Politecnico di Torino

Master's Degree in Data Science and Engineering

A.a. 2024/2025

Graduation Session July, 2025

Neuro-Symbolic Artificial for Visual Reasoning via Dynamic Logic Tensor Networks

Supervisor:

Lia Morra

Candidate:

Homayoun Afshari

Summary

Recent advancements in Large Language Models (LLMs) have demonstrated impressive abilities in contextual understanding and chain-of-thought reasoning. Yet, while neural networks excel at inductive pattern recognition, they struggle with deductive rule chaining and abductive hypothesis generation due to their lack of explicit symbolic structure. As a promising solution, Neuro-Symbolic (NeSy) AI integrates this neural perception with symbolic logic. By incorporating real logic—a form of differentiable fuzzy logic—NeSy systems gain the ability to reason over soft truth values, enabling gradient-based learning of interpretable rules and resilience to noisy or ambiguous inputs. This thesis focuses on visual reasoning through NeSy AI, a domain where perception and logic must jointly operate to interpret partially observable, visually encoded structures. Despite progress, current NeSy approaches face limitations: they are often rigidly tailored to specific tasks, generate unexplainable rules, or produce informal rules that are hard to verify automatically. To address these challenges, this work proposes a novel NeSy visual reasoning framework emphasizing flexibility, explainability, and formality. The architecture takes textual and visual descriptions of a reasoning task and processes them using a context generator. The resulting content is fed to a rule generator to create symbolic rules, while, in parallel, a visual processor translates the visual input into symbolic visual atoms. Given the generated symbolic rules and visual symbols, a rule verifier applies the rules to the symbols and checks their coherence with visual facts. The results are then used to dynamically refine both context and rule generation, enabling the system to iteratively improve in a manner similar to reinforcement learning. The proposed system is evaluated on the ViSudo-PC benchmark, a symbolic visual reasoning task that distinguishes correct from corrupted Sudoku boards using diverse visual domains (MNIST, EMNIST, KMNIST, FMNIST). Results demonstrate that the system successfully induces symbolic rules from multi-modal inputs, verifies them automatically, and progressively improves its outputs through feedback. Ultimately, this thesis shows that the proposed intermediate feedback mechanism significantly enhances the alignment between visual perception and symbolic reasoning, providing a viable path forward for flexible, explainable, and formal neuro-symbolic AI.

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Lia Morra, for her continuous guidance, support, and encouragement throughout this research. I am also thankful to the members of her team, Alessandro Russo and Amir Goudarzi, for their valuable feedback and insights. Special thanks to my colleagues and friends for the stimulating discussions and moral support during the course of this work. Finally, I am deeply grateful to my family and my beloved wife, Fatemeh, for their unwavering encouragement and patience.

Table of Contents

List of Figures

Chapter 1

Introduction

1.1 What is Reasoning?

Recently, there has been a critical shift in Artificial Intelligence (AI) research: the pursuit of models that not only excel in **Contextual Understanding**, as done by Large Language Models (LLMs), but also demonstrate powerful reasoning capabilities, as if a human brain is pulling the strings behind the scene [1]. Not from a long ago, we have observed how OpenAI's ChatGPT or Anthropic's Claude have been equipped with the ability to generate **Chains of Thought** in order to resolve analytical problems, along with their remarkable abilities in pattern recognition, language generation, and zero-shot generalization [2]. These behaviors have sparked numerous discussions in the AI community about how these **Trainable Machines** are able to perform such tasks [3]. Are they *truly* capable of reasoning, or are they merely *simulating* reasoning behaviors? This question, however, is a serious dilemma that may never be answered, but it poses a fundamental question; **What Is Reasoning**?

As defined by cognitive science, reasoning is the process of *drawing conclusions* from available information using three primary modes: **Induction**, **Deduction**, and **Abduction** [4]. Induction involves *generalizing* rules from specific observations [5]. For instance, noticing that the sun rises every morning, one can infer that it will do so again tomorrow. Deduction, on the other hand, uses established *rules* to derive specific *conclusions* [6]. For example, if all humans are mortal and Socrates is human, then one can deduce that Socrates must also be mortal. Abduction, however, is more subtle and complex. It involves inferring the *most likely explanation* from incomplete evidence, as if a detective is hypothesizing the cause of a mysterious event [7]. For example, observing a wet floor under a cloudy sky, one can believe that it must have been raining lately. Understanding these distinct forms of reasoning is essential if we want to assess how far AI models have come and how far they still have to go in achieving or mimicking true reasoning.

1.2 How can AI Reason?

At the heart of AI, **Neural Networks**, including LLMs, have demonstrated strong capabilities in induction, primarily due to their *training on large datasets* and their proficiency in pattern extraction [8]. In other words, considering that inductive reasoning involves generalizing from specific examples to broader patterns, neural networks excel in this regard because they are optimized to minimize *prediction error* over millions of instances, which stems from their ability to implicitly encode associations within high-dimensional vector spaces and capturing complex correlations *without* the need for explicit knowledge representations [5]. This property allows these models excel at generalizing from specific instances, capturing statistical regularities, and producing coherent outputs across many tasks such as language modeling, image captioning, and even basic question-answering [9].

Nevertheless, when it comes to deduction, the reasoning performance of neural networks is often *contingent* upon the presence of implicitly encoded logic within the data distribution [10]. In other words, since these AI models rely heavily on statistical associations rather than formal rule-following, they tend to lose strength when reasoning requires precise logical consistency or the chaining of abstract rules. More critically, abduction also remains a persistent reasoning challenge. Effective abductive reasoning in AI requires *integrating* world knowledge, contextual awareness, and causal inference to construct plausible but unobserved hypotheses [7]. Neural networks, *by design*, lack explicit mechanisms to support this kind of inferential leap, which makes them unreliable for tasks that require *systematical exploration* of reasoning paths beyond what the data directly supports [5].

1.3 A Solution: NeSy AI

Early paradigms in AI, known as Good Old-Fashioned AI (**GOFAI**), focused on *symbolic reasoning*, where knowledge was represented explicitly in logic-based systems and manipulated through rule-based inference [11]. Unlike neural networks, GOFAI systems were designed to perform all three major forms of reasoning by leveraging structured representations, ontologies, and well-defined search procedures [10]. However, while these symbolic systems offered explainability and formal guarantees, but often *lacked* scalability and robustness in dealing with noisy or unstructured data [5]. With the rise of neural networks, however, modern AI systems began to prioritize statistical learning over symbolic manipulation, achieving impressive results in perception and pattern recognition but *sacrificing* core reasoning capabilities in the process. This shift has created a discontinuity: we now expect neural networks to handle reasoning tasks that were once *explicitly* addressed by symbolic AI, but *without* equipping them with the tools to do so.

As a consequence, the so-called issues have *prompted* AI researchers to explore hybrid approaches that combine the inductive strength of neural networks with deductive and abductive potentials of GOFAI, aiming to develop systems that can reason more robustly across all three reasoning modes [5]. One such promising direction is Neuro-Symbolic (**NeSy**) AI, which seeks to integrate the sub-symbolic power of neural networks with the explicit and structured reasoning capabilities of symbolic systems [8]. This hybridization aims to leverage the scalability and flexibility of neural models while incorporating the transparency, explainability, and logical rigor of symbolic representations.

This architectural duality originates from two-system theory of reasoning in cognitive science: **System 1** and **System 2** [3]. The first refers to fast, automatic, and cognitive reasoning processes that are typically unconscious, effortless, and operate in parallel, which is similar to the pattern-based memory-driven nature of neural networks optimized for inductive reasoning [1]. On the other hand, system 2 represents slow, deliberate, and analytical thought through explicitly manipulating structured representations like rules, logic, and symbols, which resembles deduction and abduction in symbolic reasoning [3]. As in neural networks, system 1 enables rapid recognition and generalization, but it often *lacks* precision [12]. System 2, by contrast, enables methodical and consistent reasoning, *albeit* at a higher computational cost [5]. A successful NeSy system, therefore, aims to simulate the complementary functions of both systems, enabling NeSy models to handle a broad range of reasoning tasks with greater versatility and depth.

1.4 NeSy AI Empowered by Fuzzy Logic

In the context of NeSy AI, **Symbols** refer to abstract representations within a specific domain that carry explicit meanings, such as constants, variables, numbers, or fixed semantic labels [5]. These symbols can be combined and manipulated using formal languages like First-Order Logic (FOL), allowing for the representation of structured knowledge and reasoning over it [5]. For instance, a rule such as $\forall x : \text{isBrid}(x) \rightarrow \text{canFly}(x)$ encodes a general statement about birds' ability to fly, where each component, including quantifiers, predicates, and logical operators, is a symbolic element. This structured framework enables symbolic components of NeSy systems to draw logical conclusions and generate plausible hypotheses from incomplete observations, i.e., deduction and abduction through symbolic reasoning, thereby overcoming some of the fundamental reasoning limitations faced by neural networks [3].

Considering this importance of logical formalisms in NeSy AI, recent research has explored moving toward **Real Logic**, which is a differentiable form of fuzzy logic [5]. Unlike binary logic systems that treat statements as either entirely true or false, fuzzy logic introduces degrees of truth, allowing the system to represent and reason with uncertainty and vagueness that often characterize real-world data. Real Logic enables soft constraints to be imposed over continuous values and supports gradient-based optimization, which makes it compatible with neural networks during training [5]. As a result, symbolic rules are no longer fixed or hand-crafted, but can be learned or fine-tuned using stochastic gradient descent based on observed data [13]. This ability to merge differentiability with symbolic reasoning significantly enhances the adaptability of NeSy models in ambiguous or noisy environments, where classical logic systems typically struggle to operate effectively [8]. For instance, the rule $\text{isBird}(x) \rightarrow \text{canFly}(x)$ can be evaluated to a degree, rather than as strictly true or false, allowing for exceptions (e.g., penguins) to be incorporated naturally.

1.5 Visual Reasoning through NeSy AI

The soft interpretation of logic allows NeSy systems to achieve more robust generalization and contextual reasoning across a variety of domains, which sets the stage for applications in dynamic and perceptually complex settings like **Visual Reasoning** [14]. In such settings, NeSy systems must integrate perception from partial information with reasoning about the relationships between different visual objects, which often involves partial observations, occlusions, or ambiguous scenes [12]; the conditions under which, pure neural networks may fail to generalize and purely symbolic systems may lack flexibility [3]. Accordingly, in this thesis, we will also focus on the class of NeSy systems that are built on top of fuzzy logic for performing visual reasoning tasks.

1.6 Issues of NeSy Visual Reasoning

Regardless of all the opportunities NeSy AI provides for promising visual reasoning, several limitations still persist across different methodological paradigms. These limitations highlight ongoing trade-offs between flexibility, explainability, and formality in current approaches [12, 15]. We will discuss these approaches in detail in Section **??**; here, we only focus on high-level descriptions of their shortcomings. In this respect, we can categorize the issues into three groups: Flexible But Non-Explainable (**FBN**), Explainable But Rigid (**ENR**), and Flexible But Verbose (**FBV**) approaches.

FBN approaches emphasize learning symbolic reasoning patterns directly from visual data without hard-coded rules, often in flexible and open-ended environments. These methods embed reasoning mechanisms within neural network weights or architectures [3]. While this enables scalability and end-to-end training, it often results in latent rules that are non-explainable [15]. In other words, the visual reasoning processes become entangled with the model's internal representations, making it difficult to extract, verify, or validate the learned logic. As a result, despite solving the task effectively, these models sacrifice transparency and explainability.

In response to this, ENR methods address the explainability gap by explicitly embedding symbolic rules into the system prior to training. Therefore, by integrating known symbolic rules as inductive biases or constraints, these approaches enforce explainability and logical consistency during both training and inference [9]. However, such methods often suffer from limited generalizability. Their performance and applicability hinge on domain-specific rule sets that do not easily transfer to new tasks, unseen visual domains, or more open-ended reasoning problems.

FBV methods, on the other hand, aim to overcome both limitations by generating rules dynamically at runtime, often in the form of free-form text or structured outputs [16]. For example, LLMs with proper prompting can produce reasoning traces that explain their decisions [16]. While this introduces flexibility and an element of explainability, the resulting explanations are typically verbose, informal, or non-machine-readable [15]. This restricts their utility in downstream symbolic processing and makes verification challenging. Additionally, such free-form reasoning is highly dependent on prompt engineering and lacks the formal structure needed for robust generalization.

Taken together, these limitations suggest that current NeSy visual reasoning systems have not yet reached the peak of their promised potential. Therefore, fully addressing these problems remains an open challenge, and future work will need to investigate architectures that can learn modular, interpretable, and grounded rules that generalize across tasks without requiring manual encoding or opaque abstraction. That is what we also seek to explore in this thesis.

1.7 Novelties of the Thesis

Considering the challenges highlighted in previous sections, this thesis proposes a novel NeSy visual reasoning framework grounded in the flexibility, explainability, and formality principles. As shown in Figure 1.1, the system processes two modalities: textual and visual. The **Textual Input** can contain optional contextual information, while the **Visual Input** comprises images representing the reasoning task. These inputs, along with an *intermediate feedback*, are integrated via an **Context Generator** to produce a semantically enriched **Context**. This input is then passed to a neural **Rule Generator**, which proposes symbolic rules in a

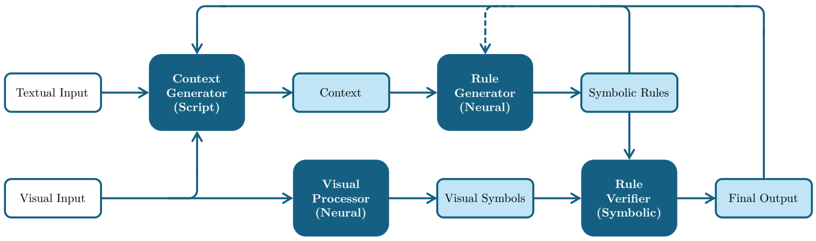

Figure 1.1: Conceptual diagram of the proposed NeSy system.

formal language such as FOL. Concurrently, the visual input is also transformed into **Visual Symbols** through a **Visual Processor**. After that, both the symbolic rules and visual symbols are processed by a symbolic **Rule Verifier**, which assesses their alignment and determines the **Final Output**.

On top of containing the results of the described experiment, both the generated symbolic rules and the final output serve as the source of the so-called intermediate feedback. This feedback is *primarily* designed to enhance the overall performance of the system by informing the context generator about possible failures or inconsistencies in previous experiments. By incorporating this feedback, the system can dynamically adjust the semantic context to better support rule generation. Additionally, the feedback can also be *directly* passed to the rule generator, which can enable it to immediately fine-tune its symbolic hypotheses based on empirical evidence. This optional feedback also increases the robustness and *expressive power* of the generated rules and promotes more efficient convergence accurate reasoning.

Consequently, the proposed system operates in an **Iterative Loop** that resembles Reinforcement Learning (**RL**) dynamics, as it iteratively uses the intermediate feedback to gradually refine its internal blocks. This process continues until a predefined **Termination Criterion** is met, such as when the generated symbolic rules adequately explain the majority of the images included in the visual input. This iterative design not only enables **Continuous Refinement of Symbolic Hypotheses** but also facilitates **Feedback-Based Learning and Convergence**. Through this mechanism, the system addresses common NeSy limitations by:

- **Providing Flexibility** in visual reasoning tasks, imposing minimal constraints on the types of problems it can handle.

- **Ensuring Explainable** both regarding its architectural design and the rules it generates at the end.

- **Upholding Formality** in rule generation, enabling the output to be understood and verified by both humans and machines.

1.8 Structure of the Thesis

This thesis is organized into four main chapters, including the current one. Chapter 2 provides a comprehensive literature review, covering foundational concepts, relevant tasks and benchmarks, and various approaches to NeSy visual reasoning. Chapter 3 presents the proposed methodology in detail, outlining each component of the system, ranging from symbolic rule generation and verification to visual processing and feedback integration. Chapter 4 discusses the experimental setup and evaluation metrics, presents the results, and offers a critical analysis of the findings. It also concludes with a summary of the contributions and outlines possible directions for future work.

Chapter 2

Literature Review

In this chapter, we review foundational concepts, benchmarks, and architectural paradigms within NeSy AI with a focus on visual reasoning tasks, where will cover different approaches for integrating neural and symbolic methods.

2.1 Foundational Concepts

As discussed earlier, NeSy AI aims to *synthesize* the complementary capabilities of neural networks and symbolic reasoning systems, moving beyond the traditional dichotomy between *connectionist* and *symbolic* paradigms [3]. A growing body of work now explores how these paradigms can be integrated to form hybrid models that are both data-efficient and cognitively plausible [1]. Importantly, NeSy systems *are not* monolithic, instead, they vary in architecture, areas of integration, and the degree to which symbolic rules influence learning or inference [17]. This diversity is not merely technical but reflects deeper assumptions about cognition and representation, *increasingly* shaped by interdisciplinary insights from cognitive science, linguistics, and formal logic [3].

To bring structure to this growing body of work, **Professor Henry Kautz** proposed a detailed taxonomy that *classifies* NeSy systems based on the nature and depth of interaction between neural and symbolic components. The taxonomy includes six types [3, 9]:

1. **symbolic Neuro symbolic**: This category resembles the standard pipeline of deep learning, where symbolic inputs (e.g., words) are *transformed* into vector embeddings, processed through neural architectures, and then mapped back into symbolic outputs. Most Natural Language Processing (**NLP**) systems fall into this category.

2. **Symbolic[Neuro]**: In this configuration, a symbolic system governs the overall

logic and control, while neural networks are used internally as *subroutines*. A well-known example is **AlphaGo**, which is an AI playing the game of GO through symbolic Monte Carlo tree search enhanced by neural networks to estimates game states.

3. **Neuro;Symbolic**: This architecture features a pipelined integration in which neural and symbolic systems operate on *distinct components* of the task. Therefore, these systems communicate in a loosely coupled manner, such as passing information back and forth to enhance mutual performance.

4. **Neuro:Symbolic → Neuro**: Here, symbolic knowledge is *embedded* directly into the structure of the neural network itself. In other words, rather than merely using symbolic inputs, these systems compile symbolic rules into architectural priors or weight initialization, influencing the neural learning process in a deeper model-intrinsic way.

5. **Neuro$_{\textbf{Symbolic}}$**: This category involves *tensorizing* symbolic structures (e.g., FOL representations) so that neural networks can perform reasoning tasks over them. These approaches maintain an interplay between formal logic and neural computation in a fully differentiable manner [5].

6. **Neuro[Symbolic]**: This is the *transpose* of the second category, where a neural architecture performs symbolic reasoning by learning structural relations or by attending to symbolic elements when required. As a prototypical example, we can consider Graph Neural Networks (**GNNs**), where the neural model effectively learns and reasons over symbolic graph structures.

The taxonomy provides a conceptual foundation for analyzing and comparing the many flavors of NeSy architectures and reflects both *practical implementation concerns* such as modularity, explainability, or scalability and *theoretical insights* into how symbolic and sub-symbolic representations can co-evolve [5]. Building on this framework, the survey **Towards Cognitive AI Systems** extends Kautz's taxonomy by introducing dimensions such as probabilistic integration, types of intermediate representations, and degrees of explainability, which offer *more comprehensive mapping* of recent developments in NeSy AI [9]. In this survey, rather than framing integration as a binary property, it is tried to treat NeSy AI as a *spectrum* that ranges from loosely coupled ensembles to deeply fused models with joint optimization. This perspective underscores how design decisions at the neural or symbolic interface affect the cognitive fidelity and explanatory power of the resulting systems [3]. Importantly, the survey highlights that these trade-offs are *particularly* impactful in domains like visual reasoning, where perception and logic must interact seamlessly to achieve meaningful interpretations.

As mentioned earlier, again, visual reasoning, in particular, serves as a compelling testbed for NeSy architectures, which requires *integrating* bottom-up perception with top-down logical inference to interpret scenes, detect causal relations, and apply structured reasoning over visual inputs [18]. However, unlike conventional visual recognition tasks, visual reasoning demands interpreting spatial and temporal patterns, analogies, and counterfactuals [7]. We will cover the examples later, but here, as an overview, we can consider tasks such as how VAR challenges models to infer plausible causes for incomplete visual sequences or how RPM tests the ability to detect and apply abstract visual rules [7, 18]. NeSy models address these challenges by *jointly* learning from visual data and structured symbolic domains, such as spatial logic, temporal constraints, or analogical schemes [8]. These capabilities not only benchmark NeSy systems but also drive innovation in architectures that balance learned representations with interpretable symbolic reasoning [1].

2.2 Different Tasks and Benchmarks

To evaluate NeSy visual reasoning systems, a range of tasks and benchmarks have been proposed, each emphasizing different aspects of reasoning capabilities [19]. One notable task is Visual Abductive Reasoning (**VAR**), which, as mentioned before, focuses on *uncovering* hidden causal relationships in visual narratives, even when the connections between events appear non-obvious or disjoint [7]. This task challenges AI systems to reason holistically over visual scenes and synthesize contextual information to infer the most plausible underlying causes. Additionally, as a task closely related to VAR, Dense Video Captioning (**DVC**) seeks to *produce* rich, multi-sentence descriptions of untrimmed videos, which demands temporally grounded, comprehensive narrative understanding [7]. Some approaches address this task by first parsing events and then generating text, while others reverse the process or unify both stages.

Building upon the VAR and DVC tasks, the **VideoABC** benchmark introduces a challenging procedural visual reasoning task [20]. Designed to assess a model's ability to *interpret* and explain physical processes in instructional videos, VideoABC emphasizes long-term dependencies and common-sense reasoning [20]. The primary task involves *selecting* the most plausible action or step to complete a visual procedure given an initial and final state. The secondary task, on the other hand, is a more demanding task that asks the model to *justify* why alternative choices are less plausible, thus discouraging superficial pattern matching. Notably, VideoABC avoids reliance on natural language inputs to provide a *purely* visual benchmark that requires genuine high-level reasoning over visual sequences [20].

Another foundational benchmark is Raven's Progressive Matrices (**RPM**), along with its variants Relational and Analogical Visual Reasoning (**RAVEN**) and Impartial RAVEN (**I-RAVEN**), which are modeled after classic IQ tests [18]. As mentioned before, these benchmarks assess abstract and analogical reasoning by requiring the model to *infer* the missing piece in a visual grid based on underlying rules of symmetry, transformation, or pattern continuation [21]. NeSy methods have shown strong performance on RPM tasks by combining visual perception with symbolic rule-based reasoning [18]. For instance, some approaches model the task as a form of probabilistic abductive reasoning, where solutions are inferred within a *symbolically* structured space constrained by prior background knowledge [1]. While this approach offers explainability and generalizability, it often incurs a high computational cost due to the complexity of the symbolic search space [21].

Beyond structured grid-based reasoning, Visual Question Answering (**VQA**) represents a more open-ended and linguistically grounded task, which challenges systems to *answer* natural language questions based on images through integrating of visual scene understanding, semantic reasoning, and multi-modal inference [19]. Successful models in this task must handle relational reasoning across spatial and semantic dimensions, combining features from both the visual input and the textual query [6]. However, early VQA benchmarks suffered from dataset biases and superficial *shortcuts* that allowed models to perform well without truly understanding the visual input [20]. To overcome these limitations, the Compositional Language and Elementary Visual Reasoning (**CLEVR**) dataset was introduced, featuring images of 3D objects and explicitly relational questions, where previous powerful models struggled with the relational aspects [6]. CLEVR has become a key benchmark for evaluating whether systems genuinely perform reasoning rather than exploit statistical patterns in language.

Finally, a benchmark tailored specifically for NeSy visual reasoning paradigms is Visual Sudoku Puzzle Classification (**ViSudo-PC**), which blends visual perception with symbolic relational constraints, requiring systems to *determine* whether a visually rendered Sudoku grid is correctly solved [22]. Unlike traditional numeric input, ViSudo-PC puzzles are built from images drawn from datasets such as Modified National Institute of Standards and Technology (**MNIST**), Extended MNIST (**EMNIST**), Kuzushiji MNIST (**KMNIST**), and Fashion MNIST **FM-NIST**, which introduces visual variability and noise [10]. The benchmark supports both 4×4 and 9×9 grid sizes to test the model's ability to integrate low-level visual recognition with high-level reasoning about Sudoku rules [22]. As such, ViSudo-PC represents a holistic NeSy challenge that demands *joint* learning of visual perception and reasoning.

2.3 Basic Visual Reasoning Frameworks

Building on the foundational concepts of NeSy AI and the tasks and benchmarks introduced previously, this section discusses several core frameworks for NeSy visual reasoning. Among these methods, one of the most influential neural components tailored for reasoning tasks is the Relation Network (**RN**) [6]. These networks are *modular* neural architectures designed explicitly for relational reasoning, which is an essential skill for many visual reasoning tasks [6]. When applied to datasets such as CLEVR, RNs significantly enhances performance by enabling models to reason about object relations, an issue that convolutional architectures alone often *struggled* to capture [6]. By augmenting perception pipelines with RNs, models can gain the ability to *implicitly* learn logical structures such as comparison, counting, and spatial relationships.

Moving from modular perception-enhancing components to integrated reasoning architectures, the Deep Symbolic Learning (**DSL**) framework represents a crucial step toward unifying perception and symbolic inference [13]. DSL enables *end-to-end* learning of symbolic representations directly from raw perceptual inputs, and simultaneously discovers the underlying symbolic rules [13]. By embedding discrete symbolic choices within differentiable neural layers, DSL offers a compositional approach to reasoning that supports generalization across tasks and domains [13]. This makes it particularly relevant in contexts where the structure of symbolic reasoning must emerge from perception, such as visual classification with latent symbolic structure [12]. However, while DSL integrates reasoning and perception tightly, another key challenge for hybrid systems is efficient training, especially when symbolic supervision is weak or indirect [12]. A practical and broadly applicable strategy is the use of Transfer Learning (**TL**) for the neural perception modules [12]. In this approach, the perception component of a NeSy system is *pre-trained* on the downstream task using standard supervision, before being integrated with the symbolic reasoning module [12]. This technique mitigates issues such as slow convergence and local minima by ensuring that the perception model already maps inputs to semantically informative representations [12]. It has shown consistent improvements in both performance and training efficiency across various NeSy setups and complex visual reasoning tasks [12].

Among more formalized integrations of symbolic logic and neural perception, Neural Probabilistic Soft Logic (**NeuPSL**) exemplifies a *general-purpose* NeSy framework that employs neural capabilities to extend Probabilistic Soft Logic (**PSL**), which is a statistical relational learning framework that, similarly to fuzzy FOL, represents logical rules as soft constraints using continuous truth values [10]. NeuPSL inherits PSL's flexible probabilistic reasoning framework but *introduces* differentiable components that interface directly with raw data via deep neural networks [10]. By supporting joint learning and inference over symbolic rules and perceptual inputs, NeuPSL demonstrates the utility of energy-based modeling for tasks like MNIST-Addition, where visual and symbolic aspects interact [10]. As part of the broader family of Neuro-Symbolic Energy-Based Models (**NeSy-EBMs**), NeuPSL contributes to a growing line of work seeking probabilistic formalism within hybrid reasoning [10].

Lastly, the challenge of symbol grounding in generative tasks is addressed by frameworks like Abductive Visual Generation (**AbdGen**), which *combine* neural visual generation models with logical rule learning via abductive reasoning [23].This generative visual reasoning requires the system to *assign* semantic symbols to latent neural factors and to infer rules that guide the generation process [23]. AbdGen achieves this purpose through a combination of quantized abduction and contrastive meta-abduction, effectively grounding symbols in visual data while maintaining a logic-based generative structure [23]. This approach enables it to learn from *limited* data and supports precise and explainable generation based on symbolic conditions. [23]

2.4 Visual Reasoning with VSAs

Vector-Symbolic Architectures (**VSAs**) are computational models that *represent* both atomic and composite concepts using high-dimensional distributed vectors [18]. These representations are manipulated through a defined set of algebraic operations, such as binding (e.g., Hadamard product), unbinding, bundling (additive superposition), permutation, and cleanup via associative memory, which together allow for the combination of symbolic structure with distributed and connectionist processing [1]. VSAs support key cognitive properties like compositionality, transparency, and robustness, which makes them well-suited for tasks that require reasoning over structured representations [24].

Early applications of VSAs to visual reasoning focused on *analogical reasoning*, often assuming symbolic inputs *without* integrating visual perception [1]. These limitations have led to the development of Neuro-Vector Symbolic Architectures (**NVSAs**), which incorporate deep neural networks for visual input processing and use VSA-based mechanisms for reasoning [1]. The NVSA framework *transforms* raw images into fixed-width VSA vectors that preserve perceptual uncertainty, which enables symbolic-like reasoning while retaining neural flexibility [1]. Then, the vector-based backend enables efficient probabilistic reasoning, such as probabilistic abduction, without resorting to *exhaustive* symbolic search [21]. In doing so, NVSA models allow perceptual processes to be shaped by the demands of downstream reasoning [1].

Building on the above ideas, the Relational Reasoning with Symbolic and Object-Level Features Using VSA processing (**RESOLVE**) architecture enhances the

NVSA approach by introducing an *attention mechanism* in high-dimensional bipolar vector spaces [24]. RESOLVE encodes both object-level features and inter-object relations through VSA operations such as binding and bundling, while its attention mechanism improves relational extraction efficiency, which is a notable challenge for transformer-based models [24]. RESOLVE demonstrates *strong* generalization in both fully relational and mixed-relational visual reasoning benchmarks [24].

Another notable approach is Probabilistic Abduction for Visual Abstract Reasoning via Learning Rules in VSAs (**Learn-VRF**), which uses VSAs to learn rule-based formulations for solving abstract reasoning problems, specifically RPM [18]. Unlike prior models that rely on *hand-crafted rule* representations, Learn-VRF learns rule structures directly from data while maintaining symbolic explainability [21]. It performs well on Out-of-Distribution (**OOD**) samples, demonstrating generalization to unseen combinations of attributes and rules [21]. More recently, the Abductive Rule Learner with Context-awareness (**ARLC**) has also extended the Learn-VRF by introducing a new training objective *specifically* tailored for abductive reasoning [18]. ARLC not only learns the rules underlying abstract reasoning tasks but also allows for the incorporation of domain knowledge through programmatic constraints [18]. It addresses prior limitations of Learn-VRF, such as its limited rule expressiveness and sub-optimal selection mechanisms, offering a more flexible and context-aware framework for rule induction [18].

Expanding further, the Abductive Visual Generation (**AbdGen**) framework also integrates logic programming with neural generative models under an abductive learning paradigm [23]. While primarily designed for visual generation tasks, AbdGen employs symbolic inference techniques, such as quantized abduction, which uses *nearest-neighbor lookups* in semantic code books to ground symbolic hypotheses in perceptual data [23]. This demonstrates the potential of VSAs in facilitating symbol grounding and abductive reasoning in visually rich contexts [23].

2.5 Visual Reasoning with LTNs

Logic Tensor Networks (**LTNs**) are presented as a significant NeSy framework designed to effectively *integrate* deep learning capabilities with symbolic reasoning, particularly relevant for visual reasoning tasks that demand processing both rich perceptual data and abstract knowledge [5]. The foundational element of LTNs is Real Logic, which, as mentioned, facilitates learning, querying, and reasoning by grounding the elements of a FOL signature onto data through neural computational graphs [5]. This framework employs fuzzy FOL semantics to transform the typically binary constraints of classical logic into continuous, differentiable operations, ultimately allowing logical knowledge to function as a differentiable regularizer in the loss function [12]. Consequently, LTNs offer a *unified* language capable of representing and computing a variety of AI tasks crucial for visual understanding and reasoning, including multi-label classification, relational learning, embedding learning, and query answering [5]. Notably, LTNs can perform reasoning on combinations of axioms not explicitly trained upon and provide high explainability, which are valuable characteristics when tackling complex visual scenarios [3].

A compelling demonstration of LTNs applied to visual reasoning tasks is explored in their application to the ViSudo-PC benchmark [8]. An LTN-based approach addresses this by integrating Convolutional Neural Networks (**CNNs**), which act as the perceptual module for tasks like digit recognition or whole-board classification, with an LTN responsible for *enforcing* the logical constraints of Sudoku [8]. Various methods for this integration and for formulating the logical constraints are investigated, which include **Indirect Solutions**, where a non-trainable predicate encodes the Sudoku rules to verify digits detected by a trainable predicate, and **Direct Solutions**, where a predicate is involved that directly calculates the validity of the entire board and, along with an auxiliary digit detection and rule enforcement predicate, is grounded by CNNs (either separate or with a shared backbone) [8]. This dual-level integration of perceptual recognition and logical reasoning exemplifies how LTNs can effectively *bridge* sub-symbolic and symbolic components in visual reasoning tasks [5].

2.6 Visual Reasoning with LLMs

As introduced in Section 1, LLMs have shown notable promise in performing reasoning tasks across multiple domains [14]. Extending this ability into the realm of visual reasoning, we can mention Vision-Language Models (VLMs), which combine natural language understanding with visual perception [14]. These models can be adapted for visual reasoning by *encoding* visual content into structured symbolic or textual representations [21]. In many current approaches, the raw visual data is preprocessed externally and transformed into a format that the LLM can interpret, such as object attributes, spatial relationships, or transformation rules [14]. Once this symbolic encoding is complete, the VLM is *prompted* to reason over the visual structure. This setup has enabled zero-shot performance on abstract reasoning benchmarks like RPMs, where VLMs can identify visual patterns and analogies without task-specific fine-tuning [21]. While the model itself does not directly interpret pixels, its general language-based reasoning abilities can be leveraged through careful design of symbolic inputs and prompts [3]..

Recent research has further explored how post-training methods influence the reasoning capabilities of LLMs and VLMs, particularly comparing Supervised Fine-Tuning (**SFT**) and RL: while SFT encourages memorization of training examples, RL fosters the emergence of generalized reasoning skills [14]. For instance, in visual tasks like RPMs, SFT-tuned models often struggle with OOD examples, whereas RL-trained models demonstrate improved adaptability and abstraction [14]. This insight is central to the development of systems like DeepSeek-R1, which applies RL to incentivize structured reasoning [2]. The DeepSeek-R1-Zero model, notably trained without prior SFT, reveals that reasoning capabilities can emerge solely from reinforcement signals [2]. Meanwhile, the full DeepSeek-R1 pipeline, which combines SFT and RL, balances stability and performance, guiding LLMs toward generating more accurate and interpretable Chain-of-Thought (**CoT**) reasoning [2].

Beyond DeepSeek, other works such as **MathPrompter** and **Auto-CoT** represent earlier or parallel efforts to enhance LLM reasoning through prompting strategies [25]. MathPrompter augments reasoning by retrieving relevant equations and concepts to scaffold mathematical problem solving, while Auto-CoT automates the generation of reasoning chains to improve model performance across logical tasks [16]. Though these methods primarily target textual reasoning, they share conceptual ties with visual reasoning when symbolic representations of images are treated analogously to structured text [21]. Together, these methods underscore a broader trend: effective visual reasoning with LLMs often hinges on external symbolic encoding and internal alignment via prompt engineering, fine-tuning, or reinforcement [21, 19, 14]. The ongoing research reflects a shift from passive language modeling to active reasoning architectures, where models are trained not just to generate plausible text, but to follow reasoning trajectories across modalities [25, 21].

Chapter 3

Methodology

3.1 More on the Proposed Method

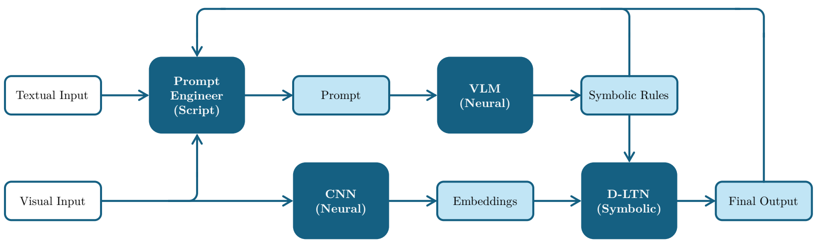

Figure 3.1: Functional diagram of the proposed NeSy system.

As discussed in Section 1.7, our objective is to overcome the limitations of the methods presented in the previous chapter by emphasizing flexibility, explainability, and formality. To this end, we transition from the high-level conceptual design shown in Figure 1.1 to a more concrete functional architecture illustrated in Figure 3.1. This refined design introduces the following key components:

- A **Prompt Engineer** serves as the context generator, combining the textual and visual inputs with intermediate feedback to construct a coherent **Prompt**. This prompt is then fed to the next processing block as the required context.

- A **VLM** acts as the rule generator. The idea behind this choice is to leverage the advanced reasoning capabilities of these modern language models for generating symbolic rules. Additionally, to obtain formal rules, we prompt the VLM to compose them in FOL and Python, which are necessary for constructing the symbolic knowledge base and performing logic-based reasoning.

- A **CNN** functions as the visual processor by converting the visual input into **Embeddings**, which are directly interpreted as visual symbols. By using a CNN, we map the images from a two-dimensional complex space into a simpler space of embedding vectors.
- An Dynamic LTN (**D-LTN**) performs rule verification. With the help of this block, the generated symbolic rules are automatically converted into an LTN, which is then tasked with applying the extracted rules to the embeddings.

In the following, we will explain the complexities and the detailed operations performed by each block of the functional diagram.

3.2 Rule Generation

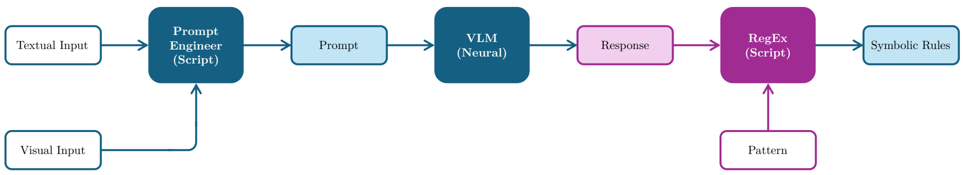

Figure 3.2: Reasoning with the help of VLM.

This section outlines the complete flow from the system inputs to the prompt engineering stage, through the VLM, and ultimately to the generation of symbolic rules. As illustrated in Figure 3.2, the process involves three additional components, the raw **Response** produced by the VLM, a Regular Expression (**RegEx**) used to extract symbolic rules from the response, and a specific **Pattern** that guides the extraction process.

3.2.1 Prompt Engineering

The following script presents our initial prompt engineering approach. For clarity, the template follows the style recommended by OpenAI in their documentation for using VLMs via Python, where the `client` object is used to access their Application Programming Interface (**API**) [26].

```
system_role = '''
You are a helpful assistant that can extract the First-OrderLogic
(FOL) from images. The grammar of the FOL is as follows:
  1. Constants: <PLACE_HOLDER>.
  2. Variable: <PLACE_HOLDER>.
  3. Functions: <PLACE_HOLDER>.
```

```
  4. Predicates: <PLACE_HOLDER>.
  6. The symbols used for AND, OR, and NOT: `<PLACE_HOLDER>`,
     `<PLACE_HOLDER>`, and `<PLACE_HOLDER>`, respectively.
  7. The symbols used for implication and equivalence:
     `<PLACE_HOLDER>`, and `<PLACE_HOLDER>`, respectively.
  8. The symbols for universal and existential quantifiers:
     `<PLACE_HOLDER>` and `<PLACE_HOLDER>`, respectively.
  9. Use parentheses for preserving operation precedence.
Act based on the following:
  1. Before FOL rule generation, deeply analyze the images.
  2. Consider that all the images must follow the same FOL rule.
  3. The FOL rule applies to the visual objects inside each image.
  4. At the end of your chain of thought, use the following
     template to present the extracted rules:
     ```JSON
     {
        "rule_1": "first possible rule",
        "rule_2": "second possible rule",
        ...
     }
     ```
  5. Then, provide the groundings of constants, functions, and
     predicates in the following template:
     ```Python
     the groundings
     ```
'''

prompt = [{
    'type': 'text',
    'text': 'These are images you can use as reference:'
}]
for base64_image in base64_image_list:
    prompt.append({
        'type': 'image_url',
        'image_url': {
            'url': f'data:image/png;base64,{base64_image}'
        }
    })

chat_completion = client.chat.completions.create(
    messages=[
        {'role': 'system', 'content': system_role},
        {'role': 'user', 'content': prompt}
    ]
```

```
)

response = chat_completion.choices[0].message.content
```

In this template, the placeholder `'<PLACE_HOLDER>'` indicates where specific information must be inserted into the `system_role` variable, which will be specified in the following sections once the grammar for the FOL language has been defined. Also, as shown in the template, the textual input of the system is provided via the `system_role` variable, while the visual input is passed to the VLM as part of the prompt, using images stored in the list `base64_image_list`. These images serve as reference examples from the training data, guiding the VLM in generating the corresponding symbolic rules. As previously mentioned, these rules are produced in two forms:

- **FOL**. According to the prompt, the VLM is first instructed to generate a JavaScript Object Notation (**JSON**) dictionary containing the possible FOL rules that describe the underlying relationships among the visual objects. The JSON format provides the flexibility for the VLM to extract and organize as many relevant rules as it identifies.
- **Python**. The VLM is also asked to produce a Python script containing the **Groundings**, which are the contextual meanings of the symbols appearing in the FOL rules. A grounding can be value or tensor assigned to a constant or a variable, a Python function or neural network represented by an FOL function or predicate, or a tensor operation performing the abstract idea of a universal quantifier. Even though every symbol of an FOL rule requires grounding in practice, the VLM is prompted to specify the groundings only for constants, functions, and predicates. We explain the details in subsequent sections.

The above approach assumes the VLM is already familiar with the concept of FOL and is capable of performing analytical reasoning. In fact, in our proposed architecture, the VLM effectively serves as the reasoning brain of the system, which allows it to operate with minimal hard-coded logic or rule-based intervention. Therefore, the reasoning process is delegated entirely to the VLM and the sophistication of the its reasoning capabilities directly impacts the quality of the results. In other words, the more advanced the model, the more reliable, nuanced, and expressive the extracted rules will be.

3.2.2 Symbolic Rule Generation

The raw response generated by the VLM is returned in plain text, which most likely includes the CoT reasoning steps it followed before producing the symbolic rules. This inclusion is typical behavior of modern language models, especially when prompted to reason step-by-step or operate under system instructions that prioritize transparency and explainability. Also, the CoT reasoning is inevitable, as it provides insight into the intermediate steps the model uses to reach its conclusions. However, from a system integration perspective, this verbose output becomes noise. What we actually require for downstream processing are only the two formal parts of the response, the JSON dictionary that holds the extracted FOL rules, and the Python script that lists the groundings. To separate these relevant segments from the surrounding explanatory text, we employ RegEx, which is a robust pattern-matching tool widely used in text processing. RegEx allows us to define flexible yet precise search patterns that can locate and extract the relevant blocks within unstructured textual output. Therefore, to extract the symbolic rule components effectively, we define two RegEx patterns, one for the FOL rules and another for the groundings:

````
fol_pattern = r'```JSON\s*(\{[^\}]*)\}\s*```'
python_pattern = r'```Python\s*([^```]*)\s*```'
````

The first pattern, fol_pattern, searches for a block of text that begins with the literal prefix '```JSON', followed by any amount of whitespace, and then captures a JSON object enclosed in curly braces {...}. The pattern ensures that it terminates at the matching closing brace before the final triple backticks '````'. Similarly, the second pattern, python_pattern, captures the Python code block. It searches for the prefix '```Python', then greedily captures all content until it encounters the closing backticks '````'. These carefully crafted expressions ensure the correct segments of the response are isolated from any explanatory text or CoT reasoning.

3.3 Rule Verification

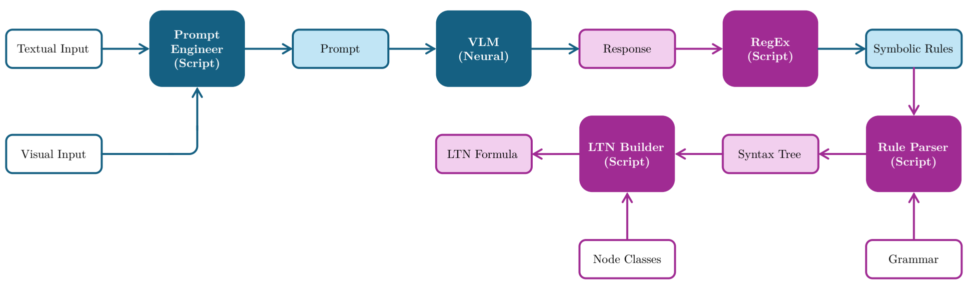

Figure 3.3: Automatic generation of LTN formulas.

With the VLM response decoded into an JSON dictionary with possible FOL rules and a Python script defining the groundings, we are now able to convert those symbolic rules into proper formats that finally lead to the implementation of a D-LTN. In summary, as illustrated in Figure 3.3, we can use a **Rule Parser** to convert the symbolic rules, stated in a pre-defined **Grammar**, into a **Syntax Tree**, and use it to generate an **LTN Formula** with the help of an **LTN Builder** by converting the raw nodes of the syntax tree into the customized **Node Classes** we already defined. In this section, we discuss the details of this process.

3.3.1 Syntax Tree Generation

A syntax tree is a hierarchy of nodes that represents the abstract syntactic structure of a set of instructions written in a formal language, in our case, FOL or Python. Each node in the tree denotes a construct occurring in the logical expression or the source code. Accordingly, in our methodology, a syntax tree serves two distinct purposes:

- The syntax tree for the FOL rules represents their logical structure, with nodes corresponding to predicates, constants, variables, functions, logical connectives, and quantifiers. This tree provides a canonical and unambiguous representation of the rule, abstracting away superficial details of the specific textual syntax.

- Similarly, a syntax tree for the Python script represents its programmatic structure, with nodes corresponding to expressions, statements, function definitions, and variable assignments. This tree is crucial for programmatically accessing and interpreting the groundings that the VLM provides.

With the use of these syntax trees, we can create well-defined formulas for our D-LTN block, ensuring that the symbolic knowledge is precisely interpreted and integrated. Accordingly, to convert the extracted FOL rules into a machine-readable format suitable for our D-LTN implementation, we leverage **Lark**, which is a powerful and flexible parsing toolkit for Python that allows for building custom parsers [27]. For the Python grounding script, on the other hand, we simply make use of Python's built-in **ast** module, which provides tools for working with syntax trees of Python source code directly. This simplifies the extraction of groundings by allowing direct traversal and inspection of the script's structure. For the FOL rules, however, the parsing process is more intricate.

As mentioned, Lark is a powerful parser generator. It uses context-free grammars to define the structure of a language and employs various parsing algorithms (such as **LALR**, **Earley**, or **CYK**) to convert plain text into syntax trees [27]. This process is similar to how development environments like Visual Studio Code (**VSCode**) utilize internal syntax parsers or textual grammars (such as **TextMate** grammars) to transform a plain Python script into a colored, syntactically highlighted visualization, regardless of its semantic meaning. In fact, a parser's sole responsibility is to construct a syntax tree based on the defined grammar. It does not concern itself with the semantic correctness or logical validity of the content. Therefore, just as VSCode does not inherently validate the runtime behavior or logical soundness of a Python script, our FOL parser similarly focuses exclusively on the grammatical correctness of the symbolic rules. The semantic validation, which involves assessing whether a syntactically correct FOL rule accurately describes the relationships within a specific domain, is a separate and crucial step that will be investigated later.

To define a context-free grammar for the conversion of plain FOL expressions into syntax trees using Lark, we primarily focus on balancing the **generalizability** of the grammar and its inherent **formality**. In other words, we want to define a grammar robust enough to encompass a broad spectrum of common FOL rules while simultaneously enforcing a strict formal structure, ensuring that all generated rules consistently adhere to the same syntactic conventions. Also, this allows the parser to render the rules unambiguously machine-readable for seamless integration into our D-LTN. According to Lark documentations, we need to define this grammar in Extended Backus-Naur Form (**EBNF**) [27], which in our case, is as follows:

```
//// Explanations ////

    // constant identifiers always start with "C"
    // variable identifiers always start with "x"
    // function identifiers always start with "f"
    // predicate identifiers always start with "P"
    // wrapper symbol is "(" and ")"
    // negation symbol is "!"
    // conjunction symbol is "&"
    // disjunction symbol is "|"
    // implication symbol is "implies"
    // equivalence symbol is "iff"
    // universal quantifier symbol is "forall"
    // existential quantifier symbol is "exists"

//// Initialization ////

    // imports
     %import common.WS
     %ignore WS

    // entry point
    ?start: expression
```

```
//// Term-Level Terminal Definitions  ////

    // Tree Structure:
    // term
    // atom
    //  constant, variable
    // mapper
    //   function

    // Abstract Terminal (no precedence)
    ?term: constant | variable | function

    // Concrete Terminals (no precedence)
    constant: /C[a-z0-9_]*/
    variable: /x[a-z0-9_]*/
    function: /f[a-z0-9_]*/ "(" term ("," term)* ")"

//// Expression-Level Terminal Definitions ////

    // Tree Structure:
    // expression
    // evaluator
    //  predicate
    // unary_connective
    //  wrapper, logical_not
    // binary_connective
    //  logical_and, logical_or, iff, implies
    // quantifier
    //   exists, forall

    // Abstract Terminal (ascending precedence)
    ?expression: level_0
    ?level_0: level_1 | exists | forall
    ?level_1: level_2 | iff | implies
    ?level_2: level_3 | logical_or
    ?level_3: level_4 | logical_and
    ?level_4: level_5 | logical_not | wrapper
    ?level_5: predicate

    // Concrete Terminals (ascending precedence)
    exists: "exists" variable ("," variable)* expression
    forall: "forall" variable ("," variable)* expression
    iff: level_1 "iff" level_2
    implies: level_1 "implies" level_2
    logical_or: level_2 "|" level_3
```

```
logical_and: level_4 "&" level_3
logical_not: "!" level_4
wrapper: "(" expression ")"
predicate: /P[a-z0-9_]*/ "(" term ("," term)* ")"
```

The provided grammar defines the formal syntax for the FOL language used in the proposed system, along with a clear hierarchy of operator precedence. At the base level, constants, variables, and functions are identified by the prefixes `C`, `x`, and `f`, respectively. On top of them, predicates, marked by the prefix `P`, act as the core evaluable units. Next, logical operators are applied in a strict order of precedence: negation (`!`) binds most tightly, followed by conjunction (`&`), disjunction (`|`), then implication (`implies`) and equivalence (`iff`), and finally universal (`forall`) and existential (`exists`) quantifiers operating at the highest level and introducing scoped variable bindings. To override this default precedence and enforce specific groupings, the grammar also includes a wrapper using parentheses, which enables unambiguous parsing of nested logical forms. According to this grammar, we can now update the prompt used by the VLM:

```
system_role = '''
You are a helpful assistant that can extract the First-OrderLogic
(FOL) from images. The grammar of the FOL is as follows:
  1. Constants: always starting with "C", e.g., "C", "C1" etc.
  2. Variable: always starting with "x", e.g., "x", "x_2", etc.
  3. Functions: always starting with "f", e.g., "f", "f_get", etc.
  4. Predicates: always starting with "P", e.g., "P", "P_equal", etc.
  6. The symbols used for AND, OR, and NOT: `&`, `|`, and `!`,
     respectively.
  7. The symbols used for implication and equivalence: `implies`
     and `iff` respectively.
  8. The symbols for universal and existential quantifiers: `forall`
     and `exists`, respectively.
  9. Use parentheses for preserving operation precedence.
Act based on the following:
  1. Before FOL rule generation, deeply analyze the images.
  2. Consider that all the images must follow the same FOL rule.
  3. The FOL rule applies to the visual objects inside each image.
  4. At the end of your chain of thought, use the following
     template to present the extracted rules:
     ```JSON
     {
        "rule_1": "first possible rule",
        "rule_2": "second possible rule",
        ...
     }
```
```

```
      ```
   5. Then, provide the groundings of constants, functions, and
      predicates in the following template:
      ```Python
      the groundings
      ```
'''

prompt = [{
    'type': 'text',
    'text': 'These are images you can use as reference:'
}]
for base64_image in base64_image_list:
    prompt.append({
        'type': 'image_url',
        'image_url': {
            'url': f'data:image/png;base64,{base64_image}'
        }
    })

chat_completion = client.chat.completions.create(
    messages=[
        {'role': 'system', 'content': system_role},
        {'role': 'user', 'content': prompt}
    ]
)

response = chat_completion.choices[0].message.content
```

### 3.3.2 LTN Implementation

Following the successful parsing of the symbolic rules, encompassing both the FOL interaction descriptions and their corresponding Python groundings, we can now proceed with their conversion into LTNs. To understand this process, it is essential to delve deeper into the fundamental nature of an LTN. At its core, an LTN represents a *fuzzified grounding* of an FOL rule, leveraging functions and neural networks to enable differentiability and trainability [5]. Therefore, the implementation of an LTN essentially involves translating this grounding process into a computational framework. For instance, consider an FOL rule such as $\forall x_1 \exists x_2 : \mathrm{Pred}(\mathrm{func}(x_1), x_2)$. The conversion starts by defining how the function func and the predicate Pred behave within a fuzzy logical space, which is actually why we previously prompted the VLM to generate Python scripts containing these concrete groundings for constants, functions, and predicates. Subsequently, we
```

must also define the fuzzy groundings for the logical quantifiers, $\exists$ (existential) and $\forall$ (universal). With these fuzzy groundings established for all components of the rule, an LTN is effectively constructed.

Consequently, given the groundings of $x1$ and $x2$, which are the input embeddings of the D-LTN block, we can then *trigger* this instantiated LTN, similar to evaluating the truth value of the original FOL rule, but in a continuous and differentiable manner. However, given our system's automatic nature, this conversion process simply cannot be manual. So, how do we transform the generated symbolic rules into an executable LTN formula? The answer lies within the syntax trees we have already discussed. In summary, we use Lark again to convert the FOL-based syntax trees into LTN-based syntax trees. Next, these LTN-based trees are grounded with the Python objects that we have prepared earlier.

The core idea behind the above approach is that traversing these newly formed LTN-based trees is functionally equivalent to feeding inputs directly into the LTN. In other words, the structure of the LTN-based syntax tree *is* the computational graph of the LTN itself. Considering our example rule, $\forall x_1 \exists x_2 : \mathrm{Pred}(\mathrm{func}(x_1), x_2)$, once converted, this rule will be represented as a tree where the root node is $\forall x_1$, its child is $\exists x_2$, whose child is Pred, and Pred then has func and x_2 as its children, with func in turn having x_1 as its child. When we *traverse* this tree, we start from the innermost node, which is x_1. Then, the node representing func, which is now a fuzzy, differentiable function defined in our Python groundings, receives the embedding for x_1 as input and computes its output. After that, the node representing Pred, which is grounded by a fuzzy predicate, takes the computed output from $\mathrm{Pred}(\mathrm{func}(x_1), x_2)$ and the embedding for x_2 as its inputs, computing its fuzzy truth value. Next, the $\exists x_2$ node, grounded as a fuzzy existential quantifier, processes the truth value from Pred by considering all possible instantiations of x_2 in the domain to find the existentially-aggregated truth value. Finally, the $\forall x_1$ node, grounded as a fuzzy universal quantifier, takes this result, considers all possible instantiations of x_1, and finds the universally-aggregated truth value.

The above sequential evaluation of an LTN-based syntax tree *is* the exact process of feeding the embeddings directly into the LTN to compute its final truth value for the rule. However, the fundamental question is how to we create an LTN-based syntax tree. Along with the ability to create custom parsers, `Lark` also provides transformers to convert raw syntax trees into customized ones, in our case, from an FOL-based syntax tree into an LTN-based syntax tree. For this purpose, we need to define custom node classes for the tree so that we achieve the functionality described in the previous example. To begin, we define the `Base` class as the superclass of all other nodes. Then, considering the grammar we defined previously, we proceed to define the remaining node classes according the hierarchy provided in the grammar.

Below the `Base` class level, we also have other high-level superclasses, which describe the fundamental roles of the nodes in the LTN-based syntax tree by grouping them into abstract categories based on their functionality. These classes are as follows:

- `Term`: extends `Base`. This abstract class serves as a superclass for all nodes representing logical terms, such as variables, constants, and function applications. The instances of this class are tensors that are fed into predicates or other expressions.
- `Expression`: extends `Base`. This abstract class serves as a superclass for all nodes representing logical expressions, including predicates, connectives, and quantified formulas. The instances of this class are responsible for computing the truth values of the expressions they represents.

Also, below the above classes, we have middle-level superclasses, which provide more specialized abstractions that further refine the structure and semantics of the LTN-based syntax tree. Each class defines a logical category of behavior that will be inherited by concrete node types. They are as follows:

- `Atom`: extends `Term`. This abstract class represents the basic terms, i.e., constants and variables.
- `Mapper`: extends `Term`. This abstract class represents functions that map one or more terms to another term, including both built-in operations (e.g., vector summation) and user-defined functions.
- `BinaryConnective`: extends `Expression`. This abstract class represents logical connectives that operate on two expressions, i.e., conjunctions, disjunctions, equivalences, and implications.
- `UnaryConnective`: extends `Expression`. This abstract class represents logical connectives that operate on a single expression, i.e., negations and groupings with parentheses.
- `Quantifier`: extends `Expression`. This abstract class represents quantified logical expressions that operate over domains of variables, i.e., universal or existential quantifiers.
- `Evaluator`: extends `Expression`. This abstract class represents logical predicates that evaluate terms to a truth value, including both built-in relations (e.g., equality, inequality) and user-defined predicates.

Finally, the low-level concrete classes are the actual building blocks that instantiate the nodes of the LTN-based syntax tree. These classes implement specific logic or structure corresponding to the elements of FOL. They are as follows:

- `Constant`: extends `Atom`. It represents a constant term.
- `Variable`: extends `Atom`. It represents a variable term.
- `Function`: extends `Mapper`. It represents a user-defined function.
- `Iff`: extends `BinaryConnective`. It represents equivalence between two expressions.
- `Implies`: extends `BinaryConnective`. It represents implication between two expressions.
- `LogicalOr`: extends `BinaryConnective`. It represents disjunction between two expressions.
- `LogicalAnd`: extends `BinaryConnective`. It represents conjunction between two expressions.
- `LogicalNot`: extends `UnaryConnective`. It represents negation of a single expression.
- `Wrapper`: extends `UnaryConnective`. It represents a syntactic wrapper for grouping expressions using parentheses.
- `Exists`: extends `Quantifier`. It represents an existential quantifier.
- `ForAll`: extends `Quantifier`. It represents a universal quantifier.
- `Predicate`: extends `Evaluator`. It represents a user-defined predicate.

We designed the above classes as general as possible to enable Lark to convert any FOL-based syntax tree into an LTN-based syntax tree. However, in the following, we assume that every LTN formula is always derived from an FOL expression and never a term. For example, a term like $f(x)$ is not an expression and it cannot serve as a valid LTN formula on its own. Therefore, the root of any LTN-based syntax tree must always be an expression, which ensures that the output of the LTN evaluation is always a truth value, and not a multi-dimensional tensor. Moreover, we intentionally did not implement concrete classes for built-in functions or predicates. If such functionality is required, it must be added explicitly by extending either `Mapper` for functions or `Evaluator` for predicates. Alternatively, possible users of these classes can define their own. For example, although the framework does not include a built-in predicate for comparing variables (e.g., greater-than), a user could extend `Evaluator` to define a custom predicate class such as `P_gt` to implement this logic.

To finalize the LTN formula, we need to equip the `Base` class with **Groundability** and **Callability** capabilities. Groundability refers to the ability to assign specific groundings to the nodes, while callability allows input values to be passed to the formula for evaluation. By adding them to the base class, all subclasses automatically inherit these features. To implement them, we define a `ground` method that accepts grounding values as named arguments, and a `call` method that also accepts inputs via named arguments. Since the resulting LTN formula is structured as a tree, these methods are only invoked at the root node. In other words, for any LTN formula, we only need to call the `ground` method once to propagate groundings to all nodes, and similarly, we call the formula once to provide input values to the entire structure.

It is worth noting that both methods described above serve the purpose of grounding the LTN formula, but in different ways. The `ground` method is used to ground all nodes that are static in nature, such as constants, functions, predicates, and any other node that is not an instance of the `Variable` class. In contrast, the `call` method is specifically used to ground variables. This separation is deliberate, as it highlights the dynamic behavior of variables in contrast to the fixed nature of other nodes. In other words, non-variable nodes are grounded once and remain unchanged throughout the evaluation, while variables may be grounded multiple times, reflecting their role as placeholders for changing input values. This dynamic nature is also the reason we have referred to variable groundings as *inputs*. Additionally, when defining the node classes, we can optionally assign default groundings to connectives and quantifiers. These elements are formal and domain-independent, so their semantics can be predefined. In contrast, constants, functions, and predicates are user-defined and domain-specific. Therefore, they must be explicitly grounded by the user or provided externally. This is the main reason we prompt the VLM to supply them along with the FOL rules.

3.4 Visual Processing

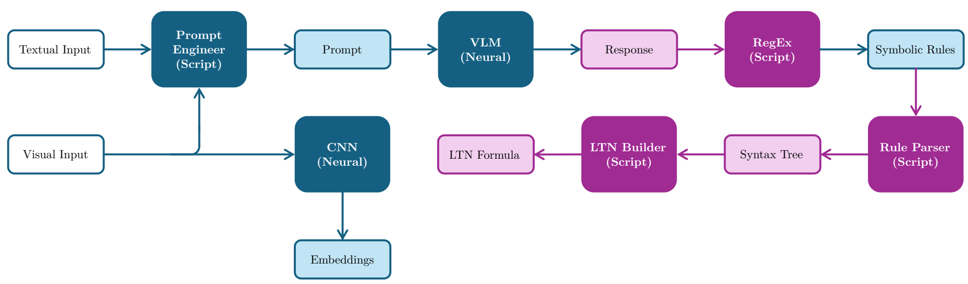

Figure 3.4: Perception with the help of CNN.

The next major component in our functional diagram is where the visual input is processed. As shown in Figure 3.4, we do not explicitly expand the diagram with additional visual processing details, but this section will describe the internal workings of the CNN block. Specifically, we explain how it processes raw visual input into embeddings and how these embeddings are subsequently used as visual symbols within our system.

3.4.1 CNN Design

The visual input consists of a batch of images, each containing multiple visual objects. Assuming a batch size of B, a maximum of O objects per image, object dimensions of $W \times H$, and C color channels, the input tensor has the shape $B \times O \times C \times W \times H$. The CNN processes this input through a series of *convolutional* layers, followed by a *linear projection* that maps each object to an embedding of dimension E, resulting in an output tensor of shape $B \times O \times E$. As previously mentioned, one key benefit of this CNN block is that it transforms complex, high-dimensional visual data into a compact and semantically meaningful embedding space. Additionally, this visual encoder plays a foundational role in the system: it is responsible for perceiving and abstracting visual information, while the rest of the architecture focuses on symbolic reasoning over these representations. For this reason, it is commonly referred to in the NeSy literature as **Perceptor**.

The perceptor can be integrated into the system in two principal ways, depending on the desired trade-off between modularity and adaptability:

- **Modular Setting (Frozen Perceptor)**: In this setup, the CNN is pre-trained independently on a task related to the visual domain, such as object classification or detection. Once trained, the perceptor is frozen and used as a fixed feature extractor during the reasoning phase. This approach can offer reduced training complexity and modularity, which allows researchers to analyze the symbolic reasoning component in isolation.

- **End-to-End Setting (Trainable Perceptor):** Here, the perceptor is not pre-trained but instead trained jointly with the reasoning module using supervision from final reasoning outcomes (e.g., logical inference labels or task-specific decisions). This setup allows the system to adapt visual representations to better align with symbolic tasks, potentially improving performance.

Both settings are valid and useful. The modular setting is often favored when pre-trained vision models are available. The end-to-end setting, on the other hand, is more suitable when domain-specific visual features are not easily captured by existing models, or when tight coupling between perception and reasoning is necessary for optimal performance.

3.4.2 Visual Symbol Definition

As discussed earlier, the embeddings produced by the CNN are used directly as visual symbols. These embeddings encode contextual properties of each object, such as shape, color, and texture, into a tensor of shape $B \times O \times E$. However, in addition to these semantic features, each visual object is also associated with positional information, such as bounding boxes or spatial indices, which play a crucial role in visual reasoning. To incorporate this information into the embeddings, we can consider two approaches:

- **Projection-Based Fusion**: Apply a separate learnable projection layer to the positional features to map them into the same embedding space, and then add the result to the semantic embeddings.

- **Concatenation-Based Fusion**: Directly concatenate the positional features to the semantic embeddings, resulting in an augmented embedding vector.

Both strategies aim to enrich the visual symbols with spatial context, enabling the reasoning component to exploit both semantic and positional cues.

3.5 Feedback Handling

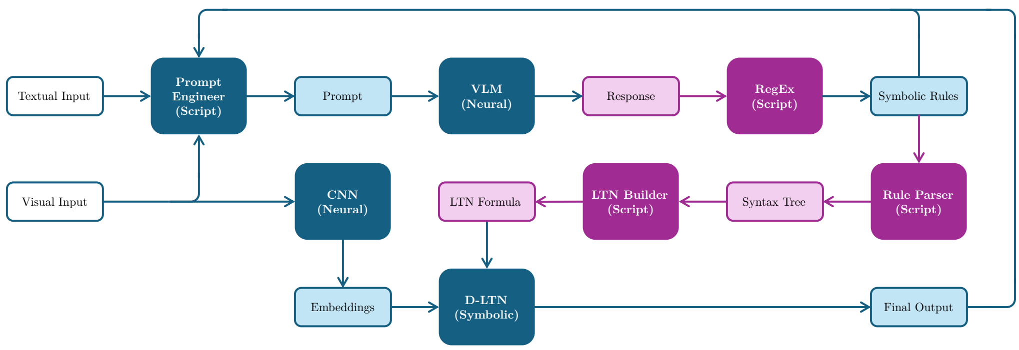

Figure 3.5: Detailed functional diagram of the proposed NeSy system.

Figure 3.5 contains the full functional diagram of the proposed system with its final feedback loop. As discussed, this loop is to provide the prompt engineer with the intermediate feedback both from generated symbolic rules and the final output of the D-LTN. In this section, we will discuss how this feedback is prepared.

3.5.1 LTN Development

Given our indirect approach to implementing LTNs (traversing the LTN-based syntax trees via the `ground` and `call` methods), we define a D-LTN as trainable if it contains at least one node that is grounded by a trainable object such as a neural network. For example, consider the formula $\forall x_1 \exists x_2 : \text{Pred}(\text{func}(x_1), x_2)$. If either `Pred` or `func` is grounded by a neural model within the corresponding LTN-based syntax tree, then the entire D-LTN becomes trainable. This capability allows us to optimize the D-LTN and shape its behavior through learning.

Therefore, to train such a system, we require a dataset consisting of appropriate textual inputs and corresponding labeled visual inputs. In addition, in order to evaluate the model's performance, we must also define a suitable metric that reflects its symbolic reasoning capabilities grounded in visual perception. Since our system is designed to extract and verify symbolic rules from images, we focus the evaluation on binary decision-making, which is determining whether the given label for an image is correct. This approach aligns with the semantics of fuzzified FOL rules, which produce a truth value $t \in [0,1]$. By setting a decision threshold $\tau \in [0,1]$, we can interpret the output as follows as if $t \geq \tau$, the system considers the rule to be satisfied, otherwise, it does not.

3.5.2 Loop Creation

Once the final output of the system is produced, the learning loop can be completed. At this stage, the system generates intermediate feedback consisting of the most recently derived symbolic rules along with the corresponding performance evaluation. This feedback is then incorporated into the next prompt, enabling the system to iteratively refine its symbolic understanding and improve performance over time. This prompt is what we have provided below, where `'<PREVIOUS_FOL_RULE'`, `'<PREVIOUS_GROUNDINGS>'`, `'<METRIC_NAME>'`, and `'<METRIC_VALUE>'` denote the most recent FOL rule with the highest performance, the most recent groundings, the evaluation metric used to evaluate the system, and the numerical performance score associated with it.

```
system_role = '''
You are a helpful assistant that can extract the First-OrderLogic
(FOL) from images. The grammar of the FOL is as follows:
  1. Constants: always starting with "C", e.g., "C", "C1" etc.
  2. Variable: always starting with "x", e.g., "x", "x_2", etc.
  3. Functions: always starting with "f", e.g., "f", "f_get", etc.
  4. Predicates: always starting with "P", e.g., "P", "P_equal", etc.
  6. The symbols used for AND, OR, and NOT: `&`, `|`, and `!`,
     respectively.
  7. The symbols used for implication and equivalence: `implies`
```

```
     and `iff` respectively.
  8. The symbols for universal and existential quantifiers: `forall`
     and `exists`, respectively.
  9. Use parentheses for preserving operation precedence.
Act based on the following:
  1. Before FOL rule generation, deeply analyze the images.
  2. Consider that all the images must follow the same FOL rule.
  3. The FOL rule applies to the visual objects inside each image.
  4. You performed the same operation previously, where you extracted
     <PREVIOUS_FOL_RULE>
     and these groundings in Python
     <PREVIOUS_GROUNDINGS>
     where you achieved <METRIC_NAME> at <METRIC_VALUE>
  5. At the end of your chain of thought, use the following
     template to present the extracted rules:
     ```JSON
     {
        "rule_1": "first possible rule",
        "rule_2": "second possible rule",
        ...
     }
     ```
  6. Then, provide the groundings of constants, functions, and
     predicates in the following template:
     ```Python
     the groundings
     ```
'''

prompt = [{
    'type': 'text',
    'text': 'These are images you can use as reference:'
}]
for base64_image in base64_image_list:
    prompt.append({
        'type': 'image_url',
        'image_url': {
            'url': f'data:image/png;base64,{base64_image}'
        }
    })

chat_completion = client.chat.completions.create(
    messages=[
        {'role': 'system', 'content': system_role},
        {'role': 'user', 'content': prompt}
```

```
    ]
)

response = chat_completion.choices[0].message.content
```

Chapter 4

Evaluation and Discussion

4.1 Setup

In this section, we detail the steps taken to evaluate our proposed NeSy visual reasoning system with the functional diagram depicted in Figure 3.5. For this purpose, we begin by describing the benchmark and dateset utilized for the experiments. Next, outline the experimental environment, which includes the hardware and software configurations. Finally, we define the evaluation metrics used to assess the system's performance.

4.1.1 Benchmark and Dataset

To assess the performance of our system, we utilize the **ViSudo-PC** benchmark. As described in Chapter 2, this benchmark evaluates a system's ability to verify the correctness of visually encoded Sudoku boards. The dataset comprises 11 splits: the first 10 are used for scoring, while the 11-th is reserved for experimentation. Each split can be configured to contain either 4×4 or 9×9 boards. Additionally, each split includes its own training (100 pairs), validation (100 pairs), and test (100 pairs) subsets. Each pair consists of two visually identical boards, except that the second board is deliberately corrupted to violate the Sudoku rules. Consequently, each split provides 200 instances: 100 labeled as correct and 100 as incorrect [22]. Moreover, each cell in every board is represented by a 28×28 grayscale image containing a handwritten digit (from MNIST), a handwritten letter representing a digit (from EMNIST), a Japanese character used as a digit (from KMNIST), or a fashion item standing in for a digit (from FMNIST). As discussed, this diverse visual representation increases the complexity of the task, requiring the system to simultaneously address both perception and reasoning.

Considering the architectural structure of our system, and in order to comply with the specifications of the ViSudo-PC benchmark and formalize our evaluation setup, we used the 11-th split for all out experiments. For this purpose, in each system iteration, we selected a random subset of the training sub-split to serve as reference inputs for the VLM. The sub-split was also used to train both the CNN block and the D-LTN. In addition, we employed the validation sub-split to monitor the system's generalization performance and guide model selection. Finally, the test sub-split was used to evaluate the overall performance of the system under unseen conditions.

4.1.2 Experimental Environment

The experiments were conducted using the computational resources provided by **Google Colaboratory**, which is a cloud-based Jupyter notebook environment offering access to Graphical Processing Units (**GPUs**) such as NVIDIA T4 and storage space for efficient model training and inference [28]. The implementation was developed in **Python 3.11.13** with the use of **PyTorch 2.6.0** for tensor operations and neural network computations due to its flexibility and support for GPU acceleration [29]. For the VLM, we utilized `llama-4-maverick-17b-128e-instruct`, which is a multi-modal language model that is composed of 17 billion activated parameters and supports a $128K$ token context window. This VLM has demonstrated robust performance in text and image reasoning tasks and aligns well with the system's requirements for scalable and responsive rule generation [30]. We accessed it via an API key provided by Groq, which is a high-throughput inference engine running on cloud [31].

In addition, during preliminary experiments, we identified the need to impose certain constraints on the prompt design. Initially, we allowed the VLM to generate Python-based groundings directly, as described in Section 3.2. However, we observed that without guidance, the VLM struggled to infer the correct FOL rules, even after multiple iterations. To address this, we restructured the prompt to present a predefined set of grounding alternatives, from which the VLM could choose. This modification improved rule accuracy and convergence. Consequently, the symbolic rules generated by the VLM were restricted to expressing only the relations between visual objects in FOL, with all groundings (except for variables) predefined in advance. We also instructed the VLM to generate only a single FOL rule per iteration, thereby reducing the complexity of the rule verification process and enhancing system stability. Accordingly, we used the following prompt:

```
system_role = '''
    You are a helpful assistant that can extract the First-Order
    Logic (FOL) rule from images.
    THE GRAMMAR OF FOL:
    - Constants: Not allowed in the rule.
    - Variables: Your options are `x1`, `x2`, ..., which
```

```
          represent visual objects.
        - Functions: Not allowed in the rule.
        - Predicates: Your options are `P_same_row`, `P_same_col`,
          `P_same_block`, `P_same_loc`, and `P_same_value`.
        - To compare variables, only use predicates.
        - The symbols used for logical AND, OR, and NOT are
          respectively `&`, `|`, and `!`.
        - The symbols used for implication and equivalence are
          respectively `implies` and `iff`.
        - The symbols used for universal and existential quantifiers
          are respectively `forall` and `exists`.
        - Use parentheses for preserving operation precedence.
        WHAT YOU MUST CONSIDER:
        - Use your own knowledge to analyze and deeply think about the
          images provided as your reference.
        - All the images must follow the same rule that you extract.
        - The rule applies to the visual objects within each image.
        - The visual objects may represent numbers rather than what
          they really are.
        - At the end of your chain of thought, put the extracted rule
          in the following template:
          EXTRACTED_RULE: "the rule you extracted"
    '''
    if len(history_list) > 0:
        n_extracted_rules = 0
        system_role += 'HISTORY OF PREVIOUS TRIALS:'
        for trial, incident in enumerate(history_list):
            error_message, extracted_fol_rule, ratio = incident
            system_role += (
                f'-  Trial {trial+1} -> '
            )
            if error_message != '':
                system_role += (
                    f'error: "{error_message}"'
                )
            else:
                n_extracted_rules += 1
                system_role += (
                    f'extracted rule: "{extracted_fol_rule}", '
                    f'conforming images: {100 * ratio:.2f}%'
                )
        if ratio < termination_threshold:
            system_role += 'IMPORTANT LESSON FROM HISTORY:'
            if n_extracted_rules == 0:
                system_role += (
```

```
                '-  Pay attention to the the instructions!'
            )
        else:
            system_role += (
                '-  The next FOL rule must be an improved version'
                ' of the above!'
            )

prompt = [{
    'type': 'text',
    'text': 'These are the reference images:'
}]
for base64_image in base64_image_list:
    prompt.append({
        'type': 'image_url',
        'image_url': {
            'url': f'data:image/png;base64,{base64_image}'
        }
    })

chat_completion = client.chat.completions.create(
    messages=[
        {'role': 'system', 'content': system_role},
        {'role': 'user', 'content': prompt}
    ]
)

response = chat_completion.choices[0].message.content
```

In the above prompt, while the VLM is asked to abstractly define the predicates in FOL, they are grounded in executable Python functions tailored to our benchmark. Each predicate captures a specific type of relationship between visual objects and is implemented using either exact or approximate similarity. Specifically, `P_same_value` checks whether two visual objects represent the same semantic concept (e.g., digit or letter) and uses an exponential similarity to tolerate slight variations in handwriting or visual style. `P_same_loc` verifies whether two objects occupy the same grid position and relies on a binary similarity, since location is a discrete and deterministic attribute. Similarly, `P_same_row`, `P_same_col`, and `P_same_block` evaluate whether two objects belong to the same row, column, or a block respectively. This design ensures that semantic relationships are modeled flexibly, while the VLM decides how to insert them into the FOL rule that it extracts.

Finally, for our CNN block, we employed a neural network composed of a configurable stack of **convolutional** layers, followed by a **fully connected** linear layer and a **dropout** layer. Each stage consists of a 2D convolutional layer with a kernel size of 3, stride of 1, and padding of 1 to preserve the spatial dimensions. This stack is followed by a **ReLU** activation to introduce non-linearity, a 2D **batch normalization** layer to stabilize training, and a 2D **max-pooling** operation with a kernel size and stride of 2 to downsample the feature maps by a factor of two. After passing through all these convolutional stages, the resulting feature maps are flattened and fed into the fully connected layer, which projects them into an embedding space. A dropout operation is then applied to the projected embeddings to mitigate overfitting.

To incorporate positional information, we adopted the concatenation-based fusion strategy described in Section 3.4. Specifically, the output vector from the convolutional stack was concatenated with the positional coordinates (row and column indices) of each visual object to produce the final embedding. This design enables the model to convert batches of symbol images and their spatial positions into rich, position-aware embeddings, where their creation is governed by three key hyperparameters: `cnn_hidden_dims`, which controls the number and size of convolutional layers; `embed_dim`, which sets the dimensionality of the output embeddings; and `drop_prob`, which determines the dropout rate applied after the linear projection.

4.1.3 Evaluation Metrics

Since the primary focus of this thesis is not on visual perception, we did not evaluate the CNN block in isolation. Instead, we adopted an end-to-end evaluation strategy, as outlined in Section 3.4, which means that rather than measuring the CNN block's accuracy independently, we assess its contribution indirectly by observing how it affects the performance of the complete system. Consequently, to account for varying levels of perceptual fidelity and reasoning capability, we conducted experiments under two distinct modes:

- **Perfect Perception**: In this mode, we assumed the CNN block operates flawlessly, generating ideal embeddings for each visual object. To simulate this, we replaced the CNN output with the ground-truth contextual representations of each object. This allowed us to isolate and evaluate the reasoning pipeline without the influence of visual recognition errors.

- **Realistic Perception**: In this mode, the CNN block was actively used to generate embeddings from raw visual inputs, which were then passed to downstream components without modification. This setup represents a more practical scenario and enables the evaluation of system performance under realistic levels of perceptual noise and uncertainty.

Accordingly, we evaluate the system from multiple perspectives using the following metrics:

- **VLM Load**: The number of images fed to the VLM. A lower value is preferred, as it indicates that fewer reference images are needed for better generalization.
- **Total Iterations**: The number of iterations required until a high-quality and simple rule is obtained. Fewer iterations reflect faster convergence and greater efficiency.
- **Area Under the Curve (AUC)**: This metric summarizes the system's performance across varying thresholds τ, providing a comprehensive measure of the trade-off between rule quality and acceptance criteria.

4.2 Findings

4.2.1 Results

Generated rules are as follows.

Rule 1:

```
forall x1, x2
    !P_same_loc(x1, x2) & (
        P_same_row(x1, x2)
        | P_same_col(x1, x2)
        | P_same_block(x1, x2)
    )
    implies !P_same_value(x1, x2)
```

Rule 2:

```
forall x1 forall x2
    (
        P_same_row(x1, x2)
        | P_same_col(x1, x2)
        | P_same_block(x1, x2)
    ) & !P_same_loc(x1, x2)
    implies !P_same_value(x1, x2)
```

Rule 3:

```
forall x1, x2
    !P_same_loc(x1, x2) & P_same_value(x1, x2)
    implies (
        !P_same_row(x1, x2)
```

```
        & !P_same_col(x1, x2)
        & !P_same_block(x1, x2)
    }
```

Dataset	VLM Load	Iterations	Extracted Rule	Test AUC
MNIST (4×4)	3	18	Rule 1	0.9974
EMNIST (4×4)	3	12	Rule 2	0.9012
KMNIST (4×4)	3	8	Rule 3	0.9517
FMNIST (4×4)	3	20	Error	-

Table 4.1: Clean and responsive table using ‘tabularx‘ and ‘booktabs‘

4.2.2 Discussions

4.3 Conclusion

4.4 Future Work

Bibliography

[1] Michael Hersche, Mustafa Zeqiri, Luca Benini, Abu Sebastian, and Abbas Rahimi. «A neuro-vector-symbolic architecture for solving Raven's progressive matrices». In: *Nature Machine Intelligence* 5.4 (2023), pp. 363–375 (cit. on pp. 1, 3, 8, 10, 11, 13).

[2] Daya Guo et al. «Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning». In: *arXiv preprint arXiv:2501.12948* (2025) (cit. on pp. 1, 16).

[3] Kyle Hamilton, Aparna Nayak, Bojan Božić, and Luca Longo. «Is neuro-symbolic AI meeting its promises in natural language processing? A structured review». In: *Semantic Web* 15.4 (2024), pp. 1265–1306 (cit. on pp. 1, 3–5, 8, 9, 15).

[4] *'Deduction' vs. 'Induction' vs. 'Abduction' — merriam-webster.com.* `https://www.merriam-webster.com/grammar/deduction-vs-induction-vs-abduction`. (Visited on 12/12/2024) (cit. on p. 1).

[5] Samy Badreddine, Artur d'Avila Garcez, Luciano Serafini, and Michael Spranger. «Logic tensor networks». In: *Artificial Intelligence* 303 (2022), p. 103649 (cit. on pp. 1–4, 9, 14, 15, 26).

[6] Adam Santoro, David Raposo, David G Barrett, Mateusz Malinowski, Razvan Pascanu, Peter Battaglia, and Timothy Lillicrap. «A simple neural network module for relational reasoning». In: *Advances in neural information processing systems* 30 (2017) (cit. on pp. 1, 11, 12).

[7] Chen Liang, Wenguan Wang, Tianfei Zhou, and Yi Yang. «Visual abductive reasoning». In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition.* 2022, pp. 15565–15575 (cit. on pp. 1, 2, 10).

[8] Lia Morra et al. «Designing Logic Tensor Networks for Visual Sudoku Puzzle Classification.» In: *NeSy.* 2023, pp. 223–232 (cit. on pp. 2–4, 10, 15).

[9] Zishen Wan et al. «Towards cognitive ai systems: a survey and prospective on neuro-symbolic ai». In: *arXiv preprint arXiv:2401.01040* (2024) (cit. on pp. 2, 5, 8, 9).

[10] Connor Pryor, Charles Dickens, Eriq Augustine, Alon Albalak, William Wang, and Lise Getoor. «Neupsl: Neural probabilistic soft logic». In: *arXiv preprint arXiv:2205.14268* (2022) (cit. on pp. 2, 11–13).

[11] Laura Von Rueden et al. «Informed machine learning–a taxonomy and survey of integrating prior knowledge into learning systems». In: *IEEE Transactions on Knowledge and Data Engineering* 35.1 (2021), pp. 614–633 (cit. on p. 2).

[12] Alessandro Daniele, Tommaso Campari, Sagar Malhotra, and Luciano Serafini. «Simple and Effective Transfer Learning for Neuro-Symbolic Integration». In: *International Conference on Neural-Symbolic Learning and Reasoning*. Springer. 2024, pp. 166–179 (cit. on pp. 3, 4, 12, 15).

[13] Alessandro Daniele, Tommaso Campari, Sagar Malhotra, and Luciano Serafini. «Deep symbolic learning: Discovering symbols and rules from perceptions». In: *arXiv preprint arXiv:2208.11561* (2022) (cit. on pp. 4, 12).

[14] Tianzhe Chu, Yuexiang Zhai, Jihan Yang, Shengbang Tong, Saining Xie, Dale Schuurmans, Quoc V Le, Sergey Levine, and Yi Ma. «Sft memorizes, rl generalizes: A comparative study of foundation model post-training». In: *arXiv preprint arXiv:2501.17161* (2025) (cit. on pp. 4, 15, 16).

[15] Xin Zhang and Victor S Sheng. «Neuro-Symbolic AI: Explainability, Challenges, and Future Trends». In: *arXiv preprint arXiv:2411.04383* (2024) (cit. on pp. 4, 5).

[16] Shima Imani, Liang Du, and Harsh Shrivastava. «Mathprompter: Mathematical reasoning using large language models». In: *arXiv preprint arXiv:2303.05398* (2023) (cit. on pp. 5, 16).

[17] Nicholas Watters, Daniel Zoran, Theophane Weber, Peter Battaglia, Razvan Pascanu, and Andrea Tacchetti. «Visual interaction networks: Learning a physics simulator from video». In: *Advances in neural information processing systems* 30 (2017) (cit. on p. 8).

[18] Giacomo Camposampiero, Michael Hersche, Aleksandar Terzić, Roger Wattenhofer, Abu Sebastian, and Abbas Rahimi. «Towards learning abductive reasoning using vsa distributed representations». In: *International Conference on Neural-Symbolic Learning and Reasoning*. Springer. 2024, pp. 370–385 (cit. on pp. 10, 11, 13, 14).

[19] Yeongbin Kim, Gautam Singh, Junyeong Park, Caglar Gulcehre, and Sungjin Ahn. «Imagine the unseen world: a benchmark for systematic generalization in visual world models». In: *Advances in Neural Information Processing Systems* 36 (2023), pp. 27880–27896 (cit. on pp. 10, 11, 16).

[20] Wenliang Zhao, Yongming Rao, Yansong Tang, Jie Zhou, and Jiwen Lu. «Videoabc: A real-world video dataset for abductive visual reasoning». In: *IEEE Transactions on Image Processing* 31 (2022), pp. 6048–6061 (cit. on pp. 10, 11).

[21] Michael Hersche, Francesco Di Stefano, Thomas Hofmann, Abu Sebastian, and Abbas Rahimi. «Probabilistic abduction for visual abstract reasoning via learning rules in vector-symbolic architectures». In: *arXiv preprint arXiv:2401.16024* (2024) (cit. on pp. 11, 13–16).

[22] Eriq Augustine, Connor Pryor, Charles Dickens, Jay Pujara, William Wang, and Lise Getoor. «Visual sudoku puzzle classification: A suite of collective neuro-symbolic tasks». In: *International Workshop on Neural-Symbolic Learning and Reasoning (NeSy)*. 2022 (cit. on pp. 11, 36).

[23] Yifei Peng, Yu Jin, Zhexu Luo, Yao-Xiang Ding, Wang-Zhou Dai, Zhong Ren, and Kun Zhou. «Generating by Understanding: Neural Visual Generation with Logical Symbol Groundings». In: *arXiv preprint arXiv:2310.17451* (2023) (cit. on pp. 13, 14).

[24] Mohamed Mejri, Chandramouli Amarnath, and Abhijit Chatterjee. «RESOLVE: Relational Reasoning with Symbolic and Object-Level Features Using Vector Symbolic Processing». In: *arXiv preprint arXiv:2411.08290* (2024) (cit. on pp. 13, 14).

[25] Paul Tarau. «On LLM-generated Logic Programs and their Inference Execution Methods». In: *arXiv preprint arXiv:2502.09209* (2025) (cit. on p. 16).

[26] OpenAI. *Images and vision: Learn how to understand or generate images*. publisher: OpenAI. URL: `https://platform.openai.com/docs/guides/images-vision?api-mode=responses` (visited on 06/09/2025) (cit. on p. 18).

[27] Erez Shinan. *Lark - a parsing toolkit for Python*. URL: `https://github.com/lark-parser/lark` (cit. on pp. 22, 23).

[28] Google Research. *Google Colaboratory: A Hosted Jupyter Notebook Service for Machine Learning and Data Science*. `https://colab.research.google.com`. Accessed: 2025-06-17. 2025 (cit. on p. 37).

[29] PyTorch Contributors. *PyTorch: An Open Source Machine Learning Framework*. `https://pytorch.org`. Version 2.5.1, Accessed: 2025-06-17. 2025 (cit. on p. 37).

[30] Meta AI. *Llama-4-Maverick-17B-128E-Instruct: A Multimodal Mixture-of-Experts Model*. `https://huggingface.co/meta-llama/Llama-4-Maverick-17B-128E-Instruct`. Accessed: 2025-06-17. 2025 (cit. on p. 37).

[31] Groq, Inc. *Groq: Fast Inference for Large Language Models.* https://groq.com. Accessed: 2025-06-17. 2025 (cit. on p. 37).