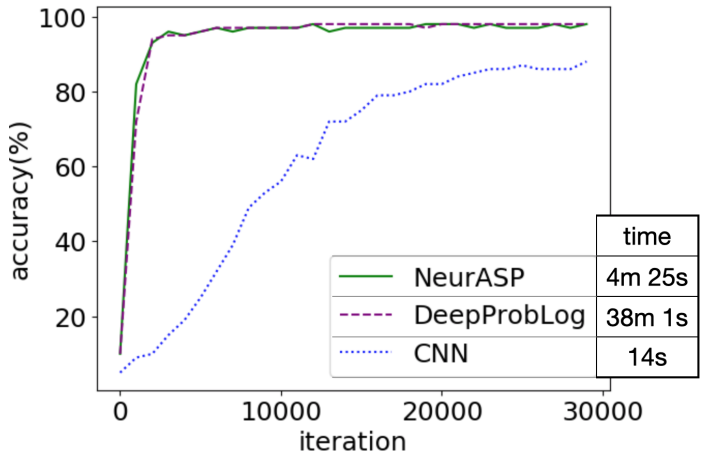

Figure 3.3: `NeurASP` v.s. DeepProbLog

Figure 3.3 shows the accuracy on the test data after each training iteration. The method CNN denotes the baseline used in (Manhaeve *et al.*, 2018) where a convolutional neural network (with more parameters) is trained to classify the concatenation of the two images into the 19 possible sums. As we can see, the neural networks trained by `NeurASP` and DeepProbLog converge much faster than CNN and have almost the same accuracy at each iteration. However, `NeurASP` spends much less time on training compared to DeepProbLog. The time reported is for one epoch (30,000 iterations in gradient descent). This is because DeepProbLog constructs an SDD (Sequential Decision Diagram) at each iteration for each training instance (i.e., each pair of images). This example illustrates that

generating many SDDs could be more time-consuming than enumerating stable models in NeurASP computation. In general, there is a trade-off between the two methods and other examples may show the opposite behavior.

3.4.2 Learning How to Solve Sudoku

In section 3.3.2, we used a neural network $M_{identify}$ to identify the numbers on a Sudoku board and used ASP rules to solve the Sudoku problem. In this section, we use a neural network to learn to solve Sudoku problems. The task is, given the *textual representation* of an unsolved Sudoku board (in the form of a 9×9 matrix where an empty cell is represented by 0), to let a neural network learn to predict the solution of the Sudoku board.

We use the neural network M_{sol} from (Park, 2018) as the baseline. M_{sol} is composed of 9 convolutional layers and a 1x1 convolution layer followed by softmax. Park (2018) trained M_{sol} using 1 million examples and achieved 70% accuracy using an "inference trick": instead of predicting digits for all empty cells at once, which leads to a poor accuracy, the most probable grid-cell value was predicted one by one.

Since the current NeurASP implementation is not as scalable as neural network training, training on 1 million examples takes too long. Thus, we construct a dataset of 63,000 + 1000 $\langle config, label \rangle$ pairs for training and testing. Using Park's method on this relatively small dataset, we observe that M_{sol}'s highest whole-board accuracy Acc_{sol}[5] is only 29.1% and M_{sol}'s highest grid-cell accuracy[6] is only 89.3% after 63 epochs of training.

We get a better result by training M_{sol} with the NeurASP program Π_{sol}. The program is almost the same as $\Pi_{identify}$ in Section 3.3.2 except that it uses M_{sol} in place of $M_{identify}$ and the first three rules of $\Pi_{identify}$ are replaced with

```
nn(sol(81, img), [1,2,3,4,5,6,7,8,9]).
```

[5]The percentage of Sudoku examples that are correctly solved.

[6]The percentage of grid cells having correct digits regardless whether the Sudoku solution is correct.

```
a(R,C,N) :- sol(Pos,img,N), R=Pos/9, C=Pos\9.
```

because we do not have to assign the value `empty` in solving Sudoku.

We trained M_{sol} using `NeurASP` where the training target is to maximize the probability of all stable models that satisfy the observation. On the same test data, after 63 epochs of training, the highest whole-board accuracy of M_{sol} trained this way is 66.5% and the highest grid-cell accuracy is 96.9% (In other words, we use rules only during training and not during testing). This indicates that including such structured knowledge sometimes helps the training of the neural network significantly.

3.4.3 Learning Shortest Path (SP)

The experiment is about, given a graph and two points, finding the shortest path between them. We use the dataset from (Xu *et al.*, 2018), which was used to demonstrate the effectiveness of semantic constraints for enhanced neural network learning. Each example is a 4 by 4 grid $G = (V, E)$, where $|V| = 16, |E| = 24$. The source and the destination nodes are randomly picked up, as well as 8 edges are randomly removed to increase the difficulty. The dataset is divided into 60/20/20 train/validation/test examples.

The following `NeurASP` program [7]

```
nn(sp(24, g), [true, false]).

sp(0,1) :- sp(1,g,true).

...

sp(X,Y) :- sp(Y,X).
```

together with the union of the following 4 constraints defines the shortest path.

```
% [nr] 1. No removed edges should be predicted
```

[7] $sp(X, g, true)$ means edge X is in the shortest path. $sp(X, Y)$ means there is a path between nodes X and Y in the shortest path.

```
:- sp(X,g,true), removed(X).


% [p] 2. Prediction must form a simple path, i.e., the degree of each node must
    be either 0 or 2
:- X=0..15, #count{Y: sp(X,Y)} = 1.
:- X=0..15, #count{Y: sp(X,Y)} >= 3.


% [r] 3. Every 2 nodes in the prediction must be reachable
reachable(X,Y) :- sp(X,Y).
reachable(X,Y) :- reachable(X,Z), sp(Z,Y).
:- sp(X,A), sp(Y,B), not reachable(X,Y).


% [o] 4. Predicted path should contain least edges
:~ sp(X,g,true). [1, X]
```

In this experiment, we trained the same neural network model M_{sp} as in (Xu *et al.*, 2018), a 5-layer Multi-Layer Perceptron (MLP), but with 4 different settings: (i) MLP only; (ii) together with `NeurASP` with the simple-path constraint **(p)** (which is the only constraint used in (Xu *et al.*, 2018)); [8] (iii) together with `NeurASP` with simple-path, reachability, and optimization constraints **(p-r-o)**; and (iv) together with `NeurASP` with all 4 constraints **(p-r-o-nr)**. [9]

Table 3.2 shows, after 500 epochs of training, the percentage of the predictions on the test data that satisfy each of the constraints **p**, **r**, and **nr**, the path constraint (i.e., **p-r**), the shortest path constraint (i.e., **p-r-o-nr**), and the accuracy w.r.t. the ground truth.

The accuracies for the first experiment (MLP Only) show that M_{sp} was not trained

[8] A path is *simple* if every node in the path other than the source and the destination has only 1 incoming edge and only 1 outgoing edge.

[9] Other combinations are either meaningless (e.g., **o**) or having similar results (e.g. **p-r** is similar to **p**).

well only by minimizing the cross-entropy loss of its prediction: 100-28.3 = 71.7% of the predictions are not even a simple-path.

In the remaining experiments (MLP (x)), instead of minimizing the cross-entropy loss, our training target is changed to maximizing the probability of all stable models under certain constraints. The accuracies under the 2nd and 3rd experiments (MLP (p) and MLP (p-r-o) columns) are increased significantly, showing that (i) including such structured knowledge helps the training of the neural network and (ii) the more structured knowledge included, the better M_{sp} is trained under `NeurASP`. Compared to the results from (Xu *et al.*, 2018), M_{sp} trained by `NeurASP` with the simple-path constraint **p** (in the 2nd experiment MLP (p) column) obtains a similar accuracy on predicting the label (28.9% v.s. 28.5%) but a higher accuracy on predicting a simple-path (96.6% v.s. 69.9%).

In the 4th experiment (MLP (p-r-o-nr) column) where we added the constraint **nr** saying that "no removed edges can be predicted", the accuracies go down. This is because the new constraint **nr** is about randomly removed edges, changing from one example to another, which is hard to be generalized.

Table 3.2: Shortest Path: Accuracy on Test Data: Columns Denote MLPs Trained with Different Rules; Each Row Represents the Percentage of Predictions that Satisfy the Constraints

Predictions satisfying	MLP Only	MLP (p)	MLP (p-r-o)	MLP (p-r-o-nr)
p	28.3%	96.6%	**100**%	30.1%
r	88.5%	**100**%	**100**%	87.3%
nr	32.9%	36.3%	45.7%	**70.5**%
p-r	28.3%	96.6%	**100**%	30.1%
p-r-o-nr	23.0%	33.2%	**45.7**%	24.2%
label (ground truth)	22.4%	28.9%	**40.1**%	22.7%

3.5 Extend NeurASP with Probabilistic Rules

Multi-valued probabilistic programs are a fragment of LP^{MLN} programs that distinguishes between probabilistic rules and regular rules. We first present the definition of Multi-Valued Probabilistic Programs (MVPP) from (Lee and Wang, 2016) with some modifications. Then we present the extended NeurASP with probabilistic rules, whose semantics is defined by a translation to MVPP. We assume that the reader is familiar with ASP-Core2 (Calimeri *et al.*, 2020).

3.5.1 Multi-Valued Probabilistic Programs

We assume a propositional signature σ that is constructed from "constants" and their "values." A *constant* c is associated with a finite set $\mathit{Dom}(c)$, called the *domain* of c. The signature σ is constructed from a finite set of constants, consisting of atoms $c{=}v$ for every constant c and every element v in $\mathit{Dom}(c)$. If the domain of c is $\{\text{FALSE}, \text{TRUE}\}$ then we say that c is *Boolean*, and abbreviate $c{=}\text{TRUE}$ as c and $c{=}\text{FALSE}$ as ${\sim}c$.[10] We assume that constants are divided into *probabilistic* constants and *non-probabilistic* constants. By σ_p, we denote the set of atoms in σ that are constructed from the probabilistic constants.

Syntax: A *probabilistic rule* is of the form

$$p_1 : c{=}v_1 \ \mid \ p_2 : c{=}v_2 \ \mid \ \ldots \ \mid \ p_n : c{=}v_n \tag{3.4}$$

where p_i are real numbers in $[0, 1]$ (denoting probabilities) such that $\sum_{i\in\{1,\ldots,n\}} p_i = 1$, and c is a probabilistic constant in σ, and $\{v_1, \ldots, v_n\} = \mathit{Dom}(c)$. If $\mathit{Dom}(c) = \{\text{FALSE}, \text{TRUE}\}$, rule $p_1 : c \ \mid \ p_2 : {\sim}c$ can be abbreviated as $p_1 : c$.

A *Multi-Valued Probabilistic Program* is the union of Π^{pr} and Π^{asp}, where Π^{pr} consists of probabilistic rules (3.4), one for each probabilistic constant c in σ_p, and Π^{asp} consists of rules of the form $\mathit{Head} \leftarrow \mathit{Body}$, following the rule format of ASP-Core2 where *Head*

[10] The use of symbol $\sim$ is intentional; the semantics of $c{=}\text{FALSE}$ works the same as strong negation.

contains no probabilistic constants.

Example 4 *Consider the game of flipping a coin where we win if we got* $head$*. Suppose the coin is biased and the probability of getting* $head$ *is 0.1, then this problem can be represented by the following MVPP program* Π_{coin} *where* $head$ *is a probabilistic constant and* win *is a non-probabilistic constant.*

```
0.1: head.
win :- head.
~win :- not win.
```

Semantics: Given an MVPP program Π, we obtain an ASP program Π' from Π by replacing each rule (3.4) with

$$1\{c{=}v_1; \ldots; c{=}v_n\}1$$

which means to choose only one atom from the set $\{c = v_1, \ldots, c = v_n\}$. In addition, Π' contains the rule

$$\leftarrow 2\{c = v_1; \ldots; c = v_n\}.$$

for each non-probabilistic constant c with $\mathit{Dom}(c) = \{v_1, \ldots, v_n\}$. That is, non-probabilistic constants are allowed to have no values.

The stable models of Π are defined as the stable models of Π'.

To define the probability of a stable model, we first define the probability of an atom $c{=}v$ in σ_p. We know there must be exactly one probabilistic rule (3.4) for each probabilistic constant c. Thus we can always find such a rule (3.4) for any atom $c = v$ in σ_p, and the probability of $c{=}v_i$, denoted by $P_{\Pi}(c{=}v_i)$, is defined as p_i in rule (3.4).

The probability of a stable model I of Π, denoted by $P_{\Pi}(I)$, is defined as the product of the probability of each atom $c = v$ in $I \cap \sigma_p$, divided by the number of stable models satisfied by $I \cap \sigma_p$. In the following equation, we use $I|_{\sigma_p}$ to denote $I \cap \sigma_p$, which is indeed

the projection of I onto σ_p. We also use $\mathit{Num}(I|_{\sigma_p}, \Pi)$ to denote the number of stable models of Π that satisfy $I|_{\sigma_p}$.

$$P_\Pi(I) = \begin{cases} \frac{\prod_{c=v \in I|_{\sigma_p}} P_\Pi(c=v)}{\mathit{Num}(I|_{\sigma_p}, \Pi)} & \text{if } I \text{ is a stable model of } \Pi; \\ 0 & \text{otherwise.} \end{cases}$$

An *observation* is a set of ASP constraints (i.e., rules of the form $\bot \leftarrow \mathit{Body}$). The probability of an observation O is defined as

$$P_\Pi(O) = \sum_{I \models O} P_\Pi(I).$$

The probability of the set $\mathbf{O} = \{O_1, \ldots, O_o\}$ of independent observations, where each O_i is a set of ASP constraints, is defined as the product of the probability of each O_i:

$$P_\Pi(\mathbf{O}) = \prod_{O_i \in \mathbf{O}} P_\Pi(O_i).$$

Example 4 Continued: The following ASP program is Π'_{coin} (the ASP counter-part of Π_{coin}).

```
win=true :- head=true.
win=false :- not win=true.


1{head=true; head=false}1.
:- 2{win=true; win=false}.
```

It has 2 stable models: $I_1 = \{head = \text{TRUE}, win = \text{TRUE}\}$ and $I_2 = \{head = \text{FALSE}, win = \text{FALSE}\}$, which are the stable models of Π_{coin}. There are 2 atoms in σ_p and their probabilities are $P_{\Pi_{coin}}(head = \text{TRUE}) = 0.1$ and $P_{\Pi_{coin}}(head = \text{FALSE}) = 0.9$. Then the probabilities of I_1 and I_2 can be computed as follows.

$$P_{\Pi_{coin}}(I_1) = \frac{0.1}{1} = 0.1 \qquad P_{\Pi_{coin}}(I_2) = \frac{0.9}{1} = 0.9$$

And the probability of $O = \{\mathit{not}\ win = \text{TRUE}\}$ is $P_{\Pi_{coin}}(O)) = P_{\Pi_{coin}}(I_2) = 0.9$.

3.5.2 Define NeurASP on a Translation to MVPP

Syntax: We first define the notion of a neural atom to describe a neural network in a logic program. Intuitively, a neural atom can be seen as the shorthand for a sequence of probabilistic rules whose atoms are defined by a syntactical translation from the neural atom and whose probabilities are the outputs of neural networks.

We assume that neural network M allows an arbitrary tensor as input whereas the output is a matrix in $\mathbb{R}^{e\times n}$, where e is the number of random events predicted by the neural network, and n is the number of possible outcomes for each random event. Each row of the matrix represents the probability distribution of the outcomes of each event. Given an input tensor $\mathbf{t}$, by $M(\mathbf{t})$, we denote the output matrix of M. $M(\mathbf{t})[i,j]$ ($i \in \{1,\ldots,e\}$, $j \in \{1,\ldots,n\}$) is the probability at the i-th row and j-th column of the matrix. For example, in a neural network for MNIST digit classification, the input is a tensor representation of a digit image, e is 1, and n is 10. For a neural network that outputs a Boolean value for each edge in a graph, e is the number of edges and n is 2.

In NeurASP, the neural network M above can be represented by a *neural atom* of the form

$$nn(m(e,t),[v_1,\ldots,v_n]), \tag{3.5}$$

where (i) nn is a reserved keyword to denote a neural atom; (ii) m is an identifier (symbolic name) of the neural network M; (iii) t is a list of terms that serves as a "pointer" to an input data; the mapping $\mathbf{D}$ is implemented by the external Python code that accepts NeurASP program as input, and can map t to different data instances by iteratively loading from the dataset; this is useful for training; (iv) $v_1,\ldots,v_n$ represent all n possible outcomes of each of the e random events.

Each neural atom (3.5) introduces propositional atoms of the form $c=v$, where $c \in \{m_1(t),\ldots,m_e(t)\}$ and $v \in \{v_1,\ldots,v_n\}$. The output of the neural network provides the

probabilities of the introduced atoms (defined in Section 3.2).

Example 5 *Let M_{digit} be a neural network that classifies an MNIST digit image. The input of M_{digit} is (a tensor representation of) an image and the output is a matrix in $\mathbb{R}^{1\times 10}$. The neural network can be represented as the neural atom*

$$nn(digit(1,d),\ [0,1,2,3,4,5,6,7,8,9])$$

which introduces propositional atoms $digit_1(d)=0$, $digit_1(d)=1$, ..., $digit_1(d)=9$.

Let M_{sp} be another neural network for finding the shortest path in a graph with 24 edges. The input is a tensor encoding the graph and the start/end nodes of the path, and the output is a matrix in $\mathbb{R}^{24\times 2}$. This neural network can be represented as the neural atom

$$nn(sp(24,g),\ [\text{TRUE}, \text{FALSE}]).$$

A `NeurASP` *program* Π is the union of Π^{mvpp} and Π^{nn} where Π^{mvpp} is an MVPP program, and Π^{nn} is a set of neural atoms. Let σ^{nn} be the set of all atoms $c=v$ that is obtained from the neural atoms in Π^{nn} as described above. We require that no atoms in σ^{nn} appear in the probabilistic rules in Π^{mvpp} or in *Head* of each ASP rule *Head* $\leftarrow$ *Body* in Π^{mvpp}.

We could allow schematic variables into Π, which are understood in terms of grounding as in standard answer set programs. We find it convenient to use rules of the form

$$nn(m(e,t),[v_1,\ldots,v_n]) \leftarrow \textit{Body} \tag{3.6}$$

where *Body* is either identified by $\top$ or $\bot$ during grounding so that (3.6) can be viewed as an abbreviation of multiple (variable-free) neural atoms (3.5).

Example 6 *An example* `NeurASP` *program Π_{digit} is as follows, where d_1 and d_2 are terms representing two images. Each image is classified by neural network M_{digit} as one of the*

values in $\{0, \ldots, 9\}$. *The addition of two digit-images is the sum of their values.*

$$img(d_1).$$
$$img(d_2).$$
$$nn(digit(1, X), [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) \leftarrow img(X).$$
$$addition(A, B, N) \leftarrow digit_1(A) = N_1, digit_1(B) = N_2, N = N_1 + N_2.$$

The neural network M_{digit} *generates 10 probabilities for each image. The addition is applied once the digits are recognized and its probability is induced from the perception as we explain in the next section.*

Semantics: For any `NeurASP` program $\Pi = \Pi^{mvpp} \cup \Pi^{nn}$, we obtain its MVPP counterpart Π' by replacing each neural atom $nn(m(e,t), [v_1, \ldots, v_n])$ in Π^{nn} with the set of probabilistic rules

$$p_{1,1} : m_1(t) = v_1 \mid \ldots \mid p_{1,n} : m_1(t) = v_n$$
$$\ldots$$
$$p_{e,1} : m_e(t) = v_1 \mid \ldots \mid p_{e,n} : m_e(t) = v_n$$

where $p_{i,j}$ denotes the probability of atom $m_i(t) = v_j$. Recall that there is an external mapping $\mathbf{D}$ that turns t into a specific input tensor of M, the value of $p_{i,j}$ is the neural network output $M(\mathbf{D}(t))[i, j]$.

The stable models of a `NeurASP` program Π are defined as the stable models of its MVPP counterpart Π'. The probability of each stable model I under Π is defined as its probability under Π'.

Example 6 Continued: The following MVPP program is Π'_{digit}, i.e., the MVPP counter-

part of Π_{digit}.

$$
\begin{aligned}
&img(d_1).\\
&img(d_2).\\
&addition(A,B,N) \leftarrow digit_1(A){=}N_1, digit_1(B){=}N_2, N = N_1 + N_2.\\
&p^{d_1}_{1,1} : digit_1(d_1){=}0 \mid \ \ldots \ \mid p^{d_1}_{1,10} : digit_1(d_1){=}9\\
&p^{d_2}_{1,1} : digit_1(d_2){=}0 \mid \ \ldots \ \mid p^{d_2}_{1,10} : digit_1(d_2){=}9
\end{aligned}
$$

Recall that $M_{digit}(\mathbf{D}(d)) \in \mathbb{R}^{1\times 10}$ is the output matrix of M_{digit} in Example 6 with input $\mathbf{D}(d)$. For $d \in \{d_1, d_2\}$ and $j \in \{0, \ldots, 9\}$, the value of $p^d_{1,j}$ is the neural network output $M_{digit}(\mathbf{D}(d))[1, j]$.

3.6 Detailed Description of Learning Algorithms for `NeurASP`

Consider a `NeurASP` program $\Pi(\boldsymbol{\theta})$ and a set $\mathbf{O}$ of observations such that $P_{\Pi(\boldsymbol{\theta})}(O) > 0$ for each $O \in \mathbf{O}$. The task is to find $\hat{\boldsymbol{\theta}}$ that maximizes the log-likelihood of observations $\mathbf{O}$ under program $\Pi(\boldsymbol{\theta})$, i.e.,

$$\hat{\boldsymbol{\theta}} \in \operatorname*{argmax}_{\boldsymbol{\theta}} \ log(P_{\Pi(\boldsymbol{\theta})}(\mathbf{O})),$$

which is equivalent to

$$\hat{\boldsymbol{\theta}} \in \operatorname*{argmax}_{\boldsymbol{\theta}} \ \sum_{O\in\mathbf{O}} log(P_{\Pi(\boldsymbol{\theta})}(O)).$$

Let $\mathbf{p}$ denote the probabilities of the atoms in σ^{nn}. Since $\mathbf{p}$ is indeed the outputs of the neural networks in $\Pi(\boldsymbol{\theta})$, we can compute the gradient of $\mathbf{p}$ w.r.t. $\boldsymbol{\theta}$ through back-propagation. Then the gradients of $\sum_{O\in\mathbf{O}} log(P_{\Pi(\boldsymbol{\theta})}(O))$ w.r.t. $\boldsymbol{\theta}$ is

$$\frac{\partial \sum_{O\in\mathbf{O}} log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial \boldsymbol{\theta}} = \sum_{O\in\mathbf{O}} \frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial \mathbf{p}} \times \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$$

where $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ can be computed through the usual neural network back-propagation, while $\frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial p}$ for each $p \in \mathbf{p}$ can be computed with Proposition 1.

Algorithm 1 shows how to update the value of $\boldsymbol{\theta}$ to maximize the log-likelihood of observations $\mathbf{O}$ under $\Pi(\boldsymbol{\theta})$.

Algorithm 1 `NeurASP` weight learning by exact computation

Input:

1. $\Pi(\boldsymbol{\theta})$: a `NeurASP` program (under signature σ) with parameters $\boldsymbol{\theta}$
2. $\mathbf{O}$: a set of observations O_i such that $P_{\Pi(\boldsymbol{\theta})}(O_i) > 0$
3. $\mathbf{D}$: a set of mappings $\{\mathbf{D_1}, \ldots, \mathbf{D_n}\}$ where each $\mathbf{D_i}$ is associated with O_i and maps terms to input tensors of neural networks in $\Pi(\boldsymbol{\theta})$
4. lr: a real number denoting learning rate
5. epoch: a positive integer denoting the number of epochs

Output: $\hat{\boldsymbol{\theta}}$: the updated parameters such that $\hat{\boldsymbol{\theta}} \in \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\ log(P_{\Pi(\boldsymbol{\theta})}(\mathbf{O}))$

Procedure:

1. Repeat for epoch number of times:
   (a) For O_i in $\mathbf{O}$:
      i. Compute the outputs $\mathbf{p}$ of the neural networks in $\Pi(\boldsymbol{\theta})$ according to $\mathbf{D_i}$
      ii. Compute $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ by back-propagation
      iii. Find all stable models $\mathbf{I}_{O_i}$ of $\Pi(\boldsymbol{\theta})$ that satisfy O_i
      iv. gradients = $gradientsSM(\Pi(\boldsymbol{\theta}), \mathbf{p}, \mathbf{I}_{O_i})$
      v. $\boldsymbol{\theta} = \boldsymbol{\theta}$ + lr * gradients * $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$
2. return $\boldsymbol{\theta}$

Algorithm 1 uses the function $gradientsSM$, which is defined in Algorithm 2 below.

Algorithm 2 $gradientsSM$: compute gradients of the probability of a set of stable models

Input:

1. $\Pi(\boldsymbol{\theta})$: a `NeurASP` program (under signature σ) with parameters $\boldsymbol{\theta}$
2. $\mathbf{p}$: the probabilities of the atoms in σ^{nn}
3. $\mathbf{I}$: a set of stable models of $\Pi(\boldsymbol{\theta})$, whose probabilities are to be maximized

Output: $\frac{\partial log(\sum_{I_i \in \mathbf{I}} P_{\Pi(\boldsymbol{\theta})}(I_i))}{\partial \mathbf{p}}$: the gradients of the log-likelihood of $\mathbf{I}$ w.r.t. $\mathbf{p}$

Procedure:

1. If $|\mathbf{I}| = 1$, for each $p \in \mathbf{p}$, where p denotes the probability of one atom $c = v \in \sigma^{nn}$:
   - if I contains $c = v$, gradient(p) = $\frac{1}{p}$
   - else, I must contain $c = v'$ for some $v' \neq v$, and gradient(p) = -$\frac{1}{P_{\Pi(\boldsymbol{\theta})}(c=v')}$
2. If $|\mathbf{I}| \geq 2$:
   (a) compute $P_{\Pi(\boldsymbol{\theta})}(I)$ for each I in $\mathbf{I}$; denominator = $\sum_{I \in \mathbf{I}} P_{\Pi(\boldsymbol{\theta})}(I)$
   (b) for each $p \in \mathbf{p}$, where p denotes the probability of one atom $c = v \in \sigma^{nn}$:
      i. numerator = 0; for each $I \in \mathbf{I}$:
         - if I contains $c = v$, numerator += $\frac{1}{p} P_{\Pi(\boldsymbol{\theta})}(I)$
         - else, I must contain $c = v'$ for some $v' \neq v$, and numerator -= $\frac{1}{P_{\Pi(\boldsymbol{\theta})}(c=v')} P_{\Pi(\boldsymbol{\theta})}(I)$
      ii. gradient(p) = $\frac{\text{numerator}}{\text{denominator}}$
3. return [gradient(p) for p in $\mathbf{p}$]

Algorithm 3 is almost the same as Algorithm 1 except that, in step 1-(a)-iii, instead of finding all stable models that satisfy O_i, it randomly samples num_of_samples stable

Algorithm 3 `NeurASP` weight learning by sampling stable models

Input:

1. $\Pi(\boldsymbol{\theta})$: a `NeurASP` program (under signature σ) with parameters $\boldsymbol{\theta}$
2. $\mathbf{O}$: a set of observations O_i such that $P_{\Pi(\boldsymbol{\theta})}(O) > 0$
3. $\mathbf{D}$: a set of mappings $\{\mathbf{D_1}, \ldots, \mathbf{D_n}\}$ where each $\mathbf{D_i}$ is associated with O_i and maps terms to input tensors of neural networks in $\Pi(\boldsymbol{\theta})$
4. num_of_samples: the number of stable models sampled for each O_i in each iteration
5. lr: a real number denoting learning rate
6. epoch: a positive integer denoting the number of epochs

Output: $\hat{\boldsymbol{\theta}}$: the updated parameters such that $\hat{\boldsymbol{\theta}} \in \underset{\boldsymbol{\theta}}{\operatorname{argmax}}\ log(P_{\Pi(\boldsymbol{\theta})}(\mathbf{O}))$

Procedure:

1. Repeat for epoch number of times:
   (a) For O_i in $\mathbf{O}$:
      i. Compute the outputs $\mathbf{p}$ of the neural networks in $\Pi(\boldsymbol{\theta})$ according to $\mathbf{D_i}$
      ii. Compute $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ by back-propagation
      iii. Sample num_of_samples stable models of $\Pi(\boldsymbol{\theta})$ that satisfy O_i according to their probability distribution:

$$\mathbf{I} = sampleSM(\Pi(\boldsymbol{\theta}), O_i, \text{num\_of\_samples})$$

      iv. gradients = $gradientsSM(\Pi(\boldsymbol{\theta}), \mathbf{p}, \mathbf{I})$
      v. $\boldsymbol{\theta}$ = $\boldsymbol{\theta}$ + lr * gradients * $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$
2. return $\boldsymbol{\theta}$

models that satisfy O_i according to their probability distribution. The function $sampleSM$ used in Algorithm 3 to sample stable models is defined in Algorithm 4.

Algorithm 4 $sampleSM$: sample num_of_samples stable models that satisfy O

Input:

1. $\Pi(\boldsymbol{\theta})$: a `NeurASP` program (under signature σ) with parameters $\boldsymbol{\theta}$
2. O: an observation in the form of a set of ASP constraints
3. num_of_samples: the number of sample stable models generated for O

Output: **I**: a list of stable models of $\Pi(\boldsymbol{\theta}) \cup O$ such that $|\mathbf{I}| \geq$ num_of_samples and the probability distribution of **I** follows the distribution defined by $\Pi(\boldsymbol{\theta})$

Procedure:

1. SM = []; obtain Π' from $\Pi(\boldsymbol{\theta})$ according to the semantics
2. while True: do
   (a) obtain an ASP program P from Π' by randomly replacing each choice rule $\{c = v_1 \ ; \ \ldots \ ; \ c = v_n\} = 1$ in Π' with a fact "$c = v_i$." according to the probability distribution $\langle P_{\Pi(\boldsymbol{\theta})}(c = v_1), \ldots, P_{\Pi(\boldsymbol{\theta})}(c = v_n)\rangle$;
   (b) generate the set S of all stable models of $P \cup O$ (by calling CLINGO on $P \cup O$);
   (c) append each element in S to SM;
   (d) if $|SM| \geq$ num_of_samples: break the loop;
3. return SM

3.7 Proof of Proposition 1

Proposition 1 *Let $\Pi(\boldsymbol{\theta})$ be a* `NeurASP` *program and let O be an observation such that $P_{\Pi(\boldsymbol{\theta})}(O) > 0$. Let p denote the probability of an atom $c = v$ in σ^{nn}, i.e., p denotes $P_{\Pi(\boldsymbol{\theta})}(c = v)$. We have that*

$$\frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial p} = \frac{\sum\limits_{\substack{I:\ I\models O \\ I\models c=v}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v)} - \sum\limits_{\substack{I,v':\ I\models O \\ I\models c=v', v\neq v'}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v')}}{\sum\limits_{I:\ I\models O} P_{\Pi(\boldsymbol{\theta})}(I)}.$$

Proof.

$$\frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial p_i}$$

$$= \frac{\partial log(P_{\Pi(\boldsymbol{\theta})}(O))}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)}$$

$$= \frac{1}{P_{\Pi(\boldsymbol{\theta})}(O)} \times \frac{\partial P_{\Pi(\boldsymbol{\theta})}(O)}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)}$$

(since $P_{\Pi(\boldsymbol{\theta})}(O) = \sum\limits_{I\models O} P_{\Pi(\boldsymbol{\theta})}(I)$)

$$= \frac{1}{\sum\limits_{I\models O} P_{\Pi(\boldsymbol{\theta})}(I)} \times \frac{\partial \sum\limits_{I\models O} P_{\Pi(\boldsymbol{\theta})}(I)}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)}$$

(since (i) $P_{\Pi(\boldsymbol{\theta})}(I) = 0$ if I is not a stable model of $\Pi(\boldsymbol{\theta})$ and

(ii) any stable model I of $\Pi(\boldsymbol{\theta})$ must satisfy $c = v^*$ for some v^*)

$$= \frac{1}{\sum\limits_{I\models O} P_{\Pi(\boldsymbol{\theta})}(I)} \times \Big(\frac{\partial \sum\limits_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I\models O \\ I\models c=v}} P_{\Pi(\boldsymbol{\theta})}(I)}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)} + \frac{\partial \sum\limits_{\substack{I,v' \mid I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I\models O \\ I\models c=v', v\neq v'}} P_{\Pi(\boldsymbol{\theta})}(I)}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)} \Big)$$

(since for any stable model I of $\Pi(\boldsymbol{\theta})$, $P_{\Pi(\boldsymbol{\theta})}(I) = \frac{\prod\limits_{c^*=v^*\in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}$)

$$= \frac{1}{\sum\limits_{I \models O} P_{\Pi(\boldsymbol{\theta})}(I)} \times \Bigg(\frac{\partial \sum\limits_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v}} \frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)} + \frac{\partial \sum\limits_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v', v \neq v'}} \frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)} \Bigg)$$

$$= \frac{1}{\sum\limits_{I \models O} P_{\Pi(\boldsymbol{\theta})}(I)} \times \Bigg(\sum_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v}} \frac{\partial \frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)} + \sum_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v', v \neq v'}} \frac{\partial \frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)} \Bigg)$$

$$= \frac{1}{\sum\limits_{I \models O} P_{\Pi(\boldsymbol{\theta})}(I)} \times \Bigg(\sum_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v}} \frac{\frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{P_{\Pi(\boldsymbol{\theta})}(c=v)} + \sum_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v', v \neq v'}} \frac{\partial \frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v')} \times \frac{\partial P_{\Pi(\boldsymbol{\theta})}(c=v')}{\partial P_{\Pi(\boldsymbol{\theta})}(c=v)} \Bigg)$$

(since $P_{\Pi(\boldsymbol{\theta})}(c=v') = 1 - P_{\Pi(\boldsymbol{\theta})}(c=v) - \dots$)

$$= \frac{1}{\sum\limits_{I \models O} P_{\Pi(\boldsymbol{\theta})}(I)} \times \Bigg(\sum_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v}} \frac{\frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{P_{\Pi(\boldsymbol{\theta})}(c=v)} + \sum_{\substack{I \text{ is a stable model of } \Pi(\boldsymbol{\theta}) \\ I \models O \\ I \vDash c=v', v \neq v'}} \frac{\frac{\prod\limits_{c^*=v^* \in I|_{\sigma_m}} P_{\Pi(\boldsymbol{\theta})}(c^*=v^*)}{Num(I|_{\sigma_m}, \Pi(\boldsymbol{\theta}))}}{P_{\Pi(\boldsymbol{\theta})}(c=v')} \times -1 \Bigg)$$

$$= \frac{1}{\sum\limits_{I \models O} P_{\Pi(\boldsymbol{\theta})}(I)} \times \Big(\sum_{\substack{I \models O \\ I \models c = v}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c = v)} - \sum_{\substack{I \models O \\ I \models c = v', v \neq v'}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c = v')} \Big)$$

∎

Chapter 4

CL-STE

`CL-STE` is a general method to encode logical constraints in propositional logic as a Constraint Loss function so that minimizing its value via a Straight-Through-Estimator (STE) makes neural network prediction follow the logical constraints.

Compared to `NeurASP`, training a NN with `CL-STE` costs significantly less time since `CL-STE` computes a gradient for each clause independently instead of computing a gradient for the whole logic program with a combinatorial computation, allowing for more efficient and parallel computation with batch training and GPUs. On the other hand, `CL-STE` requires hyper-parameters to balance the effects of different loss functions and hyper-parameter tuning is often needed to achieve the best performance.

In this chapter, we first discuss STE and its relation to Trainable Gate Function, then propose the CL-STE method, followed by four learning tasks for `CL-STE` to demonstrate its domain of usage and its pros and cons in terms of ease of representation, accuracy, and time efficiency.

4.1 STE and Trainable Gate Function

Review. STEs are used to estimate the gradients of a discrete function. Courbariaux *et al.* (2015) consider a binarization function b that transforms real-valued weights x into discrete values $b(x)$ as $b(x) = 1$ if $x \geq 0$ and $b(x) = 0$ otherwise. A loss function L is defined on binarized weights $b(x)$, but the gradient descent won't update binarized weights in small increments. However, using STE, we could update the real-valued weights x that are input to $b(x)$. In the end, a quantized model consists of binarized weights $b(x)$ only. More specifically, according to the chain rule, the gradient of loss L w.r.t. x is $\frac{\partial L}{\partial x} =$

$\frac{\partial L}{\partial b(x)} \times \frac{\partial b(x)}{\partial x}$, where $\frac{\partial b(x)}{\partial x}$ is zero almost everywhere. The idea is to replace $\frac{\partial b(x)}{\partial x}$ with an STE $\frac{\partial s(x)}{\partial x}$ for some (sub)differentiable function $s(x)$. The STE $\frac{\partial s(x)}{\partial x}$ is called the *identity STE* (iSTE) if $s(x) = x$ and is called the *saturated STE* (sSTE) if $s(x) = clip(x, [-1, 1]) = min(max(x, -1), 1)$. Since $\frac{\partial s(x)}{\partial x} = 1$, by $\frac{\partial L}{\partial x} \overset{iSTE}{\approx} \frac{\partial L}{\partial b(x)}$, we denote the identification of $\frac{\partial L}{\partial x}$ with $\frac{\partial L}{\partial b(x)}$ under iSTE.

Directly applying STE doesn't work well since the binarization function $b(x)$ passes only the sign of x while information about the magnitude of x is lost (Simons and Lee, 2019). In XNOR-Net (Rastegari *et al.*, 2016), the input x is normalized to have the zero mean and a small variance before the binarization to reduce the information loss. In this work, we normalize x by turning it into a probability using softmax or sigmoid activation functions. Indeed, several neuro-symbolic learning methods (e.g., `DeepProbLog`, `NeurASP`, `NeuroLog`) assume the neural network outputs that are fed into the logic layer are normalized as probabilities. To address a probabilistic input, we introduce a variant binarization function $b_p(x)$ for probabilities $x \in [0, 1]$: $b_p(x) = 1$ if $x \geq 0.5$ and $b_p(x) = 0$ otherwise. It is easy to see that iSTE and sSTE work the same with $b_p(x)$ since $x = clip(x, [-1, 1])$ when $x \in [0, 1]$. A vector $\mathbf{x}$ is allowed as input to the binarization functions b and b_p, in which case they are applied to each element of $\mathbf{x}$.

TGF and Its Relation to STE. The concept of STE is closely related to that of the Trainable Gate Function (TGF) from (Kim *et al.*, 2020), which was applied to channel pruning. Instead of replacing the gradient $\frac{\partial b(x)}{\partial x}$ with an STE, TGF tweaks the binarization function $b(x)$ to make it meaningfully differentiable. More specifically, a differentiable binarization function $\widetilde{b}^K$ is defined as

$$\widetilde{b}^K(x) = b(x) + s^K(x)g(x), \tag{4.1}$$

where K is a large constant; $s^K(x) = \frac{Kx - \lfloor Kx \rfloor}{K}$ is called a *gradient tweaking* function, whose value is less than $\frac{1}{K}$ and whose gradient is always 1 wherever differentiable; $g(x)$

Figure 4.1: Trainable Gate Function $\widetilde{b}^K(x)$ When $g(x) = 1$

is called a *gradient shaping* function, which could be an arbitrary function, but the authors note that the selection does not affect the results critically and $g(x) = 1$ can be adopted without significant loss of accuracy. As obvious from Figure 4.1, as K becomes large, TGF $\widetilde{b}^K(x)$ is an approximation of $b(x)$, but its gradient is 1 wherever differentiable.

Proposition 2 tells us a precise relationship between TGF and STE: when K is big enough, the binarization function $b(x)$ with iSTE or sSTE can be simulated by TGF. In other words, Proposition 2 allows us to visualize $b(x)$ with STE as the TGF $\widetilde{b}^K(x)$ with $K = \infty$ as Figure 4.1 illustrates.

Proposition 2 *When K approaches ∞ and $|g(x)| \leq c$ for some constant c, the value of $\widetilde{b}^K(x)$ converges to $b(x)$:*

$$\lim_{K \to \infty} \widetilde{b}^K(x) = b(x).$$

The gradient $\frac{\partial \widetilde{b}^K(x)}{\partial x}$, wherever defined, is exactly the iSTE of $\frac{\partial b(x)}{\partial x}$ if $g(x) = 1$, or the sSTE of $\frac{\partial b(x)}{\partial x}$ if

$$g(x) = \begin{cases} 1 & \text{if } -1 \leq x \leq 1, \\ 0 & \text{otherwise.} \end{cases}$$

Proposition 2 still holds if we replace $b(x)$ with $b_p(x)$.

The proposition yields insights into STE and TGF in terms of each other. As shown in Figure 4.1, TGF is a sawtooth function that approximates a step function as K becomes large. At large, TGF works like a discrete function, but it is differentiable almost everywhere. In view of Proposition 2, this fact gives an idea why the STE method works in

practice. On the other hand, the proposition tells that the implementation of TGF can be replaced with STE. That could be better because TGF in equation (4.1) requires that K approximate infinity and be non-differentiable when x is a multiple of $\frac{1}{K}$ whereas STE is differentiable at every x.

4.2 Enforcing Logical Constraints using STE

This section presents our method of encoding logical constraints in propositional logic as a loss function so that minimizing its value via STE makes neural network prediction follow the logical constraints.

4.2.1 Encoding CNF as a Loss Function Using STE

We first review the terminology in propositional logic. A *signature* is a set of symbols called *atoms*. Each atom represents a proposition that is true or false. A *literal* is either an atom p (*positive literal*) or its negation $\neg p$ (*negative literal*). A *clause* is a disjunction over literals, e.g., $p_1 \vee \neg p_2 \vee p_3$. A *Horn clause* is a clause with at most one positive literal. We assume a *(propositional) theory* consisting of a set of clauses (sometimes called a *CNF (Conjunctive Normal Form) theory*). A truth assignment to atoms *satisfies* (denoted by $\models$) a theory if at least one literal in each clause is true under the assignment. A theory is *satisfiable* if at least one truth assignment satisfies the theory. A theory *entails* (also denoted by $\models$) a literal if every truth assignment that satisfies the theory also satisfies that literal.

We define a general loss function L_{cnf} for any CNF theory as follows. Here, bold upper and lower letters (e.g., $\mathbf{C}$ and $\mathbf{v}$) denote matrices and vectors, respectively; $\mathbf{C}[i, j]$ and $\mathbf{v}[i]$ denote their elements.

Consider a propositional signature $\sigma = \{p_1, \ldots, p_n\}$. Given (i) a theory C consisting of m clauses (encoding domain knowledge), (ii) a set F of atoms denoting some atomic

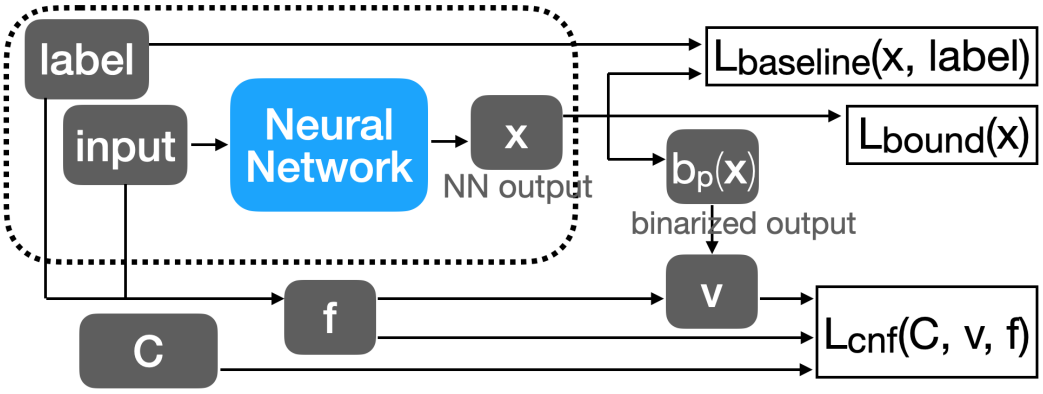

Figure 4.2: Architecture That Overlays Constraint Loss

facts that we assume known to be true (representing the ground-truth label of a data instance), and (iii) a truth assignment v such that $v \models F$, we construct their matrix/vector representations as

- the matrix $\mathbf{C} \in \{-1, 0, 1\}^{m \times n}$ to represent the theory such that $\mathbf{C}[i, j]$ is 1 (-1, resp.) if p_j ($\neg p_j$, resp.) belongs to the i-th clause in the theory, and is 0 if neither p_j nor $\neg p_j$ belongs to the clause;
- the vector $\mathbf{f} \in \{0, 1\}^n$ to represent F such that $\mathbf{f}[j]$ is 1 if $p_j \in F$ and is 0 otherwise; and
- the vector $\mathbf{v} \in \{0, 1\}^n$ to represent v such that $\mathbf{v}[j]$ is 1 if $v(p_j) = \text{TRUE}$, and is 0 if $v(p_j) = \text{FALSE}$.

Figure 4.2 shows an architecture that overlays the two loss functions L_{bound} and L_{cnf} over the neural network output, where L_{cnf} is the main loss function to encode logical constraints and L_{bound} is a regularizer to limit the raw neural network output not to grow too big (more details will follow). The part $\mathbf{input}$ is a tensor (e.g., images) for a data instance; $\mathbf{label}$ denotes the labels of input data; $\mathbf{C}$ encodes the domain knowledge, $\mathbf{x} \in [0, 1]^n$ denotes the NN output (in probability), and $\mathbf{f} \in \{0, 1\}^n$ records the known facts in that

data instance (e.g., given digits in Sudoku).[1] Let $\mathbb{1}_{\{k\}}(X)$ denote an indicator function that replaces every element in X with 1 if it is k and with 0 otherwise. Then the binary prediction $\mathbf{v}$ is constructed as $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$, where $\odot$ denotes element-wise multiplication. Intuitively, $\mathbf{v}$ is the binarized NN output with part of it strictly following the given facts specified in $\mathbf{f}$ (ensuring $v \models F$).

Example 7 *Consider a simple example* **mnistAdd** *from (Manhaeve* et al.*, 2018), where the task is, given a pair of MNIST digit images and their sum as the label, to let a neural network learn the digit classification of the input images. The example is used to demonstrate how NNs can learn from known constraints. In Figure 4.2, the input consists of two-digit images i_1 and i_2, and the label is an integer l in $\{0, ..., 18\}$ denoting the sum of i_1 and i_2. The neural network is the same Convolutional Neural Network (CNN) used in (Manhaeve* et al.*, 2018).*

The theory for this problem consists of the following clause for $l \in \{0, \ldots, 18\}$, where $sum(l)$ represents "the sum of i_1 and i_2 is l" and $pred(n_1, n_2)$ represents "the neural network predicts i_1 and i_2 as n_1 and n_2 respectively":

$$\neg sum(l) \vee \bigvee_{\substack{n_1, n_2 \in \{0,\ldots,9\}: \\ n_1 + n_2 = l}} pred(n_1, n_2).$$

This theory contains $19 + 100 = 119$ atoms for $sum/1$ and $pred/2$ respectively. We construct the matrix $\mathbf{C} \in \{-1, 0, 1\}^{19 \times 119}$, where each row represents a clause. For instance, the row for the clause $\neg sum(1) \vee pred(0, 1) \vee pred(1, 0)$ is a vector in $\{-1, 0, 1\}^{1 \times 119}$ containing a single -1 for atom $sum(1)$, two 1s for atoms $pred(0, 1)$ and $pred(1, 0)$, and 116 0s.

Vectors $\mathbf{f}$ and $\mathbf{v}$ are in $\{0, 1\}^{119}$ constructed from each data instance $\langle i_1, i_2, l \rangle$. The fact vector $\mathbf{f}$ contains a single 1 for atom $sum(l)$ (ground truth label) and 118 0s. To obtain

[1] In case the length of $\mathbf{x}$ is less than n, we pad $\mathbf{x}$ with 0s for all the atoms that are not related to NN output.

the prediction vector $\mathbf{v}$*, we (i) feed images* i_1,i_2 *into the CNN (with softmax at the last layer) from (Manhaeve* et al.*, 2018) to obtain the outputs* $\mathbf{x}_1, \mathbf{x}_2 \in [0,1]^{10}$ *(consisting of probabilities), (ii) construct the vector* $\mathbf{x} \in [0,1]^{100}$ *(for 100 atoms of* $pred/2$*) such that* $\mathbf{x}[10i+j]$ *is* $\mathbf{x}_1[i] \times \mathbf{x}_2[j]$ *for* $i, j \in \{0, \ldots, 9\}$*, (iii) update* $\mathbf{x}$ *as the concatenation of* $\mathbf{x}$ *and* $\{0\}^{19}$*, and (iv) finally, let* $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

Using $\mathbf{C}$, $\mathbf{v}$, and $\mathbf{f}$, we define the *CNF loss* $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ as follows:

$$\mathbf{L}_f = \mathbf{C} \odot \mathbf{f} \tag{4.2}$$

$$\mathbf{L}_v = \mathbb{1}_{\{1\}}(\mathbf{C}) \odot \mathbf{v} + \mathbb{1}_{\{-1\}}(\mathbf{C}) \odot (1 - \mathbf{v}) \tag{4.3}$$

$$\mathbf{deduce} = \mathbb{1}_{\{1\}}\Big(sum(\mathbf{C} \odot \mathbf{C}) - sum(\mathbb{1}_{\{-1\}}(\mathbf{L}_f))\Big) \tag{4.4}$$

$$\mathbf{unsat} = prod(1 - \mathbf{L}_v) \tag{4.5}$$

$$\mathbf{keep} = sum(\mathbb{1}_{\{1\}}(\mathbf{L}_v) \odot (1 - \mathbf{L}_v) + \mathbb{1}_{\{0\}}(\mathbf{L}_v) \odot \mathbf{L}_v) \tag{4.6}$$

$$L_{deduce} = sum(\mathbf{deduce} \odot \mathbf{unsat}) \tag{4.7}$$

$$L_{unsat} = avg(\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat}) \tag{4.8}$$

$$L_{sat} = avg(\mathbb{1}_{\{0\}}(\mathbf{unsat}) \odot \mathbf{keep}) \tag{4.9}$$

$$L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) = L_{deduce} + L_{unsat} + L_{sat} \tag{4.10}$$

where $prod(X)$, $sum(X)$, and $avg(X)$ compute the product, sum, and average of the elements in X along its last dimension. [2] Although these equations may look complex, it helps to know that they use the form $\mathbb{1}_{\{k\}}(X_1) \odot X_2$, where the indicator function $\mathbb{1}_{\{k\}}(X_1)$ can be seen as a constant that is multiplied to a trainable variable X_2. Take equation (4.8) as an example. To minimize L_{unsat}, the NN parameters will be updated towards making $\mathbf{unsat}[i]$ to be 0 whenever $\mathbb{1}_{\{1\}}(\mathbf{unsat})$ is 1, i.e., towards making unsatisfied clauses satisfied.

[2] The aggregated dimension is "squeezed," which is the default behavior in PyTorch aggregate functions (`keepdim` is False).

In equations (4.2) and (4.3), $\mathbf{f}$ and $\mathbf{v}$ are treated as matrices in $\{0,1\}^{1\times n}$ to have element-wise multiplication (with broadcasting) with a matrix in $\{-1,0,1\}^{m\times n}$. Take equation (4.2) as an example, $\mathbf{L}_f[i,j] = \mathbf{C}[i,j] \times \mathbf{f}[j]$. $\mathbf{L}_f$ is the matrix in $\{-1,0,1\}^{m\times n}$ such that (i) $\mathbf{L}_f[i,j] = 1$ iff clause i contains literal p_j and $p_j \in F$; (ii) $\mathbf{L}_f[i,j] = -1$ iff clause i contains literal $\neg p_j$ and $p_j \in F$; (iii) otherwise, $\mathbf{L}_f[i,j] = 0$.

$\mathbf{L}_v$ is the matrix in $\{0,1\}^{m\times n}$ such that $\mathbf{L}_v[i,j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal is TRUE under v.

In equations (4.4), (4.5), and (4.6), $sum(\mathbf{C} \odot \mathbf{C})$ is a vector in $\mathbb{N}^m$ representing in each clause the number of literals, and $sum(\mathbb{1}_{\{-1\}}(\mathbf{L}_f))$ is a vector in $\mathbb{N}^m$ representing in each clause the number of literals that are FALSE under F (i.e., the number of literals of the form $\neg p$ such that $p \in F$). Consequently, $\mathbf{deduce}$ is a vector in $\{0,1\}^m$ where $\mathbf{deduce}[i]$ is 1 iff clause i has all but one literal being FALSE under F. If $C \cup F$ is satisfiable and a clause has all but one literal being FALSE under F, then we can safely deduce that the remaining literal is TRUE. For instance, in a clause for Sudoku

$$\neg a(1,1,9) \vee \neg a(1,2,9), \tag{4.11}$$

if $a(1,1,9)$ is in the ground-truth label (i.e., in F) but $a(1,2,9)$ is not, we can safely deduce $\neg a(1,2,9)$ is true. It follows that such a clause is always a Horn clause. Intuitively, the vector $\mathbf{deduce}$ represents the clauses that such deduction can be applied given F.

The vector $\mathbf{unsat} \in \{0,1\}^m$ indicates which clause is not satisfied by the truth assignment v, where $\mathbf{unsat}[i]$ is 1 iff v does not satisfy the i-th clause. The vector $\mathbf{keep} \in \{0\}^m$ consists of m zeros while its gradient w.r.t. $\mathbf{v}$ consists of non-zeros. Intuitively, the gradient of $\mathbf{keep}$ tries to keep the current predictions $\mathbf{v}$ in each clause.

In equations (4.7), (4.8), and (4.9), $L_{deduce} \in \mathbb{N}$ represents the number of clauses that can deduce a literal given F and are not satisfied by v. The vector $\mathbb{1}_{\{1\}}(\mathbf{unsat}) \in \{0,1\}^m$ (and $\mathbb{1}_{\{0\}}(\mathbf{unsat})$, resp.) indicates the clauses that are not satisfied (and are satisfied, resp.)

by v. Intuitively, for all clauses, minimizing L_{unsat} makes the neural network change its predictions to decrease the number of unsatisfied clauses. In contrast, minimizing L_{sat} makes the neural network more confident in its predictions in the satisfied clauses. We use avg instead of sum in equations (4.8) and (4.9) to ensure that the gradients from L_{unsat} and L_{sat} do not overpower those from L_{deduce}. Formal statements of these intuitive explanations follow in the next section.

For any neural network output $\mathbf{x}$ consisting of probabilities, let $\mathbf{x}^r$ denote the raw value of $\mathbf{x}$ before the activation function (e.g., softmax or sigmoid) in the last layer. Without restriction, the value $\mathbf{x}^r$ may vary in a large range when trained with STE. When such an output is fed into softmax or sigmoid, it easily falls into a saturation region of the activation function (Tang *et al.*, 2017). To resolve this issue, we include another loss function to bound the scale of $\mathbf{x}^r$:

$$L_{bound}(\mathbf{x}) = avg(\mathbf{x}^r \odot \mathbf{x}^r).$$

To enforce constraints, we add the weighted sum of $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ and $L_{bound}(\mathbf{x})$ to the baseline loss (if any), where the weight of each loss is a hyperparameter. We call this way of semantic regularization the *CL-STE* (Constraint Loss via STE) method.

Example 7 Continued. *Given the matrix* $\mathbf{C}$ *for the CNF theory, a data instance* $\langle i_1, i_2, l \rangle$*, the NN outputs* $\mathbf{x}_1, \mathbf{x}_2$ *for* i_1, i_2*, and the vectors* $\mathbf{f}, \mathbf{v}$ *as constructed in Example 7, the total loss function used for* **mnistAdd** *problem is*

$$\mathcal{L} = L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + \sum_{\mathbf{x} \in \{\mathbf{x}_1, \mathbf{x}_2\}} 0.1 \times L_{bound}(\mathbf{x}).$$

4.2.2 Properties of Constraint Loss and Its Gradients

Proposition 3 shows the relation between L_{deduce}, L_{unsat}, and L_{sat} components in the constraint loss L_{cnf} and its logical counterpart.

Proposition 3 *Given a theory C, a set F of atoms, and a truth assignment v such that $v \models F$, let $\mathbf{C}, \mathbf{f}, \mathbf{v}$ denote their matrix/vector representations, respectively. Let $C_{deduce} \subseteq C$ denote the set of Horn clauses H in C such that all but one literal in H are of the form $\neg p$ such that $p \in F$.* [3] *Then*

- *the minimum values of L_{deduce}, L_{unsat}, L_{sat}, and $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ are 0;*
- *$v \models C_{deduce}$ iff L_{deduce} is 0;*
- *$v \models C$ iff L_{unsat} is 0 iff $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ is 0.*

Clause (4.11) is an example clause in C_{deduce}. There could be many other ways to design $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ to satisfy the properties in Proposition 3. Propositions 4 and 5 below justify our design choice.

Proposition 4 *Given a theory C with m clauses and n atoms and a set F of atoms such that $C \cup F$ is satisfiable, let $\mathbf{C}, \mathbf{f}$ denote their matrix/vector representations, respectively. Given a neural network output $\mathbf{x} \in [0,1]^n$ denoting probabilities, we construct $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$ and a truth assignment v such that $v(p_j) =$ TRUE if $\mathbf{v}[j]$ is 1, and $v(p_j) =$ FALSE if $\mathbf{v}[j]$ is 0. Let $C_{deduce} \subseteq C$ denote the set of Horn clauses H in C such that all but one literal in H are of the form $\neg p$ where $p \in F$. Then, for any $j \in \{1, \ldots, n\}$,*

1. *if $p_j \in F$, all of $\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]}$, $\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]}$, and $\frac{\partial L_{sat}}{\partial \mathbf{x}[j]}$ are zeros;*

[3] This implies that the remaining literal is either an atom or $\neg p$ such that $p \notin F$.

2. *if* $p_j \notin F$,

$$\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \begin{cases} -c & \textit{if } c > 0 \textit{ clauses in } C_{deduce} \\ & \textit{contain literal } p_j\textit{;} \\ c & \textit{if } c > 0 \textit{ clauses in } C_{deduce} \\ & \textit{contain literal } \neg p_j\textit{;} \\ 0 & \textit{otherwise;} \end{cases}$$

$$\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \frac{c_2 - c_1}{m}$$

$$\frac{\partial L_{sat}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \begin{cases} -\frac{c_3}{m} & \textit{if } v \models p_j \\ \frac{c_3}{m} & \textit{if } v \not\models p_j, \end{cases}$$

where $\overset{iSTE}{\approx}$ *stands for the equivalence of gradients assuming iSTE;* c_1 *(and* c_2*, resp.) is the number of clauses in* C *that are not satisfied by* v *and contain* p_j *(and* $\neg p_j$*, resp.);* c_3 *is the number of clauses in* C *that are satisfied by* v *and contain* p_j *or* $\neg p_j$*.*

Intuitively, Proposition 4 ensures the following properties of the gradient $\frac{\partial L_{cnf}(\mathbf{C},\mathbf{v},\mathbf{f})}{\partial \mathbf{x}[j]}$, which consists of $\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]}$, $\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]}$, and $\frac{\partial L_{sat}}{\partial \mathbf{x}[j]}$.

P1. If we know for sure that p_j is TRUE ($p_j \in F$), these gradients w.r.t. $\mathbf{x}[j]$ (real values corresponding to p_j) are 0, so they do not affect the truth value of p_j.

P2. Otherwise (F does not tell whether p_j is TRUE),

1. the gradient $\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]}$ is negative (positive, resp.) to increase (decrease, resp.) the value of $\mathbf{x}[j]$ by the gradient descent if $C \cup F$ entails p_j ($\neg p_j$, resp.);

2. the gradient $\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]}$ is negative (positive resp.) to increase (decrease, resp.) the value of $\mathbf{x}[j]$ by the gradient descent if, among all unsatisfied clauses, more clauses contain p_j than $\neg p_j$ ($\neg p_j$ than p_j, resp.);

3. the gradient $\frac{\partial L_{sat}}{\partial \mathbf{x}[j]}$ is negative (positive resp.) to increase (decrease, resp.) the value of $\mathbf{x}[j]$ by the gradient descent if $v \models p_j$ ($v \not\models p_j$, resp.) and there exist satisfied clauses containing literal p_j or $\neg p_j$.

Intuitively, bullet 1 in **P2** simulates a deduction step, which is always correct, while bullets 2 and 3 simulate two heuristics: "we tend to believe a literal if more unsatisfied clauses contain this literal than its negation" and "we tend to keep our prediction on an atom if many satisfied clauses contain this atom." This justifies another property below.

P3. The sign of the gradient $\frac{\partial L_{cnf}}{\partial \mathbf{x}[j]}$ is the same as the sign of $\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]}$ when the latter gradient is non-zero.

Example 8 *Consider the theory C below with $m = 2$ clauses and 3 atoms*

$$\neg a \vee \neg b \vee c$$

$$\neg a \vee b$$

and consider the set of given facts $F = \{a\}$. They are represented by matrix $\mathbf{C} = \begin{bmatrix} -1 & -1 & 1 \\ -1 & 1 & 0 \end{bmatrix}$ and vector $\mathbf{f} = [1, 0, 0]$. Suppose a neural network predicts $\mathbf{x} = [0.3, 0.1, 0.9]$ as the probabilities of the 3 atoms $\{a, b, c\}$.

With the above matrix and vectors, we can compute

$$b_p(\mathbf{x}) = [0, 0, 1],$$

$$\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x}) = [1, 0, 1].$$

From $\mathbf{v}$, we construct the truth assignment $v = \{a = \text{TRUE}, b = \text{FALSE}, c = \text{TRUE}\}$. Clearly, v satisfies the first clause but not the second one. Given $F = \{a\}$, we see C_{deduce} is $\neg a \vee b$.

According to Proposition 4,

$$\frac{\partial L_{deduce}}{\partial \mathbf{x}} \overset{iSTE}{\approx} [0, -1, 0], \quad \frac{\partial L_{unsat}}{\partial \mathbf{x}} \overset{iSTE}{\approx} [0, -\frac{1}{2}, 0], \quad \frac{\partial L_{sat}}{\partial \mathbf{x}} \overset{iSTE}{\approx} [0, \frac{1}{2}, -\frac{1}{2}],$$

$$\frac{\partial L_{cnf}}{\partial \mathbf{x}} = \frac{\partial L_{deduce}}{\partial \mathbf{x}} + \frac{\partial L_{unsat}}{\partial \mathbf{x}} + \frac{\partial L_{sat}}{\partial \mathbf{x}} \overset{iSTE}{\approx} [0, -1, -\frac{1}{2}].$$

*Intuitively, given C, F, and the current truth assignment v, (***P1***) we know a is* TRUE *($a \in F$) thus no need to update it, (***P2.1** *and* **P3***) we know for sure that the prediction for b should be changed to* TRUE *by deduction on clause $\neg a \vee b$ and the given fact $F = \{a\}$, (***P2.3***) we tend to strengthen our belief in c being* TRUE *due to the satisfied clause $\neg a \vee \neg b \vee c$.*

The proposition also holds with another binarization function $b(x)$.

Proposition 5 *Proposition 4 still holds for $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b(\mathbf{x})$.*

4.3 Evaluation of CL-STE

We conduct an experimental evaluation to answer the following questions.

Q1 Is CL-STE more scalable in injecting discrete constraints into neural network learning than existing neuro-symbolic learning methods?

Q2 Does CL-STE make neural networks learn with no or fewer labeled data by effectively utilizing the given constraints?

Q3 Is CL-STE general enough to overlay constraint loss on different types of neural networks to enforce logical constraints and improve the accuracy of existing networks?

Our implementation takes a CNF theory in DIMACS format (the standard format for input to SAT solvers).[4] Since the CL-STE method alone doesn't have associated symbolic rules, unlike `DeepProbLog`, `NeurASP`, and `NeuroLog`, in this section, we compare these methods on the classification accuracy of the trained NNs (e.g., correctly predicting

[4]All experiments in this section were done on Ubuntu 18.04.2 LTS with two 10-cores CPU Intel(R) Xeon(R) CPU E5-2640 v4 @ 2.40GHz and four GP104 [GeForce GTX 1080].

the label of an MNIST image) instead of query accuracy (e.g., correctly predicting the sum of two MNIST images).

4.3.1 mnistAdd Revisited

We introduced the CNF encoding and the loss function for the **mnistAdd** problem in Example 7. The problem was used in (Manhaeve *et al.*, 2018) and Chapter 3 as a benchmark.

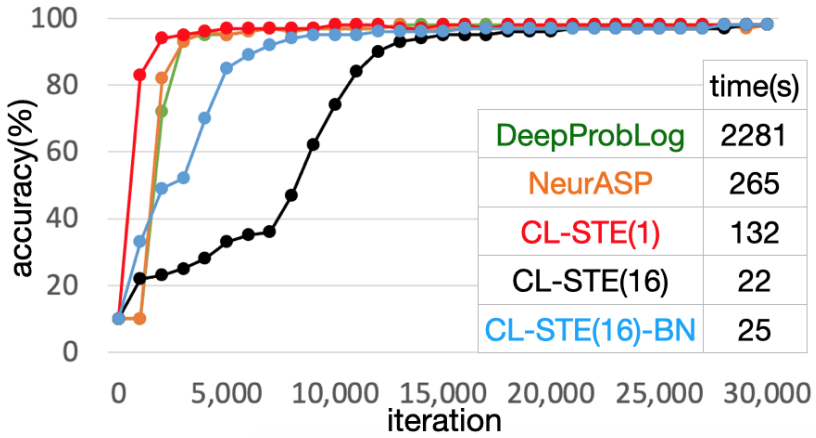

Figure 4.3: Comparison on mnistAdd

Figure 4.3 compares the MNIST digit classification accuracy of neural networks trained by different methods on a single epoch of 30,000 addition data from (Manhaeve *et al.*, 2018). "CL-STE(n)" denotes our method with $b_p(x)$ and iSTE using a batch of size n. As we see, `DeepProbLog`, `NeurASP`, and `CL-STE` with a batch size of 1 could quickly converge to near 100% test accuracy. Training time-wise, CL-STE outperforms the other approaches since it does not need to generate arithmetic circuits for every training instance as in `DeepProbLog` or enumerate all models as in `NeurASP`. Also, while `DeepProbLog` and `NeurASP` do not support batch training, CL-STE could leverage the batch training to reduce the training time to 22s with a batch size of 16 (denoted by CL-STE(16)). We observe that increasing the batch size in CL-STE also increases the number of parameter

Table 4.1: Experiments on mnistAdd

	mnistAdd	mnistAdd2	mnistAdd3
DeepProbLog	98.36% 2565s	97.57% 22699s	timeout
NeurASP	97.87% 292s	97.85% 1682s	timeout
CL-STE	97.48% 22s	98.12% 92s	97.78% 402s

updates for convergence, which we could decrease by using batch normalization as shown in the blue line denoted by CL-STE(16)-BN.

Furthermore, we apply CL-STE to the variants of **mnistAdd** by training with two-digit sums (**mnistAdd2** (Manhaeve *et al.*, 2018)) and three-digit sums (**mnistAdd3**). Table 4.1 shows that the CL-STE method scales much better than `DeepProbLog` and `NeurASP`. The time and accuracy are reported for a single epoch of training, where the cutoff time is 24 hours after which we report "timeout."

4.3.2 Benchmarks from (Tsamoura et al.*, 2021)*

The following are benchmark problems from (Tsamoura *et al.*, 2021). Like the **mnistAdd** problem, labels are not immediately associated with the data instances but with the results of logical operations applied to them.

add2x2 The input is a 2×2 grid of digit images. The output is the four sums of the pairs of digits in each row/column. The task is to train a CNN for digit classification.

apply2x2 The input is three digits and a 2×2 grid of hand-written math operator images in $\{+, -, \times\}$. The output is the four results of applying the two math operators in each row/column in the grid on the three digits. The task is to train a CNN for math operator classification.

member(n) The input is a set of n images of digits and a digit in $\{0, \ldots, 9\}$. The output is 0 or 1, indicating whether the digit appears in the set of digit images. The task is to train

Table 4.2: Comparison between CL-STE and Other Approaches: The Numbers in Parentheses Are the Times Spent by NeuroLog to Generate All Abductive Proofs.

	add2x2	apply2x2	member(3)	member(5)
accuracy(%)				
DeepProbLog	88.4±0.7	100±0	96.3±0.3	timeout
NeurASP	97.6±0.2	100±0	93.5±0.9	timeout
NeuroLog	97.5±0.4	100±0	94.5±1.5	93.9±1.5
$b(x)$ + iSTE	95.5±0.7	100±0	73.2±9.1	51.1±24.9
$b(x)$ + sSTE	95.7±0.5	100±0	83.2±8.4	88.0±7.1
$b_p(x)$ + iSTE	98.0±0.2	100±0	95.5±0.7	95.0±0.5
time(s)				
DeepProbLog	1035±71	586±9	2218±211	timeout
NeurASP	142±2	47±1	253±1	timeout
NeuroLog	2400±46 (1652)	2428±29 (2266)	427±12 (27)	682±40 (114)
$b(x)$ + iSTE	80±2	208±1	45±0	177±1
$b(x)$ + sSTE	81±2	214±8	46±1	181±10
$b_p(x)$ + iSTE	54±4	112±2	43±3	49±4

a CNN for digit classification.

Table 4.2 compares our method with `DeepProbLog`, `NeurASP`, and `NeuroLog` test accuracy-wise and training time-wise. Note that, instead of comparing the query accuracy as in (Tsamoura *et al.*, 2021), we evaluate and compare the NN classification accuracies.

Our experiments agree with (Yin *et al.*, 2019), which proves the instability issue of iSTE and the convergence guarantees with sSTE in a simple 2-layer CNN. Their experiments also observe a better performance of $b(x)$+sSTE over $b(x)$+iSTE on deep neural networks.

Our experimental results (especially for member(5) problem) also reveal the instability issue of $b(x)$+iSTE and show that $b(x)$+sSTE achieves higher and more stable accuracy. Furthermore, we observe that $b_p(x)$ works better than $b(x)$ in terms of both accuracy and time in our experiments. This is because the input x to $b_p(x)$ is normalized into probabilities before binarization, resulting in less information loss (i.e., change in magnitude "$b_p(x)-x$") when the neural network accuracy increases.

4.3.3 CNN + Constraint Loss for Sudoku

The following experimental setting from Section 3.4.2 demonstrates unsupervised learning with `NeurASP` on textual Sudoku problems. Recall that, given a textual representation of a Sudoku puzzle (in the form of a 9×9 matrix where an empty cell is represented by 0), Park (2018) trained a CNN using 1 million examples and achieved 70% test accuracy using an "inference trick". With the same CNN and inference trick, `NeurASP` achieved 66.5% accuracy with only 7% data with no supervision (i.e., 70k data instances without labels) by enforcing semantic constraints in neural network training. In this section, we consider the same unsupervised learning problem for Sudoku while we represent the Sudoku problem in CNF and use L_{cnf} to enforce logical constraints during training.

We use a CNF theory for 9×9 Sudoku problems with $9^3 = 729$ atoms and 8991 clauses as described in Section 4.4. This CNF can be represented by a matrix $\mathbf{C} \in \{-1, 0, 1\}^{8991 \times 729}$. The dataset consists of 70k data instances, 80%/20% for training/testing. Each data instance is a pair $\langle \mathbf{q}, \mathbf{l} \rangle$ where $\mathbf{q} \in \{0, 1, \ldots, 9\}^{81}$ denotes a 9×9 Sudoku board (0 denotes an empty cell) and $\mathbf{l} \in \{1, \ldots, 9\}^{81}$ denotes its solution ($\mathbf{l}$ is not used in `NeurASP` and our method during training). The non-zero values in $\mathbf{q}$ are treated as atomic facts F and we construct the matrix $\mathbf{F} \in \{0, 1\}^{81 \times 9}$ such that, for $i \in \{1, \ldots, 81\}$, the i-th row $\mathbf{F}[i, :]$ is the vector $\{0\}^9$ if $\mathbf{q}[i] = 0$ and is the one-hot vector for $\mathbf{q}[i]$ if $\mathbf{q}[i] \neq 0$. Then, the vector $\mathbf{f} \in \{0, 1\}^{729}$ is simply the flattening of $\mathbf{F}$. We feed $\mathbf{q}$ into the CNN and obtain the

Table 4.3: CNN, NeurASP, and CL-STE on Park 70k Sudoku Dataset (80%/20% split) w/ and w/o Inference Trick

Method	Supervised	Acc_{wo}	Acc_{w}	time(m)
Park's CNN	Full	0.94%	23.3%	163
Park's CNN+NeurASP	No	1.69%	66.5%	13230
Park's CNN+CL-STE	No	2.38%	93.7%	813

output $\mathbf{x} \in [0,1]^{729}$. Finally, the prediction $\mathbf{v} \in \{0,1\}^{729}$ is obtained as $\mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$, and the total loss function $\mathcal{L}$ we used is $\mathcal{L} = L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + 0.1 \times L_{bound}(\mathbf{x})$.

Table 4.3 compares the training time and the (whole-board) test accuracies with and without the inference trick (Acc_w and Acc_{wo}, resp.) using $b_p(x)$+iSTE against `NeurASP` and baseline CNN (Park, 2018). In each experiment, the same CNN is trained with only 70k (labeled/unlabeled) data instances from (Yang *et al.*, 2020) with an average of 43 given digits in a puzzle (min: 26, max: 77). As we can see, our method outperforms `NeurASP` in both accuracy and time. Accuracy-wise, the CNN model trained using CL-STE is 27.2% more accurate than the CNN model trained using `NeurASP` when we use the inference trick. Training time-wise, CL-STE is 16 times faster than `NeurASP` because we directly encode semantic constraints in a loss function, which saves the time to call a symbolic engine externally (e.g., CLINGO to enumerate all stable models as in `NeurASP`).

Table 4.4 compares CNN+CL-STE with SATNet trained on Park 70k and tested on both Park 70k and Palm Sudoku dataset (Palm *et al.*, 2018). While CNN is less tailored to logical reasoning than SATNet, our experiment shows that, when it is trained via CL-STE, it performs better than SATNet.

4.3.4 GNN + Constraint Loss for Sudoku

This section investigates if a GNN training can be improved with the constraint loss functions with STE by utilizing already known constraints without always relying on the

Table 4.4: SATNet vs. CNN+CL-STE

Method	Train Data (Supv)	Test Data	#Given	Test Accuracy
SATNet	Park 70k (Full)	Park 70k	26-77	67.78%
		Palm	17-34	6.76%
CNN+CL-STE	Park 70k (No)	Park 70k	26-77	93.70%
		Palm	17-34	27.37%

labeled data. We consider the Recurrent Relational Network (RRN) (Palm *et al.*, 2018), a state-of-the-art GNN for multi-step relational reasoning that achieves 96.6% accuracy for hardest Sudoku problems by training on 180k labeled data instances. Our focus here is to make RRN learn more effectively using fewer data by injecting known constraints.

The training dataset in (Palm *et al.*, 2018) contains 180k data instances evenly distributed in 18 difficulties with 17-34 given numbers. We use a small subset of this dataset with random sampling. Given a data instance $\langle \mathbf{q}, \mathbf{l} \rangle$ where $\mathbf{q} \in \{0, 1, \ldots, 9\}^{81}$ denotes a 9×9 Sudoku board and $\mathbf{l} \in \{1, \ldots, 9\}^{81}$ denotes its solution, RRN takes $\mathbf{q}$ as input and, after 32 iterations of message passing, outputs 32 matrices of probabilities $\mathbf{X}_s \in \mathbf{R}^{81 \times 9}$ where $s \in \{1, \ldots, 32\}$; for example, $\mathbf{X}_1$ is the RRN prediction after 1 message passing step.

The baseline loss is the sum of the cross-entropy losses between prediction $\mathbf{X}_s$ and label $\mathbf{l}$ for all s.

We evaluate if using constraint loss could further improve the performance of RRN with the same labeled data. We use the same L_{cnf} and L_{bound} defined in CNN (with weights 1 and 0.1, resp.), which are applied to $\mathbf{X}_1$ only so that the RRN could be trained to deduce new digits in a single message passing step. We also use a continuous regularizer L_{sum} below to regularize every $\mathbf{X}_s$ that "the sum of the 9 probabilities in $\mathbf{X}_s$ in the same row/-

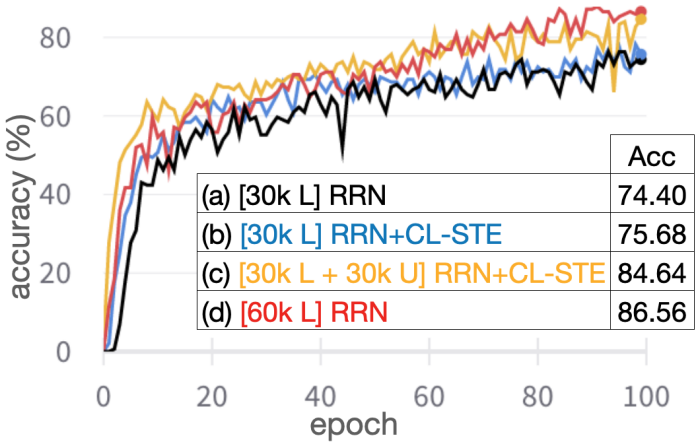

Figure 4.4: Test Accuracy on the Same Randomly Sampled 1k Data from Palm Sudoku Dataset When Trained with RRN(+STE) with 30k to 60k [L]abeled/[U]nlabeled Data

column/box must be 1":

$$L_{sum} = \sum_{\substack{s\in\{1,\ldots,32\}\\ i\in\{row,col,box\}}} avg((sum(\mathbf{X}_s^i) - 1)^2).$$

Here, $avg(X)$ and $sum(X)$ compute the average and sum of all elements in X along its last dimension; $\mathbf{X}_s^{row}, \mathbf{X}_s^{col}, \mathbf{X}_s^{box} \in \mathbb{R}^{81\times 9}$ are reshaped copies of $\mathbf{X}_s$ such that each row in, for example, $\mathbf{X}_s^{row}$ contains 9 probabilities for atoms $a(1, C, N), \ldots, a(9, C, N)$ for some C and N.

Figure 4.4 compares the test accuracy of the RRN trained for 100 epochs under 4 settings: (a) the RRN trained with baseline loss using 30k labeled data; (b) the RRN trained with both baseline loss and constraint losses (L_{sum}, L_{cnf}, and L_{bound}) using the same 30k labeled data; (c) the same setting as (b) with additional 30k unlabeled data; (d) same as (a) with additional 30k labeled data. Comparing (a) and (b) indicates the effectiveness of the constraint loss using the same number of labeled data; comparison between (b) and (c) indicates even with the same number of labeled data but adding unlabeled data could increase the accuracy (due to the constraint loss); comparison between (c) and (d) shows that the effectiveness of the constraint loss is comparable to adding additional 30k labels.

Figure 4.5 assesses the effect of constraint loss using fixed 10k labeled data and varying

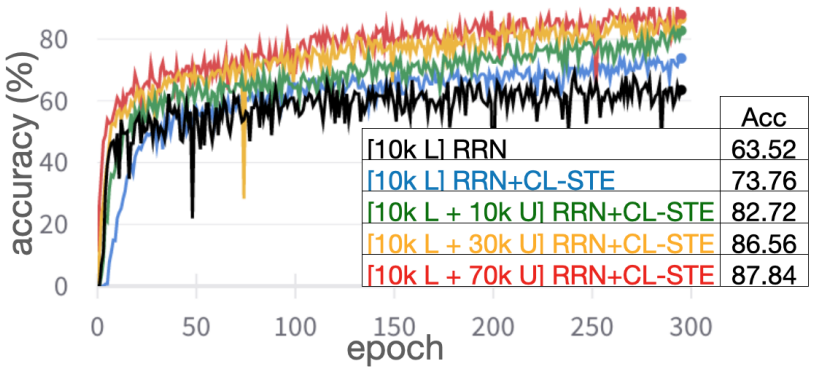

Figure 4.5: Semi-supervised Learning with RRN+STE on Sudoku Using Only 10k Labeled Data and Varying Numbers of Unlabeled Data from Palm Dataset for Training and Using the Same Randomly Sampled 1k Data for Testing

numbers (10k, 30k, 70k) of unlabeled data. We see that the baseline RRN trained with 10k labeled data ([10k L] RRN) has roughly saturated while the other methods are still slowly improving the accuracy. Training with the same number of labeled data but adding more unlabeled data makes the trained RRN achieve higher test accuracy, indicating that the constraint loss is effective in training even when the labels are unavailable.

Remark. Although the CNN in the previous section seems to show comparable accuracies as the RRN in this section, it relies on the inference trick and has to be trained on a dataset with varying difficulties (e.g., 26-77 givens) to make the inference trick work. The RRN result does not use the trick, and we find that its learning is more robust. On the other hand, the RRN could not learn well when there are no labeled data unlike the CNN.

4.3.5 Learning to Solve the Shortest Path Problem

In this section, we evaluate `CL-STE` on the shortest path problem from Section 3.4.3. The experiment is about, given a graph and two points, finding the shortest path between them. We use the same dataset from (Xu *et al.*, 2018) as in Section 3.4.3, but now the dataset is divided into 80/20 train/test examples.

Recall that each example is a 4 by 4 grid $G = (V, E)$, where $|V| = 16, |E| = 24$, two-terminal (i.e., source and the destination) nodes are randomly picked up from 16 nodes, and 8 edges are randomly removed from 24 edges to increase the difficulty. The dataset consists of 1610 data instances, each is a pair $\langle \mathbf{i}, \mathbf{l} \rangle$ where $\mathbf{i} \in \{0, 1\}^{40}$ and $\mathbf{l} \in \{0, 1\}^{24}$. The ones in the first 24 values in $\mathbf{i}$ denote the (non-removed) edges in the grid, the ones in the last 16 values in $\mathbf{i}$ denote the terminal nodes, and ones in $\mathbf{l}$ denote the edges in the shortest path.

We define a CNF with 40 atoms and 120 clauses to represent "each terminal node is connected to exactly one edge in the shortest path". To start with, let's identify each node in the 4×4 grid by a pair (i, j) for $i, j \in \{1, \ldots, 4\}$ and identity the edge between nodes (i_1, j_1) and (i_2, j_2) as $((i_1, j_1), (i_2, j_2))$. Then, we introduce the following 2 atoms.

- $terminal(i, j)$ represents that node (i, j) is one of the two terminal nodes.
- $sp((i_1, j_1), (i_2, j_2))$ represents edge $((i_1, j_1), (i_2, j_2))$ is in the shortest path.

Then, the CNF for the shortest path problem consists of 120 clauses: 16 clauses of the form

$$\neg terminal(i_1, j_1) \vee \bigvee_{\substack{i_2, j_2: \\ ((i_1, j_1), (i_2, j_2)) \\ \text{is an edge}}} sp((i_1, j_1), (i_2, j_2))$$

for $i_1, j_1 \in \{1, \ldots, 4\}$, and 104 clauses of the form

$$\neg terminal(i_1, j_1) \vee \neg sp((i_1, j_1), (i_2, j_2)) \vee \neg sp((i_1, j_1), (i_3, j_3))$$

for $i_*, j_* \in \{1, \ldots, 4\}$ such that $((i_1, j_1), (i_2, j_2))$ and $((i_1, j_1), (i_3, j_3))$ are different edges.

This CNF can be represented by a matrix $\mathbf{C} \in \{-1, 0, 1\}^{120 \times 40}$.

To construct $\mathbf{f}$ and $\mathbf{v}$ for a data instance $\langle \mathbf{i}, \mathbf{l} \rangle$, the facts $\mathbf{f} \in \{0, 1\}^{40}$ is simply the concatenation of $\mathbf{i}[24:]$ and $\{0\}^{24}$; while the prediction $\mathbf{v}$ is a vector in $\{0, 1\}^{40}$ obtained as follows. We (i) feed $\mathbf{i}$ into the same MLP from (Xu *et al.*, 2018) and obtain the NN output

$\mathbf{x} \in [0,1]^{24}$ consisting of probabilities, (ii) extend $\mathbf{x}$ with 16 $0s$ (in the beginning) so as to have a 1-1 correspondence between 40 elements in $\mathbf{x}$ and 40 atoms in the CNF, and (iii) $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

Finally, the total loss function $\mathcal{L}_{base}$ used in the baseline is

$$\mathcal{L}_{base} = L_{cross}(\mathbf{x}, \mathbf{l})$$

where L_{cross} is the cross-entropy loss.

The loss function $\mathcal{L}$ used for shortest path problem is

$$\mathcal{L} = L_{cross}(\mathbf{x}, \mathbf{l}) + \alpha L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + \beta L_{bound}(\mathbf{x})$$

where $\alpha = 0.2$ and $\beta = 1$. We set $\alpha = 0.2$ in our experiments to balance the gradients from the CNF loss and cross entropy loss. Indeed, a similar accuracy can be achieved if we compute α dynamically as $\frac{g_{cross}}{g_{cnf}}$ where g_{cnf} and g_{cross} are the maximum absolute values in the gradients $\frac{\partial L_{cnf}(\mathbf{C},\mathbf{v},\mathbf{f})}{\partial \mathbf{x}}$ and $\frac{\partial L_{cross}(\mathbf{x},\mathbf{l})}{\partial \mathbf{x}}$ respectively. Intuitively, the weight α makes sure that the semantic regularization from CL-STE won't overwrite the hints from labels.

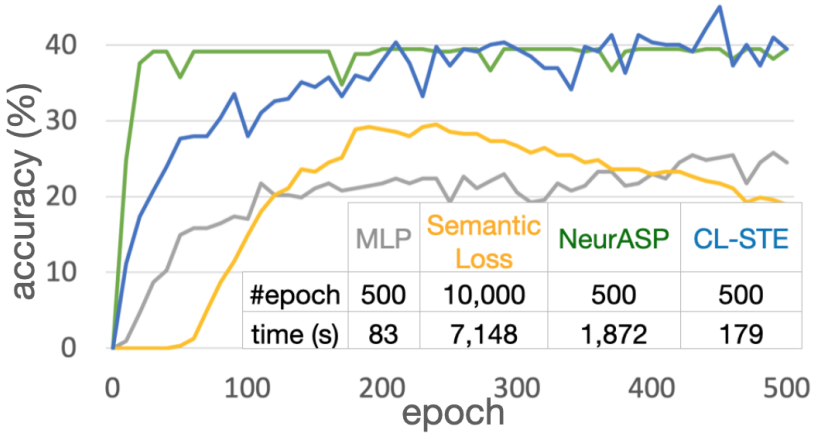

Figure 4.6: MLP+CL-STE on Shortest Path Problem

Figure 4.6 compares the test accuracy of the same Multi-Layer Perceptron (MLP) trained by different learning methods during 500 epochs of training (except that the ac-

curacy for Semantic Loss method is reported for 10k epochs). As we can see, it only took 83s for baseline and 179s for CL-STE to complete all 500 epochs (including the time to compute training and testing accuracy) since they are all trained on GPU with a batch size of 32. Besides, CL-STE achieves comparable accuracy to `NeurASP` with only about $\frac{1}{10}$ of time. The training time of Semantic Loss in Figure 4.6 was recorded when it was trained on CPU. We re-did the Semantic Loss experiment on GPU with early stopping and found that it still takes 1032s to achieve the highest accuracy 30.75% after 2900 epochs of training.

4.3.6 Semi-Supervised Learning for MNIST and FASHION Dataset

Xu *et al.* (2018) show that minimizing semantic loss could enhance semi-supervised multi-class classification results by enforcing the constraint that a model must assign a unique label even for unlabeled data. Their method achieves state-of-the-art results on the permutation invariant MNIST classification problem, a commonly used testbed for semi-supervised learning algorithms, and a slightly more challenging problem, FASHION-MNIST.

For both tasks, we apply $b(x)$+iSTE to the same MLP (without softmax in the last layer) as in (Xu *et al.*, 2018), i.e., an MLP of shape (784, 1000, 500, 250, 250, 250, 10), where the output $\mathbf{x} \in \mathbb{R}^{10}$ denotes the digit/cloth prediction.

The CNF for this problem consists of 46 clauses: 1 clause

$$pred(i, 0) \vee \cdots \vee pred(i, 9)$$

and 45 clauses of the form

$$\neg pred(i, n_1) \vee \neg pred(i, n_2)$$

for $n_1, n_2 \in \{0, \ldots, 9\}$ such that $n_1 < n_2$. Intuitively, these 2 clauses define the existence and uniqueness constraints on the label of image i. This CNF can be represented by the matrix $\mathbf{C} \in \{-1, 0, 1\}^{46 \times 10}$.

The vectors $\mathbf{f}$ and $\mathbf{v}$ are constructed in the same way for both supervised data instance $\langle i, l \rangle$ and unsupervised data instance $\langle i \rangle$. The facts $\mathbf{f}$ is simply $\{0\}^{10}$; while the prediction $\mathbf{v}$ is a vector in $\{0, 1\}^{10}$ obtained as follows. We (i) feed image i into the CNN and obtain the outputs $\mathbf{x} \in \mathbb{R}^{10}$ (consisting of real values not probabilities); and (ii) $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b(\mathbf{x})$. Then, the total loss for unsupervised data instances is defined as

$$\mathcal{L} = L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + L_{bound}(\mathbf{x}),$$

which enforces that each image should map to exactly one digit or one cloth type. The total loss for supervised data instance simply contains $\mathcal{L}$ as well as the typical cross-entropy loss.

We train the network using 100, 500, and 1,000 partially labeled data and full (60,000) labeled data, respectively. We run experiments for 50k batch updates with a batch size of 32. Each experiment is repeated 10 times, and we report the mean and the standard deviation of classification accuracy (%).

Table 4.5: Accuracy on MNIST & FASHION dataset

Method	Number of labeled examples used			
	100	500	1000	All (60,000)
MNIST (Xu et al.)	85.3±1.1	94.2±0.5	95.8±0.2	98.8±0.1
MNIST ($b(x)$+iSTE)	84.4±1.5	94.1±0.3	95.9±0.2	98.8±0.1
FASHION (Xu et al.)	70.0±2.0	78.3±0.6	80.6±0.3	87.3±0.2
FASHION ($b(x)$+iSTE)	71.0±1.2	78.6±0.7	80.7±0.5	87.2±0.1

Table 4.5 shows that the MLP with the CNF loss achieves similar accuracy with the implementation of semantic loss from (Xu *et al.*, 2018). Time-wise, each experiment using the method from (Xu *et al.*, 2018) took up about 12 minutes, and each experiment using the CL-STE method took about 10 minutes. There is not much difference in training time since

the logical constraints for this task in the implementation of semantic loss (Xu *et al.*, 2018) are simple enough to be implemented in python scripts without constructing an arithmetic circuit and inference on it.

4.3.7 Ablation Study with GNN and Constraint Loss for Sudoku

To better analyze the effect of constraint losses on general GNN, in this section, we apply constraint losses to a publicly available GNN for Sudoku problem.[5] The graph for Sudoku problem consists of 81 nodes, one for each cell in the Sudoku board, and 1620 edges, one for each pair of nodes in the same row, column, or 3×3 non-overlapping box. The GNN consists of an embedding layer, 8 iterations of message passing layers, and an output layer.

For each data instance $\langle \mathbf{q}, \mathbf{l} \rangle$, the GNN takes $\mathbf{q} \in \{0, 1, \ldots, 9\}^{81}$ as input and outputs a matrix of probabilities $\mathbf{X} \in \mathbf{R}^{81 \times 9}$ after 8 message passing steps.

The baseline loss $L_{baseline}$ is the cross-entropy loss defined on prediction $\mathbf{X}$ and label $\mathbf{l}$.

$$L_{baseline} = L_{cross\_entropy}(\mathbf{X}, \mathbf{l})$$

The constraint loss L_{cl} is the same as the total loss in Appendix 4.4.6 where $\mathbf{x}$ is the flattening of $\mathbf{X}$.

$$L_{cl} = L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + 0.1 \times L_{bound}(\mathbf{x}). \tag{4.12}$$

In addition, we designed the following domain-specific loss functions for Sudoku problem as semantic regularizers for comparison. Intuitively, L_{hint} says that "the given digits must be predicted" and L_{sum} says that "the sum of the 9 probabilities in $\mathbf{X}$ in the same

[5]The GNN is from https://www.kaggle.com/matteoturla/can-graph-neural-network-solve-sudoku, along with the dataset.

row/column/box must be 1".

$$L_{hint} = avg\Big(\mathbf{f} \odot (1 - b_p(\mathbf{x}))\Big)$$

$$L_{sum} = \sum_{\substack{s\in\{1,\ldots,32\}\\ i\in\{row,col,box\}}} avg((sum(\mathbf{X}_s^i) - 1)^2).$$

Here, $avg(X)$ and $sum(X)$ compute the average and sum of all elements in X along its last dimension; $\mathbf{X}_s^{row}, \mathbf{X}_s^{col}, \mathbf{X}_s^{box} \in \mathbb{R}^{81\times 9}$ are reshaped copies of $\mathbf{X}_s$ such that each row in, for example, $\mathbf{X}_s^{row}$ contains 9 probabilities for atoms $a(1, C, N), \ldots, a(9, C, N)$ for some C and N.

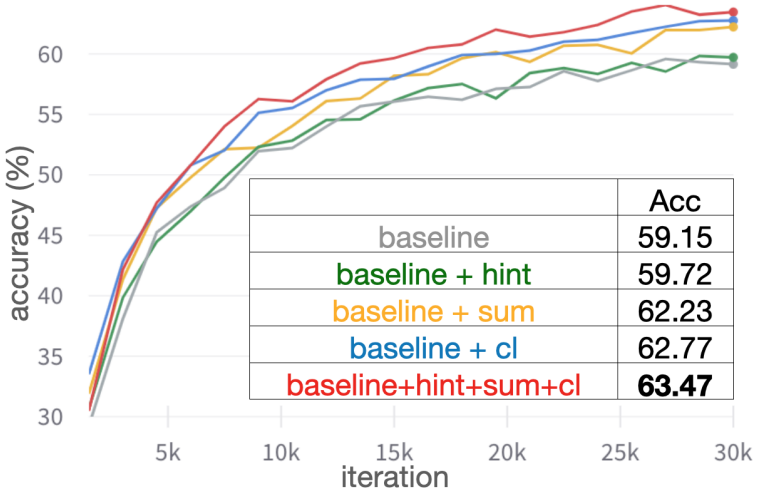

Figure 4.7: Acc with 30k Dataset under Different Losses

Figure 4.7 shows the test accuracy of the GNN after 20 epochs of training on 30k data instances (with full supervision) using different loss functions (denoted by subscripts of losses). It shows monotonic improvement from each loss and the best result is achieved when we add all losses.

Figure 4.8 further shows the monotonic improvement from each component in

$$L_{cl} = L_{deduce} + L_{sat} + L_{unsat} + 0.1 \times L_{bound}$$

where we split $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ in equation (4.12) into its 3 components. We can see that the most improvement comes from $L_{deduce} + 0.1 \times L_{bound}$, which aligns with Proposition 4

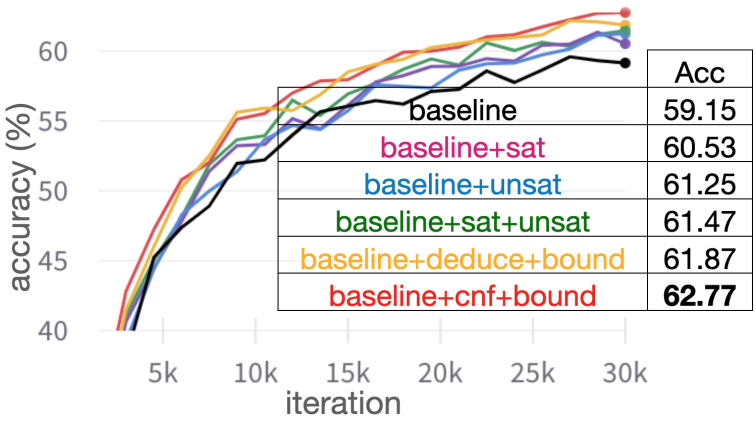

Figure 4.8: Acc with 30k Dataset under Different Losses in L_{cl}

since L_{deduce} has dominant gradients that enforces a deduction step. Noticeably, L_{bound} is necessary for L_{deduce} to bound the size of raw NN output.

Figure 4.9 shows the test accuracy of the GNN after 60 epochs of training on 60k data instances (with full supervision). We can see that the monotonic improvement from each loss is kept in the experiments with 60k data instances and the best result is still achieved when we add all losses. However, the most improvement is from L_{sum} instead of L_{cl}. This is because most semantic information in L_{cl} are from L_{deduce} (i.e., one step deduction from the given digits), which can be eventually learned by the GNN with more data instances.

4.3.8 Discussion

Regarding **Q1**, Figure 4.3, Tables 4.1 and 4.2 show that our method achieves comparable accuracy with existing neuro-symbolic formalisms but is much more scalable. Regarding **Q2**, Table 4.3 and Figures 4.4 and 4.5 illustrate our method could be used for unsupervised and semi-supervised learning by utilizing the constraints underlying the data. Regarding **Q3**, we applied constraint loss to MLP, CNN, and GNN, and observed that it improves the existing neural networks' prediction accuracy.

As we noted, the gradient computation in other neuro-symbolic approaches, such as

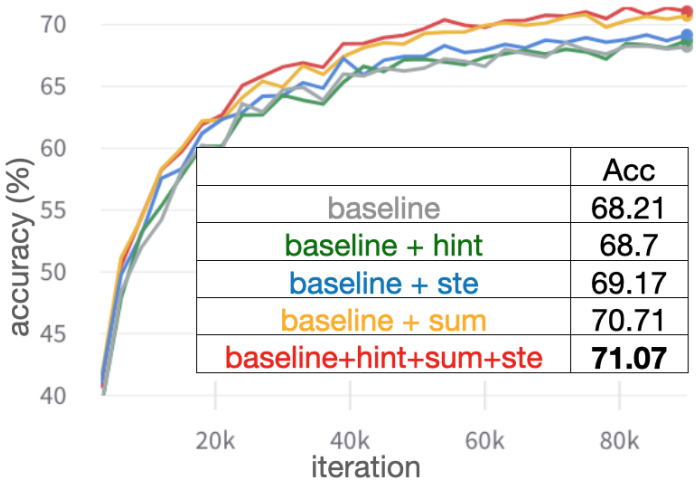

Figure 4.9: Acc with 60k Dataset under Different Losses

`NeurASP`, `DeepProbLog`, and `NeuroLog`, requires external calls to symbolic solvers to compute stable models or proofs for every data instance, which takes a long time. These approaches may give better quality gradients to navigate to feasible solutions, but their gradient computations are associated with NP-hardness (the worst case exponential size of SDD, computing all proofs or stable models). In comparison, CL-STE treats each clause independently and locally to accumulate small pieces of gradients, allowing us to leverage GPUs and batch training as in the standard deep learning. The method resembles local search and deduction in SAT, and the gradients may not reflect the global property but could be computed significantly faster. Indeed, together with the gradient signals coming from the data, our method works well even when logical constraints are hard to satisfy, e.g., in training a neural network to solve Sudoku where a single feasible solution lies among 9^{47} to 9^{64} candidates when 17-34 digits are given.

Constraint loss helps neural networks learn with fewer data, but the state-of-the-art methods require combinatorial computation to compute gradients. By leveraging STE, we demonstrate the feasibility of more scalable constraint learning in neural networks. Also,

we showed that GNNs could learn with fewer (labeled) data by utilizing known constraints. Based on the formal properties of the CNF constraint loss and the promising initial experiments here, the next step is to apply the method to larger-scale experiments.

We could also consider an extension to weighted constraints. Although we treat all clauses as equal importance in learning, one can consider assigning weights to clauses so that more important constraints are prioritized in learning. Assigning weights to loss functions reflects the idea of weighted logics, such as Markov Logic (Richardson and Domingos, 2006).

4.4 CNF and Total Loss

In this section, we explain the CNF and total loss in each domain.

4.4.1 mnistAdd2

In **mnistAdd2** problem (Manhaeve *et al.*, 2018), a data instance is a 5-tuple $\langle i_1, i_2, i_3, i_4, l\rangle$ such that i_* are images of digits and l is an integer in $\{0, \ldots, 198\}$ denoting the sum of two 2-digit numbers $i_1 i_2$ and $i_3 i_4$. The task is, given 15k data instances of $\langle i_1, i_2, i_3, i_4, l\rangle$, to train a CNN for digit classification given such weak supervision. The CNF for **mnistAdd2** consists of the 199 clauses of the form

$$\neg sum(l) \vee \bigvee_{\substack{n_1,n_2,n_3,n_4\in\{0,\ldots,9\}:\\ 10(n_1+n_3)+n_2+n_4=l}} pred(n_1, n_2, n_3, n_4)$$

for $l \in \{0, \ldots, 198\}$. Intuitively, this clause says that "if the sum of $i_1 i_2$ and $i_3 i_4$ is l, then their individual labels n_1, n_2, n_3, n_4 must satisfy $10(n_1 + n_3) + n_2 + n_4 = l$."

This CNF contains 199 clauses and $10^4{+}199 = 10199$ atoms for $pred/4$ and $sum/1$, respectively. According to the definition, we can construct the matrix $\mathbf{C} \in \{-1, 0, 1\}^{199\times 10199}$ where each row represents a clause.

To construct $\mathbf{f}$ and $\mathbf{v}$ for a data instance $\langle i_1, i_2, i_3, i_4, l\rangle$, the facts $\mathbf{f}$ is simply a vector in

$\{0,1\}^{10199}$ with 10198 0s and a single 1 for atom $sum(l)$; while the prediction $\mathbf{v}$ is a vector in $\{0,1\}^{10199}$ obtained as follows. We (i) feed images i_1,i_2,i_3,i_4 into the CNN and obtain the outputs $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4 \in \mathbb{R}^{10}$ (consisting of probabilities); (ii) construct $\mathbf{x} \in \mathbb{R}^{10000}$ such that its $(1000a + 100b + 10c + d)$-th element is $\mathbf{x}_1[a] \times \mathbf{x}_2[b] \times \mathbf{x}_3[c] \times \mathbf{x}_4[d]$ for $a, b, c, d \in \{0, \ldots, 9\}$; and (iii) $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

The loss function used for **mnistAdd2** problem is

$$\mathcal{L} = \alpha L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + \sum_{\mathbf{x} \in \{\mathbf{x}_1, \ldots, \mathbf{x}_4\}} \beta L_{bound}(\mathbf{x})$$

where $\alpha = 1$ and $\beta = 0.01$.

4.4.2 *mnistAdd using b(x) and iSTE*

In **mnistAdd** problem, a data instance is a 3-tuple $\langle i_1, i_2, l \rangle$ where i_1, i_2 are 2 images of digits and l is an integer in $\{0, \ldots, 18\}$ indicating the sum of the 2 digit images. The propositional signature σ in this problem consists of 139 atoms: 19 atoms of the form $sum(i_1, i_2, s)$ for $s \in \{0, \ldots, 18\}$, 20 atoms of the form $digit(i, n)$ for $i \in \{i_1, i_2\}$ and for $n \in \{0, \ldots, 9\}$, and 100 atoms of the form $conj(i_1, n_1, i_2, n_2)$ for $n_1, n_2 \in \{0, \ldots, 9\}$ (denoting the conjunction of $digit(i_1, n_1)$ and $digit(i_2, n_2)$). The CNF for this problem consists of 111 clauses: 19 clauses of the form

$$\neg sum(i_1, i_2, s) \vee \bigvee_{\substack{n_1, n_2 \in \{0, \ldots, 9\} \\ n_1 + n_2 = s}} conj(i_1, n_1, i_2, n_2) \tag{4.13}$$

for $s \in \{0, \ldots, 18\}$, 2 clauses of the form

$$digit(i, 0) \vee \cdots \vee digit(i, 9) \tag{4.14}$$

for $i \in \{i_1, i_2\}$, and 90 clauses of the form

$$\neg digit(i, n_1) \vee \neg digit(i, n_2) \tag{4.15}$$

for $i \in \{i_1, i_2\}$ and for $n_1, n_2 \in \{0, \ldots, 9\}$ such that $n_1 < n_2$. Intuitively, clause (4.13) says that "if the sum of i_1 and i_2 is s, then we should be able to predict the labels n_1, n_2 of i_1, i_2 such that they sum up to s." Clauses (4.14) and (4.15) define the existence and uniqueness constraints on the label of i. Note that clauses (4.14) and (4.15) are not needed if we use $b_p(x)$+iSTE since these constraints will be enforced by the softmax function in the last layer of the neural network, which is widely and inherently used in most neuro-symbolic formalisms.

This CNF can be represented by the matrix $\mathbf{C} \in \{-1, 0, 1\}^{111 \times 139}$. To construct $\mathbf{f}$ and $\mathbf{v}$ for a data instance $\langle i_1, i_2, l \rangle$, the facts $\mathbf{f}$ is simply a vector in $\{0, 1\}^{139}$ with 138 0s and a single 1 for atom $sum(i_1, i_2, l)$; while the prediction $\mathbf{v}$ is a vector in $\{0, 1\}^{139}$ obtained as follows. We (i) feed images i_1,i_2 into the CNN and obtain the outputs $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^{10}$ (consisting of probabilities); (ii) construct $\mathbf{x} \in \mathbb{R}^{139}$ such that its $(10a + b)$-th element is $\mathbf{x}_1[a] \times \mathbf{x}_2[b]$ for $a, b \in \{0, \ldots, 9\}$ and its remaining elements are 0; and (iii) $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

Then, the total loss is defined as

$$\mathcal{L} = \alpha L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + \sum_{\mathbf{x} \in \{\mathbf{x}_1, \mathbf{x}_2\}} \beta L_{bound}(\mathbf{x})$$

where $\alpha = 1$ and $\beta = 0.1$.

4.4.3 add2x2

In **add2x2** problem, a data instance is a 8-tuple $\langle i_1, i_2, i_3, i_4, \ row_1, row_2, col_1, col_2 \rangle$ where i_* are 4 images of digits arranged in the following order in a grid

$$i_1 \ \ i_2$$

$$i_3 \ \ i_4 \ ,$$

and each row_* or col_* is an integer in $\{0, \ldots, 18\}$ denoting the sum of 2 digits on the specified row/column in the above grid. The task is to train a CNN for digit classification

given such weak supervision.

For $o, o' \in \{(i_1, i_2), (i_3, i_4), (i_1, i_3), (i_2, i_4)\}$, and for $r \in \{0, \ldots, 18\}$ the CNF contains the following clause:

$$\neg sum(o, o', r) \vee \bigvee_{\substack{i,j \in \{0,\ldots,9\} \\ i+j=r}} conj(o, i, o', j).$$

This clause can be read as "if the sum of o and o' is r, then o and o' must be some values i and j such that $i + j = r$." This CNF contains $4 \times 19 = 76$ clauses and $76 + 4 \times 10 \times 10 = 476$ atoms (for $sum/3$ and $conj/4$, resp.). This CNF can be represented by the matrix $\mathbf{C} \in \{-1, 0, 1\}^{76 \times 476}$.

To construct $\mathbf{f}$ and $\mathbf{v}$ for a data instance $\langle i_1, i_2, i_3, i_4, row_1, row_2, col_1, col_2 \rangle$, the facts $\mathbf{f}$ is simply a vector in $\{0, 1\}^{476}$ with 472 0s and four 1s for atoms $sum(i_1, i_2, row_1)$, $sum(i_3, i_4, row_2)$, $sum(i_1, i_3, col_1)$, and $sum(i_2, i_4, col_2)$; while the prediction $\mathbf{v}$ is a vector in $\{0, 1\}^{476}$ obtained as follows. We (i) feed images i_1,i_2,i_3,i_4 into the CNN and obtain the outputs $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4 \in \mathbb{R}^{10}$ (consisting of probabilities); (ii) construct $\mathbf{x} \in \mathbb{R}^{476}$ as the concatenation of $\langle \mathbf{v}_1, \mathbf{v}_2, \mathbf{v}_3, \mathbf{v}_4, \{0\}^{76} \rangle$ where

$$\mathbf{v}_1 = \mathbf{x}_1^T \cdot \mathbf{x}_2, \qquad \mathbf{v}_2 = \mathbf{x}_3^T \cdot \mathbf{x}_4,$$
$$\mathbf{v}_3 = \mathbf{x}_1^T \cdot \mathbf{x}_3, \qquad \mathbf{v}_4 = \mathbf{x}_2^T \cdot \mathbf{x}_4;$$

and (iii) $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

Then, the total loss is defined as

$$\mathcal{L} = \alpha L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + \sum_{\mathbf{x} \in \{\mathbf{x}_1, \ldots, \mathbf{x}_4\}} \beta L_{bound}(\mathbf{x})$$

where $\alpha = 1$ and $\beta = 0.1$.

4.4.4 apply2x2

In **apply2x2** problem, a data instance is a 11-tuple $\langle d_1, d_2, d_3, o_{11}, o_{12}, o_{21}, o_{22}, row_1, row_2, col_1, col_2 \rangle$ where d_* are digits in $\{0, \ldots, 9\}$, o_* are 4 images of operators in $\{+, -, \times\}$

arranged in the following order in a grid

$$o_{11} \;\; o_{12}$$

$$o_{21} \;\; o_{22} \;,$$

and each row_* or col_* is an integer denoting the value of the formula (e.g., $(4 \times 7) - 9$)

$$(d_1 \; o_1 \; d_2) \; o_2 \; d_3 \tag{4.16}$$

where $(o_1, o_2) \in \{(o_{11}, o_{12}), (o_{21}, o_{22}), (o_{11}, o_{21}), (o_{12}, o_{22})\}$ denotes the two operators on the specified row/column in the above grid. The task is to train a CNN for digit classification given such weak supervision.

We construct a CNF to relate formula (4.16) and its value and will apply the CNF loss for $(o_1, o_2) \in \{ (o_{11}, o_{12}), (o_{21}, o_{22}), (o_{11}, o_{21}), (o_{12}, o_{22})\}$.

For $d_1, d_2, d_3 \in \{0, \ldots, 10\}$, and for all possible r such that $(d_1 \; Op_1 \; d_2) \; Op_2 \; d_3 = r$ for some $Op_1, Op_2 \in \{+, -, \times\}$, the CNF contains the following clause:

$$\neg apply(d_1, o_1, d_2, o_2, d_3, r) \vee \bigvee_{\substack{Op_1, Op_2 \in \{+,-,\times\} \\ (d_1 \;\; Op_1 \;\; d_2) \;\; Op_2 \;\; d_3 = r}} (operators(o_1, Op_1, o_2, Op_2)).$$

This clause can be read as "if the result is r after applying o_1 and o_2 to the three digits, then o_1 and o_2 must be some values Op_1 and Op_2 such that $(d_1 \; Op_1 \; d_2) \; Op_2 \; d_3 = r$." This CNF contains 10597 clauses and 10606 atoms and can be represented by the matrix $\mathbf{C} \in \{-1, 0, 1\}^{10597 \times 10606}$.

Given a data instance $\langle d_1, \; d_2, \; d_3, \; o_{11}, \; o_{12}, \; o_{21}, \; o_{22}, \; row_1, \; row_2, \; col_1, \; col_2 \rangle$, we construct $\mathbf{v}_i, \mathbf{f}_i \in \{0, 1\}^{10606}$ for $i \in \{1, \ldots, 4\}$, one for each $\langle o_1, o_2, r \rangle \in \{\langle o_{11}, o_{12}, row_1 \rangle, \langle o_{21}, o_{22}, row_2 \rangle, \langle o_{11}, o_{21}, col_1 \rangle, \langle o_{12}, o_{22}, col_2 \rangle\}$. The detailed steps to construct $\mathbf{f}$ and $\mathbf{v}$ for $\langle o_1, o_2, r \rangle$ is as follows.

First, the facts $\mathbf{f}$ is simply a vector in $\{0, 1\}^{10606}$ with 10605 0s and a single 1 for atom $apply(d_1, o_1, d_2, o_2, d_3, r)$. Second, the prediction $\mathbf{v}$ is a vector in $\{0, 1\}^{10606}$ obtained

as follows. We (i) feed images o_1,o_2 into the CNN and obtain the outputs $\mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^3$ (consisting of probabilities); (ii) construct $\mathbf{x} \in \mathbb{R}^{10606}$ such that its $(3a + b)$-th element is $\mathbf{x}_1[a] \times \mathbf{x}_2[b]$ for $a, b \in \{0, \ldots, 2\}$ and its remaining elements are 0; and (iii) $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

Then, the total loss is defined as

$$\mathcal{L} = \sum_{i \in \{1,\ldots,4\}} \alpha L_{cnf}(\mathbf{C}, \mathbf{v}_i, \mathbf{f}_i) + \sum_{\mathbf{x} \in \{\mathbf{x}_1,\ldots,\mathbf{x}_4\}} \beta L_{bound}(\mathbf{x})$$

where $\alpha = 1$ and $\beta = 0.1$.

Note that, to avoid unnecessary computation, we filter out the clauses in C (by masking rows in $\mathbf{C}$) that are already evaluated as TRUE given the labels in F (encoded in $\mathbf{f}$).

4.4.5 member(n)

We take **member(3)** problem as an example. In **member(3)** problem, a data instance is a 5-tuple $\langle i_1, i_2, i_3, d, l \rangle$ where i_1, i_2, i_3 are 3 images of digits, d is a digit in $\{0, \ldots, 9\}$, and l is an integer in $\{0, 1\}$ indicating whether d appears in the set of digit images. The task is to train a CNN for digit classification given such weak supervision. The CNF for this problem consists of the 2 kinds of clauses in table 4.6.

Table 4.6: Clauses in the CNF for member(3) Problem

Clause	Reading
$\neg in(d, 1) \vee digit(i_1, d) \vee digit(i_2, d) \vee digit(i_3, d)$ (for $d \in \{0, \ldots, 9\}$)	if d appears in the 3 images, then i_1 or i_2 or i_3 must be digit d
$\neg in(d, 0) \vee \neg digit(i, d)$ (for $d \in \{0, \ldots, 9\}$ and $i \in \{i_1, i_2, i_3\}$)	if d doesn't appear in the 3 images, then each image i cannot be digit d

This CNF contains $10 + 10 \times 3 = 40$ clauses and $3 \times 10 + 2 \times 10 = 50$ atoms for $digit/2$ and $in/2$ respectively. According to the definition, we can construct the matrix $\mathbf{C} \in \{-1, 0, 1\}^{40 \times 50}$ where each row represents a clause. For instance, the row for the clause

$\neg in(5,1) \vee digit(i_1,5) \vee digit(i_2,5) \vee digit(i_3,5)$ is a row vector in $\{-1,0,1\}^{1\times 50}$ containing 46 0s, a single -1 for atom $in(5,1)$, and three 1s for atoms $digit(i_1,5)$, $digit(i_2,5)$, $digit(i_3,5)$.

To construct $\mathbf{f}$ and $\mathbf{v}$ for a data instance $\langle i_1,i_2,i_3,d,l\rangle$, the facts $\mathbf{f}$ is simply a vector in $\{0,1\}^{50}$ with 49 0s and a single 1 for atom $in(d,l)$; while the prediction $\mathbf{v}$ is a vector in $\{0,1\}^{50}$ obtained as follows. We (i) feed images i_1,i_2,i_3 into the CNN and obtain the NN outputs $\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3 \in \mathbb{R}^{10}$ consisting of probabilities, (ii) construct $\mathbf{x} \in \mathbb{R}^{50}$ by concatenating $\mathbf{x}_1,\mathbf{x}_2,\mathbf{x}_3$ and the vector $\{0\}^{20}$, and (iii) $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

The total loss function is

$$\mathcal{L} = \alpha L_{cnf}(\mathbf{C},\mathbf{v},\mathbf{f}) + \sum_{\mathbf{x}\in\{\mathbf{x}_1,\ldots,\mathbf{x}_3\}} \beta L_{bound}(\mathbf{x})$$

where $\alpha = 1$ and $\beta = 0.1$.

4.4.6 Sudoku

We use a typical CNF for 9×9 Sudoku problem. The CNF is defined on a propositional signature $\sigma = \{a(R,C,N) \mid R,C,N \in \{1,\ldots,9\}\}$ where $a(R,C,N)$ represents "digit N is assigned at row R column C". The CNF consists of the following $1 + \binom{9}{2} = 37$ clauses for each of the $4 \times 9 \times 9 = 324$ different sets A of atoms

$$\bigvee_{p\in A} p$$

$$\neg p_i \vee \neg p_j \quad (\text{for } p_i,p_j \in A \text{ and } i<j)$$

where the $4 \times 9 \times 9$ definitions of A can be split into the following 4 categories, each consisting of 9×9 definitions.

1. (UEC on row indices)

   For $C,N \in \{1,\ldots,9\}$, A is the set of atoms $\{a(1,C,N),\ldots,a(9,C,N)\}$.

2. (UEC on column indices)

   For $R, N \in \{1, \ldots, 9\}$, A is the set of atoms $\{a(R, 1, N), \ldots, a(R, 9, N)\}$.

3. (UEC on 9 values in each cell)

   For $R, C \in \{1, \ldots, 9\}$, A is the set of atoms $\{a(R, C, 1), \ldots, a(R, C, 9)\}$.

4. (Optional: UEC on 9 cells in the same 3×3 box)

   For $B, N \in \{1, \ldots, 9\}$, A is the set of atoms $\{a(R_1, C_1, N), \ldots, a(R_9, C_9, N)\}$ such that the 9 cells (R_i, C_i) for $i \in \{1 \ldots, 9\}$ are the 9 cells in the B-th box in the 9×9 grid for value N. Note that the clauses in bullet 4 are optional under the setting $b_p(x)$+iSTE since they are already enforced by the softmax function used in the last layer to turn NN output to probabilities.

   This CNF can be represented by a matrix $\mathbf{C} \in \{-1, 0, 1\}^{8991 \times 729}$. The dataset in the CNN experiments consists of 70k data instances, 20% supervised for testing, and 80% unsupervised for training. Each unsupervised data instance is a single vector $\mathbf{q} \in \{0, 1, \ldots, 9\}^{81}$ representing a 9×9 Sudoku board (0 denotes an empty cell). The non-zero values in $\mathbf{q}$ are treated as atomic facts F and we construct the matrix $\mathbf{F} \in \{0, 1\}^{81 \times 9}$ such that, for $i \in \{1, \ldots, 81\}$, the i-th row $\mathbf{F}[i, :]$ is the vector $\{0\}^9$ if $\mathbf{q}[i] = 0$ and is the one-hot vector for $\mathbf{q}[i]$ if $\mathbf{q}[i] \neq 0$. Then, the vector $\mathbf{f} \in \{0, 1\}^{729}$ is simply the flattened version of $\mathbf{F}$. We feed $\mathbf{q}$ into the CNN and obtain the output $\mathbf{x} \in \mathbb{R}^{729}$ consisting of probabilities. The prediction $\mathbf{v} \in \{0, 1\}^{729}$ is obtained as $\mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$.

   Then, the total loss function $\mathcal{L}$ used to train the CNN for Sudoku is

$$\mathcal{L} = \alpha L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) + \beta L_{bound}(\mathbf{x})$$

   where $\alpha = 1$ and $\beta = 0.1$.

4.5 Proofs

4.5.1 Proof of Proposition 2

Proposition 2 When K approaches ∞ and $|g(x)| \leq c$ for a constant c, the value of $\widetilde{b}^K(x)$ converges to $b(x)$.

$$\lim_{K \to \infty} \widetilde{b}^K(x) = b(x)$$

The gradient $\frac{\partial \widetilde{b}^K(x)}{\partial x}$, whenever defined, is exactly the iSTE of $\frac{\partial b(x)}{\partial x}$ if $g(x) = 1$, or the sSTE of $\frac{\partial b(x)}{\partial x}$ if

$$g(x) = \begin{cases} 1 & \text{if } -1 \leq x \leq 1 \\ 0 & \text{otherwise.} \end{cases}$$

[Remark]: Proposition 2 in our paper is similar to proposition 1 in (Kim *et al.*, 2020) but not the same. For the value of $\widetilde{b}^K(x)$, we don't have a condition that $g'(x)$ should be bounded. For the gradient of $\widetilde{b}^K(x)$, we have a stronger statement specific for STEs and don't have the condition for K approaching ∞.

Proof. Recall the definition of $\widetilde{b}^K(x)$

$$\widetilde{b}^K(x) = b(x) + s^K(x)g(x)$$

where K is a constant; $s^K(x) = \frac{Kx - \lfloor Kx \rfloor}{K}$ is a gradient tweaking function whose value is less than $\frac{1}{K}$ and whose gradient is always 1 whenever differentiable; and $g(x)$ is a gradient shaping function.

[First], we will prove $\lim_{K \to \infty} \widetilde{b}^K(x) = b(x)$. Since $\widetilde{b}^K(x) = b(x) + s^K(x)g(x)$, it's equivalent to proving

$$\lim_{K \to \infty} s^K(x)g(x) = 0$$

Since $s^K(x) = \frac{Kx - \lfloor Kx \rfloor}{K}$ and $0 \leq Kx - \lfloor Kx \rfloor \leq 1$, we know $0 \leq s^K(x) \leq \frac{1}{K}$. Since $|g(x)| \leq c$ where c is a constant, $-\frac{c}{K} \leq s^K(x)g(x) \leq \frac{c}{K}$. Thus $0 \leq \lim_{K\to\infty} s^K(x)g(x) \leq 0$, and consequently, $\lim_{K\to\infty} s^K(x)g(x) = 0$.

[Second], we will prove

- when $g(x) = 1$ and $s(x) = x$ (i.e., under iSTE),

$$\frac{\partial \widetilde{b}^K(x)}{\partial x} = \begin{cases} \frac{\partial s(x)}{\partial x} & \text{(if } Kx \neq \lfloor Kx \rfloor\text{)} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Let's prove some general properties of the gradients. Since $s(x) = x$, $g(x) = 1$, and $s^K(x) = \frac{Kx - \lfloor Kx \rfloor}{K}$,

- $\frac{\partial s(x)}{\partial x} = 1$, $\frac{\partial g(x)}{\partial x} = 0$, and
- $\frac{\partial s^K(x)}{\partial x} = 1$ whenever differentiable (i.e., whenever $Kx \neq \lfloor Kx \rfloor$).

Then,

$$\begin{aligned} \frac{\partial \widetilde{b}^K(x)}{\partial x} &= \frac{\partial (b(x) + s^K(x)g(x))}{\partial x} \\ &= \frac{\partial (s^K(x) \times g(x))}{\partial x} \\ &= \frac{\partial s^K(x)}{\partial x} \\ &= \begin{cases} 1 & \text{(if } Kx \neq \lfloor Kx \rfloor\text{)} \\ \text{undefined} & \text{otherwise.} \end{cases} \end{aligned}$$

[Third], we will prove

- when $s(x) = min(max(x, -1), 1)$, and $g(x) = 1$ if $-1 \leq x \leq 1$ and $g(x) = 0$ otherwise (i.e., under sSTE),

$$\frac{\partial \widetilde{b}^K(x)}{\partial x} = \begin{cases} \frac{\partial s(x)}{\partial x} & \text{(if } Kx \neq \lfloor Kx \rfloor\text{)} \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Let's prove some general properties of the gradients. Since $s(x) = min(max(x, -1), 1)$, $g(x) = 1$ if $-1 \leq x \leq 1$ and $g(x) = 0$ otherwise, and $s^K(x) = \frac{Kx - \lfloor Kx \rfloor}{K}$,

- $\frac{\partial s(x)}{\partial x} = 1$ if $-1 \leq x \leq 1$ and $\frac{\partial s(x)}{\partial x} = 0$ otherwise,
- $\frac{\partial g(x)}{\partial x} = 0$, and
- $\frac{\partial s^K(x)}{\partial x} = 1$ whenever differentiable (i.e., whenever $Kx \neq \lfloor Kx \rfloor$).

Then,

$$\begin{aligned}
\frac{\partial \widetilde{b}^K(x)}{\partial x} &= \frac{\partial (b(x) + s^K(x) g(x))}{\partial x} \\
&= \frac{\partial (s^K(x) \times g(x))}{\partial x} \\
&= g(x) \times \frac{\partial s^K(x)}{\partial x} + s^K(x) \times \frac{\partial g(x)}{\partial x} \\
&= g(x) \times \frac{\partial s^K(x)}{\partial x} \\
&= \begin{cases} g(x) & \text{(if } Kx \neq \lfloor Kx \rfloor\text{)} \\ \text{undefined} & \text{otherwise.} \end{cases} \\
&= \begin{cases} \frac{\partial s(x)}{\partial x} & \text{(if } Kx \neq \lfloor Kx \rfloor\text{)} \\ \text{undefined} & \text{otherwise.} \end{cases}
\end{aligned}$$

∎

4.5.2 Proof of Proposition 3

Proposition 3 Given a CNF theory C, a set F of atoms, and a truth assignment v such that $v \models F$, let $\mathbf{C}, \mathbf{f}, \mathbf{v}$ denote their matrix/vector representations, respectively. Let $C_{deduce} \subseteq C$ denote the set of Horn clauses H in C such that all but one literal in H are of the form $\neg p$ where $p \in F$. Then

- the minimum values of L_{deduce}, L_{unsat}, L_{sat}, and $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ are 0;

- $v \models C_{deduce}$ iff L_{deduce} is 0;
- $v \models C$ iff L_{unsat} is 0 iff $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ is 0.

Proof. Recall the definition of L_{cnf}

$$\begin{aligned}
\mathbf{L}_f &= \mathbf{C} \odot \mathbf{f} \\
\mathbf{L}_v &= \mathbb{1}_{\{1\}}(\mathbf{C}) \odot \mathbf{v} + \mathbb{1}_{\{-1\}}(\mathbf{C}) \odot (1 - \mathbf{v}) \\
\mathbf{deduce} &= \mathbb{1}_{\{1\}}\Big(sum(\mathbf{C} \odot \mathbf{C}) - sum(\mathbb{1}_{\{-1\}}(\mathbf{L}_f))\Big) \\
\mathbf{unsat} &= prod(1 - \mathbf{L}_v) \\
\mathbf{keep} &= sum(\mathbb{1}_{\{1\}}(\mathbf{L}_v) \odot (1 - \mathbf{L}_v) + \mathbb{1}_{\{0\}}(\mathbf{L}_v) \odot \mathbf{L}_v) \\
L_{deduce} &= sum(\mathbf{deduce} \odot \mathbf{unsat}) \\
L_{unsat} &= avg(\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat}) \\
L_{sat} &= avg(\mathbb{1}_{\{0\}}(\mathbf{unsat}) \odot \mathbf{keep})
\end{aligned}$$

$$L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) = L_{deduce} + L_{unsat} + L_{sat}$$

and the definitions of $\mathbf{C}, \mathbf{f}, \mathbf{v}$ below.

- the matrix $\mathbf{C}$ is in $\{-1, 0, 1\}^{m \times n}$ such that $\mathbf{C}[i, j]$ is 1 (-1, resp.) if p_j ($\neg p_j$, resp.) belongs to the i-th clause, and is 0 if neither p_j nor $\neg p_j$ belongs to the clause;
- the vector $\mathbf{f}$ is in $\{0, 1\}^n$ to represent F such that $\mathbf{f}[j]$ is 1 if $p_j \in F$ and is 0 otherwise; and
- the vector $\mathbf{v}$ is in $\{0, 1\}^n$ to represent v such that $\mathbf{v}[j]$ is 1 if $v(p_j) = \text{TRUE}$, and is 0 if $v(p_j) = \text{FALSE}$.

We will prove each bullet in Proposition 3 as follows.

1. **[First]**, we will prove

- $\mathbf{L}_f$ is the matrix in $\{-1, 0, 1\}^{m \times n}$ such that (i) $\mathbf{L}_f[i, j] = 1$ iff clause i contains literal p_j and $p_j \in F$; and (ii) $\mathbf{L}_f[i, j] = -1$ iff clause i contains literal $\neg p_j$ and $p_j \in F$.
- $\mathbf{L}_v$ is the matrix in $\{0, 1\}^{m \times n}$ such that $\mathbf{L}_v[i, j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v.

According to the definition, $\mathbf{L}_f[i, j] = \mathbf{C}[i, j] \times \mathbf{f}[j]$. Since $\mathbf{f}[j] \in \{0, 1\}$, we have $\mathbf{L}_f[i, j] = 1$ iff "$\mathbf{C}[i, j] = 1$ and $\mathbf{f}[j] = 1$", and according to the definition of $\mathbf{C}$ and $\mathbf{f}$, we have $\mathbf{L}_f[i, j] = 1$ iff "clause i contains literal p_j and $p_j \in F$". Similarly, we have $\mathbf{L}_f[i, j] = -1$ iff "$\mathbf{C}[i, j] = -1$ and $\mathbf{f}[j] = 1$" iff "clause i contains literal $\neg p_j$ and $p_j \in F$".

According to the definition, $\mathbf{L}_v[i, j] = \mathbb{1}_{\{1\}}(\mathbf{C})[i, j] \times \mathbf{v}[j] + \mathbb{1}_{\{-1\}}(\mathbf{C})[i, j] \times (1 - \mathbf{v}[j])$. Since $\mathbb{1}_{\{1\}}(\mathbf{C})[i, j]$ and $\mathbb{1}_{\{-1\}}(\mathbf{C})[i, j]$ cannot be 1 at the same time and $\mathbf{v}[j] \in \{0, 1\}$, we have $\mathbf{L}_v[i, j] = 1$ iff "$\mathbf{C}[i, j] = 1$ and $\mathbf{v}[j] = 1$" or "$\mathbf{C}[i, j] = -1$ and $\mathbf{v}[j] = 0$". According to the definition of $\mathbf{C}$ and $\mathbf{v}$, we have $\mathbf{L}_v[i, j] = 1$ iff "clause i contains literal p_j, which evaluates to TRUE under v" or "clause i contains literal $\neg p_j$, which evaluates to TRUE under v".

[Second], we will prove

- $\mathbf{deduce}$ is the vector in $\{0, 1\}^m$ such that $\mathbf{deduce}[i] = 1$ iff clause i has all but one literal of the form $\neg p_j$ such that $p_j \in F$.
- $\mathbf{unsat}$ is the vector in $\{0, 1\}^m$ such that $\mathbf{unsat}[i] = 1$ iff clause i evaluates to FALSE under v.
- $\mathbf{keep}$ is the vector $\{0\}^m$.

From the definition of $\mathbf{C}$, the matrix $\mathbf{C} \odot \mathbf{C}$ is in $\{0, 1\}^{m \times n}$ such that the element at position (i,j) is 1 iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j. Since

$sum(X)$ computes the sum of elements in each row of matrix X, for $i \in \{1, \ldots, m\}$ and $k \in \{1, \ldots, n\}$, $sum(\mathbf{C} \odot \mathbf{C})[i] = k$ iff clause i contains k literals. Recall that we proved that $\mathbf{L}_f[i, j] = -1$ iff "clause i contains literal $\neg p_j$ and $p_j \in F$". Consequently, $\mathbb{1}_{\{-1\}}(\mathbf{L}_f)$ is the matrix in $\{0, 1\}^{m \times n}$ such that $\mathbb{1}_{\{-1\}}(\mathbf{L}_f)[i, j] = 1$ iff "clause i contains literal $\neg p_j$ and $p_j \in F$". As a result, $sum(\mathbf{C} \odot \mathbf{C}) - sum(\mathbb{1}_{\{-1\}}(\mathbf{L}_f))$ is the vector in $\{0, \ldots, n\}^m$ such that its i-th element is 1 iff "clause i contains all but one literal of the form $\neg p_j$ such that $p_j \in F$". Thus $\mathbf{deduce}$ is the vector in $\{0, 1\}^m$ such that $\mathbf{deduce}[i] = 1$ iff "clause i has all but one literal of the form $\neg p_j$ such that $p_j \in F$".

Since $prod(X)$ computes the product of elements in each row of matrix X, for $i \in \{1, \ldots, m\}$, $\mathbf{unsat}[i] = \prod_{j \in \{1, \ldots, n\}} (1 - \mathbf{L}_v[i, j])$. Recall that we proved that $\mathbf{L}_v$ is the matrix in $\{0, 1\}^{m \times n}$ such that $\mathbf{L}_v[i, j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v. Thus $\mathbf{unsat}[i] \in \{0, 1\}$ and $\mathbf{unsat}[i] = 1$ iff "$\mathbf{L}_v[i, j] = 0$ for $j \in \{1, \ldots, n\}$" iff "for $j \in \{1, \ldots, n\}$, clause i either does not contain a literal for atom p_j or contains a literal for atom p_j while this literal evaluates to FALSE under v" iff "clause i evaluates to FALSE under v". In other words, $\mathbf{unsat}$ is the vector in $\{0, 1\}^m$ such that $\mathbf{unsat}[i] = 1$ iff clause i evaluates to FALSE under v.

Since $\mathbf{L}_v$ is the matrix in $\{0, 1\}^{m \times n}$, for $i \in \{1, \ldots, m\}$ and $j \in \{1, \ldots, n\}$, $\mathbb{1}_{\{1\}}(\mathbf{L}_v)[i, j] = 1$ iff $\mathbf{L}_v[i, j] = 1$ iff $(1 - \mathbf{L}_v[i, j]) = 0$. Thus $\mathbb{1}_{\{1\}}(\mathbf{L}_v) \odot (1 - \mathbf{L}_v)$ is the matrix $\{0\}^{m \times n}$ of all zeros. Similarly, $\mathbb{1}_{\{0\}}(\mathbf{L}_v) \odot \mathbf{L}_v)$ is also the matrix $\{0\}^{m \times n}$. As a result, $\mathbf{keep}$ is the vector $\{0\}^m$.

[Third], we will prove

- L_{deduce} is an integer in $\{0, \ldots, m\}$ such that $L_{deduce} = k$ iff there are k clauses in C_{deduce} that are evaluated to FALSE under v.

- L_{unsat} is a number in $\{0, \frac{1}{m}, \ldots, \frac{m}{m}\}$ such that $L_{unsat} = \frac{k}{m}$ iff there are k clauses that are evaluated to FALSE under v.
- L_{sat} is 0.

Recall that we proved that $\mathbf{deduce}$ is the vector in $\{0,1\}^m$ such that $\mathbf{deduce}[i] = 1$ iff clause i has all but one literal of the form $\neg p_j$ such that $p_j \in F$; and $\mathbf{unsat}$ is the vector in $\{0,1\}^m$ such that $\mathbf{unsat}[i] = 1$ iff clause i evaluates to FALSE under v. According to the definition of C_{deduce}, $\mathbf{deduce} \odot \mathbf{unsat}$ is the vector in $\{0,1\}^m$ such that its i-th element is 1 iff clause i is in C_{deduce} and evaluates to FALSE under v. As a result, L_{deduce} is an integer in $\{0, \ldots, m\}$ such that $L_{deduce} = k$ iff there are k clauses in C_{deduce} that are evaluated as FALSE under v.

Since $\mathbf{unsat}$ is the vector in $\{0,1\}^m$ such that $\mathbf{unsat}[i] = 1$ iff clause i evaluates to FALSE under v, and since $\mathbf{unsat}[i] = 1$ iff $\mathbb{1}_{\{1\}}(\mathbf{unsat})[i] = 1$, we know the i-th element in $\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat}$ is 1 iff clause i evaluates to FALSE under v. $L_{unsat} = avg(\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat})$ is a number in $\{0, \frac{1}{m}, \ldots, \frac{m}{m}\}$ such that $L_{unsat} = \frac{k}{m}$ iff there are k clauses that are evaluated as FALSE under v.

Recall that we proved that $\mathbf{keep}$ is the vector $\{0\}^m$. Thus $\mathbb{1}_{\{0\}}(\mathbf{unsat}) \odot \mathbf{keep}$ is the vector $\{0\}^m$. Thus L_{sat} is 0.

[Fourth], we will prove

- the minimum values of L_{deduce}, L_{unsat}, L_{sat}, $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ are 0.

Recall that we proved that L_{deduce} is an integer in $\{0, \ldots, m\}$, L_{unsat} is a number in $\{0, \frac{1}{m}, \ldots, \frac{m}{m}\}$, and L_{sat} is 0. It's obvious that the minimum values of L_{deduce}, L_{unsat}, and L_{sat} are 0. Since (i) $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) = L_{deduce} + L_{unsat} + L_{sat}$, (ii) $L_{deduce} = 0$ when all clauses in C_{deduce} are evaluated to TRUE under v, and (iii) $L_{unsat} = 0$ when all clauses in C are evaluated to TRUE under v, the minimum value of $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$

is 0 and is achieved when all clauses in C are evaluated to TRUE under v.

2. We will prove

   - $v \models C_{deduce}$ iff L_{deduce} is 0.

   Recall that we proved that L_{deduce} is an integer in $\{0, \ldots, m\}$ such that $L_{deduce} = k$ iff there are k clauses in C_{deduce} that are evaluated as FALSE under v. Then L_{deduce} is 0 iff "there is no clause in C_{deduce} that evaluates to FALSE under v" iff "every clause in C_{deduce} evaluates to TRUE under v" iff $v \models C_{deduce}$.

3. We will prove

   - $v \models C$ iff L_{unsat} is 0 iff $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ is 0.

   Recall that we proved that L_{unsat} is a number in $\{0, \frac{1}{m}, \ldots, \frac{m}{m}\}$ such that $L_{unsat} = \frac{k}{m}$ iff there are k clauses that are evaluated as FALSE under v. Then L_{unsat} is 0 iff "there is no clause in C that evaluates to FALSE under v" iff $v \models C$.

   Assume L_{unsat} is 0. Then "there is no clause in C that evaluates to FALSE under v". Consequently, "there is no clause in C_{deduce} that is evaluated to FALSE under v". Recall that we proved that L_{deduce} is an integer in $\{0, \ldots, m\}$ such that $L_{deduce} = k$ iff there are k clauses in C_{deduce} that are evaluated as FALSE under v. Then L_{deduce} is 0. Since L_{sat} is 0, $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ is 0.

   Assume $L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f})$ is 0, which is the minimum value L_{cnf} can take. It is easy to see that L_{unsat} must be 0.

∎

4.5.3 *Proof of Proposition 4*

Proposition 4 Given a CNF theory C of m clauses and n atoms and a set F of atoms such that $C \cup F$ is satisfiable, let $\mathbf{C}, \mathbf{f}$ denote their matrix/vector representations, respectively. Given a neural network output $\mathbf{x} \in [0,1]^n$ denoting probabilities, we construct $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$ and a truth assignment v such that $v(p_j) = \text{TRUE}$ if $\mathbf{v}[j]$ is 1, and $v(p_j) = \text{FALSE}$ if $\mathbf{v}[j]$ is 0. Let $C_{deduce} \subseteq C$ denote the set of Horn clauses H in C such that all but one literal in H are of the form $\neg p$ and $p \in F$. Then, for any $j \in \{1, \ldots, n\}$,

1. if $p_j \in F$, all of $\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]}$, $\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]}$, and $\frac{\partial L_{sat}}{\partial \mathbf{x}[j]}$ are zeros;

2. if $p_j \notin F$,

$$\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \begin{cases} -c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } p_j; \\ c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } \neg p_j; \\ 0 & \text{otherwise;} \end{cases}$$

$$\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \frac{c_2 - c_1}{m}$$

$$\frac{\partial L_{sat}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \begin{cases} -\frac{c_3}{m} & \text{if } v \models p_j, \\ \frac{c_3}{m} & \text{if } v \not\models p_j. \end{cases}$$

where $\overset{iSTE}{\approx}$ stands for the equivalence of gradients under iSTE; c_1 (and c_2, resp.) is the number of clauses in C that are not satisfied by v and contain p_j (and $\neg p_j$, resp.); c_3 is the number of clauses in C that are satisfied by v and contain p_j or $\neg p_j$.

Proof. We will prove each bullet in Proposition 4 as follows.

1. Take any $k \in \{1, \ldots, n\}$, let's focus on $\mathbf{x}[k]$ and compute the gradient of $L \in \{L_{deduce}, L_{unsat}, L_{sat}\}$ to it with iSTE. According to the chain rule and since $\frac{\partial \mathbf{v}[i]}{\partial b_p(\mathbf{x})[j]} = 0$ for $i \neq j$, we have

$$\frac{\partial L}{\partial \mathbf{x}[k]} = \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial \mathbf{v}[k]}{\partial b_p(\mathbf{x}[k])} \times \frac{\partial b_p(\mathbf{x}[k])}{\partial \mathbf{x}[k]}.$$

Under iSTE, the last term $\frac{\partial b_p(\mathbf{x}[k])}{\partial \mathbf{x}[k]}$ is replaced with $\frac{\partial s(\mathbf{x}[k])}{\partial \mathbf{x}[k]} = \frac{\partial \mathbf{x}[k]}{\partial \mathbf{x}[k]} = 1$. Thus

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{x}[k]} =& \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial \mathbf{v}[k]}{\partial b_p(\mathbf{x}[k])} \times \frac{\partial b_p(\mathbf{x}[k])}{\partial \mathbf{x}[k]} \\
\overset{iSTE}{\approx}& \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial \mathbf{v}[k]}{\partial b_p(\mathbf{x}[k])} \text{ (under iSTE)} \\
=& \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial (\mathbf{f}[k] + \mathbb{1}_{\{0\}}(\mathbf{f}[k]) \times b_p(\mathbf{x}[k]))}{\partial b_p(\mathbf{x}[k])} \\
=& \begin{cases} \frac{\partial L}{\partial \mathbf{v}[k]} & \text{if } \mathbf{f}[k] = 0, \\ 0 & \text{if } \mathbf{f}[k] = 1. \end{cases}
\end{aligned}$$

Since $\mathbf{f}[k] = 1$ iff $p_k \in F$, if $p_k \in F$, then all of $\frac{\partial L_{deduce}}{\partial \mathbf{x}[k]}$, $\frac{\partial L_{unsat}}{\partial \mathbf{x}[k]}$, and $\frac{\partial L_{sat}}{\partial \mathbf{x}[k]}$ are zeros.

2. Recall the definition of L_{cnf}

$$\begin{aligned}
\mathbf{L}_f &= \mathbf{C} \odot \mathbf{f} \\
\mathbf{L}_v &= \mathbb{1}_{\{1\}}(\mathbf{C}) \odot \mathbf{v} + \mathbb{1}_{\{-1\}}(\mathbf{C}) \odot (1 - \mathbf{v}) \\
\mathbf{deduce} &= \mathbb{1}_{\{1\}}\Big(sum(\mathbf{C} \odot \mathbf{C}) - sum(\mathbb{1}_{\{-1\}}(\mathbf{L}_f))\Big) \\
\mathbf{unsat} &= prod(1 - \mathbf{L}_v) \\
\mathbf{keep} &= sum(\mathbb{1}_{\{1\}}(\mathbf{L}_v) \odot (1 - \mathbf{L}_v) + \mathbb{1}_{\{0\}}(\mathbf{L}_v) \odot \mathbf{L}_v) \\
L_{deduce} &= sum(\mathbf{deduce} \odot \mathbf{unsat}) \\
L_{unsat} &= avg(\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat}) \\
L_{sat} &= avg(\mathbb{1}_{\{0\}}(\mathbf{unsat}) \odot \mathbf{keep}) \\
L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) &= L_{deduce} + L_{unsat} + L_{sat}
\end{aligned}$$

We know $p_k \notin F$ iff $\mathbf{f}[k] = 0$. As proved in the first bullet, for $L \in \{L_{deduce}, L_{unsat}, L_{sat}\}$, if $p_k \notin F$, then $\frac{\partial L}{\partial \mathbf{x}[k]} \overset{iSTE}{\approx} \frac{\partial L}{\partial \mathbf{v}[k]}$. We further analyze the value of $\frac{\partial L}{\partial \mathbf{v}[k]}$ for each L under the condition that $\mathbf{f}[k] = 0$.

[L_{deduce}] According to the definition,

$$\begin{aligned}
L_{deduce} &= \sum_{i\in\{1,\dots,m\}} \Big(\mathbf{deduce}[i] \times \mathbf{unsat}[i]\Big) \\
&= \sum_{i\in\{1,\dots,m\}} \Big(\mathbf{deduce}[i] \times \prod_{j\in\{1,\dots,n\}} (1 - \mathbf{L}_v[i,j])\Big) \\
\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} &= \sum_{i\in\{1,\dots,m\}} \frac{\partial\Big(\mathbf{deduce}[i] \times \prod_{j\in\{1,\dots,n\}} (1 - \mathbf{L}_v[i,j])\Big)}{\partial \mathbf{v}[k]} \\
&= \sum_{i\in\{1,\dots,m\}} \Big(\frac{\partial \mathbf{deduce}[i]}{\partial \mathbf{v}[k]} \times \prod_{j\in\{1,\dots,n\}} (1 - \mathbf{L}_v[i,j]) + \\
&\qquad\qquad \mathbf{deduce}[i] \times \frac{\partial \prod_{j\in\{1,\dots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}\Big)
\end{aligned}$$

Since $\mathbf{deduce}$ is the result of an indicator function, $\frac{\partial \mathbf{deduce}[i]}{\partial \mathbf{v}[k]} = 0$. Then,

$$\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} = \sum_{i\in\{1,\dots,m\}} \Big(\mathbf{deduce}[i] \times \frac{\partial \prod_{j\in\{1,\dots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}\Big).$$

Let $U \subseteq \{1,\dots,m\}$ denote the set of indices of all clauses in C_{deduce}. Since $\mathbf{deduce}[i] = 1$ iff $i \in U$,

$$\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} = \sum_{i\in U} \Big(\frac{\partial \prod_{j\in\{1,\dots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}\Big).$$

Let $G_{i,k}$ denote $\frac{\partial \prod_{j\in\{1,\dots,n\}} (1-\mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}$. Then

$$\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} = \sum_{i\in U} G_{i,k}.$$

Let's analyze the value of $G_{i,k}$ where $i \in U$ and $k \in \{1, \ldots, n\}$ such that $\mathbf{f}[k] = 0$. According to the product rule below,

$$\frac{d}{dx}\left[\prod_{i=1}^{k} f_i(x)\right] = \left(\prod_{i=1}^{k} f_i(x)\right)\left(\sum_{i=1}^{k} \frac{f_i'(x)}{f_i(x)}\right)$$

we have

$$\begin{aligned} G_{i,k} =& \frac{\partial \prod\limits_{j\in\{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \\ =& \left(\prod_{j\in\{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])\right) \times \sum_{j\in\{1,\ldots,n\}} \frac{\frac{\partial(1-\mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}}{1 - \mathbf{L}_v[i,j]} \end{aligned}$$

Since $\mathbf{L}_v[i,j] = \mathbb{1}_{\{1\}}(\mathbf{C})[i,j] \times \mathbf{v}[j] + \mathbb{1}_{\{-1\}}(\mathbf{C})[i,j] \times (1 - \mathbf{v}[j])$, we know

(a) for $j \in \{1, \ldots, n\}$ such that $j \neq k$, $\frac{\partial(1-\mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} = 0$ and $\frac{\partial \mathbf{L}_v[i,j]}{\partial \mathbf{v}[k]} = 0$;

(b) when clause i doesn't contain a literal for atom p_k, $\frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} = 0$ and $\frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} = 0$;

(c) when clause i contains literal p_k, $\frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} = -1$ and $\frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} = 1$;

(d) when clause i contains literal $\neg p_k$, $\frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} = 1$ and $\frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} = -1$.

We will refer to the above 4 bullets with their identifiers.

Since $i \in U$, we know clause i has all but one literal of the form $\neg p_j$ such that $p_j \in F$. Since $\mathbf{f}[k] = 0$, we know $p_k \notin F$. Then, when clause i contains literal p_k or $\neg p_k$, all other literals in clause i must be of the form $\neg p_j$ where $p_j \in F$. For every literal $\neg p_j$ in clause i where $j \neq k$, we know $p_j \in F$, thus $\mathbf{f}[j] = 1$; since $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b_p(\mathbf{x})$, then $\mathbf{v}[j] = 1$; consequently, the literal $\neg p_j$ evaluates to FALSE under v. Recall that $\mathbf{L}_v[i,j] \in \{0,1\}$, and $\mathbf{L}_v[i,j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v, then we know

- when $i \in U$, $\mathbf{f}[k] = 0$, and clause i contains literal p_k or $\neg p_k$, $\mathbf{L}_v[i,j] = 0$ for $j \in \{1, \ldots, n\}$ such that $j \neq k$.

Then we have

$$
\begin{aligned}
G_{i,k} &= \left(\prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \right) \times \sum_{j \in \{1,\ldots,n\}} \frac{\frac{\partial(1-\mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}}{1 - \mathbf{L}_v[i,j]} \\
&= \left(\prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \right) \times \frac{\frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]}}{1 - \mathbf{L}_v[i,k]} \text{ (due to \textbf{(a)})} \\
&= \frac{\partial(1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \times \prod_{\substack{j \in \{1,\ldots,n\} \\ j \neq k}} (1 - \mathbf{L}_v[i,j]) \\
&= \begin{cases}
0 & \text{if clause } i \text{ doesn't contain a literal} \\
 & \text{for atom } p_k \text{ (due to \textbf{(b)})} \\
-1 & \text{if clause } i \text{ contains a literal } p_k \text{ (due to \textbf{(c)})} \\
1 & \text{if clause } i \text{ contains a literal } \neg p_k \text{ (due to \textbf{(d)})}
\end{cases}
\end{aligned}
$$

Since $i \in U$ and $\mathbf{f}[k] = 0$, when clause i contains a literal l_k for atom p_k, we know $F \not\models l_j$ for every literal l_j in clause i such that $j \neq k$. Since $C \cup F$ is satisfiable, we know $C \cup F \models l_k$ and there cannot be two clauses in C_{deduce} containing different literals p_k and $\neg p_k$. Thus, when $\mathbf{f}[k] = 0$,

$$
\begin{aligned}
\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} &= \sum_{i \in U} G_{i,k} \\
&= \begin{cases}
-c & \text{if } c > 0 \text{ clauses in } C_{deduce} \text{ contain literal } p_k, \\
c & \text{if } c > 0 \text{ clauses in } C_{deduce} \text{ contain literal } \neg p_k, \\
0 & \text{otherwise.}
\end{cases}
\end{aligned}
$$

Note that the first 2 cases above are disjoint since there cannot be two clauses in C_{deduce} containing different literals p_k and $\neg p_k$.

Finally, if $p_k \notin F$,

$$\frac{\partial L_{deduce}}{\partial \mathbf{x}[k]} \overset{iSTE}{\approx} \frac{\partial L_{deduce}}{\partial \mathbf{v}[k]}$$
$$= \begin{cases} -c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } p_k; \\ c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } \neg p_k; \\ 0 & \text{otherwise;} \end{cases}$$

[L_{unsat}] According to the definition,

$$\begin{aligned} L_{unsat} =& avg(\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat}) \\ =& \frac{1}{m} \sum_{i \in \{1,\ldots,m\}} \Big(\mathbb{1}_{\{1\}}(\mathbf{unsat}[i]) \times \prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \Big) \end{aligned}$$

Recall that we proved that $\mathbb{1}_{\{1\}}(\mathbf{unsat})[i] \in \{0,1\}$ is the output of an indicator function whose value is 1 iff clause i evaluates to FALSE under v. Let $U \subseteq \{1,\ldots,m\}$ denote the set of indices of clauses in C that are evaluated as FALSE under v.

$$L_{unsat} = \frac{1}{m} \sum_{i \in U} \Big(\prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \Big)$$

Then the gradient of L_{unsat} w.r.t. $\mathbf{v}[k]$ is

$$\frac{\partial L_{unsat}}{\partial \mathbf{v}[k]} = \frac{1}{m} \sum_{i \in U} \Big(\frac{\partial \prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \Big).$$

Recall that $\mathbf{L}_v[i,j] \in \{0,1\}$, and $\mathbf{L}_v[i,j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v. When $i \in U$, clause i evaluates to FALSE under v. Thus when $i \in U$, all literals in clause i must be evaluated as FALSE under v, and consequently, $\mathbf{L}_v[i,j] = 0$ for all $j \in \{1,\ldots,m\}$.

Then

$$\begin{aligned}\frac{\partial L_{unsat}}{\partial \mathbf{v}[k]} =& \frac{1}{m}\sum_{i\in U}\Big(\frac{\partial \prod_{j\in\{1,\dots,n\}}(1-\mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}\Big)\\ =& \frac{1}{m}\sum_{i\in U}\Big(\frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]}\Big) \text{ (due to } \textbf{(a)}\text{)}\\ =& \frac{c_2-c_1}{m} \text{ (due to } \textbf{(b), (c), (d)}\text{)}\end{aligned}$$

where c_1 (and c_2, resp.) is the number of clauses in U that contain p_k (and $\neg p_k$, resp.). Finally, if $p_k \notin F$,

$$\frac{\partial L_{unsat}}{\partial \mathbf{x}[k]} \overset{iSTE}{\approx} \frac{\partial L_{unsat}}{\partial \mathbf{v}[k]} = \frac{c_2-c_1}{m}$$

where c_1 (and c_2, resp.) is the number of clauses in C that are not satisfied by v and contain p_k (and $\neg p_k$, resp.).

[L_{sat}] Recall that we proved that $\mathbb{1}_{\{0\}}(\mathbf{unsat})[i] \in \{0,1\}$ is the output of an indicator function whose value is 1 iff clause i evaluates to TRUE under v. Let $S \subseteq \{1,\dots,m\}$ denote the set of indices of clauses in C that are evaluated as TRUE under v. Then

$$\begin{aligned}L_{sat} =& avg(\mathbb{1}_{\{0\}}(\mathbf{unsat}) \odot \mathbf{keep})\\ =& \frac{1}{m}\sum_{i\in\{1,\dots,m\}}\Big(\mathbb{1}_{\{0\}}(\mathbf{unsat}[i]) \times \mathbf{keep}[i]\Big)\\ =& \frac{1}{m}\sum_{i\in S}\mathbf{keep}[i]\\ =& \frac{1}{m}\sum_{i\in S}\sum_{j\in\{1,\dots,n\}}\Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,j]) \times (1-\mathbf{L}_v[i,j])\\ &\qquad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,j]) \times \mathbf{L}_v[i,j]\Big)\end{aligned}$$

Then the gradient of L_{sat} w.r.t. $\mathbf{v}[k]$ is

$$\frac{\partial L_{sat}}{\partial \mathbf{v}[k]} = \frac{1}{m}\sum_{i\in S}\sum_{j\in\{1,\dots,n\}}\Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,j]) \times \frac{\partial(1-\mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}$$

$$
\begin{aligned}
&\quad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,j]) \times \frac{\partial \mathbf{L}_v[i,j]}{\partial \mathbf{v}[k]}\Big) \\
=&\frac{1}{m}\sum_{i\in S}\Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) \times \frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \\
&\quad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \times \frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]}\Big) \text{ (due to \textbf{(a)})} \\
=&\frac{1}{m}\sum_{\substack{i\in S \\ \text{clause } i \text{ contains} \\ \text{literal } p_k}}\Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) \times \frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \\
&\quad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \times \frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]}\Big) + \\
&\frac{1}{m}\sum_{\substack{i\in S \\ \text{clause } i \text{ contains} \\ \text{literal } \neg p_k}}\Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) \times \frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \\
&\quad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \times \frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]}\Big) \text{ (due to \textbf{(b)})} \\
=&\frac{1}{m}\sum_{\substack{i\in S \\ \text{clause } i \text{ contains} \\ \text{literal } p_k}}\Big(-\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k])\Big) \\
&+ \frac{1}{m}\sum_{\substack{i\in S \\ \text{clause } i \text{ contains} \\ \text{literal } \neg p_k}}\Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) - \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k])\Big) \text{ (due to \textbf{(c)} and \textbf{(d)})}
\end{aligned}
$$

Recall that $\mathbf{L}_v[i,j] \in \{0,1\}$, and $\mathbf{L}_v[i,j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v. It's easy to check that

- when clause i contains literal p_k, the value of $-\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k])$ is -1 if $v \models p_k$ and is 1 if $v \not\models p_k$;
- when clause i contains literal $\neg p_k$, the value of $\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) - \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k])$ is -1 if $v \models p_k$ and is 1 if $v \not\models p_k$.

Thus

$$
\frac{\partial L_{sat}}{\partial \mathbf{v}[k]} = \begin{cases} -\frac{c}{m} & \text{if } v \models p_k, \\ \frac{c}{m} & \text{if } v \not\models p_k. \end{cases}
$$

where c is the number of clauses in S that contain a literal for atom p_k. Finally, if $p_k \notin F$,

$$\frac{\partial L_{sat}}{\partial \mathbf{x}[k]} \overset{iSTE}{\approx} \frac{\partial L_{sat}}{\partial \mathbf{v}[k]} = \begin{cases} -\frac{c}{m} & \text{if } v \models p_k, \\ \frac{c}{m} & \text{if } v \not\models p_k; \end{cases}$$

where c is the number of clauses in C that are satisfied by v and contain p_k or $\neg p_k$.

∎

4.5.4 Proof of Proposition 5

Proposition 5 Proposition 4 still holds for $\mathbf{x} \in \mathbb{R}^n$ and $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b(\mathbf{x})$.

[Complete Statement] Given a CNF theory C of m clauses and n atoms and a set F of atoms such that $C \cup F$ is satisfiable, let $\mathbf{C}, \mathbf{f}$ denote their matrix/vector representations, respectively. Given a neural network output $\mathbf{x} \in \mathbb{R}^n$ in logits (i.e., real numbers instead of probabilities), we construct $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b(\mathbf{x})$ and a truth assignment v such that $v(p_j) = \text{TRUE}$ if $\mathbf{v}[j]$ is 1, and $v(p_j) = \text{FALSE}$ if $\mathbf{v}[j]$ is 0. Let $C_{deduce} \subseteq C$ denote the set of Horn clauses H in C such that all but one literal in H are of the form $\neg p$ and $p \in F$. Then, for any $j \in \{1, \ldots, n\}$,

1. if $p_j \in F$, all of $\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]}$, $\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]}$, and $\frac{\partial L_{sat}}{\partial \mathbf{x}[j]}$ are zeros;
2. if $p_j \notin F$,

$$\frac{\partial L_{deduce}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \begin{cases} -c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } p_j; \\ c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } \neg p_j; \\ 0 & \text{otherwise;} \end{cases}$$

$$\frac{\partial L_{unsat}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \frac{c_2 - c_1}{m}$$

$$\frac{\partial L_{sat}}{\partial \mathbf{x}[j]} \overset{iSTE}{\approx} \begin{cases} -\frac{c_3}{m} & \text{if } v \models p_j, \\ \frac{c_3}{m} & \text{if } v \not\models p_j. \end{cases}$$

where $\overset{iSTE}{\approx}$ stands for the equivalence of gradients under iSTE; c_1 (and c_2, resp.) is the number of clauses in C that are not satisfied by v and contain p_j (and $\neg p_j$, resp.); c_3 is the number of clauses in C that are satisfied by v and contain p_j or $\neg p_j$.

Proof. Recall the definition of L_{cnf}

$$\begin{aligned}
\mathbf{L}_f &= \mathbf{C} \odot \mathbf{f} \\
\mathbf{L}_v &= \mathbb{1}_{\{1\}}(\mathbf{C}) \odot \mathbf{v} + \mathbb{1}_{\{-1\}}(\mathbf{C}) \odot (1 - \mathbf{v}) \\
\mathbf{deduce} &= \mathbb{1}_{\{1\}}\Big(sum(\mathbf{C} \odot \mathbf{C}) - sum(\mathbb{1}_{\{-1\}}(\mathbf{L}_f))\Big) \\
\mathbf{unsat} &= prod(1 - \mathbf{L}_v) \\
\mathbf{keep} &= sum(\mathbb{1}_{\{1\}}(\mathbf{L}_v) \odot (1 - \mathbf{L}_v) + \mathbb{1}_{\{0\}}(\mathbf{L}_v) \odot \mathbf{L}_v) \\
L_{deduce} &= sum(\mathbf{deduce} \odot \mathbf{unsat}) \\
L_{unsat} &= avg(\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat}) \\
L_{sat} &= avg(\mathbb{1}_{\{0\}}(\mathbf{unsat}) \odot \mathbf{keep}) \\
L_{cnf}(\mathbf{C}, \mathbf{v}, \mathbf{f}) &= L_{deduce} + L_{unsat} + L_{sat}
\end{aligned}$$

We will prove each bullet in Proposition 5 as follows. This proof is almost the same as the proof for Proposition 4 since the choice of $b(x)$ v.s. $b_p(x)$ doesn't affect the gradient computation from L_{cnf} to $\mathbf{x}$ under iSTE.

1. Take any $k \in \{1, \ldots, n\}$, let's focus on $\mathbf{x}[k]$ and compute the gradient of $L \in \{L_{deduce}, L_{unsat}, L_{sat}\}$ to it with iSTE. According to the chain rule and since $\frac{\partial \mathbf{v}[i]}{\partial b(\mathbf{x})[j]} =$

0 for $i \neq j$, we have

$$\frac{\partial L}{\partial \mathbf{x}[k]} = \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial \mathbf{v}[k]}{\partial b(\mathbf{x}[k])} \times \frac{\partial b(\mathbf{x}[k])}{\partial \mathbf{x}[k]}.$$

Under iSTE, the last term $\frac{\partial b(\mathbf{x}[k])}{\partial \mathbf{x}[k]}$ is replaced with $\frac{\partial s(\mathbf{x}[k])}{\partial \mathbf{x}[k]} = \frac{\partial \mathbf{x}[k]}{\partial \mathbf{x}[k]} = 1$. Thus

$$\begin{aligned}
\frac{\partial L}{\partial \mathbf{x}[k]} =& \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial \mathbf{v}[k]}{\partial b(\mathbf{x}[k])} \times \frac{\partial b(\mathbf{x}[k])}{\partial \mathbf{x}[k]} \\
\overset{iSTE}{\approx}& \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial \mathbf{v}[k]}{\partial b(\mathbf{x}[k])} \quad \text{(under iSTE)} \\
=& \frac{\partial L}{\partial \mathbf{v}[k]} \times \frac{\partial (\mathbf{f}[k] + \mathbb{1}_{\{0\}}(\mathbf{f}[k]) \times b(\mathbf{x}[k]))}{\partial b(\mathbf{x}[k])} \\
=& \begin{cases} \frac{\partial L}{\partial \mathbf{v}[k]} & \text{if } \mathbf{f}[k] = 0, \\ 0 & \text{if } \mathbf{f}[k] = 1. \end{cases}
\end{aligned}$$

Since $\mathbf{f}[k] = 1$ iff $p_k \in F$, if $p_k \in F$, then all of $\frac{\partial L_{deduce}}{\partial \mathbf{x}[k]}$, $\frac{\partial L_{unsat}}{\partial \mathbf{x}[k]}$, and $\frac{\partial L_{sat}}{\partial \mathbf{x}[k]}$ are zeros.

2. We know $p_k \notin F$ iff $\mathbf{f}[k] = 0$. As proved in the first bullet, for $L \in \{L_{deduce}, L_{unsat}, L_{sat}\}$, if $p_k \notin F$, then $\frac{\partial L}{\partial \mathbf{x}[k]} = \frac{\partial L}{\partial \mathbf{v}[k]}$. We further analyze the value of $\frac{\partial L}{\partial \mathbf{v}[k]}$ for each L under the condition that $\mathbf{f}[k] = 0$.

   [L_{deduce}] According to the definition,

$$\begin{aligned}
L_{deduce} =& \sum_{i \in \{1,\dots,m\}} \Big(\mathbf{deduce}[i] \times \mathbf{unsat}[i] \Big) \\
=& \sum_{i \in \{1,\dots,m\}} \Big(\mathbf{deduce}[i] \times \prod_{j \in \{1,\dots,n\}} (1 - \mathbf{L}_v[i,j]) \Big) \\
\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} =& \sum_{i \in \{1,\dots,m\}} \frac{\partial \Big(\mathbf{deduce}[i] \times \prod_{j \in \{1,\dots,n\}} (1 - \mathbf{L}_v[i,j]) \Big)}{\partial \mathbf{v}[k]} \\
=& \sum_{i \in \{1,\dots,m\}} \Big(\frac{\partial \mathbf{deduce}[i]}{\partial \mathbf{v}[k]} \times \prod_{j \in \{1,\dots,n\}} (1 - \mathbf{L}_v[i,j]) + \\
& \qquad \mathbf{deduce}[i] \times \frac{\partial \prod_{j \in \{1,\dots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \Big)
\end{aligned}$$

Since $\mathbf{deduce}$ is the result of an indicator function, $\frac{\partial \mathbf{deduce}[i]}{\partial \mathbf{v}[k]} = 0$. Then,

$$\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} = \sum_{i \in \{1,\ldots,m\}} \Big(\mathbf{deduce}[i] \times \frac{\partial \prod\limits_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \Big).$$

Let $U \subseteq \{1,\ldots,m\}$ denote the set of indices of all clauses in C_{deduce}. Since $\mathbf{deduce}[i] = 1$ iff $i \in U$,

$$\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} = \sum_{i \in U} \Big(\frac{\partial \prod\limits_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \Big).$$

Let $G_{i,k}$ denote $\frac{\partial \prod\limits_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}$. Then

$$\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} = \sum_{i \in U} G_{i,k}.$$

Let's analyze the value of $G_{i,k}$ where $i \in U$ and $k \in \{1,\ldots,n\}$ such that $\mathbf{f}[k] = 0$. According to the product rule below,

$$\frac{d}{dx}\left[\prod_{i=1}^{k} f_i(x)\right] = \left(\prod_{i=1}^{k} f_i(x)\right)\left(\sum_{i=1}^{k} \frac{f_i'(x)}{f_i(x)}\right)$$

we have

$$\begin{aligned} G_{i,k} =& \frac{\partial \prod\limits_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \\ =& \left(\prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \right) \times \sum_{j \in \{1,\ldots,n\}} \frac{\frac{\partial (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}}{1 - \mathbf{L}_v[i,j]} \end{aligned}$$

Since $\mathbf{L}_v[i,j] = \mathbb{1}_{\{1\}}(\mathbf{C})[i,j] \times \mathbf{v}[j] + \mathbb{1}_{\{-1\}}(\mathbf{C})[i,j] \times (1 - \mathbf{v}[j])$, we know

(a) for $j \in \{1,\ldots,n\}$ such that $j \neq k$, $\frac{\partial (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} = 0$ and $\frac{\partial \mathbf{L}_v[i,j]}{\partial \mathbf{v}[k]} = 0$;

(b) when clause i doesn't contain a literal for atom p_k, $\frac{\partial (1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} = 0$ and $\frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} = 0$;

(c) when clause i contains literal p_k, $\frac{\partial (1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} = -1$ and $\frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} = 1$;

(d) when clause i contains literal $\neg p_k$, $\frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} = 1$ and $\frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} = -1$.

We will refer to the above 4 bullets with their identifiers.

Since $i \in U$, we know clause i has all but one literal of the form $\neg p_j$ such that $p_j \in F$. Since $\mathbf{f}[k] = 0$, we know $p_k \notin F$. Then, when clause i contains literal p_k or $\neg p_k$, all other literals in clause i must be of the form $\neg p_j$ where $p_j \in F$. For every literal $\neg p_j$ in clause i where $j \neq k$, we know $p_j \in F$, thus $\mathbf{f}[j] = 1$; since $\mathbf{v} = \mathbf{f} + \mathbb{1}_{\{0\}}(\mathbf{f}) \odot b(\mathbf{x})$, then $\mathbf{v}[j] = 1$; consequently, the literal $\neg p_j$ evaluates to FALSE under v. Recall that $\mathbf{L}_v[i,j] \in \{0,1\}$, and $\mathbf{L}_v[i,j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v, then we know

- when $i \in U$, $\mathbf{f}[k] = 0$, and clause i contains literal p_k or $\neg p_k$, $\mathbf{L}_v[i,j] = 0$ for $j \in \{1, \ldots, n\}$ such that $j \neq k$.

Then we have

$$
\begin{aligned}
G_{i,k} &= \left(\prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \right) \times \sum_{j \in \{1,\ldots,n\}} \frac{\frac{\partial(1-\mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]}}{1 - \mathbf{L}_v[i,j]} \\
&= \left(\prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \right) \times \frac{\frac{\partial(1-\mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]}}{1 - \mathbf{L}_v[i,k]} \text{ (due to \textbf{(a)})} \\
&= \frac{\partial(1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \times \prod_{\substack{j \in \{1,\ldots,n\} \\ j \neq k}} (1 - \mathbf{L}_v[i,j]) \\
&= \begin{cases}
0 & \text{if clause } i \text{ doesn't contain a literal} \\
 & \text{for atom } p_k \text{ (due to \textbf{(b)})} \\
-1 & \text{if clause } i \text{ contains a literal } p_k \text{ (due to \textbf{(c)})} \\
1 & \text{if clause } i \text{ contains a literal } \neg p_k \text{ (due to \textbf{(d)})}
\end{cases}
\end{aligned}
$$

Since $i \in U$ and $\mathbf{f}[k] = 0$, when clause i contains a literal l_k for atom p_k, we know $F \not\models l_j$ for every literal l_j in clause i such that $j \neq k$. Since $C \cup F$ is satisfiable, we know $C \cup F \models l_k$ and there cannot be two clauses in C_{deduce} containing different literals p_k and $\neg p_k$. Thus, when $\mathbf{f}[k] = 0$,

$$\begin{aligned}\frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} =& \sum_{i \in U} G_{i,k} \\ =& \begin{cases} -c & \text{if } c > 0 \text{ clauses in } C_{deduce} \text{ contain literal } p_k, \\ c & \text{if } c > 0 \text{ clauses in } C_{deduce} \text{ contain literal } \neg p_k, \\ 0 & \text{otherwise.} \end{cases}\end{aligned}$$

Note that the first 2 cases above are disjoint since there cannot be two clauses in C_{deduce} containing different literals p_k and $\neg p_k$.

Finally, if $p_k \notin F$,

$$\begin{aligned}\frac{\partial L_{deduce}}{\partial \mathbf{x}[k]} \overset{iSTE}{\approx} & \frac{\partial L_{deduce}}{\partial \mathbf{v}[k]} \\ =& \begin{cases} -c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } p_k; \\ c & \text{if } c > 0 \text{ clauses in } C_{deduce} \\ & \text{contain literal } \neg p_k; \\ 0 & \text{otherwise;} \end{cases}\end{aligned}$$

[L_{unsat}] According to the definition,

$$\begin{aligned}L_{unsat} =& avg(\mathbb{1}_{\{1\}}(\mathbf{unsat}) \odot \mathbf{unsat}) \\ =& \frac{1}{m} \sum_{i \in \{1,\ldots,m\}} \Big(\mathbb{1}_{\{1\}}(\mathbf{unsat}[i]) \times \prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \Big)\end{aligned}$$

Recall that we proved that $\mathbb{1}_{\{1\}}(\mathbf{unsat})[i] \in \{0,1\}$ is the output of an indicator function whose value is 1 iff clause i evaluates to FALSE under v. Let $U \subseteq \{1,\ldots,m\}$ denote the set of indices of clauses in C that are evaluated as FALSE under v.

$$L_{unsat} = \frac{1}{m} \sum_{i \in U} \Big(\prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j]) \Big)$$

Then the gradient of L_{unsat} w.r.t. $\mathbf{v}[k]$ is

$$\frac{\partial L_{unsat}}{\partial \mathbf{v}[k]} = \frac{1}{m} \sum_{i \in U} \Big(\frac{\partial \prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \Big).$$

Recall that $\mathbf{L}_v[i,j] \in \{0,1\}$, and $\mathbf{L}_v[i,j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v. When $i \in U$, clause i evaluates to FALSE under v. Thus when $i \in U$, all literals in clause i must be evaluated as FALSE under v, and consequently, $\mathbf{L}_v[i,j] = 0$ for all $j \in \{1,\ldots,m\}$. Then

$$\begin{aligned}
\frac{\partial L_{unsat}}{\partial \mathbf{v}[k]} &= \frac{1}{m} \sum_{i \in U} \Big(\frac{\partial \prod_{j \in \{1,\ldots,n\}} (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \Big) \\
&= \frac{1}{m} \sum_{i \in U} \Big(\frac{\partial (1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \Big) \text{ (due to \textbf{(a)})} \\
&= \frac{c_2 - c_1}{m}
\end{aligned}$$

where c_1 (and c_2, resp.) is the number of clauses in U that contain p_k (and $\neg p_k$, resp.). Finally, if $p_k \notin F$,

$$\frac{\partial L_{unsat}}{\partial \mathbf{x}[k]} \overset{iSTE}{\approx} \frac{\partial L_{unsat}}{\partial \mathbf{v}[k]} = \frac{c_2 - c_1}{m}$$

where c_1 (and c_2, resp.) is the number of clauses in C that are not satisfied by v and contain p_k (and $\neg p_k$, resp.).

[L_{sat}] Recall that we proved that $\mathbb{1}_{\{0\}}(\mathbf{unsat})[i] \in \{0,1\}$ is the output of an indicator function whose value is 1 iff clause i evaluates to TRUE under v. Let

$S \subseteq \{1, \ldots, m\}$ denote the set of indices of clauses in C that are evaluated as TRUE under v. Then

$$
\begin{aligned}
L_{sat} =& avg(\mathbb{1}_{\{0\}}(\mathbf{unsat}) \odot \mathbf{keep}) \\
=& \frac{1}{m} \sum_{i \in \{1,\ldots,m\}} \Big(\mathbb{1}_{\{0\}}(\mathbf{unsat}[i]) \times \mathbf{keep}[i] \Big) \\
=& \frac{1}{m} \sum_{i \in S} \mathbf{keep}[i] \\
=& \frac{1}{m} \sum_{i \in S} \sum_{j \in \{1,\ldots,n\}} \Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,j]) \times (1 - \mathbf{L}_v[i,j]) + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,j]) \times \mathbf{L}_v[i,j] \Big)
\end{aligned}
$$

Then the gradient of L_{sat} w.r.t. $\mathbf{v}[k]$ is

$$
\begin{aligned}
\frac{\partial L_{sat}}{\partial \mathbf{v}[k]} =& \frac{1}{m} \sum_{i \in S} \sum_{j \in \{1,\ldots,n\}} \Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,j]) \times \frac{\partial (1 - \mathbf{L}_v[i,j])}{\partial \mathbf{v}[k]} \\
& \qquad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,j]) \times \frac{\partial \mathbf{L}_v[i,j]}{\partial \mathbf{v}[k]} \Big) \\
=& \frac{1}{m} \sum_{i \in S} \Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) \times \frac{\partial (1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \\
& \qquad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \times \frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} \Big) \text{ (due to \textbf{(a)})} \\
=& \frac{1}{m} \sum_{\substack{i \in S \\ \text{clause } i \text{ contains} \\ \text{literal } p_k}} \Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) \times \frac{\partial (1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \\
& \qquad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \times \frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} \Big) + \\
& \frac{1}{m} \sum_{\substack{i \in S \\ \text{clause } i \text{ contains} \\ \text{literal } \neg p_k}} \Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) \times \frac{\partial (1 - \mathbf{L}_v[i,k])}{\partial \mathbf{v}[k]} \\
& \qquad + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \times \frac{\partial \mathbf{L}_v[i,k]}{\partial \mathbf{v}[k]} \Big) \text{ (due to \textbf{(b)})} \\
=& \frac{1}{m} \sum_{\substack{i \in S \\ \text{clause } i \text{ contains} \\ \text{literal } p_k}} \Big(- \mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \Big) \\
& + \frac{1}{m} \sum_{\substack{i \in S \\ \text{clause } i \text{ contains} \\ \text{literal } \neg p_k}} \Big(\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) - \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k]) \Big) \text{ (due to \textbf{(c)} and \textbf{(d)})}
\end{aligned}
$$

Recall that $\mathbf{L}_v[i,j] \in \{0,1\}$, and $\mathbf{L}_v[i,j] = 1$ iff clause i contains a literal (p_j or $\neg p_j$) for atom p_j and this literal evaluates to TRUE under v. It's easy to check that

- when clause i contains literal p_k, the value of $-\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) + \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k])$ is -1 if $v \models p_k$ and is 1 if $v \not\models p_k$;
- when clause i contains literal $\neg p_k$, the value of $\mathbb{1}_{\{1\}}(\mathbf{L}_v[i,k]) - \mathbb{1}_{\{0\}}(\mathbf{L}_v[i,k])$ is -1 if $v \models p_k$ and is 1 if $v \not\models p_k$.

Thus

$$\frac{\partial L_{sat}}{\partial \mathbf{v}[k]} = \begin{cases} -\frac{c}{m} & \text{if } v \models p_k, \\ \frac{c}{m} & \text{if } v \not\models p_k. \end{cases}$$

where c is the number of clauses in S that contain a literal for atom p_k. Finally, if $p_k \notin F$,

$$\frac{\partial L_{sat}}{\partial \mathbf{x}[k]} \overset{iSTE}{\approx} \frac{\partial L_{sat}}{\partial \mathbf{v}[k]} = \begin{cases} -\frac{c}{m} & \text{if } v \models p_k, \\ \frac{c}{m} & \text{if } v \not\models p_k; \end{cases}$$

where c is the number of clauses in C that are satisfied by v and contain p_k or $\neg p_k$.

∎

Chapter 5

RECURRENT TRANSFORMER

Constraint Satisfaction Problems (CSPs) are about finding values for variables that satisfy the given constraints. They have been much studied in symbolic AI with the emphasis on designing efficient algorithms for deductively finding solutions for explicitly stated constraints. On the other hand, there are rising interests in CSPs with the deep learning approach, where the focus is on inductively learning the constraints and solving them in an end-to-end manner. One of the main methods is graph neural networks (GNNs). For example, Recurrent Relational Network (RRN) (Palm *et al.*, 2018) uses message passing over graph structures to learn logical constraints, achieving high accuracy on Sudoku. On the other hand, it uses hand-coded information about Sudoku constraints, namely, which variables are allowed to interact. Moreover, it is limited to textual input. SATNet (Wang *et al.*, 2019) is a differentiable MAXSAT solver that can infer logical rules and be integrated into DNNs. SATNet was shown to solve even Visual Sudoku, where the input is a hand-written Sudoku board. The problem is harder because a model has to learn how to map visual inputs to symbolic digits without explicit supervision. However, Chang *et al.* (2020) noted that the experiment had a label leakage issue. With the proper evaluation, SATNet's performance on Visual Sudoku dropped to 0%. Moreover, SATNet's evaluation is limited to easy puzzles, and it does not perform well on hard puzzles that RRN could solve.

On another aspect, although these models could learn complicated constraints purely from data, in many cases, (part of) constraints are already known, and exploiting such deductive knowledge in inductive learning could be helpful for sample-efficient and robust learning. The problem is challenging, especially if the knowledge is in the form of *discrete* constraints, whereas standard deep learning is mainly about *continuous* and *differentiable*

parameter optimization.

This chapter provides a viable solution to the limitations of the above models based on the Transformer architecture. Transformer-based models have not been shown effective for CSPs despite their widespread applications in language (Vaswani *et al.*, 2017; Zhang *et al.*, 2020; Helwe *et al.*, 2021; Li *et al.*, 2020) and vision (Dosovitskiy *et al.*, 2020; Gabeur *et al.*, 2020). Creswell *et al.* (2022) asserted that Transformer-based large language models (LLMs) tend to perform poorly on multi-step logical reasoning problems. In the case of Sudoku, the typical solving requires about 20 to 60 steps of reasoning. Despite the various ideas on prompting GPT-3, GPT-3 is not able to solve Sudoku. Nye *et al.* (2021) note that LLMs work well for system 1 intuitive thinking but not for system 2 logical thinking. Given the superiority of other models on CSPs, one might conclude that Transformers are unsuitable for CSPs.

We find that Transformers needs to incorporate recurrence to successfully apply to CSPs. The added recurrence encourages the Transformer model to apply multi-step reasoning similar to RRNs. Interestingly, this simple change already yields better results than the other models above and gives several other advantages. The learning is more robust than SATNet's. By looking at the learned attention matrices, we could interpret what the Transformer has learned. Intuitively, multi-head attention extracts distinct information about the problem structure. Adding more attention blocks and recurrences tends to make the model learn better. Analogous to the extension Vision Transformer (Dosovitskiy *et al.*, 2020) made, our model can be easily extended to process the visual input. Moreover, the model avoids the symbol grounding problem that SATNet encountered.

In addition, we present a way to inject discrete constraints into Recurrent Transformer training, borrowing the idea from CL-STE in Chapter 4. We apply this idea to Recurrent Transformers with some modifications. We note that adding explicit constraint loss to all recurrent layers helps Transformers learn more effectively. We also add a constraint loss

to the attention matrix so that constraints can help learn better attention values. Including these constraint losses in training improves accuracy and lets the Transformer learn with fewer labeled data (semi-supervised learning).

5.1 CSP

A constraint satisfaction problem (CSP) [1] is defined as $\langle \mathbb{X}, \mathbb{D}, \mathbb{C} \rangle$ where $\mathbb{X} = \{\mathcal{X}_1, \ldots, \mathcal{X}_t\}$ is a set of t logical variables; $\mathbb{D} = \{\mathbb{D}_1, \ldots, \mathbb{D}_t\}$ and each $\mathbb{D}_i$ is a finite set of domain values for logical variable $\mathcal{X}_i$; and $\mathbb{C}$ is a set of constraints. An *atom* (i.e., value assignment) is of the form $\mathcal{X}_i = v$ where $v \in \mathbb{D}_i$. A *constraint* on a sequence $\langle \mathcal{X}_i, \ldots, \mathcal{X}_j \rangle$ of variables is a mapping: $\mathbb{D}_i \times \cdots \times \mathbb{D}_j \rightarrow \{\text{TRUE}, \text{FALSE}\}$ that specifies the set of atoms that can or cannot hold at the same time. A (complete) *evaluation* is a set of t atoms $\{\mathcal{X}_i = v \mid i \in \{1, \ldots, t\}, v \in \mathbb{D}_i\}$. An evaluation is a *solution* if it does not violate any constraint in $\mathbb{C}$, i.e., it maps all constraints to TRUE.

One of the commonly used constraints is the *cardinality constraint*:

$$l \leq |\{\mathcal{X}_i = v_i, \ldots, \mathcal{X}_j = v_j\}| \leq u \tag{5.1}$$

where l and u are nonnegative integers denoting bounds, and for $k \in \{i, \ldots, j\}$, $\mathcal{X}_k \in \mathbb{X}$ and $v_k \in \mathbb{D}_k$. Cardinality constraint (5.1) is TRUE iff the number of atoms that are true in it is between l and u. If $l = u$, constraint (5.1) can be simplified to

$$|\{\mathcal{X}_i = v_i, \ldots, \mathcal{X}_j = v_j\}| = l \tag{5.2}$$

which is TRUE iff the number of atoms in the given set is exactly l. If $i = j$ and $l = 1$, constraint (5.2) can be further simplified to $\mathcal{X}_i = v_i$.

Example 9 (CSP for Sudoku) *A CSP for Sudoku puzzle is such that* $\mathbb{X} = \{cell_1, \ldots, cell_{81}\}$ *denotes all 81 cells in a Sudoku board;* $\mathbb{D} = \{\mathbb{D}_1, \ldots, \mathbb{D}_{81}\}$ *and* $\mathbb{D}_i = \{1, \ldots, 9\}$

[1] https://en.wikipedia.org/wiki/Constraint_satisfaction_problem

denotes all possible values in each cell; and $\mathbb{C}$ consists of constraints $cell_i = d$ for each given digit d at cell i, and constraints

$$|\{cell_i = d, \ldots, cell_j = d\}| = 1 \tag{5.3}$$

for $d \in \{1, \ldots, 9\}$ and any set $\{i, \ldots, j\}$ of 9 cell indices that belong to the same row/-column/box, saying that "each digit d should appear exactly once in each row/column/box." The solution to the CSP corresponds to the solution to the Sudoku puzzle.

5.2 Model Design

Given a constraint satisfaction problem $\langle \mathbb{X}, \mathbb{D}, \mathbb{C} \rangle$ such that, for $i \in \{1, \ldots, t\}$, $1 \leq |\mathbb{D}_i| \leq c$ for a constant c, the Recurrent Transformer takes as an input the sequence $\langle \mathcal{X}_1, \ldots, \mathcal{X}_t \rangle$ of logical variables, and outputs the probability distribution over the values in the domain $\mathbb{D}_i$ of each $\mathcal{X}_i$. Let c_i be the domain size of $\mathcal{X}_i$. Without loss of generality, we assume values in $\mathbb{D}_i$ are represented by their indices, i.e., $\mathbb{D}_i = \{1, \ldots, c_i\}$. The probability of $\mathcal{X}_i = j$ is given by the j-th value of the output for $\mathcal{X}_i$.

A logical variable $\mathcal{X}_i$ is treated as a token whose token embedding is a vector of length d_h encoding the given information about this logical variable (e.g., some numbers, a textual description, an image, etc). The position embedding of $\mathcal{X}_i$ is a randomly initialized vector of length d_h and is to be learned to record data-invariant information for logical variable $\mathcal{X}_i$. Let $\boldsymbol{E}_{tok}, \boldsymbol{E}_{pos} \in \mathbb{R}^{t \times d_h}$ denote the token and position embeddings of t logical variables. The r-th recurrent step in a Recurrent Transformer with L self-attention blocks and R recurrences can be formulated as follows ($r \in \{1, \ldots, R\}$):

$$\begin{aligned}
\boldsymbol{H}^{(r,0)} &= \boldsymbol{H}^{(r-1,L)} \\
\boldsymbol{H}^{(r,l)} &= \mathrm{block}_l(\boldsymbol{H}^{(r,l-1)}) && \forall l \in \{1, \ldots, L\} \\
\boldsymbol{X}^{(r,l)} &= \mathrm{softmax}(\mathrm{layer_norm}(\boldsymbol{H}^{(r,l)}) \cdot \boldsymbol{W}_{out}) && \forall l \in \{1, \ldots, L\}
\end{aligned}$$

where the initial hidden embedding $\boldsymbol{H}^{(0,L)}$ is $\boldsymbol{E}_{tok} + \boldsymbol{E}_{pos}$ and $+$ denotes element-wise addition; $\boldsymbol{H}^{(r,l)} \in \mathbb{R}^{t \times d_h}$ denotes the hidden embedding of t logical variables after the l-th (self-attention) block in the r-th recurrent step; $block_l$ denotes the l-th Transformer block in the model; $\mathrm{layer\_norm}$ denotes layer normalization; $\cdot$ denotes matrix multiplication, $\boldsymbol{W}_{out} \in \mathbb{R}^{d_h \times c}$ is the weight of the output layer; and $\boldsymbol{X}^{(r,l)} \in [0,1]^{t \times c}$ denotes the NN output with the hidden embedding $\boldsymbol{H}^{(r,l)}$.

Each $block_l$ is defined on weights $\boldsymbol{W}_K^{(l)}, \boldsymbol{W}_Q^{(l)}, \boldsymbol{W}_V^{(l)}, \boldsymbol{W}_P^{(l)} \in \mathbb{R}^{d_h \times d_h}$ (we describe a single head case for simplicity) and a Multi-Layer Perceptron MLP_l with output size d_h.

$$\mathbf{K}^{(r,l)} = \mathrm{layer\_norm}(\boldsymbol{H}^{(r,l)}) \cdot \boldsymbol{W}_K^{(l)} \qquad \mathbf{Q}^{(r,l)} = \mathrm{layer\_norm}(\boldsymbol{H}^{(r,l)}) \cdot \boldsymbol{W}_Q^{(l)}$$

$$\mathbf{V}^{(r,l)} = \mathrm{layer\_norm}(\boldsymbol{H}^{(r,l)}) \cdot \boldsymbol{W}_V^{(l)} \qquad \mathbf{A}^{(r,l)} = \mathrm{softmax}(\frac{\mathbf{Q}^{(r,l)}(\mathbf{K}^{(r,l)})^T}{\sqrt{d_h}})$$

$$\mathbf{V}^* = (\mathbf{A}^{(r,l)} \cdot \mathbf{V}^{(r,l)}) \cdot \boldsymbol{W}_P^{(l)} + \boldsymbol{H}^{(r,l)}$$

$$\mathrm{block}_l(\boldsymbol{H}^{(r,l)}) = \mathrm{MLP}_l(\mathrm{layer\_norm}(\mathbf{V}^*)) + \mathbf{V}^*$$

Here, $\boldsymbol{H}^{(r,l)}, \mathbf{K}^{(r,l)}, \mathbf{Q}^{(r,l)}, \mathbf{V}^{(r,l)}, \mathbf{V}^* \in \mathbb{R}^{t \times d_h}$ and $\mathbf{A}^{(r,l)} \in [0,1]^{t \times t}$.

Figure 5.1 shows a multi-layer Transformer encoder architecture (**a**) and the Recurrent Transformer architecture in our work (**b**), where every dotted box denotes a self-attention block. An output layer consists of a layer normalization, a linear layer, and a softmax activation function. In (**b**), all output layers share the same parameters, while every self-attention block has its own parameters.

For logical variable $\mathcal{X}_i$ and its domain $\mathbb{D}_i = \{1, \ldots, c_i\}$ where $c_i \leq c$, the scalar $X_{i,j}^{(r,l)}$ (i.e., element i, j of matrix $\boldsymbol{X}^{(r,l)}$) is interpreted as the probability of atom $\mathcal{X}_i = j$ for $j \in \{1, \ldots, c_i\}$.

Example 10 (Recurrent Transformer for Visual Sudoku) *Figure 5.2 shows how a Recurrent Transformer is used to solve the Visual Sudoku problem from (Wang* et al.*, 2019). Here, a Sudoku board is represented by* $9 \times 9 = 81$ *MNIST digit images where empty*

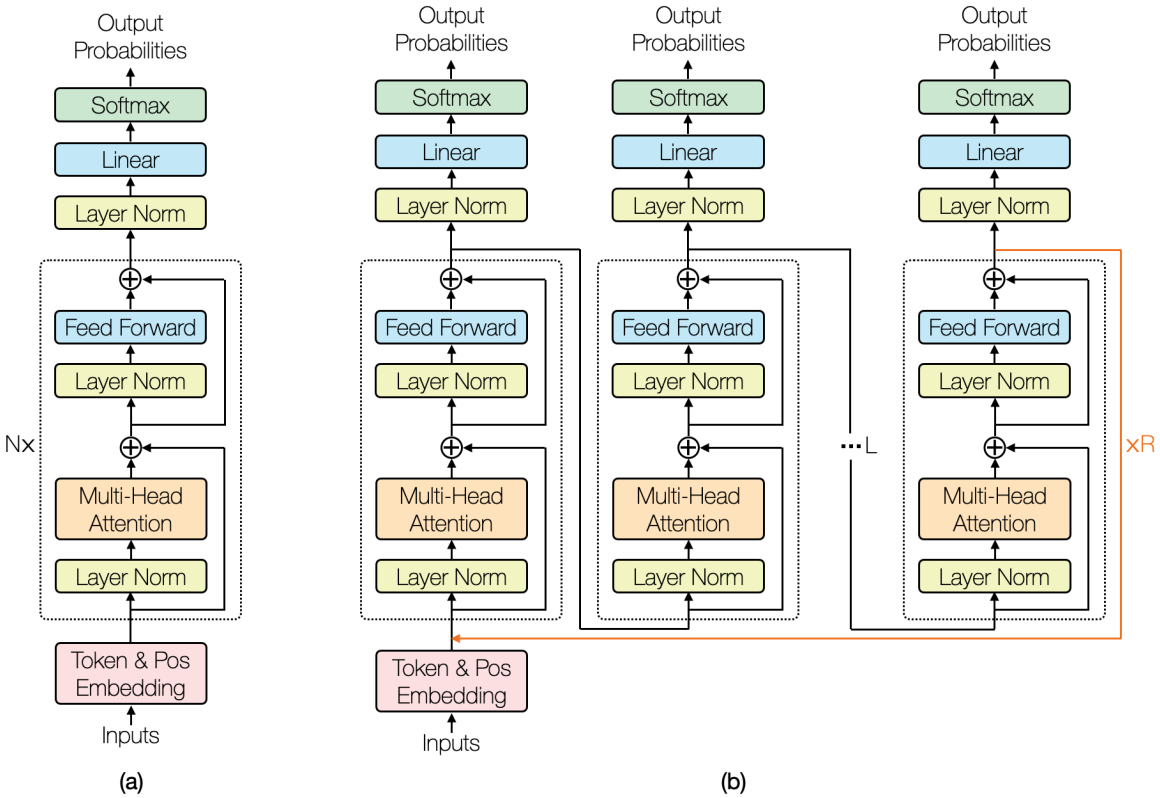

Figure 5.1: (a) Transformer Encoder. (b) Recurrent Transformer.

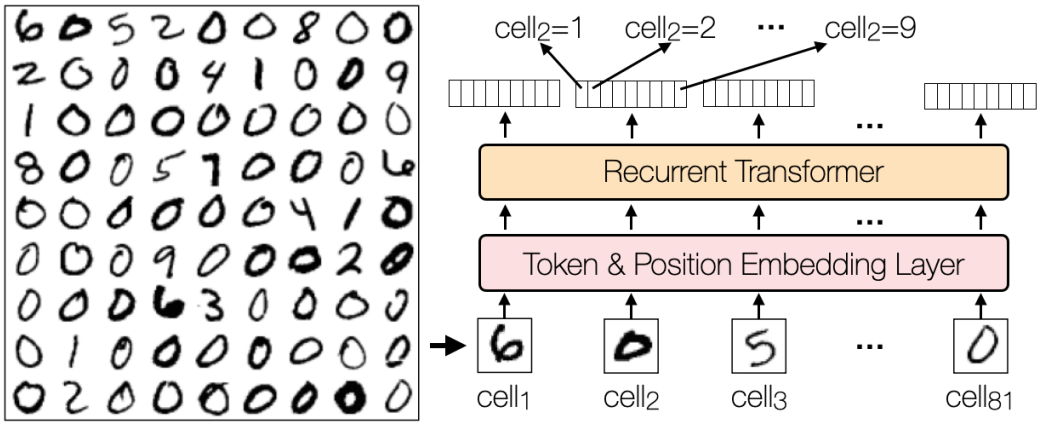

Figure 5.2: Recurrent Transformer for Visual Sudoku Problem.

cells are represented by images of digit 0. The Recurrent Transformer takes as an input the sequence $\langle cell_1, \ldots, cell_{81} \rangle$ *of logical variables, and outputs the probability distribution over atoms* $cell_i = v$ *for* $i \in \{1, \ldots, 81\}, v \in \{1, \ldots, 9\}$*. The given information for each logical variable* $cell_i$ *is the MNIST digit image in the* i*-th cell. Within the Recurrent*

Transformer, the token, position, and hidden embeddings $\boldsymbol{E}_{tok}, \boldsymbol{E}_{pos}, \boldsymbol{H}^{(r,l)}$ *are in* $\mathbb{R}^{81\times128}$*, and the output* $\boldsymbol{X}^{(r,l)}$ *is in* $\mathbb{R}^{81\times9}$*.*

Discussion about Recurrence in Transformers. Adding recurrence to standard Transformer is not a new idea, but its application to CSP is novel. Other implementations of recurrence in Transformers (Dehghani *et al.*, 2019; Hao *et al.*, 2019) largely follow the traditional training and inference procedure, where causal attention encourages the model to focus on the next token to be generated. In contrast, we use an encoder-only model and force the Transformer to update all unknown variables at every recurrence, which encourages it to learn a convergent reasoning algorithm while considering the global picture to generate a solution incrementally in any order. We find that with this framework, the Transformer solves the CSP problems incrementally, gradually predicting more unknown variables after it is confident in others. Since Universal Transformers (Dehghani *et al.*, 2019) produce outputs sequentially, they are forced to predict output variables in an arbitrary fixed order, only attending to previous sequence elements, which may not be suitable for reasoning about many CSPs. We remove masked attention during training and inference, and instead allow for attention to every other input variable. With our design, during inference time, an arbitrary number of recurrences can be used (different number from the training time), which can further boost performance. Furthermore, instead of computing a single loss on the final output as in Universal Transformer, we accumulate loss for each output from every attention block at every recurrent step, which yields better performance, as shown in Section 5.4.3.

5.3 Training Objective

Consider a labeled data instance $\langle \boldsymbol{t}, \boldsymbol{l} \rangle$ where $\boldsymbol{t}$ is t input tokens (that will be turned into the token and position embeddings $\mathbf{E}_{tok}$, $\mathbf{E}_{pos}$) and $\boldsymbol{l} \in \{na, 1, \ldots, c\}^t$ is a label for $\boldsymbol{t}$, where na denotes unknown label. Let $\boldsymbol{X}^{(r,l)} \in \mathbb{R}^{t\times c}$ be the NN output with input $\boldsymbol{t}$ at

recurrent step $r \in \{1, \ldots, R\}$ and block $l \in \{1, \ldots, L\}$. The cross-entropy loss $\mathcal{L}_{cross}$ is defined as follows, where l_i denotes element i in $\boldsymbol{l}$.

$$\mathcal{L}_{cross}(\boldsymbol{X}^{(r,l)}, \boldsymbol{l}) = - \sum_{i \in \{1,\ldots,t\},\ j \in \{1,\ldots,c\},\ l_i = j} log(X_{i,j}^{(r,l)}).$$

For example, in ungrounded Visual Sudoku, $\boldsymbol{t}$ is a list of $t = 81$ MNIST images and $\boldsymbol{l} \in \{na, 1, \ldots, 9\}^{81}$ is the "ungrounded" solution for the Sudoku puzzle where the label for all given digits is na. The cross-entropy loss on NN output $\boldsymbol{X}^{(r,l)} \in \mathbb{R}^{81 \times 9}$ depends only on empty cell predictions. In other words, no supervision for given digits is provided during training.

The baseline loss $\mathcal{L}_{base}$ is the sum of $\mathcal{L}_{cross}$ over NN output $\boldsymbol{X}^{(r,l)}$ from all recurrent steps and blocks.

$$\mathcal{L}_{base} = \sum_{r \in \{1,\ldots,R\},\ l \in \{1,\ldots,L\}} \mathcal{L}_{cross}(\boldsymbol{X}^{(r,l)}, \boldsymbol{l}).$$

Note that we apply cross-entropy loss to the NN outputs from all recurrent steps and all layers instead of from the very last one. We find that this makes the Recurrent Transformer converge faster.

5.4 Evaluation

We use LxRyHz to denote our Recurrent Transformer with $L = x$ self-attention blocks, $R = y$ recurrent steps, and z self-attention heads. If omitted, the number of heads z is 4 and the embedding size d_h is 128.

5.4.1 Textual and Visual Sudoku

In this section, we apply Recurrent Transformers to solve Sudoku problem, where a board can be either textual or visual.

Sudoku Datasets

For textual Sudoku, we use the SATNet dataset from (Wang *et al.*, 2019) and the RRN dataset from (Palm *et al.*, 2018). The difference is that RRN dataset is much harder and bigger (with 17-34 given digits in each puzzle and 180k/18k training/test data) than the SATNet dataset (with 31-42 given digits and 9k/1k training/test data). Each labeled data instance in textual Sudoku is $\langle \boldsymbol{t}, \boldsymbol{l} \rangle$ where $\boldsymbol{t} \in \{0, \ldots, 9\}^{81}$ denotes a Sudoku puzzle (0 represents an empty cell) and $\boldsymbol{l} \in \{1, \ldots, 9\}^{81}$ is the solution to the puzzle. For Visual Sudoku, we use the ungrounded SATNet-V dataset from (Topan *et al.*, 2021). SATNet-V was created based on the SATNet dataset where (i) each textual input in $\{0, \ldots, 9\}$ in the training (or testing resp.) split is replaced with a randomly-selected MNIST image in the MNIST training (or testing resp.) dataset, and (ii) the label for each given digit is na, i.e., unknown label that cannot be used to help training. In addition to SATNet-V, we created a new ungrounded dataset, RRN-V, following the same procedure based on RRN dataset. For faster evaluation on RRN-V dataset, we randomly sampled 9k/1k training/test data and denoted it by "RRN-V (9k/1k)".

Baselines and our Model for Sudoku

We take RRN and SATNet as the baselines for textual Sudoku, and take RRN, SATNet, and SATNet* (Topan *et al.*, 2021) (which resolves the symbol grounding issue of SATNet by clustering the input images) as the baselines for Visual Sudoku. As RRN was not designed for Visual Sudoku, we applied the same convolutional neural network (CNN) from (Wang *et al.*, 2019) to turn each MNIST image into the initial number embedding of that cell in RRN. For our method on both textual and Visual Sudoku, we apply Recurrent Transformer as in Figure 5.2 where the only difference is that the token embedding layer is a linear embedding layer for textual Sudoku and is the same CNN for Visual Sudoku. All evaluations

of our model use 32 recurrence steps for training and 64 for evaluation, as is done in (Palm *et al.*, 2018).

Table 5.1: Whole Board Accuracy on Different Sudoku Datasets. RRN-hardest Consists of a Copy of the RRN Training Set, While the Testing Set Consists of Only the Hardest Puzzles with 17 Given Digits in the RRN Test Set. We Compare with 3 Baselines: RRN (Palm *et al.*, 2018), SATNet (Wang *et al.*, 2019), And SATNet* (Topan *et al.*, 2021).

		Textual Sudoku			Visual Sudoku (Ungrounded)	
	dataset	SATNet	RRN	RRN-hardest	SATNet-V	RRN-V
	#given	31-42	17-34	17-34	31-42	17-34
	(#train/#test)	(9k/1k)	(180k/18k)	(180k/1k)	(9k/1k)	(9k/1k)
Models	#Param (text/visual)	Accuracy on test data				
RRN	201k / 692k	**100%**	98.9%	96.6%	0%	0%
SATNet	618k / 1049k	98.3%	6.1%	0%	0%	0%
SATNet*	– / 1049k + 13M(InfoGAN)	–	–	–	64.8%	0%
L1R32H4 (ours)	211k / 702k	**100%**	**99.5%**	**96.7%**	**93.5%**	**75.6%**

Table 5.1 shows that our method outperforms state-of-the-art neural network models for both textual and Visual Sudoku in different difficulties. Note that among all methods, only RRN requires prior knowledge about Sudoku rules (i.e., there is an edge in the graph between every 2 nodes if their related cells are in the same row/column/box). Both RRN and SATNet fail on the (ungrounded) SATNet-V dataset due to the symbol grounding issue. While SATNet* could learn to solve Visual Sudoku with the ungrounded dataset, it requires training an InfoGAN with 13M parameters to cluster the inputs. Unlike SATNet*, our model works out-of-the-box on Visual Sudoku without carefully adjusting the structure and outperforms SATNet* by a large margin.

For textual Sudoku problem, Table 5.2 shows the number of parameters in our model and where they come from. There are a total of 211,328 parameters in a L1R32H4 model computed based on input vocabulary size (v), context size (t), number of classes (c), hidden embedding size (d_h), and the hidden layer size (d_{MLP}) of MLP_l, which is of shape (d_h, d_{MLP}, d_h). For SATNet (Wang *et al.*, 2019), the number of parameters is 618,000 total.

Table 5.2: Parameter Values And Counts for L1R32H4 Model for Textual Sudoku

Operation	Parameters	Parameter Count
Token Embedding	$v \times d_h$	$10 \times 128 = 1280$
Positional Embedding	$t \times d_h$	$81 \times 128 = 10,368$
Multi-Head Self-Attention	$4(d_h^2 + d_h)$	$4(128^2 + 128)$
$(\boldsymbol{W}_K^{(l)}, \boldsymbol{W}_Q^{(l)}, \boldsymbol{W}_V^{(l)}, \boldsymbol{W}_P^{(l)})$	(the d_h is for bias)	$= 66,048$
Layer normalization	$3 \times 2d_h$	$3 \times 2 \times 128 = 768$
MLP_l	$d_h d_{MLP} + d_{MLP}$	$2 \times 128 \times 512 + 512$
(d_h, d_{MLP}, d_h)	$+d_{MLP} d_h + d_h$	$+128 = 131,712$
Output layer $\boldsymbol{W}_{out}$	$d_h \times c$	$128 \times 9 = 1,152$

This is $(n + 1 + aux) \times m$, where n is the number of input variables, aux is the number of auxiliary variables, and m is the rank of the clause matrix. The RRN (Palm *et al.*, 2018) has a total of 201,194 parameters, which come from the row, column, and number embeddings, and the three MLPs used for node updates, message passing, and producing output probabilities.

Although the L1R32H4 model has already achieved the new state-of-the-art results, we may further improve the accuracy by increasing the number L of attention blocks, the number H of heads, or the hidden embedding size d_h, with a trade-off of bigger model size. We will analyze the effects of these decision choices on smaller datasets in Section 5.4.3.

5.4.2 More Domains

16x16 Sudoku

We train our L1R32H8 model ($d_h = 256$) on 16x16 textual Sudoku. We generate two 10k (9k/1k training/test split) datasets of difficulty "simple" with an average of 111 givens

and "medium" with an average of 95 givens. With 64 recurrent steps during inference, we achieved 99.9% accuracy on both test sets, meaning there is only one wrongly predicted board solution. Due to the absence of a 16x16 Sudoku generator that can produce fewer givens, we could not test on harder boards.

MNIST Mapping

The MNIST Mapping problem was proposed in (Chang *et al.*, 2020) as a simple test for the symbol grounding problem. It requires learning a bijection that maps an image of MNIST digit to one of the 10 symbolic digits. (Chang *et al.*, 2020) shows that SATNet is sensitive to this task and often fails without delicate tuning. Our Recurrent Transformer achieves 99% accuracy.

Nonograms

Nonogram (https://en.wikipedia.org/wiki/Nonogram) is a game that consists of an initially empty $N \times N$ grid representing a binary image, where each cell must take on a value of 0 or 1. Each row and column have constraints that must be satisfied to complete the image successfully. A constraint for a row/column is a list of numbers, where each number corresponds to contiguous blocks of cells for a row/column. We created two datasets for 7x7 and 15x15 grids, each has a 9k/1k training/test split. We use the same Recurrent Transformer model as in previous experiments. A sample $N \times N$ grid is input as a N^2 long sequence, where each element is a concatenated representation of the row and column constraints associated with the element. For example, for 15x15 grids, a given cell with the column constraint [1,7,4] and row constraint [2,2,4,1] would have a corresponding sequence element of the concatenation of the two constraint vectors [0,0,1,7,4] and [0,2,2,4,1] (assuming the maximum constraint length is 5). With this simple input encoding only, our Recurrent Transformer L1R16H4 achieved 97.5% test accuracy on 7x7 grids and 78.3%

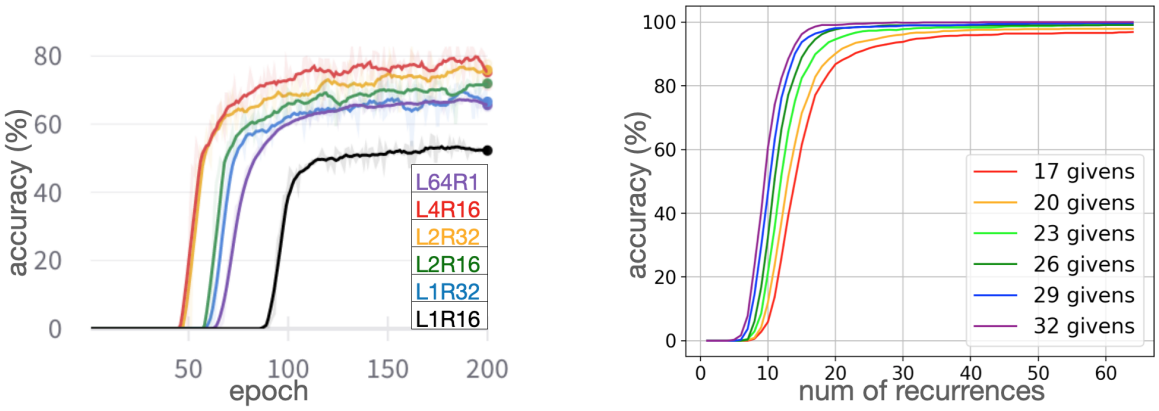

Figure 5.3: (left) Running Average of the Test Accuracy for Every 10 Epochs of a Recurrent Transformer with Different L And R Trained on the Same 8k RRN Data. (right) Test Accuracy as a Function of the Number T of Recurrences When Testing on Different Difficulty Puzzles in RRN Dataset, Using the Same L1R32 Model Trained on 180k RRN Data.

test accuracy on 15x15 grids.

5.4.3 Ablation Study

Effects of Blocks and Recurrences

To analyze the effects of blocks and recurrences, we trained six LxRy(H4) models with different numbers of self-attention blocks L and recurrences R on 8k/2k (training/test) RRN dataset with $\mathcal{L}_{base}$. Figure 5.3 (**left**) compares the whole board accuracy of these models, showing that more self-attention steps (equal to $L \times R$) lead to higher accuracy. With the same number of self-attention steps, when L is small (e.g., $L \leq 4$), more parameters introduced by a bigger L slightly increase the accuracy. On the other hand, adding recurrences is essential, and the non-recurrent model L64R1 performs poorly compared to the Recurrent Transformers.

The number of recurrences T during testing can be higher than R during training. Indeed, Figure 5.3 (**right**) shows that when $T \leq 64$, the more recurrent steps T is, the higher accuracy is achieved with the same L1R32 model trained on 32 steps, and the improvement is bigger for harder puzzles.

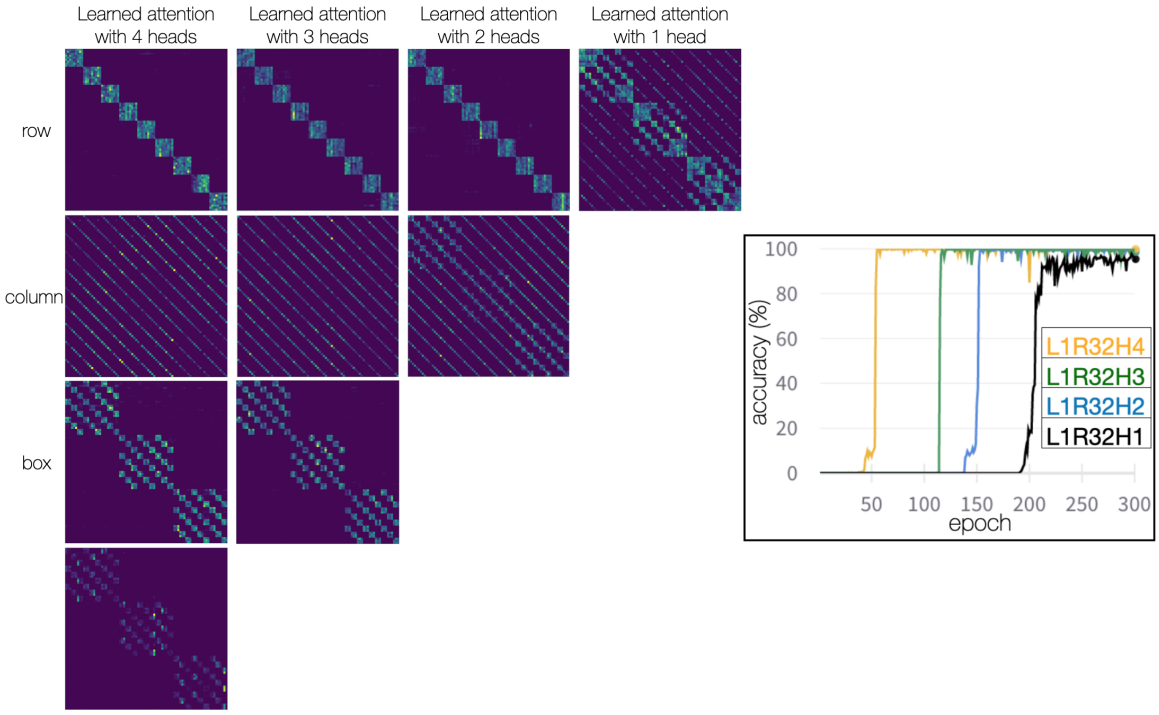

Figure 5.4: (left) Heatmaps of the Learned 81x81 Attention Matrices in the L1R32 Recurrent Transformer with Varying Numbers of Heads. (right) Test Accuracy VS. Epochs for These Models.

Effects and Visualization of Multi-Head Attention

Without prior knowledge of the Sudoku game, the Recurrent Transformer learns purely from $\langle puzzle, solution \rangle$ pairs so that each cell should pay attention to all cells in the same row, column, and 3×3 box through the attention mechanism. We trained an L1R32 model with 1 to 4 self-attention heads on the SATNet dataset with $\mathcal{L}_{base}$. Figure 5.4 (**left**) visualizes the learned attention matrices – they correctly pay attention to each row, column, and box, respectively. For example, the first row of the top-left attention matrix in Figure 5.4 (**left**) learns purely from data about the 9 atoms to pay attention in constraint (5.3) where $\{i, \ldots, j\} = \{1, \ldots, 9\}$ and $d = 1$. These attentions are clearly separated in different heads if the number of heads is greater or equal to 3 and would merge otherwise. Figure 5.4 (**right**) compares the whole board accuracy of these models, showing that more attention heads help faster convergence, and the accuracy may decrease if the number of

attention heads is too small to capture different semantic meanings.

Effect of Positional Embedding

To evaluate the effect of positional embedding, we trained the L1R32 model without positional embedding on the SATNet dataset with $\mathcal{L}_{base}$, finding that removing positional embedding decreases the test accuracy from 100% to 0%. This is because positional embedding is essential for a CSP as it is the only source to differentiate logical variables (e.g., $cell_1, \ldots, cell_{81}$) with the same given information (e.g., digit 2 in both cell 4 and cell 10 in Figure 5.2).

Analyses on Symbol Grounding with Visual Sudoku

In Visual Sudoku, we observed similar effects of different model designs as in textual Sudoku. In this section, we analyze how the symbol grounding issue is resolved in Recurrent Transformer by applying the same L1R32 model on both RRN-V dataset and its grounded version (i.e., the label for every given digit is provided instead of na). For each of the two trained models, we evaluate their (i) whole board accuracy, (ii) *solution* accuracy where a board is counted correct if the prediction on all non-given cells are correct (even when the given digits are incorrectly classified), and (iii) *givens cell* accuracy, i.e., per-cell classification accuracy of the given cells.

When trained on the grounded dataset, the L1R32 model quickly learns to classify the givens in 1 epoch, as shown in Figure 5.5 (**right**). On the other hand, with the ungrounded dataset, the L1R32 model starts to classify the givens correctly at around epoch 85. In Figure 5.5 (**left**) and (**right**), the solution accuracy and givens cell accuracy (with the ungrounded dataset) increase around the same time, indicating that digit classification is being jointly learned with solving. Interestingly, our model achieves 99.36% classification (givens cell) accuracy without explicitly training for it. Furthermore, the solution

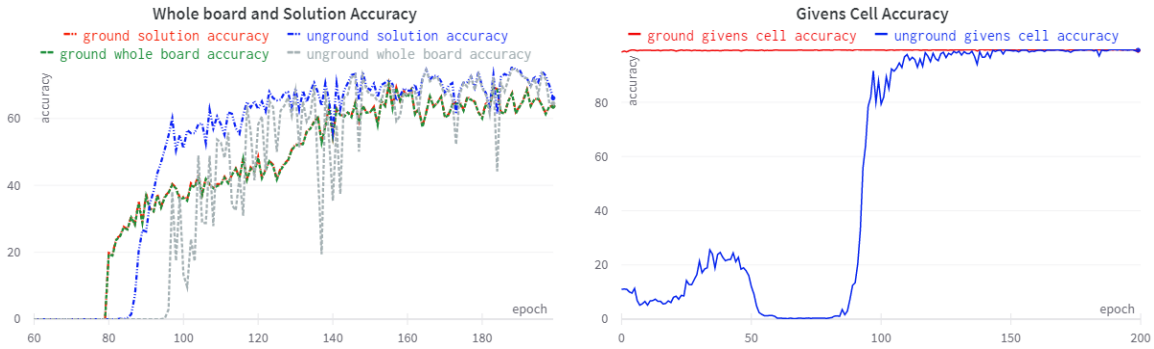

Figure 5.5: (left) The Whole Board Accuracy And Solution Accuracy of the L1R32 Model Trained on Grounded or Ungrounded RRN-V Dataset (9k/1k). (right) The Givens Cell Accuracy of the Same Models.

accuracy (75.5%) is consistently higher than the whole board accuracy (74.8%) as shown in Figure 5.5 (**left**), meaning that even when givens are not correctly classified, the solution can still be attained. We attribute this to the fact that reasoning is on the latent space instead of classifying and solving in two steps, as SATNet does.

Recurrent Transformer vs. Vanilla Transformer

There are two main decision choices in Recurrent Transformer: adding recurrence and applying losses to all blocks at all recurrent steps. To justify our decision choices, we compared 3 Transformer designs on the textual Sudoku problem under 3 settings. Figures 5.3, 5.4, and 5.5 show the experimental results on textual Sudoku on SATNet (9k/1k for training/testing), Palm (9k/1k), and Palm (3k/1k) datasets where

- the black line denotes the vanilla Transformer L32R1 with 32 blocks and with the cross-entropy loss applied to the final output;
- the yellow line denotes the Recurrent Transformer L1R32 with a single block, 32 recurrences, and with the cross-entropy loss applied to the last output;
- the red line denotes the Recurrent Transformer L1R32 with a single block, 32 recurrences, and with the cross-entropy loss applied to 32 outputs.

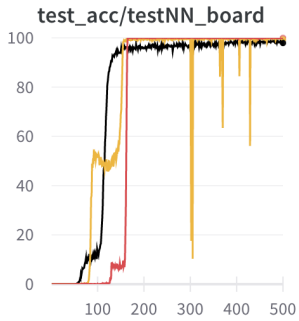

Table 5.3: Whole-board Test Accuracy on SATNet (9k/1k)

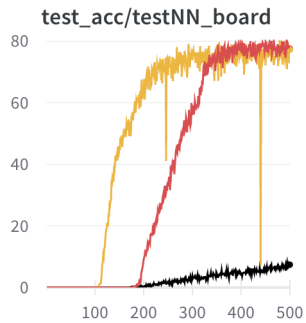

Table 5.4: Whole-board Test Accuracy on Palm (9k/1k)

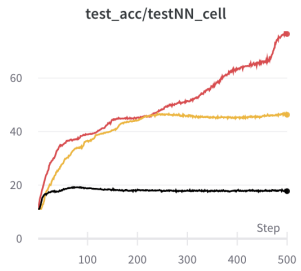

Table 5.5: Cell Test Accuracy on Palm (3k/1k)

We can see that

- (comparing black and yellow lines) adding recurrences allows the model to achieves higher accuracy (especially for harder problems in Palm dataset) with a much fewer parameters in the model (1 block vs. 32 blocks);
- (comparing yellow and red lines) applying losses to all blocks makes the Recurrent Transformer model more stable and achieve higher accuracy than the Recurrent Transformer with a single loss;
- (comparing Figure 5.5 with the other 2 figures) the benefit of recurrence and losses on all blocks is more when the number of data is less. Figure 5.5 compares the cell accuracy under the above 3 settings when trained on only 3k Palm data. In this figure, using recurrent blocks increases the converged cell accuracy from 17.7% to 46.3%, and applying losses to all blocks further improves the cell accuracy to 76.5%, and it has not converged.

5.5 Cardinality Constraint

5.5.1 Injecting General Cardinality Constraints via STE

Although Recurrent Transformers can learn to solve CSPs purely from labeled data, we could inject the known constraints to help Transformers learn with fewer labeled data. In this section, we follow the idea from CL-STE and propose a lightweight constraint loss method for a special family of constraints in CSP, namely the cardinality constraint that restricts the number of atoms in a set that can hold at the same time.

While CL-STE has successfully injected discrete constraints into NN training, representing cardinality constraints in CNF is tedious. On the other hand, we notice that the binarization function $\mathtt{t}(x)$ enables direct counting on discrete values. Under this observation, for the cardinality constraint

$$l \leq |\{\mathcal{X}_{i1} = v_1, \ldots, \mathcal{X}_{ik} = v_k\}| \leq u$$

we construct a vector $\boldsymbol{x} \in \mathbb{R}^k$ of probabilities of the atoms in the given set such that x_j (i.e., element j in $\boldsymbol{x}$) is the probability of $\mathcal{X}_{ij} = v_j$ for $j \in \{1, \ldots, k\}$, and design a constraint loss as follows

$$\mathcal{L}_{[l,u]}(\boldsymbol{x}) = \mathbf{1}_{c(\boldsymbol{x})<l} \times (c(\boldsymbol{x}) - l)^2 + \mathbf{1}_{c(\boldsymbol{x})>u} \times (c(\boldsymbol{x}) - u)^2 \tag{5.4}$$

where scalar $c(\boldsymbol{x}) = \sum \mathtt{t}(\boldsymbol{x}) = \sum_j \mathtt{t}(x_j)$, and $\mathbf{1}_{\text{condition}}$ is 1 if condition is true, 0 otherwise. Similarly, constraint $|\{\mathcal{X}_{i1} = v_1, \ldots, \mathcal{X}_{ik} = v_k\}| = n$ can be encoded in the following loss.

$$\mathcal{L}_{[n]}(\boldsymbol{x}) = (c(\boldsymbol{x}) - n)^2. \tag{5.5}$$

In constraint losses (5.4) and (5.5), $c(\boldsymbol{x})$ is the number of 1s in the binarized vector $\mathtt{t}(\boldsymbol{x})$, which corresponds to counting the number of true atoms in constraints (5.1) and

(5.2). Note that binarization function $\mathtt{t}(x)$ enables the counting, but its gradient $\frac{\partial \mathtt{t}(x)}{\partial x}$ is always 0 whenever differentiable, so minimizing (5.4) and (5.5) won't work on updating NN parameters.

As with CL-STE, we use the identity STE to replace the gradient $\frac{\partial \mathtt{t}(x)}{\partial x}$ with 1 so that the gradient of each constraint loss to $\mathtt{t}(x)$ becomes the "straight-though estimator" of the gradient to x. In this way, we can do counting on the NN output with meaningful gradients. While CL-STE could also represent the constraint loss (5.5) for $n = 1$ (uniqueness and existence of values), the size of the CNF representation could be huge.

Example 11 (Constraint Loss on Output) *Cardinality constraint loss (5.5) can be used to define the constraints in Sudoku problem*

$$\mathcal{L}_{Sudoku}(\boldsymbol{X}^{(r,l)}) = \sum_{k\in\{row,col,box\}} \sum_{i\in\{1,\ldots,81\}} \mathcal{L}_{[1]}(\boldsymbol{X}^k_{i,:}),$$

where $\boldsymbol{X}^{(r,l)} \in \mathbb{R}^{81\times 9}$ is the NN output; $\mathbf{X}^{row}, \mathbf{X}^{col}, \mathbf{X}^{box} \in \mathbb{R}^{81\times 9}$ are reshaped copies of $\boldsymbol{X}^{(r,l)}$ such that each row in them contains the predictions in the same row/column/box; and $\boldsymbol{X}^k_{i,:}$ denotes row i of matrix $\boldsymbol{X}^k$. Intuitively, $\mathcal{L}_{Sudoku}$ says that "exactly one digit in $\{1,\ldots,9\}$ can be predicted in the same row/column/box". Note that, in CL-STE, the same Sudoku constraints are represented by a CNF with 729 atoms and 8991 clauses, which requires computation on a big matrix in $\{-1,0,1\}^{8991\times 729}$.

Furthermore, since the values in vector $\boldsymbol{x}$ are not limited to probabilities in NN outputs, we can apply these cardinality constraint losses to an attention matrix, representing additional constraints (not in the original CSP) that should be satisfied by the attention.

Example 12 (Constraint Loss on Attention) *In Sudoku problem, an attention matrix $\boldsymbol{A}^{(r,l)} \in \mathbb{R}^{81\times 81}$ is computed in the l-th block at the r-th recurrence where $A^{(r,l)}_{i,j}$ is a normalized attention weight that can be interpreted as the percentage of attention from cell*

i to cell j. The cardinality constraint loss (5.5) can also be used to define the following constraint loss

$$\mathcal{L}_{attention}(\boldsymbol{A}^{(r,l)}) = \mathcal{L}_{[81]}(\boldsymbol{x})$$

where $\boldsymbol{x} = \sum_j (\boldsymbol{A}^{(r,l)}_{:,j} \odot \boldsymbol{M}_{:,j})$; $\boldsymbol{M}$ is the adjacency matrix in $\{0,1\}^{81\times 81}$ such that $M_{i,j}$ is 1 iff cells i and j are in the same row, column, or box; $\odot$ denotes element-wise multiplication. Intuitively, the i-th element in $\boldsymbol{x} \in \mathbb{R}^{81}$ denotes the probability of the i-th cell paying attention to its adjacent cells. Minimizing $\mathcal{L}_{attention}(\boldsymbol{A}^{(r,l)})$ makes all 81 cells pay attention to their adjacent cells.

Similar to the baseline loss $\mathcal{L}_{base}$, which is the sum of $\mathcal{L}_{cross}$ over NN output $\boldsymbol{X}^{(r,l)}$ from all recurrent steps and blocks, the total constraint loss $\mathcal{L}_{constraint}$ is also accumulated over all NN outputs. The total loss with constraint loss is $\mathcal{L}_{total} = \mathcal{L}_{base} + \mathcal{L}_{constraint}$.

The constraint loss for Sudoku problem is

$$\mathcal{L}_{constraint} = \sum_{r\in\{1,\ldots,R\},\ l\in\{1,\ldots,L\}} \Big(\alpha \mathcal{L}_{Sudoku}(\boldsymbol{X}^{(r,l)}) + \beta \mathcal{L}_{attention}(\boldsymbol{A}^{(r,l)})\Big),$$

where α, β are reals in $[0,1]$ that are hyper-parameters specified in the next section.

5.5.2 Effect of Combinations of Constraints

Applying Constraint Losses to Sudoku

To evaluate the effects of different logical constraint losses, we trained the L1R32 model on 9k/1k (training/test) RRN dataset and 9k/1k RRN-V dataset for 300 epochs till convergence with and without constraint losses. Table 5.6 shows that the same Recurrent Transformer model can further be improved if, in the total loss, we include $\mathcal{L}_{attention}$ and/or $\mathcal{L}_{Sudoku}$ on each neural network output, where the accuracy is evaluated with 32 or 64 recurrent steps T during testing. We also observe a better performance gain with $\mathcal{L}_{Sudoku}$ than with $\mathcal{L}_{attention}$

Table 5.6: Effect of Adding Constraint Losses $\mathcal{L}_{attention}$ (att) And $\mathcal{L}_{Sudoku}$ (sud) to the Baseline Loss $\mathcal{L}_{base}$ When Training the Same L1R32 Model on 9k RRN or RRN-V Training Data.

		Textual Sudoku		Visual Sudoku	
att	sud	T=32	T=64	T=32	T=64
–	–	80.3%	81.9%	72.0%	75.6%
–	✓	80.1%	84.4%	74.4%	79.3%
✓	–	83.8%	86.3%	76.4%	79.1%
✓	✓	83.3%	87.0%	79.9%	83.6%

Table 5.7: Effect of Adding Constraint Loss $\mathcal{L}_{constraint}$ And x Thousand **U**nlabeled Data (Denoted by xkU) When Training the Same L1R32 Model on 4k **L**abeled RRN or RRN-V Training Data (Denoted by 4kL).

Data	Textual Sudoku		Visual Sudoku	
	T=32	T=64	T=32	T=64
4kL	58.0%	62.0%	40.9%	44.0%
4kL + $\mathcal{L}_{constraint}$	65.4%	69.2%	47.5%	50.4%
4kL + 4kU + $\mathcal{L}_{constraint}$	65.8%	69.9%	57.2%	61.0%
4kL + 8kU + $\mathcal{L}_{constraint}$	70.7%	73.3%	60.8%	64.4%

because the baseline model (trained with $\mathcal{L}_{base}$ only) already learns the attention matrices well, as shown in Figure 5.4 in Section 5.4.3. Besides, when we use 64 recurrences during testing (whereas trained with 32 recurrences), the same Recurrent Transformer model has bigger improvements on the test accuracy when it is also trained with constraint losses.

Since constraint loss $\mathcal{L}_{constraint}$ (accumulated by $\mathcal{L}_{Sudoku}$ and $\mathcal{L}_{attention}$) doesn't require labels, we could use it for semi-supervised learning tasks. Table 5.7 shows that, with only 4k labeled data, adding $\mathcal{L}_{constraint}$ increases the whole board accuracy of the same 1k test data, which can further be improved by adding additional 4k and 8k unlabeled data along

with $\mathcal{L}_{constraint}$.

In Table 5.7, we showed how constraint loss helps in a semi-supervised setting for textual and visual Sudoku. To analyze the effect of constraint loss on more unlabeled data instances, we continued the experiments for both textual and visual Sudoku and recorded the running average of the test accuracy for every 10 epochs in Figures 5.6 and 5.7.

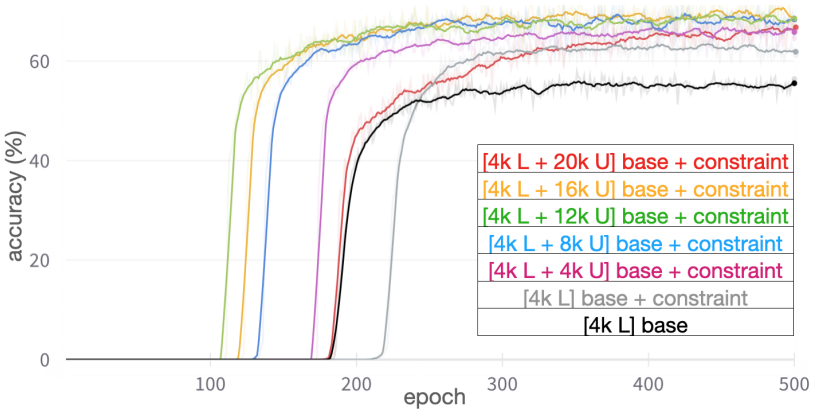

Figure 5.6: Effect of adding constraint loss $\mathcal{L}_{constraint}$ and x thousand **U**nlabeled data (denoted by xkU) when training the same L1R32 model on 4k **L**abeled RRN training data (denoted by 4kL).

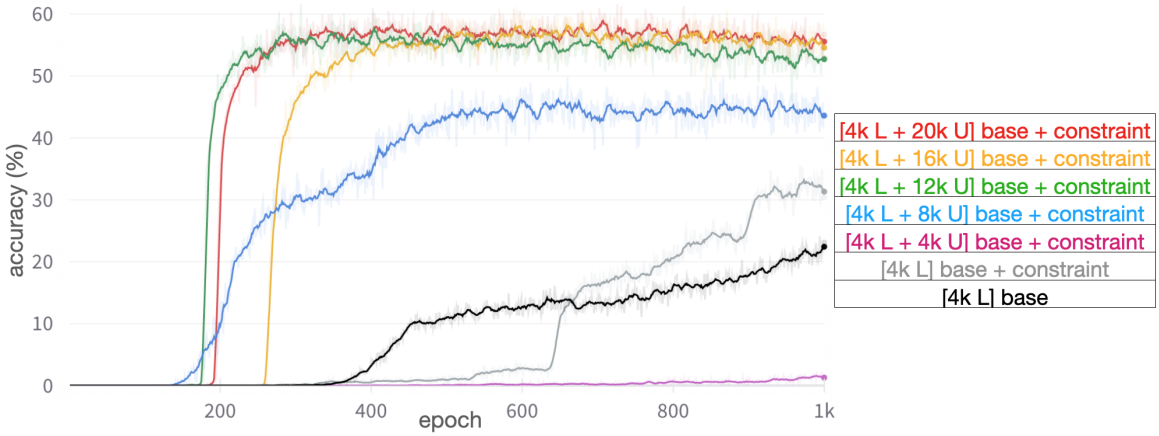

Figure 5.7: Effect of adding constraint loss $\mathcal{L}_{constraint}$ and x thousand **U**nlabeled data (denoted by xkU) when training the same L1R32 model on 4k **L**abeled RRN-V training data (denoted by 4kL).

We set the batch size to around 64 in the experiments in Figure 5.6 and to around 128 in the experiments in Figure 5.7. The batch sizes are slightly adjusted to have integer number split on labeled and unlabeled data in each batch. To reduce the effect of hyper-parameter

tuning on the weights of constraint losses, in all experiments, the weights $\langle \alpha, \beta \rangle$ of the constraint losses $\mathcal{L}_{sudoku}$ and $\mathcal{L}_{attention}$ are set to $\langle 1, 0 \rangle$, i.e., only the constraint loss $\mathcal{L}_{sudoku}$ is used with fixed weight 1. We can see that

- adding constraint loss improves the baseline accuracy by a large margin when trained with limited labeled data;
- with the help of constraint loss, adding more unlabeled data improves the accuracy in the beginning but the improvement is getting smaller;
- if we don't lower the weight for the constraint loss and keep increasing the number of unlabeled data, adding unlabeled data may lower the accuracy at some point as the signals (i.e., gradients) from constraint loss may overwrite the signals from the labels.

Applying Constraint Losses to Shortest Path Problem

A shortest path problem can be viewed as a CSP where $\mathbb{X} = \{node_1, \ldots, node_m, edge_1, \ldots, edge_n\}$ denotes all m nodes and n edges in a graph; $\mathbb{D} = \{\mathbb{D}_1, \ldots, \mathbb{D}_{m+n}\}$ and $\mathbb{D}_i = \{\text{FALSE}, \text{TRUE}\}$; and $\mathbb{C}$ is the set of constraints specifying the two end nodes in the graph and that "the selected edges form a path between the end nodes with minimum length". The goal is to find the solution of this CSP, which represents the solution of the shortest path problem. Here, $node_i = \text{TRUE}$ (or $edge_i = \text{TRUE}$ resp.) represents that node i (or edge i resp.) is in the shortest path. Let $n_1, n_2 \in \{1, \ldots, m\}$ denote the indices of the 2 end nodes. $\mathbb{C}$ contains constraints $node_{n_1} = \text{TRUE}$ and $node_{n_2} = \text{TRUE}$, the following constraint for end nodes $i \in \{n_1, n_2\}$,

$$|\{edge_{i1} = \text{TRUE}, \ldots, edge_{ik} = \text{TRUE}\}| = 1 \tag{5.6}$$

and the following constraint for non-end nodes $i \in \{1, \ldots, m\} \setminus \{n_1, n_2\}$ in the shortest path (given from label),

$$|\{edge_{i1} = \text{TRUE}, \ldots, edge_{ik} = \text{TRUE}\}| = 2 \tag{5.7}$$

where $\{edge_{i1}, \ldots, edge_{ik}\}$ are the edges connected to node i in the graph. The first constraint says that "each end node should connect to exactly 1 edge in the path" and the second constraint says that "each non-end node in the path should connect to exactly 2 edges in the path".

Dataset. We use the shortest path dataset SP4 from (Xu *et al.*, 2018) to illustrate our method where each graph is a 4×4 grid with $m = 16$ nodes and $n = 24$ edges. SP4 has 1610 data instances and, as in (Xu *et al.*, 2018), we split the dataset into 60%/20%/20% training/test/validation examples. In addition, we created a more challenging dataset SP12 where each graph is a 12×12 grid with $m = 144$ nodes and $n = 264$ edges. SP12 has 22k data instances, split into 20k/1k/1k training/test/validation examples. In each problem, two end nodes are randomly picked up, as well as $\frac{n}{3}$ edges are randomly removed to increase the difficulty. A labeled data instance is $\langle \boldsymbol{t}, \boldsymbol{l} \rangle$ where $\boldsymbol{t} \in \{0, 1\}^{m+n}$ such that $t_i = 1$ denotes "node i is a terminal node" when $i \leq m$, and denotes "edge '$i - m$' is not removed" when $i > m$; and $\boldsymbol{l} \in \{0, 1\}^n$ such that $l_i = 1$ denotes "edge i is in the shortest path."

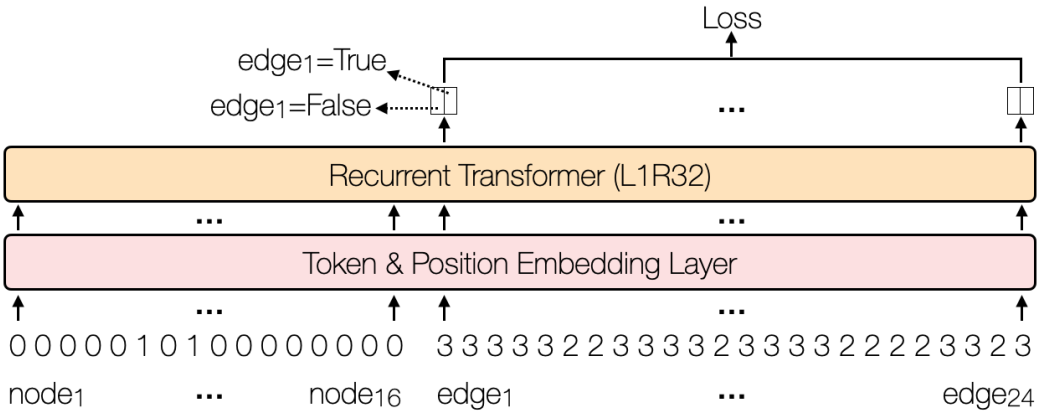

Figure 5.8: Recurrent Transformer for Shortest Path.

Figure 5.8 shows how a Recurrent Transformer is used to solve the shortest path problem in a 4×4 grid with $m = 16$ nodes and $n = 24$ edges. The given information for each logical variable $node_i$ ($i \in \{1, \ldots, 16\}$) is a single digit 1 or 0 denoting that node i is an end node or not. The given information for $edge_i$ ($i \in \{1, \ldots, 24\}$) is a single digit 2 or 3 denoting that the edge i is removed or not. Given a NN output $\boldsymbol{X}^{(r,l)} \in \mathbb{R}^{40\times 2}$ for 40 logical variables, we construct the vector $\boldsymbol{v} \in \mathbb{R}^{24}$ such that $v_i = X^{(r,l)}_{i+16,2}$, denoting the probabilities of $edge_i = \text{TRUE}$ for $i \in \{1, \ldots, 24\}$. Let $\boldsymbol{l} \in \{0, 1\}^{24}$ denote the label, i.e., $l_i = 1$ iff edge i is in the shortest path. The cross-entropy loss is defined on $\boldsymbol{v}$ and $\boldsymbol{l}$.

For the optional constraint loss, let $\boldsymbol{M}$ be the matrix in $\{0, 1\}^{16\times 24}$ such that $M_{i,j} = 1$ iff node i is connected with edge j in the graph. Let $\boldsymbol{c} \in \{0, 1, 2, 3, 4\}^{16}$ be $\boldsymbol{M} \cdot \boldsymbol{l}$. Intuitively, c_i denotes the number of edges in the shortest path containing node i, and $c_i > 0$ means that node i is in the shortest path. Then, constraints (5.6) and (5.7) can be encoded as follows

$$\mathcal{L}_{path}(\boldsymbol{X}^{(r,l)}) = \sum_{i\in\{n_1,n_2\}} \Big(\mathcal{L}_{[1]}(\boldsymbol{M}_{i,:} \odot \boldsymbol{v})\Big) + \sum_{i\in\{1,\ldots,16\}\setminus\{n_1,n_2\}} \Big(\mathbf{1}_{c_i>0} \times \mathcal{L}_{[2]}(\boldsymbol{M}_{i,:} \odot \boldsymbol{v})\Big),$$

where the non-zero values in $\boldsymbol{M}_{i,:} \odot \boldsymbol{v} \in \mathbb{R}^{24}$ are the probabilities of $edge_j = \text{TRUE}$ for all edge j that contains node i.

Table 5.8: Constraint Accuracy on SP4 Test Data for the Shortest Path Problem

Method	Constraint accuracy		
	Path	No removed edges	Shortest path
MLP	28.3%	32.9%	23.0%
MLP + Semantic Loss (Xu *et al.*, 2018)	69.9%	–	–
MLP + NeurASP (Yang *et al.*, 2020)	96.6%	36.3%	33.2%
L1R32 (ours)	84.5%	100%	83.5%
L1R32 + $\mathcal{L}_{path}$ (ours)	91.9%	100%	91.0%

Table 5.8 compares the constraint accuracy achieved by (i) the baseline Multi-Layer

Perceptron (MLP) introduced in (Xu *et al.*, 2018) for the shortest path problem, (ii) the NeurASP method that encodes a path constraint to help train the MLP, (iii) our Recurrent Transformer (L1R32), and (iv) the same Recurrent Transformer enhanced by the constraint loss $\mathcal{L}_{path}$. The constraint accuracy are the percentage of the predictions that (i) form a valid path between end nodes, or (ii) do not include removed edges, or (iii) form a shortest path between end nodes. Table 5.8 shows that the Recurrent Transformer significantly outperforms the baseline MLP. Besides, the constraint loss $\mathcal{L}_{path}$ further improves the accuracy of the same Recurrent Transformer for predicting a valid path (or a shortest path resp.) from 84.5% (or 83.5%) to 91.9% (or 91.0%).

Furthermore, we applied the same L1R32 model to the more challenging SP12 dataset. After 2,000 epochs of training, we achieved 72.3% accuracy when trained with cross-entropy loss only and 76.0% when trained with both cross-entropy loss and constraint loss $\mathcal{L}_{path}$.

5.5.3 Computation Size

The proposed cadinality constraint loss is different from the constraint loss in CL-STE (Yang *et al.*, 2022). We tried the original design of constraint loss in CL-STE but it computes too slow due to the exponential size of CNF used to represent a cardinality constraint. Thus, we invented a new constraint loss for a general cardinality constraint based on counting discrete values in either NN output or attention matrix, which, to the best of our knowledge, is new to the field.

In Table 5.9, we applied the cross-entropy loss, the CL-STE loss, and the cardinality constraint loss to train the same RRN (Palm *et al.*, 2018) on SATNet textual Sudoku dataset (Wang *et al.*, 2019). The cross-entropy loss serves as the baseline loss and is used in all four rows in Table 5.9 during training. Here, R is the number of recurrent steps and is 32 in the RRN model; $NumAtom$ is the number of Boolean atoms in Sudoku and is $81 \times 9 = 729$;

Table 5.9: Computation Size of Different Losses ($R = 32$, $NumAtom = 729$, $NumClause = 8991$)

Loss	Applied To	Computation Size	Time/Epoch
Cross Entropy	all recurrent steps	$O(R \times NumAtom)$	120s
CL-STE	first recurrent step	$O(1 \times NumAtom \times NumClause)$	211s
CL-STE	all recurrent steps	$O(R \times NumAtom \times NumClause)$	3796s
Cardinality (ours)	all recurrent steps	$O(R \times NumAtom)$	122s

and $NumClause$ is 8991 which is the number of clauses in the CNF for Sudoku. As we can see, the proposed cardinality constraint loss has the same computation size as the cross-entropy loss, thus almost doesn't affect the training time. On the other hand, the constraint loss in CL-STE computes much slower since the computation size is propositional to the number of clauses in a CNF, whose size is exponential to represent a cardinality constraint. In addition to the cross-entropy loss that is applied to the output from all recurrent steps, if we only apply the CL-STE loss to the output from the first recurrent step as done in the CL-STE paper, the training time per epoch is 211s. Note that batch size is not included in the computation size for simplicity. All experiments are using a batch size of 16 except for the third row (applying CL-STE loss to the outputs from all recurrent steps). If we apply the CL-STE loss to all recurrent steps, we have to decrease the batch size by 8 times to fit the GPU memory and the training time per epoch is increased to 3796s.

Chapter 6

LLM AND ASP

While large language models (LLMs), such as GPT-3, appear to be robust and general, their reasoning ability is not at a level to compete with the best models trained for specific natural language reasoning problems (Nye *et al.*, 2021; Valmeekam *et al.*, 2022). In this chapter, we observe that a large language model could serve as a highly effective few-shot semantic parser that turns natural language sentences into a logical form that can be used as input to answer set programs (Lifschitz, 2008a; Brewka *et al.*, 2011b). The combination leads to a robust and general system that works across multiple QA tasks without the need to retrain for new tasks. It requires only a few examples to direct an LLM to tune to an individual task, along with ASP knowledge modules that can be reused over multiple tasks. We demonstrate that this method achieves state-of-the-art performance on several NLP benchmarks, such as bAbI (Weston *et al.*, 2016), StepGame (Shi *et al.*, 2022), CLUTRR (Sinha *et al.*, 2019), and gSCAN (Ruis *et al.*, 2020), and also handles robot planning tasks that an LLM alone fails to solve.

6.1 Method: LLM+ASP

We call our framework `LLM+ASP` where `LLM` denotes a large pre-trained network such as GPT-3, which we use as a semantic parser to generate input to the ASP reasoner. More specifically, we assume data instances of the form $\langle S, q, a\rangle$, where S is a context story in natural language, q is a natural language query associated with S, and a is the answer. We use an LLM to convert a problem description (that is, context S and query q) into atomic facts, which are inputted into the ASP solver, together with background knowledge encoded as ASP rules. The output of the ASP solver is interpreted as the prediction to this

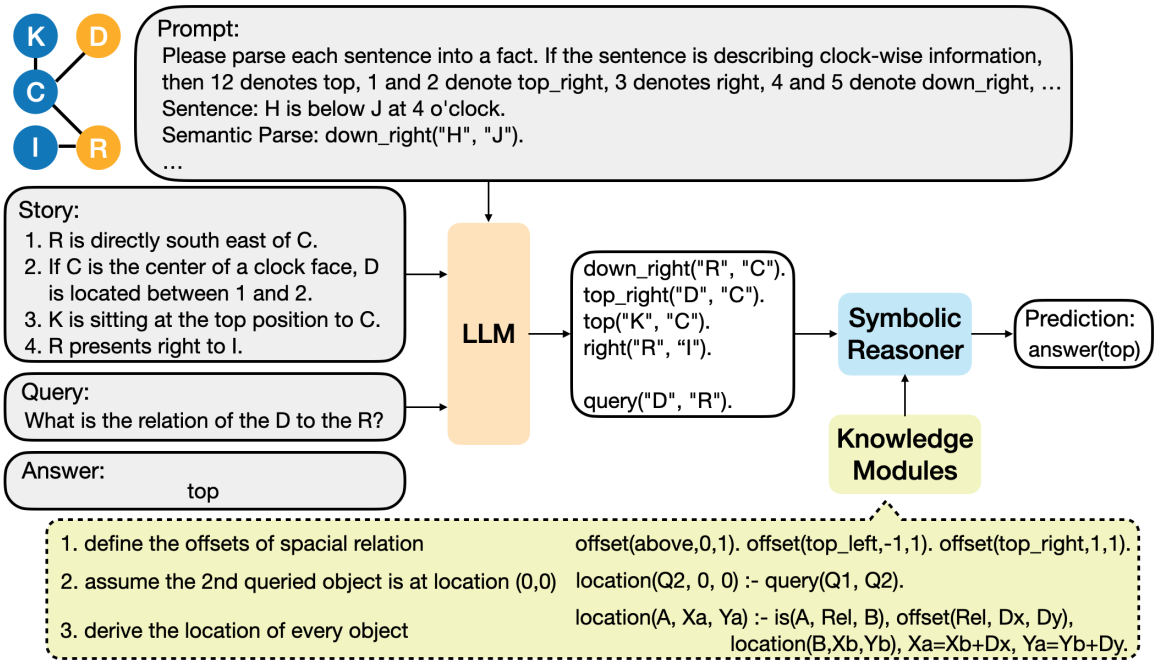

Figure 6.1: The LLM+ASP Pipeline for the StepGame Dataset.

data instance. Figure 6.1 illustrates the inference flow in the context of StepGame. The pipeline is simple but general enough to apply to various tasks without retraining; it only requires replacing few-shot prompts to the LLM and the ASP background knowledge with those appropriate for the new tasks.

Combining LLMs and ASP this way allows symbolic reasoning to be robust to varying and raw textual input; the ASP knowledge modules are not affected by the various forms of text input that express the same facts. Our method does not require training datasets. Instead, a few examples that turn natural language sentences into atomic facts are sufficient to build a semantic parser thanks to learned representations in LLMs. ASP knowledge modules can be reused for different tasks.

6.1.1 Prompts for Fact Extraction

We use an LLM (i.e., GPT-3) to extract atomic facts from the story and query. Most of the time, giving several examples yields accurate semantic parsing. The following is an

example prompt for bAbI.

```
Please parse the following statements into facts. The available keywords are:
   pickup, drop, and go.
Sentence: Max journeyed to the bathroom.
Semantic parse: go(Max, bathroom).

Sentence: Mary grabbed the football there.
Semantic parse: pickup(Mary, football).
...
```

We find that GPT-3 is highly tolerable to linguistic variability. For example, in StepGame, GPT-3 can turn various sentences below into the same atomic fact `top_right("C","D")`.

```
C is to the top right of D.
C is to the right and above D at an angle of about 45 degrees.
C is at a 45 degree angle to D, in the upper righthand corner.
C is directly north east of D.
C is below D at 2 o'clock.
```

In the experiments to follow, we find that the following strategy works well for fact extraction.

1. In general, we find that if the information in a story (or query) can be extracted independently, parsing each sentence separately (using the same prompt multiple times) typically works better than parsing the whole story.

2. There is certain commonsense knowledge that GPT-3 is not able to leverage from the examples in the prompt. In this case, detailing the missing knowledge in the prompt could work. For example, in StepGame, clock numbers are used to denote cardinal directions, but GPT-3 couldn't translate correctly even with a few examples

in the prompt. It works after enumerating all cases ("12 denotes top, 1 and 2 denote top_right, 3 denotes right, . . . ") in the prompt.

3. Semantic parsing tends to work better if we instruct GPT-3 to use a predicate name that better reflects the intended meaning of the sentence. For example, "A is there and B is at the 5 position of a clock face" is better to be turned into `down_right(B,A)` than `top_left(A,B)` although, logically speaking, the relations are symmetric.

The complete set of prompts for semantic parsing is given in Section 6.5.

6.1.2 Knowledge Modules

Instead of constructing a minimal world model for each task in Python code (Nye *et al.*, 2021), we use ASP knowledge modules. While some knowledge could be lengthy to be described in English, it could be concisely expressed in ASP. For example, the **location** module contains rules for spatial reasoning in a 2D grid space and is used for bAbI, StepGame, and gSCAN. Below is the main rule in the **location** module that computes the location `(Xa,Ya)` of object `A` from the location `(Xb,Yb)` of object `B` by adding the offsets `(Dx,Dy)` defined by the spatial relation `R` between `A` and `B`.

```
location(A, Xa, Ya) :- location(B, Xb, Yb), is(A, R, B), offset(R, Dx, Dy),
    Xa=Xb+Dx, Ya=Yb+Dy.
```

The **location** module also includes 9 predefined offsets, e.g., `offset(left,-1,0)`, that can be used to model multi-hop spatial relations of objects or effects of a robot's moving in a 2D space. For example, queries in StepGame are about the spatial relation `R` of object `A` to `B`. Using the **location** module, one can fix `B`'s location to be `(0,0)` and compute the spatial relation `R` based on the location of `A` as follows.

```
location(B, 0, 0) :- query(A, B).
answer(R) :- query(A, B), location(A, X, Y), offset(R, Dx, Dy),
```

```
Dx=-1: X<0;  Dx=0: X=0;  Dx=1: X>0;
Dy=-1: Y<0;  Dy=0: Y=0;  Dy=1: Y>0.
```

The second rule above contains six *conditional literals* among which `Dx=-1:X<0` says that "`Dx` must be -1 if `X<0`." For example, if `A`'s location `(X,Y)` is `(-3,0)`, then `(Dx,Dy)` is `(-1,0)` and the answer `R` is `left`. Similar rules can also be applied to bAbI task 17, which asks if `A` is `R` of `B`.

In the above rules, the relation `R` in, e.g., `is(A,R,B)`, is a variable and can be substituted by any binary relation. Such high-order representation turns out to be quite general and applicable to many tasks that query relation or its arguments.

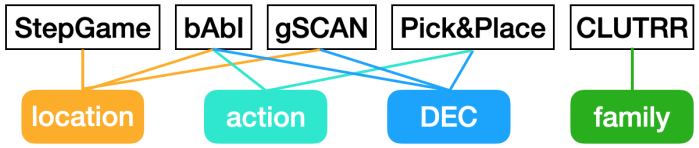

Figure 6.2: The Knowledge Modules at the Bottom Are Used in Each Task on the Top.

Figure 6.2 shows the knowledge modules used in this paper, where **DEC** denotes the Discrete Event Calculus axioms from (Mueller, 2006; Lee and Palla, 2012). In this section, we explained the main rules in the **location** module. The complete ASP knowledge modules are given in Section 6.6.

6.2 Evaluation

We apply the method in the previous section to four datasets and the Pick&Place domain. Recall that we do few-shot in-context learning, but do not use the training set included in these datasets. We use the same pipeline as in Figure 6.1 with different prompts and knowledge modules for each dataset.

Since this section evaluates 3 models of GPT-3, i.e., text-curie-001, text-davinci-002, and text-davinci-003, we use GPT-3(c1)+ASP, GPT-3(d2)+ASP, and GPT-3(d3)+ASP to

Table 6.1: Test Accuracy on 20 Tasks in bAbI Data

Task	**GPT-3(d3)**	**GPT-3(d3)**	**GPT-3(d3)**	**STM**(Le *et al.*, 2020)	**QRN**(Seo *et al.*, 2017)	
	Few-Shot	CoT	+ASP	(10k train)	(10k train)	(1k train)
1: Single supporting fact	98.4	97.3	100.0	100.0 ± 0.0	100.0	100.0
2: Two supporting facts	60.8	72.2	100.0	99.79 ± 0.23	100.0	99.3
3: Three supporting facts	39.6	54.1	100.0	97.87 ± 1.14	100.0	94.3
4: Two arg relations	60.4	72.7	100.0	100.0 ± 0.0	100.0	100.0
5: Three arg relations	88.2	89.1	99.8	99.43 ± 0.18	100.0	98.9
6: Yes/no questions	97.4	97.3	100.0	100.0 ± 0.0	100.0	99.1
7: Counting	90.6	88.6	100.0	99.19 ± 0.27	100.0	90.4
8: Lists/sets	96.2	97.1	100.0	99.88 ± 0.07	99.6	94.4
9 : Simple negation	98.4	98.2	100.0	100.0 ± 0.0	100.0	100.0
10: Indefinite knowledge	93.6	92.4	100.0	99.97 ± 0.06	100.0	100.0
11: Basic coreference	93.6	99.2	100.0	99.99 ± 0.03	100.0	100.0
12: Conjunction	88.6	88.8	100.0	99.96 ± 0.05	100.0	100.0
13: Compound coreference	98.4	97.3	100.0	99.99 ± 0.03	100.0	100.0
14: Time reasoning	78.0	91.5	100.0	99.84 ± 0.17	99.9	99.2
15: Basic deduction	57.0	95.0	100.0	100.0 ± 0.0	100.0	100.0
16: Basic induction	90.8	97.5	100.0	99.71 ± 0.15	100.0	47.0
17: Positional reasoning	66.0	70.8	100.0	98.82 ± 1.07	95.9	65.6
18: Size reasoning	89.8	97.1	100.0	99.73 ± 0.28	99.3	92.1
19: Path finding	21.0	28.7	100.0	97.94 ± 2.79	99.9	21.3
20: Agents motivations	100.0	100.0	100.0	100.0 ± 0.0	100.0	99.8
Average	80.34	86.18	**99.99**	99.85	99.70	90.1

denote our method `LLM+ASP` with a different model of GPT-3 respectively.

6.2.1 bAbI

The bAbI dataset (Weston *et al.*, 2016) is a collection of 20 QA tasks that have been widely applied to test various natural language reasoning problems, such as deduction, path-finding, spatial reasoning, and counting. State-of-the-art models, such as self-attentive associative-based two-memory model (STM) (Le *et al.*, 2020) and Query-Reduction networks (QRN) (Seo *et al.*, 2017) achieve close to 100% accuracy after training with 10k

instances while QRN's accuracy drops to 90% with 1k training instances.

We first designed two GPT-3 baselines, one with few shot prompts (containing a few example questions and answers) and the other with Chain-of-Thought (CoT) prompts (Wei *et al.*, 2022), which state the relevant information to derive the answer.

We also apply LLM+ASP. For example, we use GPT-3 to turn "the kitchen is south of the bathroom" into an atomic fact `is(kitchen, southOf, bathroom)` by giving a few examples of the same kind. Regarding knowledge modules, Tasks 1–3, 6–9, 10–14, and 19 are about events over time and use the **DEC** knowledge module. Tasks 4, 17, and 19 require various domain knowledge modules such as **location** and **action** knowledge modules. The remaining tasks do not require domain knowledge and rely only on simple rules to extract answers from parsed facts.

Table 6.1.2 compares our method with the two GPT-3 baselines, as well as two state-of-the-art methods on bAbI datasets, STM and QRN. Interestingly, the new GPT-3, text-davinci-003 (denoted GPT-3 (d3)), with basic few-shot prompting achieves 80.34% accuracy, while CoT improves it to 86.18%. GPT-3(d3)+ASP achieves state-of-the-art performance on bAbI with 99.99% average performance among all tasks, producing only two answers that disagree with the labels in the dataset. It turns out that the two questions are malformed since the answers are ambiguous, and our model's answers can be considered correct. The detailed examples are available in Section 6.4.1.

6.2.2 StepGame

Although bAbI has been extensively tested, it has several problems. Shi *et al.* (2022) note data leakage between the train and the test sets where named entities are fixed and only a small number of relations are used. Palm *et al.* (2018) point out that models do not need multi-hop reasoning to solve the bAbI dataset. To address the issues, Shi *et al.* (2022) propose the StepGame dataset. It is a contextual QA dataset in which the system is required

to interpret a story S about spatial relationships among several entities and answers a query q about the relative position of two of those entities, as illustrated in Figure 6.1. Unlike the bAbI dataset, StepGame uses a large number of named entities, and requires multi-hop reasoning up to as many as 10 reasoning steps.

In the basic form of the StepGame dataset, each story consists of k sentences that describe k spatial relationships between $k + 1$ entities in a chain-like shape. In this chapter, we evaluate the StepGame dataset with noise, where the original chain is extended with noise statements by branching out with new entities and relations.

Similarly to bAbI, we designed two GPT-3 baselines and applied our method to the StepGame data set.

Table 6.2: Test Accuracy on the StepGame Test Dataset, Where (c1), (d2), and (d3) Denote text-curie-001, text-davinci-002, and text-davinci-003 Models, Respectively

Method	k=1	k=2	k=3	k=4	k=5	k=6	k=7	k=8	k=9	k=10
RN	22.6	17.1	15.1	12.8	11.5	11.1	11.5	11.2	11.1	11.3
RRN	24.1	20.0	16.0	13.2	12.3	11.6	11.4	11.8	11.2	11.7
UT	45.1	28.4	17.4	14.1	13.5	12.7	12.1	11.4	11.4	11.7
STM	53.4	36.0	23.0	18.5	15.1	13.8	12.6	11.5	11.3	11.8
TPR-RNN	70.3	46.0	36.1	26.8	24.8	22.3	19.9	15.5	13.0	12.7
TP-MANN	85.8	60.3	50.2	37.5	31.3	28.5	26.5	23.7	22.5	21.5
SynSup	**98.6**	**95.0**	**92.0**	79.1	70.3	63.4	58.7	52.1	48.4	45.7
Few-Shot (d3)	55.0	37.0	25.0	30.0	32.0	29.0	21.0	22.0	34.0	31.0
CoT (d3)	61.0	45.0	30.0	35.0	35.0	27.0	22.0	24.0	23.0	25.0
GPT-3(c1)+ASP	44.7	38.8	40.5	58.8	62.4	57.4	56.2	58.0	56.5	54.1
GPT-3(d2)+ASP	92.6	89.9	89.1	**93.8**	**92.9**	**91.6**	**91.2**	**90.4**	**89.0**	**88.3**

For each $k \in \{1, \ldots, 10\}$, the StepGame dataset with noise consists of 30,000 training

samples, 1000 validation samples, and 10,000 test samples. To save the API cost for GPT-3, we only evaluated the two GPT-3 baselines on the first 100 test samples and evaluated our method on the first 1,000 test samples for each $k \in \{1, \ldots, 10\}$. Table 6.2 compares the accuracy of our method with the two baselines of GPT-3 and the current methods, i.e. RN (Santoro *et al.*, 2017), RRN (Palm *et al.*, 2018), UT (Dehghani *et al.*, 2019), STM (Le *et al.*, 2020), TPR-RNN (Schlag and Schmidhuber, 2018), TP-MANN (Shi *et al.*, 2022), and SynSup (with pre-training on the SPARTUN dataset) (Mirzaee and Kordjamshidi, 2022). Surprisingly, the GPT-3 baselines could achieve accuracy comparable to other models (except for SynSup) for large k values. CoT does not always help and decreases the accuracy with big ks. This may be because there is a higher chance of making a mistake in a long chain of thought. GPT-3(d2)+ASP outperforms all state-of-the-art methods and the GPT-3 baselines by a large margin for $k = 4, \ldots, 10$. Although SynSup achieves a higher accuracy for $k = 1, 2, 3$, this is misleading due to errors in the dataset. As we analyze below, about 10.7% labels in the data are wrong. The SynSup training makes the model learn to make the same mistakes over the test dataset, which is why its performance looks better than ours.

The modular design of LLM+ASP enables us to analyze the reasons behind its wrong predictions. We collected the first 100 data instances for each $k \in \{1, \ldots, 10\}$ and manually analyzed the predictions on them.

Among 1000 predictions of GPT-3(d2)+ASP, 108 of them disagree with the dataset labels, and we found that 107 of those have errors in the labels. For example, given the story and question "*J and Y are horizontal and J is to the right of Y. What is the relation of the agent Y with the agent J?*", the label in the dataset is "right" while the correct relation should be "left".[1] Recall that our method is interpretable, so we could easily identify the

[1]The remaining disagreeing case is due to text-davinci-002's mistake. For the sentence, "*if E is the center of a clock face, H is located between 2 and 3*." text-davinci-002 turns it into "right(H, E)" whereas text-

source of errors.

6.2.3 CLUTRR

CLUTRR (Sinha *et al.*, 2019) is a contextual QA dataset that requires inferring family relationships from a story. Sentences in CLUTRR are generated using 6k template narratives written by Amazon Mechanical Turk crowd-workers, and thus are more realistic and complex compared to those in bAbI and StepGame.

CLUTRR consists of two subtasks, *systematic generalization* that evaluates stories containing unseen combinations of logical rules (Minervini *et al.*, 2020; Bergen *et al.*, 2021) and *robust reasoning* that evaluates stories with noisy descriptions (Tian *et al.*, 2021). Since we use ASP for logical reasoning, which easily works for any combination of logical rules, we focus on the robust reasoning task.

Table 6.3: Test Accuracy on 4 Categories in CLUTRR 1.0 and CLUTRR 1.3 Datasets

Method	CLUTTR	clean	supp.	irre.	disc.
RN	1.0	49	68	50	45
MAC	1.0	63	65	56	40
Bi-att	1.0	58	67	51	57
GSM	1.0	**68.5**	48.6	62.9	52.8
GPT-3(d3)+ASP	1.0	**68.5**	**82.8**	**74.8**	**67.4**
GPT-3(d3)+ASP	1.3	97.0	84.0	92.0	90.0

Table 6.3 compares our method with RN (Santoro *et al.*, 2017), MAC (Hudson and Manning, 2018), BiLSTM-attention (Sinha *et al.*, 2019), and GSM (Tian *et al.*, 2021) on the original CLUTRR dataset, namely CLUTRR 1.0, in four categories of data instances:

davinci-003 turns it into "top-right(H, E)" correctly. To save API cost for GPT-3, we did not re-run the whole experiments with text-davinci-003.

clean, supporting, irrelevant, and disconnected. Except for our method, all other models are trained on the corresponding category of CLUTRR training data. Although our method achieves similar or higher accuracies in all categories, they are still much lower than we expected.

We found that such low accuracy is due to the errors in CLUTRR dataset, originating mostly from errors in the template narratives or the generated family graphs that violate common sense. The authors of CLUTRR recently published CLUTRR 1.3 codes to partially resolve this issue. [2] With the new code, we created a new dataset, namely CLUTRR 1.3, consisting of 400 data instances with 100 for each of the four categories. The last row in Table 6.3 shows that our method actually performs well on realistic sentences in CLUTRR. Indeed, with our method (text-davinci-003) on CLUTRR 1.3 dataset, 363 out of 400 predictions are correct, 16 are still wrong due to data mistakes (e.g., the label says "Maryann has an uncle Bruno" while the noise sentence added to the story is "Maryann told her son Bruno to give the dog a bath"), and 21 are wrong due to GPT-3's parsing mistakes (e.g., GPT-3 turned the sentence "Watt and Celestine asked their mother, if they could go play in the pool" into `mother("Watt", "Celestine")`. Since the sentences in CLUTRR 1.3 are more realistic than those of bAbI and StepGame, GPT-3 makes more mistakes even after reasonable efforts of prompt engineering. More details on data errors and GPT-3 errors are available in Section 6.4.2 and Section 6.3.

Table 6.4: Test Accuracy on CLUTRR-S Dataset

Method	clean	supp.	irre.	disc.
DeepProbLog	100	100	100	94
GPT-3(d2)+ASP	100	100	97	97
GPT-3(d3)+ASP	100	100	100	100

[2] https://github.com/facebookresearch/clutrr/tree/develop

We also evaluated our method on a simpler and cleaner variant of the CLUTRR data set, namely CLUTRR-S, that was used as a benchmark problem for a state-of-the-art neuro-symbolic approach DeepProbLog (Manhaeve *et al.*, 2021). Table 6.4 compares the accuracy of our method and DeepProbLog in all 4 categories of test data. GPT-3(d3)+ASP achieves 100% accuracy, outperforming DeepProbLog without the need for training.

Remark: Due to the modular structure, our method could serve as a data set validation tool to detect errors in a dataset. We detected 107 wrong data instances in the first 1000 data in StepGame and 16 wrong data instances in the 400 data in CLUTRR 1.3.

6.2.4 gSCAN

The gSCAN dataset (Ruis *et al.*, 2020) poses a task in which an agent must execute action sequences to achieve a goal (specified by a command in a natural language sentence) in a grid-based visual navigation environment. The dataset consists of two tasks, and we evaluate our method on the data splits from the compositional generalization task. There is one shared training set, one test set (split A) randomly sampled from the same distribution of the training set, and seven test sets (splits B to H) with only held-out data instances (i.e., not appearing in the training set) in different ways.

In the gSCAN dataset, each data instance is a tuple $\langle G, q, a\rangle$ where G is the grid configuration (in JSON format) describing the size of the gird, the location and direction of the agent, and the location and features of each object in the grid; q is a query (e.g., "pull a yellow small cylinder hesitantly"); and a is the answer in the form of a sequence of actions (e.g., "turn right, walk, stay, pull, stay, pull, stay"). For each data instance, we (i) use a Python script to extract atomic facts (e.g., `pos(agent,(2,3))`) from the grid configuration G; (ii) extract atomic facts from query q into atomic facts (e.g., `query(pull)`, `queryDesc(yellow)`, `while(hesitantly)`) using GPT-3; and (iii) predict the sequence of actions for this query using ASP.

Table 6.5: Test Accuracy on the gSCAN Dataset

Method	A	B	C	D
GECA	87.60	34.92	78.77	0.00
DualSys	74.7	81.3	78.1	0.01
Vilbert+CMA	99.95	99.90	99.25	0.00
GPT-3(c1)+ASP	98.30	**100**	**100**	**100**
GPT-3(d2)+ASP	**100**	**100**	**100**	**100**

Method	E	F	G	H
GECA	33.19	85.99	0.00	11.83
DualSys	53.6	76.2	0.0	21.8
Vilbert+CMA	99.02	99.98	0.00	22.16
GPT-3(c1)+ASP	**100**	**100**	**100**	**100**
GPT-3(d2)+ASP	**100**	**100**	**100**	**100**

Table 6.5 compares the accuracy of our method and the state-of-the-art methods, i.e., GECA (Ruis *et al.*, 2020), DualSys (Nye *et al.*, 2021) and Vilbert+CMA (Qiu *et al.*, 2021), on the gSCAN test dataset in eight splits. To save API cost for GPT-3, we only evaluated the first 1000 data instances of each split. With text-davinci-002, our method achieves 100% accuracy. With text-curie-001, our accuracy is slightly lower, making 17 errors in split A. The errors are of two kinds. The language model fails to extract adverbs in the correct format for 11 data instances (e.g., GPT-3 responded `queryDesc(while spinning)` instead of `while(spinning)`) and didn't ground the last word in a query for 6 data instances (e.g., for query `walk to a small square`, GPT-3 missed an atomic fact `queryDesc(square)`). Once the parsed results are correct, ASP does not make a mistake in producing plans.

6.2.5 *Robot Planning*

Recently, there has been increasing interest in using large language models to find a sequence of robot executable actions to achieve a high-level goal in natural languages, such as SayCan (Ahn *et al.*, 2022) and Innermonologue (Huang *et al.*, 2022). On the other hand, actions found by the LLM are "loosely" connected and do not consider the intermediate state changes while the actions are executed.

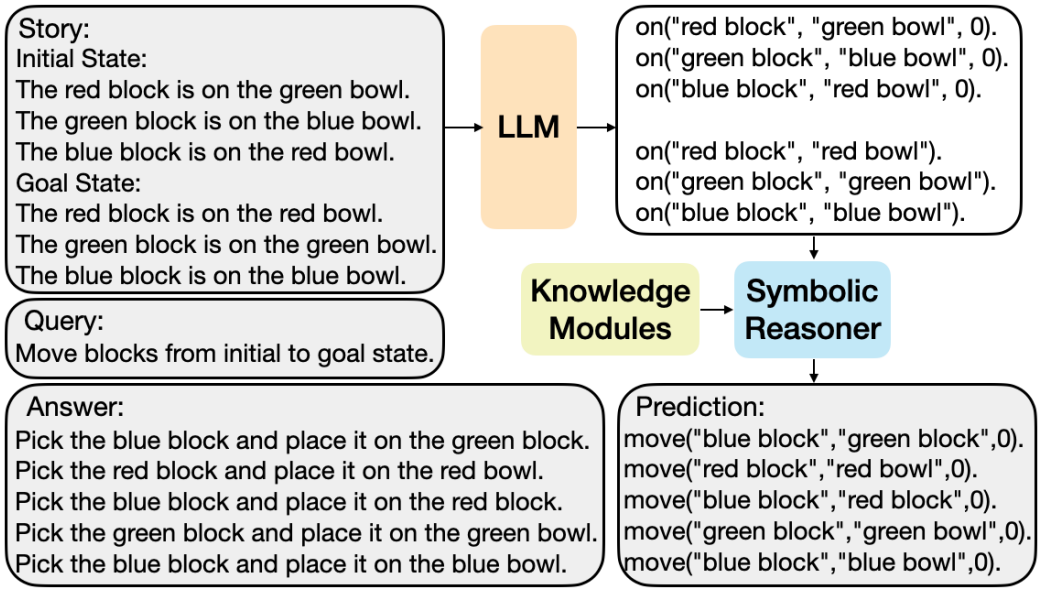

Figure 6.3: The LLM+ASP Pipeline for Pick&Place

We build the work on SayCan's open source virtual tabletop environment [3], where a robot is asked to achieve a goal, e.g., "stack the blocks" on a table with colored blocks and bowls. We observe that the successful plans demonstrated by SayCan are limited to simple one-step look-ahead plans that do not consider intermediate state changes.

We randomly sampled 40 data instances of the form $\langle S_i, S_g, L \rangle$ in the Pick&Place domain with 4 to 7 blocks and 3 to 7 bowls, possibly stacked together and with 3 to 10 steps of pick_and_place actions required by the robot to change the initial state S_i to the goal

[3]https://github.com/google-research/google-research/tree/master/saycan

state S_g. Here, the label L is the set of instructions to achieve the goals (e.g., "1. Move the violet block onto the blue block. 2..."). Among 40 data instances, 20 data instances contain only blocks that can be placed on the table while 20 data instances contain both blocks and bowls and assume all blocks must be on the bowls.

The baseline for this dataset follows the method in SayCan's open-source virtual table-top environment, where GPT-3 is used as the large language model to directly find the sequence of actions from S_i to S_g. However, the baseline fails to find successful plans for all 40 randomly sampled data instances. This result confirms the claim by (Valmeekam *et al.*, 2022) that large language models are not suitable as planners.

We also applied our method to this task. We let GPT-3 turn the states S_i and S_g into atomic facts of the form $on(A, B, 0)$ and $on(A, B)$, respectively. Then, an ASP program for the Pick&Place domain is used to find an optimal plan. We found that while GPT-3 has only 0% accuracy in predicting the whole plan, it has 100% accuracy in fact extraction under the provided format. When we apply symbolic reasoning to these extracted atomic facts with an ASP program, we could achieve 100% accuracy on the predicted plans. Figure 6.4 visualizes a simple plan predicted by LLM+ASP.

Table 6.6: Test Accuracy on the Pick&Place Dataset. (d3) Denotes the text-davinci-003 Model.

Method	Blocks	Blocks+Bowls
GPT-3(d3)	0	0
GPT-3(d3)+ASP	**100**	**100**

6.2.6 Findings

The following summarizes the findings of the experimental evaluation.

- Our experiments confirm that LLMs like GPT-3 are still not good at multi-step rea-

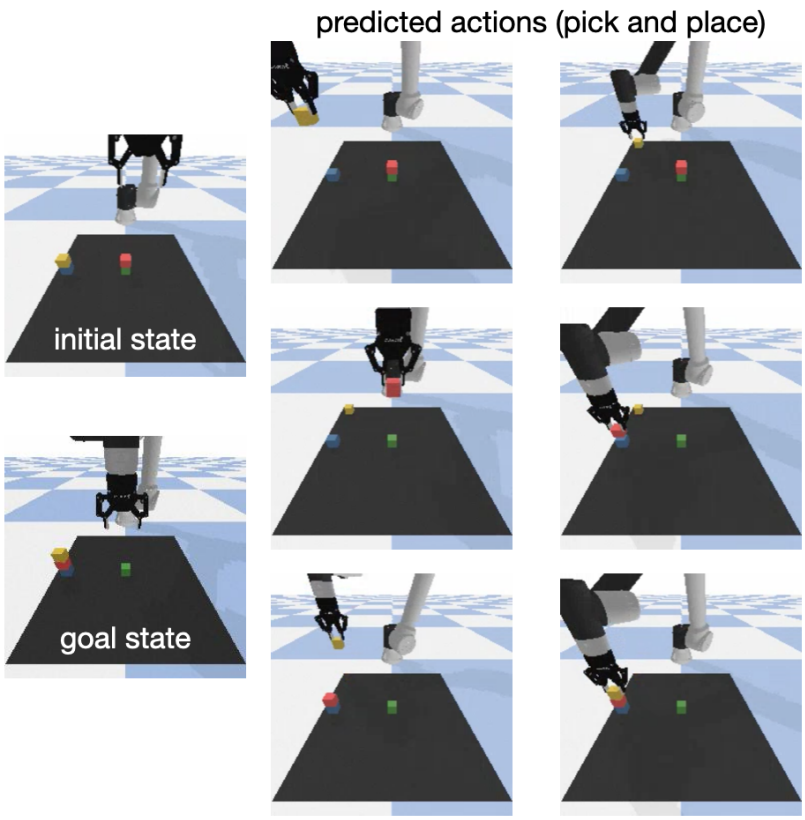

Figure 6.4: A Simple Plan Predicted by LLM+ASP in the Pick&Place Domain.

soning despite various prompts we tried. Chain-of-Thought is less likely to improve accuracy when a long chain of thought is required.

- On the other hand, LLMs are surprisingly good at turning a variety of expressions into a "canonical form" of information extraction. This in turn allows ASP knowledge modules to be isolated from linguistic variability in the input.

- Even for generating simple atomic facts, larger models tend to perform better. For example, in StepGame and gSCAN, text-curie-001 performs significantly worse compared to text-davinci-002 (Tables 6.2 and 6.5).

- The total amount of knowledge that needs to be encoded for all of the above datasets is not too large. This is in part due to the fact that GPT-3 "normalized" various forms

of input sentences for ASP to process and that knowledge modules could be reused across different datasets.

- The modular design of our approach makes it possible to locate the root cause of each failed prediction in the training data and improve upon it. There are three sources of errors: semantic parsing in LLMs, symbolic constraints, and the dataset itself, and we can resolve the first two issues by improving the prompts and updating the constraints, respectively.

- Our framework could serve as a few-shot dataset justifier and corrector. Among all predictions by our method that do not align with the labels, almost all of them (with only a few exceptions discussed in the paper) are due to errors in the dataset.

6.3 GPT-3 Errors in Semantic Parsing

In this section, we group and record the errors in the GPT-3 responses in tables where each row records a 3-tuple $\langle$ dataset, sentence(s), GPT-3 response $\rangle$. In this section, we list the following.

- all 21 errors for the CLUTRR 1.3 dataset with text-davinci-003;

- the single mistake in the first 100 data instances for every $k \in \{1, \ldots, 10\}$ in the StepGame dataset with text-davinci-002.

6.3.1 Argument Misorder

A common mistake in the GPT-3 response is that the relation and arguments for an atom are correctly extracted, but the order of the arguments is incorrect. Such mistakes can be greatly alleviated by proper few-shot prompting where the orders of arguments in the example target atoms follow their orders in the stories.

CLUTRR 1.3	[Erasmus] bought to dress for his father [Joeseph]	father(“Joeseph”, “Erasmus”)
CLUTRR 1.3	[Baldwin] bought to dress for his father [Sidney]	father(“Sidney”, “Baldwin”)
CLUTRR 1.3	[Elizabeth] is the uncle of [Leopold].	uncle(“Elizabeth”, “Leopold”)

There are only 3 errors in CLUTRR 1.3 due to argument misorder. The first 2 mistakes are indeed due to their missing periods at the end of the sentences – if we simply add the periods back, their GPT-3 responses would become correct.

6.3.2 Wrong Relation

Sometimes the arguments are correct, but the relations extracted by GPT-3 are incorrect or cannot be recognized by the ASP program.

CLUTRR 1.3	[Leila] and [Enoch] have been married for 35 years.	married(“Leila”, “Enoch”)
CLUTRR 1.3	[Lottie] and her grandsons [Clinton] and [Enoch] picked berries from the garden.	grandsons(“Lottie”, “Clinton”) grandsons(“Lottie”, “Enoch”)
CLUTRR 1.3	[Felix] loved going to the store with his mom [Christena].	mom(“Felix”, “Christena”)
CLUTRR 1.3	[Archibald] asked his father, [Abraham], to watch over his two sons, [Justus] and [Merle], for the weekend.	sons(“Archibald”, “Justus”, “Merle”)
StepGame	If E is the center of a clock face, H is located between 2 and 3.	right(“H”, “E”)

These kinds of mistake may be resolved by restricting the space of possible relations. For example, the mistakes in the first four rows can be resolved by simply adding the sentence “Use spouse(‘Person’, ‘Person’) if two persons are couples.” in the prompt.

6.3.3 Ambiguious or Incorrect Co-reference

CLUTRR 1.3	[Katherine] was eating lunch with his son [Daniel] and his wife [Adelle].	son(“Katherine”, “Daniel”) wife(“Daniel”, “Adelle”)
CLUTRR 1.3	[Sylvester] and his son [Rollie] went to his mother [Ottilia]’s home	son(“Sylvester”, “Rollie”) mother(“Rollie”, “Ottilia”)
CLUTRR 1.3	[Elsie] and [Lewis] did the Father daughter dance at the recital and his wife [Dollie] was so happy she cried	father(“Lewis”, “Dollie”) wife(“Lewis”, “Dollie”)

6.3.4 Anonymous Argument

CLUTRR 1.3	[Abe] is proud of his daughter’s son. His name is [Johnie].	son(“Johnie”, “Abe”)
CLUTRR 1.3	[Watt] and [Celestine] asked their mother, if they could go play in the pool.	mother(“Watt”, “Celestine”)
CLUTRR 1.3	[Davie] told his wife [Kitty] to pick up their daughter.	wife(“Davie”, “Kitty”) daughter(“Davie”, “Kitty”)
CLUTRR 1.3	[Ethyl] just got married to her now husband, and in the process she got herself a daughter-in-law named [Henry].	husband(“Ethyl”, “Michael”). daughter_in_law(“Ethyl”, “Henry”)
CLUTRR 1.3	[Barnett] and [Manie] asked their mother, if they could go play in the pool.	mother(“Barnett”, “Manie”)
CLUTRR 1.3	[Santiago] told his wife [Octavia] to pick up their daughter.	wife(“Santiago”, “Octavia”) daughter(“Santiago”, “Octavia”)

6.3.5 Missed to Generate Some Atoms

CLUTRR 1.3	[Elizabeth] bought to dress for his father [Leopold] [Orson] took his wife [Abbie] out for dinner.	father(“Leopold”, “Orson”)
CLUTRR 1.3	[Asa] felt lonely when his wife [Magdalena] was gone to see her mother [Josiephine].	wife(“Asa”, “Magdalena”)
CLUTRR 1.3	[Warner]’s father, [Johnny], and grandfather, [Bryant], went hiking during the first weekend of spring.	male(“Johnny”) male(“Bryant”)
CLUTRR 1.3	[Hollie] and [Rosanna], the happy couple, just got married last week.	–
CLUTRR 1.3	[Violet] took her brother [Travis] to the park, but left her sister [Serena] at home.	brother(“Violet”, “Travis”)

6.4 Dataset Errors

This section enumerates the errors in the datasets we found.

6.4.1 bAbI

In task 5, the dataset has two errors with regard to the labels.

Error #1. In the following example, the answer is ambiguous since Bill gives Mary both the football and the apple.

```
CONTEXT:
Mary journeyed to the kitchen.
Mary went to the bedroom.
Mary moved to the bathroom.
```

Mary grabbed the football there.

Mary moved to the garden.

Mary dropped the football.

Fred went back to the kitchen.

Jeff went back to the office.

Jeff went to the bathroom.

Bill took the apple there.

Mary picked up the milk there.

Mary picked up the football there.

Bill went back to the kitchen.

Bill went back to the hallway.

Fred journeyed to the office.

Bill discarded the apple.

Mary journeyed to the kitchen.

Fred journeyed to the garden.

Mary went to the hallway.

Mary gave the football to Bill.

Bill passed the football to Mary.

Bill took the apple there.

Bill gave the apple to Mary.

Jeff travelled to the kitchen.

QUERY:

What did Bill give to Mary?

PREDICTION:

apple

```
Answer:
football
```

Error #2. In the following example, the answer is ambiguous since Fred gives Bill both the milk and the apple.

```
CONTEXT:
Mary journeyed to the bathroom.
Mary moved to the hallway.
Mary went to the kitchen.
Bill went back to the bedroom.
Bill grabbed the apple there.
Fred went back to the garden.
Mary went to the garden.
Fred took the milk there.
Jeff moved to the hallway.
Bill dropped the apple there.
Fred handed the milk to Mary.
Mary handed the milk to Fred.
Fred went back to the bedroom.
Fred passed the milk to Bill.
Fred took the apple there.
Fred gave the apple to Bill.
Jeff went to the kitchen.
Bill dropped the milk.

QUERY:
What did Fred give to Bill?
```

```
PREDICTION:
apple

Answer:
milk
```

6.4.2 CLUTRR

We detected 16 data errors in the CLUTRR 1.3 dataset using our method. These errors can be grouped into the following 4 categories.

- 5 data instances are due to incorrect relation graphs. For example, one relation graph contains the main part "A-son-B-daughter-C-aunt-D" and a noise (supporting) relation "B-spouse-D". However, if B and D are couples, then C should have mother D instead of aunt D.
- 9 data instances have a correct relation graph (e.g., A-son-B-grandmother-C-brother-D with a noise supporting relation B-mother-A) but the noise relation is translated into a sentence with a wrong person name (e.g., "D has mother A" instead of "B has mother A").
- 1 data instance has a correct relation graph and story, but has a wrong label (i.e., the label should be mother_in_law instead of mother).
- 1 data instance has a correct relation graph and story, but the query cannot be answered due to the ambiguity of a sentence. It uses "A has grandsons B and C" to represent brother(B, C), while B and C may have different parents.

6.5 Prompts for Semantic Parsing

Below, we present the details of the general knowledge of the prompts that we summarized and applied in this work, followed by some examples. The full prompts for bAbI, StepGame, CLUTRR, gSCAN, and Pick&Place are available in our lab Github repository https://github.com/azreasoners/LLM-ASP.

1. If the information in a story (or query) can be extracted independently, parsing each sentence separately (using the same prompt multiple times) typically works better than parsing the whole story. Since people usually cache all GPT-3 responses to save cost by avoiding duplicated GPT-3 requests for the same prompt, parsing each sentence separately also yields better usage of cached responses. Below are some examples.

    - In most bAbI tasks (except for tasks 11 and 13), the sentences in a story (including the query sentence) are independent of each other. We parse each sentence separately using GPT-3.
    - In the stepGame dataset, each sentence in a story describes the spatial relation between 2 objects. There are 4 sentences in a story when $k = 1$ and about 20 sentences when $k = 10$. If we ask GPT-3 to extract all the atomic facts from the whole story, it always misses some atoms or predicts wrong atoms. Since every sentence is independent of each other as shown in Figure 6.1, we use the following (truncated) prompt multiple times for each data instance where each time `[INPUT]` is replaced with one sentence in the story or the query. This yields a much higher accuracy as in Section 6.2.3.

```
Please parse each sentence into a fact. If the sentence is describing
    clock-wise information, then 12 denotes top, 1 and 2 denote
```

```
    top_right, 3 denotes right, ... If the sentence is describing
    cardinal directions, then north denotes top, ...


Sentence: What is the relation of the agent X to the agent K?
Semantic Parse: query("X", "K").


Sentence: H is positioned in the front right corner of M.
Semantic Parse: top_right("H", "M").
...
Sentence: [INPUT]
Semantic Parse:
```

However, if some sentences in a story are dependent, splitting them may lead to unexpected results in the GPT-3 response. Below are some examples.

- In bAbI task #11 and #13, a story may contain the two consecutive sentences "Mary went back to the bathroom. After that she went to the bedroom." There is a dependency on the sentences to understand that "she" in the second sentence refers to "Mary" in the first. For this reason, task #11 stories are parsed as a whole. This is similar for task #13.
- In the CLUTRR dataset, a story may contain sentences with coreferences like "Shirley enjoys playing cards with her brother. His name is Henry." where the latter sentence depends on the former one, and a family relation can be correctly extracted only with both sentences. Thus for CLUTRR datasets (i.e., CLUTRR 1.0, CLUTRR 1.3, and CLUTRR-S), we extract the family relations and gender relations from the whole story.

2. There is certain commonsense knowledge that GPT-3 is not aware of, and describing

the missing knowledge in the prompt works better than adding examples only. This happens when GPT-3 cannot generalize such knowledge well with a few examples.

- For example, in StepGame dataset, clock numbers are used to denote cardinal directions, e.g., "H is below J at 4 o'clock" means "H is on the bottom-right of J". Such knowledge in the dataset is not well captured by GPT-3 and enumerating examples in the prompt doesn't work well. On the other hand, describing such knowledge at the beginning of the prompt increases the accuracy by a large margin.

6.6 ASP Knowledge Modules

The full ASP programs of all knowledge modules (i.e., Discrete Event Calculus Axioms Module, Action Module, Location Module, and Family Module) as well as all domain-specific ASP rules for each task are available in our lab Github repository https://github.com/azreasoners/LLM-ASP. Table 6.7 summarizes the knowledge modules used for each of the tasks.

Task	DEC Axioms	Action	Location	Family Relation
1: Single supporting fact	✓	✓		
2: Two supporting facts	✓	✓		
3: Three supporting facts	✓	✓		
4: Two arg relations			✓	
5: Three arg relations		✓		
6: Yes/no questions	✓	✓		
7: Counting	✓	✓		
8: Lists/sets	✓	✓		
9 : Simple negation	✓	✓		
10: Indefinite knowledge	✓	✓		
11: Basic coreference	✓	✓		
12: Conjunction	✓	✓		
13: Compound coreference	✓	✓		
14: Time reasoning	✓	✓		
15: Basic deduction				
16: Basic induction				
17: Positional reasoning			✓	
18: Size reasoning				
19: Path finding	✓	✓	✓	
20: Agents motivations				
StepGame			✓	
gSCAN	✓		✓	
CLUTRR				✓
Pick&Place	✓	✓		

Table 6.7: Knowledge Modules Used for Each of the Tasks. Note That DEC Axioms, Action, and Location Modules are Used in at Least Two Datasets.

Chapter 7

CONCLUSION

Despite the unprecedented success of deep neural networks, there is a growing awareness of their limitations: deep neural networks usually are data hungry; have difficulty in injecting knowledge or solving tasks that require complex reasoning; and lack transparency, explainability, and justification for predictions.

This dissertation presents three neuro-symbolic AI methods, namely NeurASP, CL-STE, and GPT3-ASP, to address the above issues by combining the strengths of deep neural networks and symbolic AI approaches. It also presents a Recurrent Transformer that extends Transformers with the power for multi-step reasoning through a recurrent architecture and a full attention.

This dissertation shows that neural network inference and training can benefit from explicit knowledge or constraints. For example, Chapter 3 shows that perception mistakes can be corrected by explicit constraints in ASP. Chapters 3, 4, and 5 show that a neural network can be trained better with less data or label with the gradients from a "semantic loss" expressed by either a formal probabilistic language or a regularization function. Chapter 6 shows that a large language model can serve as a highly effective few-shot semantic parser that turns natural language sentences into atomic facts – this information itself is not that important but can be utilized in symbolic computation to answer various questions.

This dissertation also shows that multi-step reasoning can be simulated in a neural network structure or simply appended as a symbolic computation layer to the output of neural networks. Chapter 5 presents a Recurrent Transformer that is capable of doing multi-step reasoning through recurrent calls to the same transformer blocks. Chapter 6 presents a dual-process neuro-symbolic reasoning system `LLM+ASP`, which achieves state-of-the-

art performance on several NLP benchmarks, including bAbI, StepGame, CLUTRR, and gSCAN, and also handles robot planning tasks that an LLM alone fails to solve.

Here, we summarize the major contributions of the dissertation and some directions for future work.

7.1 Summary of Contributions

We summarize the contributions of this dissertation as follows:

- **We designed and developed the framework NeurASP.** `NeurASP` is a neuro-symbolic framework that combines neural networks, probability, and ASP. We designed the syntax and semantics of `NeurASP`. We designed both the inference and learning algorithms, proved the correctness of the gradient computation (which is based on stable models and probabilities), and implemented a prototype system. We showed that, from ASP's perspective, `NeurASP` inherits the expressivity of ASP and extends its application domain to vector spaces and probabilistic reasoning. From neural networks' perspective, `NeurASP` provides a convenient way to represent structured knowledge used for inference and training. We designed experiments and showed that `NeurASP` can improve the neural network's perception result by applying reasoning over perceived objects and also can help neural network learn better by compensating the small size data with knowledge and constraints.

- **We designed and implemented a discrete semantic loss CL-STE.** `CL-STE` is a regularization method that systematically encodes logical constraints in propositional logic as a loss function in neural network learning. We proved the properties of this semantic loss as well as the properties of its gradients. We implemented CL-STE in PyTorch, conducted experiments, and demonstrated that minimizing this loss function via STE enforces the logical constraints in neural network learning so that neural networks learn from the explicit constraints. We also showed that leveraging

GPUs and batch training, `CL-STE` scales significantly better on a big range of existing benchmark problems compared to state-of-the-art neuro-symbolic methods that use heavy symbolic computation as a blackbox for computing gradients.

- **We designed and implemented Recurrent Transformer.** We designed and implemented `Recurrent Transformer`, a encoder-based neural network structure that extends Transformers with recurrence and full-attention. We applied this new model to challenging problems such as ungrounded visual Sudoku, showing that `Recurrent Transformer` is a viable approach to learning to solve CSPs, with clear advantages over state-of-the-art methods, such as RRN and SATNet. We also designed an efficient semantic loss for cardinality constraints and showed how to inject discrete logical constraints into `Recurrent Transformer` training to achieve sample-efficient learning and semi-supervised learning for CSPs.

- **We designed and developed a neuro-symbolic dual-system `LLM+ASP`.** We combined ASP with a large language model, i.e., GPT-3, that learned distributed representations. More specifically, we let GPT-3 serve as a general-purpose few-shot semantic parser that can convert linguistically variable natural language sentences into atomic facts and let ASP provide interpretable and explainable reasoning on the parsed results. We implemented this dual-system, evaluated the system to challenging question answering datasets, and achieved new state-of-the-art accuracy. We also showed that the knowledge modules are reusable in similar domains and adapting to a new domain does not require massive training.

- **We studied the relations among extensions of ASP.** In addition to the above contributions, we studied the relationships among some logic languages that are extensions of ASP. We proved reductions from higher-level languages to low-level ones, which helps us to have a better theoretical understanding of these languages and, in some

cases, yields more efficient solvers for the former using the solvers of the latter.

7.2 Future Directions

A few interesting directions for future work include:

- **Design and implement more efficient computation for NeurASP.** The current design of `NeurASP` has poor efficiency and the bottleneck lies in the probability computation for a data instance, which requires enumerating all stable models. There are many potential ways to resolve this issue. One way is to embed ASP (or possibly a fragment of ASP) directly in neural networks as a regularization function. Our work `CL-STE` is along this direction while the expressivity of ASP is not kept. It remains an open question to explore whether we can accelerate the computation in `NeurASP` by encoding ASP in `CL-STE`. Another way is to have an approximate inference by sampling a small amount of intended models instead of enumerating all of them. The obstacle is that the existing probabilistic sampling methods, e.g., MCASP (Lee and Wang, 2018) which uses a uniform sampler XORRO (Gebser *et al.*, 2016) for answer set programs, still take much time to sample an intended model. A more scalable sampling method needs to be designed. The third way is to compile an ASP program into a circuit, such as SDD (Kisa *et al.*, 2014) or d-DNNF (Darwiche, 2004), both of which are able to complete weighted model counting in polytime. The weighted model counting results are then used to compute probabilities and gradients for `NeurASP`.

- **Apply Neuro-Symbolic learning to larger-scale domains.** Although the `CL-STE` method has been shown to scale significantly better than existing neuro-symbolic methods that use heavy symbolic computation as a blackbox for computing gradients, the scale of the current experiments is still limited compared to the problems

targeted in deep learning community. We plan to apply `CL-STE` to larger-scale problems with domain knowledge available, starting from benchmark problems such as CIFAR-10 and CIFAR-100 datasets to which existing neuro-symbolic method (Roychowdhury *et al.*, 2021) has been applied using the prior knowledge about the relationships among classes, e.g., only bird and plane can fly. In addition to logic puzzles and vision problems, we will apply `CL-STE` to more kinds of tasks such as Word Algebra Problems as in `DeepProbLog` (Manhaeve *et al.*, 2018), which applies the knowledge about addition, subtraction, and product, and regression tasks such as temperature prediction where the constraints on the highest or lowest temperature as well as the change of temperature are summarized based on historical data and applied during training and inference.

- **Improve the Generalizability of Recurrent Transformer.** One big limitation of `Recurrent Transformer` is its generalizability. Since the window size in the current design of `Recurrent Transformer` is fixed and equal to the number of logical variables, one cannot transfer a trained model (e.g., on 9x9 Sudoku) to a new domain with different number of variables (e.g., 16x16 Sudoku). Dedicated research on more flexible window design (e.g., a sliding window on logical variables with a position embedding on variable indices) is necessary.

- **Rule generation using large language models.** Large language model (LLMs) keep rich information in vector space. In chapter 6, we showed that a large language model (LLM) could serve as a highly effective few-shot semantic parser that turns natural language sentences into atomic facts. It would be more interesting if we can generate rules using LLMs such as GPT-3. Similar to `LLM+ASP`, the prompts would play an essential role. LLM-based rule generation basically extracts a prompt-specified subset of the knowledge hidden in LLMs, and present the knowledge in an

expressive logic programming rule form.

REFERENCES

Adam, P., G. Sam, C. Soumith, C. Gregory, Y. Edward, D. Zachary, L. Zeming, D. Alban, A. Luca and L. Adam, "Automatic differentiation in PyTorch", in "Proceedings of Neural Information Processing Systems", (2017).

Ahn, M., A. Brohan, Y. Chebotar, C. Finn, K. Hausman, A. Herzog, D. Ho, J. Ibarz, A. Irpan, E. Jang, R. Julian *et al.*, "Do as I can, not as I say: Grounding language in robotic affordances", in "6th Annual Conference on Robot Learning", (2022).

Andreas, J., M. Rohrbach, T. Darrell and D. Klein, "Learning to compose neural networks for question answering", in "Proceedings of the 2016 Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies", pp. 1545–1554 (2016).

Bai, Y., D. Chen and C. P. Gomes, "CLR-DRNets: Curriculum learning with restarts to solve visual combinatorial games", in "27th International Conference on Principles and Practice of Constraint Programming (CP 2021)", (Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2021).

Battaglia, P. W., J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner *et al.*, "Relational inductive biases, deep learning, and graph networks", arXiv preprint arXiv:1806.01261 (2018).

Bergen, L., T. O'Donnell and D. Bahdanau, "Systematic generalization with edge transformers", Advances in Neural Information Processing Systems **34**, 1390–1402 (2021).

Besold, T. R., A. d. Garcez, S. Bader, H. Bowman, P. Domingos, P. Hitzler, K.-U. Kühnberger, L. C. Lamb, D. Lowd, P. M. V. Lima *et al.*, "Neural-symbolic learning and reasoning: A survey and interpretation", arXiv preprint arXiv:1711.03902 (2017).

Brewka, G., T. Eiter and M. Truszczyński, "Answer set programming at a glance", Communications of the ACM **54**, 12, 92–103 (2011a).

Brewka, G., I. Niemelä and M. Truszczynski, "Answer set programming at a glance", Communications of the ACM **54(12)**, 92–103 (2011b).

Brown, T., B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell *et al.*, "Language models are few-shot learners", Advances in neural information processing systems **33**, 1877–1901 (2020).

Buccafurri, F., N. Leone and P. Rullo, "Enhancing disjunctive datalog by constraints", IEEE Transactions on Knowledge and Data Engineering **12**, 5, 845–860 (2000).

Calimeri, F., W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, M. Maratea, F. Ricca and T. Schaub, "ASP-Core-2 input language format", Theory and Practice of Logic Programming **20**, 2, 294–309 (2020).

Calimeri, F., W. Faber, M. Gebser, G. Ianni, R. Kaminski, T. Krennwallner, N. Leone, F. Ricca and T. Schaub, "ASP-Core-2: Input language format", ASP Standardization Working Group, Tech. Rep (2012).

Chang, O., L. Flokas, H. Lipson and M. Spranger, "Assessing SATNet's ability to solve the symbol grounding problem", Advances in Neural Information Processing Systems **33**, 1428–1439 (2020).

Chen, Z., J. Mao, J. Wu, K.-Y. K. Wong, J. B. Tenenbaum and C. Gan, "Grounding physical concepts of objects and events through dynamic visual reasoning", in "International Conference on Learning Representations", (2020).

Cohen, W. W., F. Yang and K. R. Mazaitis, "Tensorlog: Deep learning meets probabilistic databases", Journal of Artificial Intelligence Research **1**, 1–15 (2018).

Courbariaux, M., Y. Bengio and J.-P. David, "Binaryconnect: training deep neural networks with binary weights during propagations", in "Proceedings of the 28th International Conference on Neural Information Processing Systems-Volume 2", pp. 3123–3131 (2015).

Creswell, A., M. Shanahan and I. Higgins, "Selection-inference: Exploiting large language models for interpretable logical reasoning", arXiv preprint arXiv:2205.09712 (2022).

Darwiche, A., "New advances in compiling CNF into decomposable negation normal form", in "ECAI", pp. 328–332 (2004).

Darwiche, A., "SDD: A new canonical representation of propositional knowledge bases", in "IJCAI Proceedings-International Joint Conference on Artificial Intelligence", p. 819 (2011).

De Raedt, L., R. Manhaeve, S. Dumancic, T. Demeester and A. Kimmig, "Neuro-symbolic = neural + logical + probabilistic", in "Proceedings of the 2019 International Workshop on Neural- Symbolic Learning and Reasoning", p. 4 (2019), URL https://sites.google.com/view/nesy2019/home.

Dehghani, M., S. Gouws, O. Vinyals, J. Uszkoreit and L. Kaiser, "Universal transformers", in "International Conference on Learning Representations", (2019), URL https://openreview.net/forum?id=HyzdRiR9Y7.

Ding, M., Z. Chen, T. Du, P. Luo, J. Tenenbaum and C. Gan, "Dynamic visual reasoning by learning differentiable physics models from video and language", Advances in Neural Information Processing Systems **34** (2021).

Donadello, I., L. Serafini and A. D. Garcez, "Logic tensor networks for semantic image interpretation", in "Proceedings of the 26th International Joint Conference on Artificial Intelligence", pp. 1596–1602 (AAAI Press, 2017).

Dosovitskiy, A., L. Beyer, A. Kolesnikov, D. Weissenborn, X. Zhai, T. Unterthiner, M. Dehghani, M. Minderer, G. Heigold, S. Gelly *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale", in "International Conference on Learning Representations", (2020).

Gabeur, V., C. Sun, K. Alahari and C. Schmid, "Multi-modal transformer for video retrieval", in "European Conference on Computer Vision", pp. 214–229 (Springer, 2020).

Garcez, A. d., M. Gori, L. C. Lamb, L. Serafini, M. Spranger and S. N. Tran, "Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning", arXiv preprint arXiv:1905.06088 (2019).

Gebser, M., B. Kaufmann, R. Kaminski, M. Ostrowski, T. Schaub and M. Schneider, "Potassco: The potsdam answer set solving collection", AI Communications **24**, 2, 107–124 (2011).

Gebser, M., T. Schaub, S. Marius and S. Thiele, "xorro: Near uniform sampling of answer sets by means of xor", Https://potassco.org/labs/2016/09/20/xorro.html (2016).

Gelfond, M. and V. Lifschitz, "The stable model semantics for logic programming", in "Proceedings of International Logic Programming Conference and Symposium", edited by R. Kowalski and K. Bowen, pp. 1070–1080 (MIT Press, 1988).

Goldman, O., V. Latcinnik, E. Nave, A. Globerson and J. Berant, "Weakly supervised semantic parsing with abstract examples", in "Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)", pp. 1809–1819 (2018).

Hao, J., X. Wang, B. Yang, L. Wang, J. Zhang and Z. Tu, "Modeling recurrence for transformer", in "Proceedings of NAACL-HLT", pp. 1198–1207 (2019).

Helwe, C., C. Clavel and F. M. Suchanek, "Reasoning with transformer-based models: Deep learning, but shallow reasoning", in "3rd Conference on Automated Knowledge Base Construction", (2021).

Huang, W., F. Xia, T. Xiao, H. Chan, J. Liang, P. Florence, A. Zeng, J. Tompson, I. Mordatch, Y. Chebotar, P. Sermanet, T. Jackson, N. Brown, L. Luu, S. Levine, K. Hausman and brian ichter, "Inner monologue: Embodied reasoning through planning with language models", in "6th Annual Conference on Robot Learning", (2022), URL https://openreview.net/forum?id=3R3Pz5i0tye.

Hudson, D. A. and C. D. Manning, "Compositional attention networks for machine reasoning", in "International Conference on Learning Representations", (2018).

Kahneman, D., *Thinking, fast and slow* (macmillan, 2011).

Katzouris, N. and A. Artikis, "Woled: a tool for online learning weighted answer set rules for temporal reasoning under uncertainty", in "Proceedings of the International Conference on Principles of Knowledge Representation and Reasoning", vol. 17, pp. 790–799 (2020).

Kazemi, S. M. and D. Poole, "Relnn: A deep neural model for relational learning", in "Proceedings of the 32nd AAAI Conference on Artificial Intelligence", (2018).

Kim, J., C. Park, H.-J. Jung and Y. Choe, "Plug-in, trainable gate for streamlining arbitrary neural networks", in "AAAI Conference on Artificial Intelligence (AAAI)", (2020).

Kipf, T. N. and M. Welling, "Semi-supervised classification with graph convolutional networks", in "Proceedings of the 5th International Conference on Learning Representations, ICLR 2017", (2017).

Kisa, D., G. Van den Broeck, A. Choi and A. Darwiche, "Probabilistic sentential decision diagrams", in "Proceedings of the 14th international conference on principles of knowledge representation and reasoning (KR)", pp. 1–10 (2014).

Lamb, L. C., A. Garcez, M. Gori, M. Prates, P. Avelar and M. Vardi, "Graph neural networks meet neural-symbolic computing: A survey and perspective", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 4877–4884 (2020).

Le, H., T. Tran and S. Venkatesh, "Self-attentive associative memory", in "International Conference on Machine Learning", pp. 5682–5691 (PMLR, 2020).

Lee, J. and R. Palla, "Reformulating the situation calculus and the event calculus in the general theory of stable models and in answer set programming", Journal of Artificial Inteligence Research (JAIR) **43**, 571–620 (2012).

Lee, J. and Y. Wang, "Weighted rules under the stable model semantics", in "Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)", pp. 145–154 (2016).

Lee, J. and Y. Wang, "Weight learning in a probabilistic extension of answer set programs", in "Proceedings of International Conference on Principles of Knowledge Representation and Reasoning (KR)", pp. 22–31 (2018).

Lee, J. and Z. Yang, "Statistical relational extension of answer set programming", Reasoning Web (2023).

Li, W., L. Yu, Y. Wu and L. C. Paulson, "Isarstep: a benchmark for high-level mathematical reasoning", in "International Conference on Learning Representations", (2020).

Lifschitz, V., "What is answer set programming?", in "Proceedings of the AAAI Conference on Artificial Intelligence", pp. 1594–1597 (MIT Press, 2008a).

Lifschitz, V., "What is answer set programming?", in "Proceedings of the 23rd National Conference on Artificial Intelligence - Volume 3", AAAI'08, p. 1594–1597 (AAAI Press, 2008b).

Lifschitz, V., D. Pearce and A. Valverde, "Strongly equivalent logic programs", ACM Transactions on Computational Logic **2**, 526–541 (2001).

Lin, B. Y., X. Chen, J. Chen and X. Ren, "Kagnet: Knowledge-aware graph networks for commonsense reasoning", in "Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)", pp. 2822–2832 (2019).

Manhaeve, R., S. Dumancic, A. Kimmig, T. Demeester and L. De Raedt, "Deepproblog: Neural probabilistic logic programming", in "Proceedings of Advances in Neural Information Processing Systems", pp. 3749–3759 (2018).

Manhaeve, R., S. Dumančić, A. Kimmig, T. Demeester and L. De Raedt, "Neural probabilistic logic programming in deepproblog", Artificial Intelligence **298**, 103504 (2021).

Mao, J., C. Gan, P. Kohli, J. B. Tenenbaum and J. Wu, "The neuro-symbolic concept learner: interpreting scenes, words, and sentences from natural supervision", in "Proceedings of International Conference on Learning Representations", (2019), URL https://openreview.net/forum?id=rJgMlhRctm.

Marcus, G., "Deep learning: A critical appraisal", arXiv preprint arXiv:1801.00631 (2018).

Minervini, P., S. Riedel, P. Stenetorp, E. Grefenstette and T. Rocktäschel, "Learning reasoning strategies in end-to-end differentiable proving", in "International Conference on Machine Learning", pp. 6938–6949 (PMLR, 2020).

Mirzaee, R. and P. Kordjamshidi, "Transfer learning with synthetic corpora for spatial role labeling and reasoning", in "Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing", p. 6148–6165 (Association for Computational Linguistics, 2022), URL https://aclanthology.org/2022.emnlp-main.413.

Mueller, E., *Commonsense reasoning* (Elsevier, 2006).

Nye, M., M. Tessler, J. Tenenbaum and B. M. Lake, "Improving coherence and consistency in neural sequence models with dual-system, neuro-symbolic reasoning", Advances in Neural Information Processing Systems **34**, 25192–25204 (2021).

Palm, R., U. Paquet and O. Winther, "Recurrent relational networks", in "Proceedings of Advances in Neural Information Processing Systems", pp. 3368–3378 (2018).

Park, K., "Can convolutional neural networks crack sudoku puzzles?", https://github.com/Kyubyong/sudoku (2018).

Pogancic, M. V., A. Paulus, V. Musil, G. Martius and M. Rolinek, "Differentiation of blackbox combinatorial solvers.", in "ICLR", (2020).

Qiu, L., H. Hu, B. Zhang, P. Shaw and F. Sha, "Systematic generalization on gscan: What is nearly solved and what is next?", in "Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing", pp. 2180–2188 (2021).

Rastegari, M., V. Ordonez, J. Redmon and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks", in "European conference on computer vision", pp. 525–542 (Springer, 2016).

Richardson, M. and P. Domingos, "Markov logic networks", Machine Learning **62**, 1-2, 107–136 (2006).

Rocktäschel, T. and S. Riedel, "End-to-end differentiable proving", in "Proceedings of Advances in Neural Information Processing Systems", pp. 3788–3800 (2017).

Roychowdhury, S., M. Diligenti and M. Gori, "Regularizing deep networks with prior knowledge: A constraint-based approach", Knowledge-Based Systems **222**, 106989 (2021).

Ruis, L., J. Andreas, M. Baroni, D. Bouchacourt and B. M. Lake, "A benchmark for systematic generalization in grounded language understanding", Advances in neural information processing systems **33**, 19861–19872 (2020).

Sampat, S. and J. Lee, "A model-based approach to visual reasoning on cnlvr dataset", in "Sixteenth International Conference on Principles of Knowledge Representation and Reasoning", (2018).

Santoro, A., D. Raposo, D. G. Barrett, M. Malinowski, R. Pascanu, P. Battaglia and T. Lillicrap, "A simple neural network module for relational reasoning", in "Advances in neural information processing systems", pp. 4967–4976 (2017).

Sarker, M. K., L. Zhou, A. Eberhart and P. Hitzler, "Neuro-symbolic artificial intelligence", AI Communications pp. 1–13 (2021).

Schlag, I. and J. Schmidhuber, "Learning to reason with third order tensor products", Advances in neural information processing systems **31** (2018).

Selsam, D., M. Lamm, B. Bünz, P. Liang, L. de Moura and D. L. Dill, "Learning a SAT solver from single-bit supervision", in "International Conference on Learning Representations (ICLR 2019)", (2019).

Seo, M. J., S. Min, A. Farhadi and H. Hajishirzi, "Query-reduction networks for question answering", in "5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings", (2017), URL https://openreview.net/forum?id=B1MRcPclx.

Serafini, L. and A. d. Garcez, "Logic tensor networks: Deep learning and logical reasoning from data and knowledge", arXiv preprint arXiv:1606.04422 (2016).

Shi, Z., Q. Zhang and A. Lipani, "Stepgame: A new benchmark for robust multi-hop spatial reasoning in texts", Association for the Advancement of Artificial Intelligence (2022).

Simons, T. and D.-J. Lee, "A review of binarized neural networks", Electronics **8**, 6, 661 (2019).

Sinha, K., S. Sodhani, J. Dong, J. Pineau and W. L. Hamilton, "Clutrr: A diagnostic benchmark for inductive reasoning from text", in "Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)", pp. 4506–4515 (2019).

Šourek, G., V. Aschenbrenner, F. Železny and O. Kuželka, "Lifted relational neural networks", in "Proceedings of the 2015th International Conference on Cognitive Computation: Integrating Neural and Symbolic Approaches-Volume 1583", pp. 52–60 (CEUR-WS. org, 2015).

Tang, W., G. Hua and L. Wang, "How to train a compact binary neural network with high accuracy?", in "Thirty-First AAAI conference on artificial intelligence", (2017).

Tian, J., Y. Li, W. Chen, H. Hao and Y. Jin, "A generative-symbolic model for logical reasoning in nlu", in "Is Neuro-Symbolic SOTA still a myth for Natural Language Inference? The first workshop", (2021).

Topan, S., D. Rolnick and X. Si, "Techniques for symbol grounding with SATNet", Advances in Neural Information Processing Systems **34** (2021).

Tsamoura, E., T. Hospedales and L. Michael, "Neural-symbolic integration: A compositional perspective", in "Proceedings of the AAAI Conference on Artificial Intelligence", pp. 5051–5060 (2021).

Valmeekam, K., A. Olmo, S. Sreedharan and S. Kambhampati, "Large language models still can't plan (a benchmark for LLMs on planning and reasoning about change)", in "NeurIPS 2022 Foundation Models for Decision Making Workshop", (2022), URL https://openreview.net/forum?id=wUU-7XTL5XO.

Vaswani, A., N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser and I. Polosukhin, "Attention is all you need", arXiv preprint arXiv:1706.03762 (2017).

Verreet, V., V. Derkinderen, P. Z. Dos Martires and L. De Raedt, "Inference and learning with model uncertainty in probabilistic logic programs", in "Proceedings of the AAAI Conference on Artificial Intelligence", vol. 36, pp. 10060–10069 (2022).

Wang, P.-W., P. L. Donti, B. Wilder and Z. Kolter, "SATNet: Bridging deep learning and logical reasoning using a differentiable satisfiability solver", in "Proceedings of the 35th International Conference on Machine Learning (ICML)", (2019).

Wei, J., X. Wang, D. Schuurmans, M. Bosma, brian ichter, F. Xia, E. H. Chi, Q. V. Le and D. Zhou, "Chain of thought prompting elicits reasoning in large language models", in "Advances in Neural Information Processing Systems", edited by A. H. Oh, A. Agarwal, D. Belgrave and K. Cho (2022), URL https://openreview.net/forum?id=_VjQlMeSB_J.

Weston, J., A. Bordes, S. Chopra and T. Mikolov, "Towards ai-complete question answering: A set of prerequisite toy tasks", in "4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings", edited by Y. Bengio and Y. LeCun (2016), URL http://arxiv.org/abs/1502.05698.

Winters, T., G. Marra, R. Manhaeve and L. De Raedt, "Deepstochlog: Neural stochastic logic programming", arXiv preprint arXiv:2106.12574 (2021).

Xu, J., Z. Zhang, T. Friedman, Y. Liang and G. Van den Broeck, "A semantic loss function for deep learning with symbolic knowledge", in "Proceedings of the 35th International Conference on Machine Learning (ICML)", (2018), URL http://starai.cs.ucla.edu/papers/XuICML18.pdf.

Yang, Z., A. Ishay and J. Lee, "NeurASP: Embracing neural networks into answer set programming", in "Proceedings of International Joint Conference on Artificial Intelligence (IJCAI)", pp. 1755–1762 (2020).

Yang, Z., A. Ishay and J. Lee, "Learning to solve constraint satisfaction problems with recurrent transformer", in "The Eleventh International Conference on Learning Representations", (2023), URL https://openreview.net/forum?id=udNhDCr2KQe.

Yang, Z., J. Lee and C. Park, "Injecting logical constraints into neural networks via straight-through estimators", in "International Conference on Machine Learning", pp. 25096–25122 (PMLR, 2022).

Yi, K., C. Gan, Y. Li, P. Kohli, J. Wu, A. Torralba and J. B. Tenenbaum, "CLEVRER: Collision events for video representation and reasoning", in "ICLR", (2019).

Yin, P., J. Lyu, S. Zhang, S. Osher, Y. Qi and J. Xin, "Understanding straight-through estimator in training activation quantized neural nets", in "International Conference on Learning Representations", (2019).

Zhang, J., Y. Zhao, M. Saleh and P. Liu, "Pegasus: Pre-training with extracted gap-sentences for abstractive summarization", in "International Conference on Machine Learning", pp. 11328–11339 (PMLR, 2020).

Zhang, Y., X. Chen, Y. Yang, A. Ramamurthy, B. Li, Y. Qi and L. Song, "Efficient probabilistic logic reasoning with graph neural networks", in "International Conference on Learning Representations", (2019).

Zhou, D., N. Schärli, L. Hou, J. Wei, N. Scales, X. Wang, D. Schuurmans, O. Bousquet, Q. Le and E. Chi, "Least-to-most prompting enables complex reasoning in large language models", arXiv preprint arXiv:2205.10625 (2022).