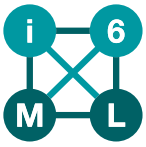


Chair of Machine
Learning and Reasoning
(Computer Science 6)

RWTH AACHEN
UNIVERSITY

This thesis was submitted to the Chair of Machine Learning and Reasoning.

Learning Description Logic Formulas via Gradient Descent

Master Thesis

Presented by

Jan Dornhege
434697

Supervised by Niklas Jansen, M.Sc.

1st Examiner Prof. Hector Geffner, Ph.D.

2nd Examiner Prof. Dr. rer. nat. Christopher Morris

Aachen, June 8, 2026

Abstract

Description logics are knowledge representation formalisms with applications in areas including generalized planning. This thesis studies the synthesis of description logic formulas from examples using both symbolic and neural methods.

First, it presents a general symbolic algorithm for constructing description logic formulas in the form of decision lists. It also introduces an encoding that links description logic to B-RASP. The main contribution is a novel neural architecture, Neural Description Logic Machines (NDLM), designed to operate on description-logic-structured data represented in terms of concepts and roles. NDLMs are trained in a supervised manner and can process inputs of arbitrary size. In addition, the expressivity of a class of NDLMs is shown to be exactly captured by a corresponding description logic.

Both the symbolic approach and the NDLM architecture are evaluated on the 1D-ARC benchmark, a one-dimensional simplification of the ARC challenge, where both solve a majority of the tasks. The thesis also formulates action model learning from potentially incomplete state-action transition sequences as a description logic synthesis problem. NDLMs successfully represent the action dynamics in several domains and generalize to larger instances than those seen during training. Further experiments on additional description logic synthesis tasks highlight both the capabilities and limitations of the approach.

Contents

1 Introduction

Learning logical classifiers or formulas from examples is a long-standing problem in artificial intelligence, particularly in the subfield of Inductive Logic Programming (ILP), and it also arises in the context of planning. In planning, data is often represented in terms of logical predicates, for example in STRIPS-style representations. Description logics (DLs) [1] provide a principled framework for representing knowledge in such formalisms and offer a favorable trade-off between expressive power and computational tractability. In particular, DLs have been successfully applied to learning generalized policies in planning domains [2]. However, the combinatorial explosion of possible formulas typically restricts symbolic approaches to formulas of relatively low complexity. Neural approaches, on the other hand, can learn more complex formulas but often lack interpretability and may require large amounts of data.

This thesis explores the problem of synthesizing description logic formulas from a small number of examples. First, a symbolic enumerative approach is presented that searches for DL-formulas within a fixed hypothesis space that are consistent with the provided examples. Due to the combinatorial growth of the hypothesis space, this approach is inherently limited to formulas of low complexity. To overcome this limitation, neural approaches are considered.

A construction is presented that translates arbitrary DL-formulas into programs in the Boolean-Restricted Access Sequence Processing Language (B-RASP) [20]. B-RASP has been shown to be expressively equivalent to masked hard-attention transformers. However, this translation appears impractical as the number of layers of a corresponding transformer would grow rapidly, which motivates a more direct neural approach.

The main contribution of this thesis is a neural architecture called Neural Description Logic Machines (NDLM) that separates data into concepts and roles and operates on these distinct tensors in a manner that mirrors the evaluation semantics of description logics. The resulting architecture is capable of representing expressive DL-formulas and can be trained end-to-end using supervised learning and gradient descent.

To empirically ground the architecture, the 1D-ARC dataset [19] is used as a benchmark. The 1D-ARC dataset is derived from the Abstraction and Reasoning Corpus (ARC) [4] and requires reasoning about one-dimensional arrays of colored cells. The proposed methods are evaluated on this dataset, demonstrating their ability to learn logical formulas from examples. Additionally, pathfinding in mazes and the problem of learning action models from observed state-action trajectories are considered.

This thesis is structured as follows: Chapter 2 introduces the relevant background concepts, including an overview of the B-RASP language, description logics, and SYNTH as an action model learning algorithm. Chapter 3 formally defines the learning task, and formulates different instances of the task. Chapter 4 reviews related work, including Neural Logic Machines and Graph Neural Networks. Chapter 5 presents the enumerative approach for learning DL-formulas, the B-RASP encoding, and finally the NDLM architecture, which constitutes the main contribution of this thesis. Chapter 6 evaluates these approaches empirically on several benchmarks. Finally, Chapter 7 summarizes the findings, discusses limitations, and outlines potential directions for future research.

2 Background

This chapter introduces relevant background knowledge used throughout this work. This includes Boolean-Restricted Access Sequence Processing Language (B-RASP) (Section 2.1), a programming language related to the expressivity of masked hard-attention transformers. Furthermore, relevant concepts from the field of description logics (Section 2.2) are presented. Section 2.3 give a brief introduction to Multilayer Perceptrons (MLPs) and Gradient Descent. Finally, classical planning and the task of action model learning is introduced in Section 2.4.

2.1 B-RASP

B-RASP is a programming language that is used as an intermediate language to translate masked hard-attention transformers into Linear Temporal Logic (LTL) and vice versa [20]. These formalisms are shown to have equivalent expressive power. In the following, the syntax and semantics of this programming language are introduced, and relevant results regarding its expressive power are presented. LTL and the formalization of masked hard-attention transformers are not introduced here, since they are not the focus of this work. Definitions can be found in [20].

A B-RASP program assumes a single input string $w \in \Sigma^*$, where Σ is a finite alphabet. From this string, initial vectors Q_σ are derived for each $\sigma \in \Sigma$, such that $Q_{\sigma,i} = \delta_{w_i,\sigma}$. Here, δ denotes the Kronecker delta (i.e. $\delta_{x,y} = 1$ iff $x = y$ and 0 otherwise).

The use of B-RASP in this work operates on a slight variation of this input format. The Q_σ vectors are not used; instead, it is assumed that the input is given as a finite set of Boolean vectors. This does not change expressivity, as the content of these initial vectors can be encoded into a finite alphabet Σ, after which the vectors can be derived from the resulting Q_σ vectors.

Given these initial vectors, a B-RASP program consists of a sequence of lines, where each line defines a new Boolean vector P_{k+1} as a function of previously defined vectors $P_1, \dots, P_k$.

The set of functions that can be used to define P_{k+1} is restricted to two types of operations: position-wise operations and attention operations.

Position-wise operations are Boolean functions of zero or more vectors $P_1, \dots, P_k$. For each position i, the value of the new vector is obtained by evaluating this Boolean function on the i-th entries of its input vectors, i.e.,

$$P_{k+1}[i] = f(P_1[i], \dots, P_k[i])$$

for some Boolean function f. Thus, these operations are applied entrywise and independently at each position i.

Attention operations allow for communication or interaction between positions. To achieve this, the operation consists of multiple functions that determine, for a given position i, the position j from which the value is taken and how the information at position j is used to determine the value at position i. Additionally, a default value is specified for the case that no such position j exists.

Syntactically, these operations are of the following two forms:

$$P_{k+1}[i] := \blacktriangleleft_j [M(i,j), S(i,j)] V(i,j) \,:\, D(i)$$

$$P_{k+1}[i] := \blacktriangleright_j [M(i,j), S(i,j)] V(i,j) \,:\, D(i)$$

where

- $M(i,j)$ is a binary mask function ($i < j$, $i > j$, or $\top$),
- the score predicate $S(i,j)$ and the value predicate $V(i,j)$ are Boolean combinations of $\{P_1[i], \ldots, P_k[i], P_1[j], \ldots, P_k[j]\}$,
- $D(i)$, the default value predicate, is a Boolean combination of $\{P_1[i], \ldots, P_k[i]\}$.

The semantics of these attention operations are as follows. For each position i, the first index j (for $\blacktriangleleft$) or the last index j (for $\blacktriangleright$) is selected such that both the masking predicate and the scoring predicate are satisfied. If such an index j exists, the i-th entry of P_{k+1} is obtained by evaluating $V(i,j)$; otherwise, it is given by the default value $D(i)$.

For language recognition tasks, the output of the program is determined by a designated output vector P_{out}, where the input string $w \in \{0,1\}^n$ is accepted iff $P_{out}[n] = 1$, i.e. the last entry of the output vector is 1.

The use of B-RASP in this work also deviates from this standard definition. The output is not a Boolean decision; instead, one or more full output vectors are considered as the output. The task usually involves learning a mapping from input to output vectors such that the output vectors are consistent with the provided examples.

The following two examples illustrate the use of B-RASP and its two available operations.

Example 1: This first simple example constructs a vector P_{shift} from an input vector P_{in} such that all entries are shifted one position to the right compared to P_{in}, and the first entry is always 0.

$$P_{shift}[i] := \blacktriangleright_j [j < i, 1] P_{in}[j] \,:\, 0$$

In this operation (for $i > 0$), j is evaluated to $i - 1$, as it is the largest position strictly smaller than i that satisfies the scoring predicate. The value is then taken from $P_{in}[j]$. If no such j exists (i.e. for $i = 0$), the default value 0 is used.

Example 2: Next, a B-RASP program is presented that constructs a Boolean vector $P_{exactly\_one}$ from an input vector P_{in} such that $P_{exactly\_one}$ is 1 everywhere iff there exists exactly one 1 in P_{in}.

$$
\begin{aligned}
P_l[i] &:= \blacktriangleright_j[j < i, P_{in}[j]]1 : 0 \\
P_r[i] &:= \blacktriangleleft_j[j > i, P_{in}[j]]1 : 0 \\
P_3[i] &:= \neg P_l[i] \wedge \neg P_r[i] \wedge P_{in}[i] \\
P_{exactly\_one}[i] &:= \blacktriangleleft_j[\top, P_3[j]]1 : 0
\end{aligned}
$$

P_l and P_r represent whether there exists an entry with value 1 to the left or right of position i, respectively. If neither exists and P_{in} is true, then P_3 is 1 at position i, indicating that there is exactly one 1 in the input vector. If this is successful, and there is in fact one position that is 1, this 1 is distributed to all positions, resulting in $P_{exactly\_one}$ being 1 everywhere. Otherwise, it evaluates to 0.

The main results presented in [20] regarding the expressive power of B-RASP state that, for a given language $L \subseteq \Sigma^*$, the following statements are equivalent:

- There exists a masked hard-attention transformer that recognizes L.
- There exists a B-RASP program that recognizes L.
- There exists an LTL formula that recognizes L.

Thus, B-RASP, LTL, and masked hard-attention transformers all have the same expressive power.

2.2 Description Logics

In this thesis, Description Logic (DL) is used as a formal language for representing structural properties of examples and for synthesizing formulas that characterize desired outputs. It provides a compact and semantically precise way to combine primitive concepts and roles into more expressive formulas.

DL[1] refers to a family of formal knowledge representation logics used to represent the knowledge of an application domain in a structured way. DLs represent knowledge in terms of concepts and roles, i.e. unary and binary predicates, respectively. Concepts represent properties

of individuals as sets of objects in the domain, while roles represent relationships between individuals as sets of pairs of objects.

Among other applications, especially in semantic web and knowledge representation tasks, description logics have also been used successfully to represent general policies in classical planning domains, where learned policies classify state transitions by observing quantitative changes in the evaluation of DL formulas [2].

In knowledge representation, a common distinction is between the ABox (assertional box) and the TBox (terminological box). The ABox contains ground facts describing what is known, whereas the TBox contains general rules about the domain [1]. This is not the perspective adopted in this work. Instead, the use of DL in this thesis is slightly reframed. The tasks considered here include an initial set of concepts and roles whose denotations are fully known; these are called *primitive* concepts and roles. The goal is then to synthesize one or more formulas in a given description logic from training examples. These examples specify the evaluation of the primitive concepts and roles on a finite set of individuals, together with the desired evaluation of the target formulas. The formulas are constructed from a fixed grammar that provides operators for combining concepts and roles with the given primitive symbols. This is formulated as a learning problem in Chapter 3.

While tasks such as satisfiability checking or subsumption checking are computationally tractable only for restricted fragments of DL, evaluating a DL formula on a finite interpretation is generally efficient (model checking).

The implementation of the symbolic approach relies on the DLplan library[1] [7], which offers tools for working with DL and planning problems, in particular those expressed as STRIPS states. It provides a function to exhaustively generate a pool of DL formulas for a set of STRIPS states such that no two formulas have the same evaluation on all of these states. The library also offers a wide range of DL operators, allowing for fairly expressive description logics.

Formally, a description logic formula is defined over a signature of primitive concept names N_C and role names N_R. The grammar of the formulas is defined by a set of operators that can be applied to these primitive symbols and to previously constructed formulas. An interpretation $I = (\Delta, \cdot^I)$ consists of a non-empty domain Δ and an interpretation function $\cdot^I$. The interpretation function maps concepts to subsets of Δ and roles to subsets of $\Delta \times \Delta$. Concepts in N_C are directly mapped into a subset of Δ, and roles in N_R are directly mapped into a subset of $\Delta \times \Delta$. The denotations of formulas consisting of logical operators are defined recursively based on the semantics of the operators used to construct them. The domain Δ is assumed to be finite in this work, and the evaluation of a formula F by an interpretation I is denoted as F^I.

[1] github.com/rleap-project/dlplan

Figures 2.1 and 2.2 summarize the semantics of the operators supported by the DLplan library and used throughout this thesis. All of these operators are available for formula construction, although some are disabled by default in the standard configuration. The experiments using the DLplan library follow these standard settings.

Figure 2.1: Semantics of concept operators in the `DLplan` library. Operators marked with * are disabled by default.

Concepts	Semantics	
$some(R, C)$	$some(R, C)^I$	$:= \{a \in \Delta \mid \exists b \in C^I : (a, b) \in R^I\}$
$all(R, C)$	$all(R, C)^I$	$:= \{a \in \Delta \mid \forall b \in \Delta : (a, b) \in R^I \implies b \in C^I\}$
$and(C_1, C_2)$	$and(C_1, C_2)^I$	$:= C_1^I \cap C_2^I$
bot_c	$\bot_c^I$	$:= \emptyset$
$*diff(C_1, C_2)$	$diff(C_1, C_2)^I$	$:= C_1^I \setminus C_2^I$
$eq(R_1, R_2)$	$eq(R_1, R_2)^I$	$:= \{a \in \Delta \mid \forall b \in \Delta : (a, b) \in R_1^I \iff (a, b) \in R_2^I\}$
$not(C)$	$not(C)^I$	$:= \Delta \setminus C^I$
$*or(C_1, C_2)$	$or(C_1, C_2)^I$	$:= C_1^I \cup C_2^I$
$*projection(R)$	$projection(R)^I$	$:= \{b \in \Delta \mid \exists a \in \Delta : (a, b) \in R^I\}$
$*subset(R_1, R_2)$	$subset(R_1, R_2)^I$	$:= \{a \in \Delta \mid \forall b \in \Delta : (a, b) \in R_1^I \implies (a, b) \in R_2^I\}$
top_c	top_c^I	$:= \Delta$

Figure 2.2: Semantics of role operators in the `DLplan` library. Operators marked with * are disabled by default.

Roles	Semantics	
$and(R_1, R_2)$	$and(R_1, R_2)^I$	$:= R_1^I \cap R_2^I$
$*compose(R_1, R_2)$	$compose(R_1, R_2)^I$	$:= \{(a, c) \in \Delta \times \Delta \mid \exists b \in \Delta : (a, b) \in R_1^I \wedge (b, c) \in R_2^I\}$
$*diff(R_1, R_2)$	$diff(R_1, R_2)^I$	$:= R_1^I \setminus R_2^I$
$id(C)$	$id(C)^I$	$:= \{(a, a) \mid a \in C^I\}$
$inverse(R)$	$inverse(R)^I$	$:= \{(b, a) \mid (a, b) \in R^I\}$
$*not(R)$	$not(R)^I$	$:= \Delta \times \Delta \setminus R^I$
$*or(R_1, R_2)$	$or(R_1, R_2)^I$	$:= R_1^I \cup R_2^I$
$restrict(R, C)$	$restrict(R, C)^I$	$:= \{(a, b) \in R^I \mid b \in C^I\}$
$*top_r$	top_r^I	$:= \Delta \times \Delta$
$transitive\_closure(R)$	$transitive\_closure(R)^I$	:= "transitive closure of R^I"

The logic induced by all of these operators except *transitive_closure* is referred to as $\mathcal{L}$. The extension obtained by additionally allowing *transitive_closure* is referred to as $\mathcal{L}^+$.

As a small example, consider the family interpretation shown in Figure 2.3. The domain is the set of all persons. The primitive concept names are $N_C = \{\mathbf{woman}, \mathbf{man}\}$. The primitive role names are $N_R = \{\mathbf{child}, \mathbf{married}\}$, where **child** relates a person to their children and **married** relates a person to their spouse. The interpretation of these primitive concepts and roles is given by the following sets:

$$\Delta = \{Lea, Finn, Sindy, Franz, Vivi, Karla\}.$$

The primitive concepts are interpreted as

$$\mathbf{woman}^I = \{Lea, Sindy, Vivi, Karla\} \quad \text{and} \quad \mathbf{man}^I = \{Finn, Franz\},$$

and the primitive roles are interpreted as

$$\begin{aligned}\mathbf{child}^I =& \{(Franz, Vivi), (Karla, Vivi),\\ & (Finn, Lea), (Finn, Sindy), (Vivi, Lea),\\ & (Vivi, Sindy)\},\\ \mathbf{married}^I =& \{(Vivi, Finn), (Finn, Karla), (Franz, Karla), (Karla, Franz)\}\end{aligned}$$

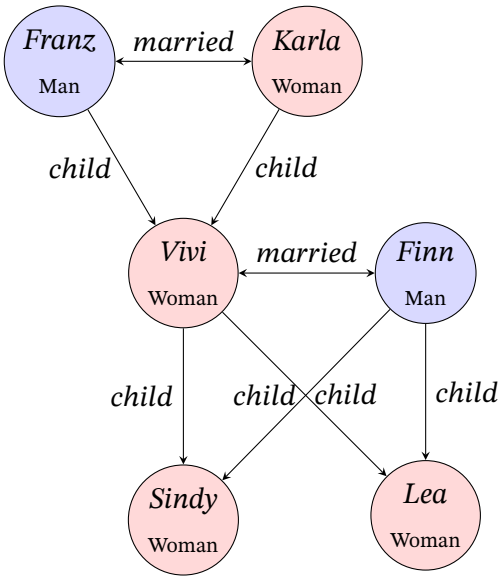


Figure 2.3: A small family.

The following examples illustrate formulas in $\mathcal{L}$ and $\mathcal{L}^+$ using the family interpretation from Figure 2.3.

- Role: Grandchild

$$\begin{aligned} Grandchild &:= compose(\mathbf{child}, \mathbf{child}) \\ &= child \circ child \\ Grandchild^I &= \{(Franz, Lea), (Karla, Lea), (Franz, Sindy), (Karla, Sindy)\}. \end{aligned}$$

- Concept: Persons that have a daughter (some child is a woman)

$$\begin{aligned} HasDaughter &:= some(\mathbf{child}, \mathbf{woman}) \\ &= \exists \mathbf{child}.\mathbf{woman} \\ HasDaughter^I &= \{Karla, Vivi\}. \end{aligned}$$

- Role: Mother (relates a person to its mother)

$$\begin{aligned} Mother &:= restrict(inverse(\mathbf{child}), \mathbf{woman}) \\ &= \mathbf{child}^{-1} \mid_{\mathbf{woman}} \\ Mother^I &= \{(Vivi, Karla), (Lea, Vivi), (Sindy, Vivi)\}. \end{aligned}$$

- Role: Sibling (a child of one's parents that is not oneself)

$$\begin{aligned} Sibling &:= diff(compose(\mathbf{child}, inverse(\mathbf{child})), id(top\_c)) \\ &= \left(\mathbf{child} \circ \mathbf{child}^{-1}\right) \setminus id(top\_c) \\ Sibling^I &= \{(Sindy, Lea), (Lea, Sindy)\}. \end{aligned}$$

- Role: Ancestor

$$\begin{aligned} Ancestor &:= transitive\_closure(inverse(\mathbf{child})) \\ &= \left(\mathbf{child}^{-1}\right)^+ \\ Ancestor^I &= \{(Lea, Finn), (Lea, Vivi), (Lea, Franz), (Lea, Karla), \\ &\quad (Sindy, Finn), (Sindy, Vivi), (Sindy, Franz), (Sindy, Karla), \\ &\quad (Finn, Franz), (Finn, Karla), \\ &\quad (Vivi, Franz), (Vivi, Karla)\}. \end{aligned}$$

- Concept: All persons with no children

$$\begin{aligned} Childless &:= not(some(\mathbf{child}, top_c)) \\ &= \neg \exists \mathbf{child}.\top_c \\ Childless^I &= \{Lea, Sindy\}. \end{aligned}$$

2.3 Multilayer Perceptrons and Gradient Descent

Multilayer perceptrons (MLPs) are a class of feedforward neural networks that consist of multiple layers of fully connected neurons. In each layer a linear transformation is applied to the input, followed by a non-linear activation function that is applied entrywise. The output of one layer serves as the input to the next layer, allowing for the modeling of complex relationships between inputs and outputs. MLPs are usually trained using gradient descent, where given some training data input and target outputs the gradient of some loss function is computed with respect to the parameters of the network. These parameters are updated by moving small steps in the direction of the negative gradient, which is expected to reduce the loss. This process is repeated iteratively until convergence, i.e., until the loss function reaches a minimum or a predefined number of iterations is reached.

This simple process is improved by introducing some modifications including a regularizer that penalizes large weights, or by using a momentum term that adds a fraction of the previous gradient to the current update. These modifications can help to speed up convergence and improve the generalization performance of the model. Experiments use the Adam optimizer [11], which is a popular variant of gradient descent that incorporates adaptive learning rates and momentum.

2.4 Planning and Action Model Learning

Classical planning is a branch of artificial intelligence concerned with finding a sequence of actions that transforms an initial state into a desired goal state. A common formalism for representing such problems is STRIPS [8]. A STRIPS planning task is a pair $P = (D, I)$, such that D describes information about the domain and I represents information about a concrete instance following the dynamics described in the domain. The domain D specifies a set of predicate symbols and a set of action schemas. An action schema $a(\bar{x})$ consists of a set of preconditions $\text{Pre}(a)$, an add list $\text{Add}(a)$, and a delete list $\text{Del}(a)$:

$$a(\bar{x}) = (\text{Pre}(a), \text{Add}(a), \text{Del}(a)).$$

For lifted actions, the preconditions, add list, and delete list are sets of predicates that contain variables from $\bar{x}$. Grounding the variables $\bar{x}$ with objects from the instance I yields a concrete action a, where the variables are replaced with specific objects from O.

The instance is represented as a tuple $I = (O, s_0, g)$, where O is a finite set of objects, s_0 is the initial state, and g is the goal condition. A state is represented as a set of grounded predicates over the objects O.

Let P denote the set of all grounded predicates. Then a state is given by $s \subseteq P$. Under the closed world assumption, every predicate not contained in s is considered false. The goal condition g is also represented as a set of grounded predicates and a state s is considered a goal state if $g \subseteq s$.

Applying a grounded action a to a state s produces a successor state $\gamma(s, a)$ defined as

$$\gamma(s, a) = (s \setminus \mathrm{Del}(a)) \cup \mathrm{Add}(a).$$

Thus, predicates in the delete list are removed from the state, and predicates in the add list are added. An action a is applicable in a state s if its preconditions are satisfied, i.e., $\mathrm{Pre}(a) \subseteq s$. The objective of a planner is to find a sequence of applicable actions that transforms the initial state s_0 into a state s_n such that $g \subseteq s_n$, i.e. s_n is a goal state.

One of the problems considered in this thesis is the task of learning action schemas from observed states and action traces. This is formulated as learning description logic formulas that capture the preconditions and effects of actions from a set of observed state-action trajectories. An additional challenge considered here is that actions may be only partially observed, such that some of their arguments are not visible. This follows the setting of the SYNTH [9] algorithm, in which state-action trajectories from a hidden *STRIPS+* domain are observed.

STRIPS+ is a variant of *STRIPS* in which some action arguments may be omitted. An argument can be omitted if it occurs only in the preconditions and, given the remaining arguments, replacing it with an existentially quantified variable does not change action applicability. Alternatively, an argument may also occur in an effect, but in that case the corresponding object must still be uniquely identifiable from the remaining arguments. These conditions ensure that removing such arguments does not change the behavior of the action, while still allowing the underlying *STRIPS+* domain to be learned from incomplete state-action traces.

As an example, consider a Blocksworld domain with predicates $on/2$, $clear/1$, $ontable/1$, $handempty/1$, and $holding/1$, and action schemas $pickup(x)$, $putdown(x)$, $stack(x, y)$, and $unstack(x, y)$. The blocksworld domain and an instance definition in PPDL syntax are provided in the Appendix (Chapter A).

In a *STRIPS+* formulation of this domain, some arguments may be omitted under the conditions described above. The *pickup* action cannot be simplified, because the argument x is needed to identify the object being picked up. The *putdown* action can be simplified to $putdown/0$, because the argument x is not needed to identify the object being put down: it is the only object currently being held. The *stack* action only requires the second argument, because the first argument can be uniquely identified as the object currently being held. The *unstack* action allows two different simplifications: Given the first argument, the second can be dropped as it can be uniquely identified to be the object underneath the first one, i.e., the

unique object y satisfying $on(x, y)$. Conversely, given the second argument, the first can be uniquely identified as the object on top of the second argument and can hence be dropped.

The SYNTH algorithm derives the preconditions and effects of actions of a hidden *STRIPS*+ domain from observed state-action trajectories. The algorithm learns the missing arguments via conjunctive queries and finds minimal preconditions matching the observed trajectories.

3 The Learning Task

The following formulates the learning task considered in this thesis formally and introduces the different problems that are investigated as instances of this general learning task.

Given are a set of primitive concept and role names N_c and N_r.

The hypothesis space is the set of all description logic formulas in $\mathcal{L}$ over primitive concepts and roles N_c and N_r. These are formulas constructed by the operators defined in Figures 2.1 and 2.2.

The learning algorithm is given a set of training examples of the form $E = \{(I_1, F^{I_1}), \dots, (I_m, F^{I_m})\}$ of finite description logic interpretations and their evaluations of some unknown formula $F \in \mathcal{L}$.

The task is to find some formula $F' \in \mathcal{L}$ that is consistent with the training examples, i.e. such that $F'^{I_i} = F^{I_i}$ for all $i \in [m]$.

Additionally the found formula F' should generalize to unseen interpretations, i.e. $F'^{I} = F^{I}$ for (all) finite interpretations I over N_c and N_r (or some test set). While this is desired, it generally can not be guaranteed in this setting.

In the following, different problems are formulated as instances of this general learning task.

3.1 1D-ARC

The 1D-ARC dataset [19] is a benchmark derived from ARC [4]. It consists of 18 tasks over one-dimensional arrays of colored cells, where each cell takes a value in $\{0, \dots, 9\}$. Each task provides three training examples and one test example. In total, each task contains 50 such input-output instances, each consisting of three training pairs and one test pair. The objective is to infer a transformation from input arrays to output arrays that is consistent with the training examples and generalizes to the test example.

This dataset is used as a benchmark for evaluating the approaches presented in this thesis. It is sufficiently diverse to cover a range of nontrivial transformation patterns, while remaining simple enough that many tasks can be represented by logical formulas of relatively low complexity. It therefore provides a suitable testbed for assessing the expressiveness and practical usefulness of the proposed methods.

Figure 3.1 shows the first example of the task *1d_move_1p*. In this task, the transformation shifts all colored cells one position to the right and sets the leftmost cell to black.

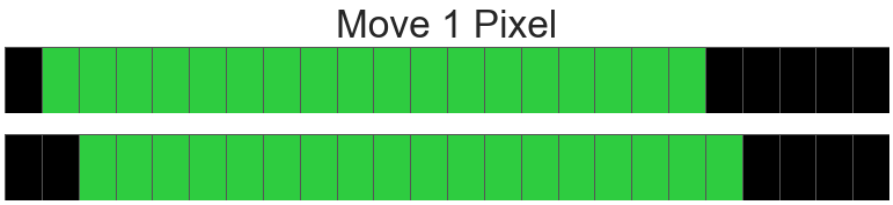


Figure 3.1: Input-output example from the 1D-ARC task *1d_move_1p*.

To formulate the problem as an instance of the learning task we make the assumption that the output color can be identified by some hidden description logic formula that is unknown.

The following presents a encoding, where each cell of the image and each color is represented as an object. The initial coloring and the neighborhood structure of the cells are represented by roles and a single target cell is marked as a concept. The output is then a single concept indicating the color of the targeted cell in the output image. This encoding is used both as input to the DLplan library and to the NDLM networks. Since the DLplan library assumes STRIPS states, concepts and roles are naturally mapped to unary and binary predicates. Let there be input-output image pair (Img, Img^o) with $Img = (Img_1, \dots, Img_n) \in [10]^n$ and $Img^o = (Img^o_1, \dots, Img^o_n) \in [10]^n$. We consider the problem of predicting the output value Img^o_t of pixel $t \in [n]$. Then the description logic interpretation $I = I_{Img,t} = \left(\Delta, \cdot^I\right)$ describing the image is defined as follows.

For each $i \in [n]$ an object $\mathbf{cell_i}$ and for each color $c \in [10]$ an object $\mathbf{color_c}$ is created in the domain. Let $\Delta = \{\mathbf{cell_i} \mid i \in [n]\} \cup \{\mathbf{color_c} \mid c \in [10]\}$ be the set of all these objects. Then the image is encoded into primitive concepts *cell*, *color*, $target$, and $color_i$ for $i \in [10]$ and roles *at* and *conn*, with the following denotations.

- $cell^I = \{\mathbf{cell_i} \mid i \in [n]\}$
- $color^I = \{\mathbf{color_c} \mid c \in [10]\}$
- $color^I_c = \{\mathbf{color_c}\}$ (for each color c)
- $target^I = \{\mathbf{cell_t}\}$ (indicates that cell t is a target cell that should be predicted)
- $at^I = \{(\mathbf{cell_i}, \mathbf{color_c}) \mid Img_i = c\}$
- $conn^I = \{(\mathbf{cell_i}, \mathbf{cell_{i+1}}) \mid i \in [n-1]\}$

Then, the learning task is defined by concept names $N_c = \{cell, color, target\} \cup \{color_i \mid i \in [10]\}$ and role names $N_r = \{at, conn\}$, the hypothesis space is the set of all description logic formulas over these primitive concepts and roles. For the three training image transformations $(Img_0, Img^o_0), (Img_1, Img^o_1), (Img_2, Img^o_2)$, the training examples are given by the interpretations $I_{Img_i,t}$ and the corresponding color of pixel t in the output image $Img^o_{i,t}$.

$$
\begin{aligned}
E = \{&\Big(I_{Img_0,i}, \{color_{Img^o_{0,i}}\}\Big) \mid i \in [|Img_0|]\} \\
\cup \{&\Big(I_{Img_1,i}, \{color_{Img^o_{1,i}}\}\Big) \mid i \in [|Img_1|]\} \\
\cup \{&\Big(I_{Img_2,i}, \{color_{Img^o_{2,i}}\}\Big) \mid i \in [|Img_2|]\}
\end{aligned}
$$

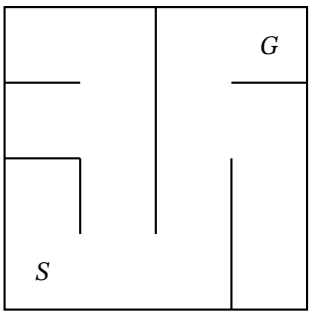
The experiments are conducted either on a reduced setting that uses only the first of the 50 train-test versions, or on the full dataset, which includes the three training examples from each of the 50 versions together with all 50 corresponding test examples.

3.2 Mazes

Spies et al. [17] study the ability of transformers to learn to solve mazes and investigate how the learned representations can be interpreted. They show that transformers can learn this task and that their internal representations can be interpreted as causal models.

The input is encoded as an adjacency list together with the start and goal positions. Each undirected edge in the adjacency list is represented as a token sequence, for example
$(0,0)$, $\leftrightarrow$, $(1,0)$, $;$,
corresponding to an edge between cell $(0,0)$ and cell $(1,0)$.

The following figure shows a simple example of such a 4×4 maze, where the agent starts in the bottom-left corner and the goal is in the top-right corner.



Spies et al. further train transformer architectures with different positional encoding schemes. Training is performed on datasets of 5×5 and 6×6 mazes, while evaluation is carried out on 7×7 mazes.

They analyze the trained models using linear probes and sparse autoencoders and find that the learned features encode maze connectivity rather than merely memorizing token sequences. By intervening on these features, they show that the model's behavior changes in a way that is consistent with the causal role of these features in decision-making. This suggests that the

model learns an internal representation that captures the underlying structure of the maze and supports reasoning about paths and connectivity, rather than simply memorizing input-output mappings. This interpretation is further supported by the model's ability to generalize to mazes larger than those seen during training.

The problem of finding a path in a maze can also be formulated as an instance of the general learning task:

The mazes for generating a training set are generated using a DFS algorithm, as used by Spies et al.[17] in their work. This creates connected acyclic mazes on $n \times n$ grids. To translate this into a description logic interpretation, each cell is treated as an object and a single primitive role encodes the adjacency between these cells. Additionally two primitive concepts encode the current position of the agent and the position of the goal.

Thus primitive concepts $N_c = \{pos, goal\}$ and a single primitive role $N_r = \{conn\}$ are used to encode the maze.

A $n \times n$ maze with connectivity relation $E \subseteq \{(c_1, c_2) \mid c_1, c_2 \in [n]^2\}$, start cell $s = (x_s, y_s) \in [n] \times [n]$, and goal cell $g = (x_g, y_g) \in [n] \times [n]$ is encoded by an interpretation $I = (\Delta, \cdot^I)$, where the domain $\Delta = \{\mathbf{cell_{i,j}} \mid i, j \in [n]\}$ consists of the n^2 cells.

Additionally, there are primitive concepts *pos* and *goal* and a primitive role *conn* with the following denotations.

- $pos^I = \{\mathbf{cell_{x_s,y_s}}\}$; indicates the current position of the agent.
- $goal^I = \{\mathbf{cell_{x_g,y_g}}\}$; indicates the position of the goal.
- $conn^I = \{(\mathbf{cell_{i,j}}, \mathbf{cell_{i',j'}}) \mid ((i,j),(i',j')) \in E\}$; encodes the connectivity between the cells, such that there is an edge between two cells if they are directly connected.

The training set is obtained by, for multiple mazes, considering a path from the start cell to the goal cell and treating each step of this path as a training example. The task is then to learn a formula that predicts the next position of the agent given the current position, the position of the goal, and the connectivity structure of the maze.

Results of the experiments conducted on the maze problems are presented in Section 6.4.

For general graph reasoning problems, with potentially initial node labels, the adjacency matrix of the graph can be used to directly encode the graph as a role. Initial and target node labels can be encoded as concepts. Section 6.3 make use of this encoding to investigate the logical limitations of the different network configurations on a set of selected small graph problems.

3.3 Action Model Learning

The goal of this task is to learn an action model from transition examples. The scope is limited to *STRIPS* and *STRIPS+* domains containing only nullary, unary, and binary predicates, but no higher-arity predicates. Each state can then be represented as a description logic interpretation, where objects in the state correspond to objects in the interpretation, unary predicates correspond to concepts, and binary predicates correspond to roles. Nullary predicates are also allowed and can be encoded as a concept that either contains all objects, if the predicate is true in the state, or no objects, if the predicate is false in the state.

Actions are also encoded in the input. The action arguments, either all or only a subset, are represented as concepts. For each argument, there is one concept that contains the object assigned to this argument.

The task consists of two problems, learning the preconditions and learning the effects, both of which are formulated as instances of the general learning task:

For each action schema one task is to learn the preconditions of the action. In the training set positive and negative examples are provided, where in positive examples the action is applicable in the state and in negative examples it is not. The training examples are given by the description logic interpretations encoding the state and the action arguments, and the target output is a single concept that is either satisfied by all objects in the state (if the action is applicable) or by no objects (if the action is not applicable).

Learning the effects is also formulated as instances of the learning task. Given the interpretations encoding the state and action arguments, the targets are obtained from the difference between the given state and the state observed after applying the action. For each action schema and each predicate two formulas are learned. The first one captures the *add* effects, containing all objects or pairs of objects that are added to the state by applying the action, while the second one captures the *delete* effects, containing all objects or pairs of objects that are removed from the state by applying the action.

The data generation for the experiments follows a DFS algorithm through the state space, starting from the initial state and expanding all applicable actions. These transitions are then grouped by the action schema to create distinct training sets for each action schema. The test set is generated by applying the same procedure in one or more different instances of the same domain.

Negative examples for the precondition learning are generated by initially generating a set of random ground actions and in each of the states observed in the DFS-sampling for applicable actions, evaluating which of these actions are not applicable. Then the action state pair is added as a negative example to the training set of the corresponding action-schema.

As an example, consider the *STRIPS* domain blocksworld from Section 2.4 with predicates $on/2$, $clear/1, ontable/1, handempty/0, holding/1$ and action schemas $pickup(x)$, $putdown(x)$, $stack(x, y)$ and $unstack(x, y)$.

Following, illustrates how states and actions are encoded as the description logic interpretations used to create the learning sets of the learning tasks.

Consider an instance with blocks a, b, c and the initial state where a is on the table, b is on a, c is on the table, and the hand is empty:
$s_0 = \{on(b, a), ontable(b), ontable(c), handempty, clear(a), clear(c)\}$. In all interpretations the domain is chosen to be $\Delta = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$.

Now only the action $pickup$ is considered. Assume we are just provided with a single positive examples $(s_0, pickup(c), s_1)$ and one negative example $(s_0, pickup(b), \emptyset)$, where s_1 is the state obtained by applying $pickup(c)$ in s_0.

Then, the learning task for the preconditions of $pickup$ as defined above is given by following learning set: $E = \{(I_0, \Delta), (I_1, \emptyset)\}$, where I_0 and I_1 are the interpretations encoding the state and action arguments of the positive and negative example, respectively.

The interpretation $I_0 = (\Delta, \cdot^I)$ captures s_0 and the action $pickup(c)$. It is defined by following assingments of the primitive concepts and roles:

- $clear^I = \{\mathbf{a}, \mathbf{c}\}$; indicates that blocks a and c are clear.
- $ontable^I = \{\mathbf{a}, \mathbf{c}\}$; indicates that blocks a and c are on the table.
- $handempty^I = \{\mathbf{a}, \mathbf{b}, \mathbf{c}\}$; indicates that the hand is empty.
- $holding^I = \emptyset$; indicates that no block is currently held.
- $on^I = \{(\mathbf{b}, \mathbf{a})\}$; indicates that block b is on block a.
- $arg_1^I = \{\mathbf{c}\}$; indicates that the argument of the action schema is assigned to block c.

For the effect network assume that only a single example $(s_0, pickup(c), s_1)$ is provided.

The learning sets for the *add* effects for each predicate are given by the following sets of examples:

$$
\begin{aligned}
E_{clear_{add}} &= \{(I_0, \emptyset)\}, \\
E_{ontable_{add}} &= \{(I_0, \emptyset)\}, \\
E_{holding_{add}} &= \{(I_0, \{\mathbf{c}\})\}, \\
E_{handempty_{add}} &= \{(I_0, \emptyset)\}, \\
E_{on_{add}} &= \{(I_0, \emptyset)\}
\end{aligned}
$$

And the learning sets for the *delete* effects for each predicate are given by the following sets of examples:

$$
\begin{aligned}
E_{clear_{delete}} &= \{(I_0, \varnothing)\}, \\
E_{ontable_{delete}} &= \{(I_0, \{\mathbf{c}\})\}, \\
E_{holding_{delete}} &= \{(I_0, \varnothing)\}, \\
E_{handempty_{delete}} &= \{(I_0, \{\mathbf{a}, \mathbf{b}, \mathbf{c}\})\}, \\
E_{on_{delete}} &= \{(I_0, \varnothing)\}
\end{aligned}
$$

Such learning sets are created for all action schemas and all predicates, leading to a total of $|A| \cdot (2 \cdot |P| + 1)$ learning tasks for a domain with action schemas A and predicates P.

Our method for solving the learning task has the capability of learning multiple concept and role formulas in parallel, given the same set of Interpretations, this is used in the experiments, such that only $2|A|$ models need to be trained, one for the preconditions and one grouping all formulas describing the effects of each action schema.

4 Related Work

Following presents a brief overview of related work particularly the Neural Logic Machines (Section 4.1) a work closely related to the methods developed in this thesis. Furthermore, Graph Neural Networks are presented in Section 4.2 as these are also tightly connected to this work.

4.1 Neural Logic Machines

Neural Logic Machines (NLMs) are a neurosymbolic architecture for learning from relational data in a way that resembles logical reasoning. Their input and hidden states are represented as tensors grouped by predicate arity. This preserves structural information and enables operations that respect relational structure: unary tensors encode properties of individual objects, binary tensors encode relations between pairs of objects, and higher-order tensors encode higher-arity relations.

Each NLM layer applies a small set of differentiable operations for relational reasoning. The expansion operation maps a k-ary tensor to a $(k+1)$-ary tensor by adding one dimension through repetition, allowing reasoning over an additional argument. The reduction operation maps a $(k+1)$-ary tensor back to a k-ary tensor and can be interpreted as aggregation or quantification over the added dimension (e.g., max pooling for existential-style behavior, min pooling for universal-style behavior). The permutation operation rearranges tensor dimensions to reason over different argument orders. After these structural operations, an MLP updates tensors of each arity to learn task-specific transformations.

A key property is that MLP weights are shared across all entries of tensors with the same arity, which makes NLMs object-permutation equivariant and applicable to varying numbers of objects.

Dong et al. [6] demonstrate the effectiveness of NLMs on tasks such as family-relation inference, shortest-path problems, and Blocksworld planning. Their results show stronger generalization to larger instances than several neural baselines. A central limitation, however, is scalability: the computational cost grows quickly with both predicate arity and the number of objects.

Zimmer et al. [21] propose an extension of NLMs that replaces the MLP component with logical modules, allowing explicit and interpretable logical formulas to be learned. This architecture performs comparably to the original NLM on the same tasks, while also providing explicit logical formulas that can be extracted from the learned model. Although scalability issues

during training persist, the resulting formulas are directly interpretable without additional post-processing.

4.2 Graph Neural Networks

GNNs are a class of neural networks designed to operate on graph-structured data. They learn to represent nodes, edges, and entire graphs by iteratively passing messages between neighboring nodes and updating node representations based on these messages. This allows GNNs to capture complex relationships and dependencies in graph data, making them suitable for tasks such as node classification, link prediction, and graph classification. While GNNs have proven successful in many applications, they are inherently limited in their ability to distinguish certain graph structures. This class of limitations is formalized by the Weisfeiler-Lehman (WL) test of graph isomorphism, which provides a theoretical framework for understanding the expressive power of GNNs[14]. While there are extensions of GNNs that can overcome some of these limitations, they often come with increased computational complexity.

There are many variants of GNNs that have been proposed to address specific challenges or to improve performance on certain tasks [10]. For example, Graph Convolutional Networks (GCNs)[12] apply convolutional operations to graph data, while Graph Attention Networks (GATs)[18] use attention mechanisms to weigh the importance of neighboring nodes. Each of these variants has its own strengths and weaknesses, and the choice of which to use depends on the specific characteristics of the problem being addressed.

According to Morris et. al.[14], a general definition of Graph Neural Networks is following: Initially, each node is embedded into some vector space, usually $\mathbb{R}^d$ for some $d \in \mathbb{N}$. Then, for each node v in the graph, the updated representation is computed by aggregating previous representations of its neighbors using some permutation-invariant and differentiable function. This is combined with the previous representation of the node by some other differentiable function. Formally functions $f_{agg}^{W_2}$ and $f_{comb}^{W_1}$, parameterized by learnable weights W_1 and W_2, are used to compute the new representation of node v into $\mathbb{R}^{1\times e}$ for some $e \in \mathbb{N}$ at layer t as follows:

$$f^{(t)}(v) = f_{comb}^{W_1}\Big(f^{(t-1)}(v), f_{agg}^{W_2}\big(\{f^{(t-1)}(u) : u \in \mathcal{N}(v)\}\big)\Big).$$

In a simple implementation of this general architecture, each neighbor's representation is multiplied with some learnable weight matrix and a sum aggregation function is applied. The current node's representation is also multiplied with a learnable weight matrix and added to the aggregated messages, followed by applying a non-linear activation function. Formally, for each node v in the graph, its representation at layer t is updated based on the representations of its neighbors at layer $t-1$ by the following update rule:

$$f^{(t)}(v) = \sigma\left(W_1^{(t)} \cdot f^{(t-1)}(v) + \sum_{u \in \mathcal{N}(v)} W_2^{(t)} \cdot f^{(t-1)}(u) \right).$$

where $f^{(t)}(v) \in \mathbb{R}^e$ is the representation of node v at layer t, $\mathcal{N}(v)$ is the set of neighbors of node v, $W_1^{(t)} \in \mathbb{R}^{e \times d}$ and $W_2^{(t)} \in \mathbb{R}^{e \times d}$ are learnable weight matrices, and σ is a non-linear activation function [14].

The initial embeddings of the nodes can capture node labels and some variants allow edge features to be incorporated. Thus, GNNs could in principle be applied to description logic reasoning tasks, operating on a complete graph with node and edge labels, where nodes correspond to individuals, which are labeled with concepts, and edge labels correspond to the role evaluations. However, the logical limitations of GNNs do not allow learning all of the relations expressible in the target description logic $\mathcal{L}$. In particular, GNNs only modify node representations, while edge representations remain fixed, disallowing the learning of role formulas. Even when only concept formulas in $\mathcal{L}$ are considered, these may contain roles as subformulas, which require reasoning about three nodes, which is not possible in a GNN style architecture. This follows from the relation between GNNs, the WL test, and its relation to C^2 logic, which only allows reasoning about pairs of nodes [3].

5 Methods

This chapter presents three approaches to learning description logic formulas from examples. The first approach is a symbolic enumerative method that constructs decision lists of DL-formulas consistent with the given examples. The second approach encodes DL-formulas in a neural architecture using B-RASP programs. The third approach is a novel differentiable architecture designed to learn DL-formulas directly from examples.

These approaches build on one another. The symbolic method provides a precise and interpretable baseline, but is limited by the combinatorial growth of the search space. The B-RASP encoding shows that DL-formulas can, in principle, be represented within a transformer-like architecture. Motivated by the limitations of the encoding into B-RASP, a dedicated differentiable architecture is then developed, whose expressivity is investigated.

Finally, the NDLM architecture is compared to Graph Neural Networks and Neural Logic Machines.

5.1 Symbolic: Enumerative DL Decision-List Learning

This section presents a symbolic algorithm for learning a decision list of description logic formulas. This algorithm is later used as a baseline to situate the performance of the NDLM approach on the 1D-ARC dataset. A decision list is an ordered sequence of DL-formulas, where each formula is evaluated successively until one formula applies to the current input. In 1D-ARC, the targets of the learning task are always a single object, i.e. the color of the target cell. Hence, the decision list is evaluated until a formula returns a non-empty set.

The method is an implementation of a greedy algorithm [5], originally studied in the context of learning referring expressions but also used for learning decision lists of logical formulas [13]. The formulas in the list are selected from a pool of candidate formulas, which are generated by the DLplan library introduced in Section 2.2. The algorithm iteratively selects the formula that correctly predicts the correct objects for the largest number of remaining unsolved training examples and appends it to the decision list. All training examples that are correctly covered by this formula are removed from the set of remaining examples, and the process is repeated until either all training examples are solved or no applicable formula remains.

The corresponding pseudocode is given in Algorithm 1. Line 5 calls the DLplan library to generate a pool of candidate formulas. The complexity parameter controls the maximum complexity of the generated formulas, which is defined as the number of operators in the

Algorithm 1 Greedy algorithm for finding a decision list of DL-formulas in Pseudocode.

```
1  Input: S=[(s_1, c_1), ..., (s_n,c_n)]
2  # s_i:strips states
3  # c_i: color to predict for target in s_i
4
5  F← DLPLAN_GENERATE_FEATURE_POOL(S,complexity)
6
7  feature_hierarchy=[]
8
9  while True:
10     best_feature ← None
11     max_solved ← []
12
13     for f in F:
14         applicable ← True
15         solved_states ← []
16         for s, c in S:
17             if len(f.evaluate(s))==1 and f.evaluate(s)[0]==c:
18                 solved_states.append(s)
19             elif len(f.evaluate(s))!=0:
20                 applicable ← False
21                 break
22         if applicable and len(solved_states)>len(max_solved):
23             best_feature ← f
24             max_solved ← solved_states
25     if best_feature == None or len(S) == 0:
26         break
27     S ← S \ max_solved
28     feature_hierarchy.append(best_feature)
29
30 feature_hierarchy, len(S)== 0
```

formula. The rest of the algorithm iterates until no applicable formula is found or all training examples are solved. The loop in line 13 iterates over all remaining candidate formulas to find the one that solves the most training examples. For each candidate formula, the inner loop in line 16 finds the training examples that are solved by this formula. Then, the best formula is selected as the one that solves the most training examples, and these examples are removed from the set of remaining examples (line 27). Finally, the selected formula is appended to the decision list (line 28).

The described algorithm is tailored to the 1D-ARC dataset, where the target is always a single object. The algorithm can be adapted to other settings by changing the way the applicability of

a formula is evaluated, such that potentially multiple objects can be predicted for each training example.

The experimental setup and the empirical evaluation of this approach are described in Section 6.1.

5.2 Neural 1: DL-formulas to B-RASP

This section explores how the equivalence between B-RASP and transformers can be utilized to obtain a transformer architecture that is able to represent certain DL-formulas. Towards this end, the first step is to encode DL-formulas in B-RASP. First, a given interpretation is encoded in the B-RASP format. Then, B-RASP programs can be constructed to represent the individual DL-operators, which can be combined to represent more complex DL-formulas.

The general idea is to encode concepts by boolean vectors of length n, where the i-th entry indicates whether the i-th object in the domain is an instance of the concept. Roles are encoded by their adjacency matrix, which is given as n boolean vectors of length n, where the a-th vector encodes the outgoing edges of the a-th object in the domain. DL-operators can then be encoded as B-RASP programs that manipulate these vectors and matrices to produce new vectors and matrices that represent the result of applying the operator.

The encoding assumes a fixed domain size $n \in \mathbb{N}$ and a fixed signature of concept names $N_C = \{C_1, \dots, C_k\}$ and role names $N_R = \{R_1, \dots, R_l\}$.

Let $I = \left(\Delta, \cdot^I\right)$ be a description logic interpretation, where Δ is a *finite* non-empty domain with n elements. First, fix an enumeration of the objects $\Delta = \{obj_1, \dots, obj_n\}$. Then, the input for the B-RASP program is constructed as follows.

For each concept C_i for $i \in [k]$ create a vector $P_{C_i} \in \{0,1\}^n$ such that

$$P_{C_i}[j] = \begin{cases} 1 & \text{if } obj_j \in C_i^I, \\ 0 & \text{otherwise.} \end{cases}$$

The roles can be represented by their adjacency matrix, which is given as n vectors of length n. For each R_j for $j \in [l]$ create n boolean vectors $P_{R_j,a} \in \{0,1\}^n$ for each $a \in [n]$ such that

$$P_{R_j,a}[b] = \begin{cases} 1 & \text{if } (obj_a, obj_b) \in R_j^I, \\ 0 & \text{otherwise.} \end{cases}$$

To capture the full expressivity of $\mathcal{L}$, the input additionally needs the identity matrix, which can be represented by n boolean vectors $P_{id,a} \in \{0,1\}^n$ for each $a \in [n]$ such that

$$P_{id,a}[b] = \begin{cases} 1 & \text{if } a = b, \\ 0 & \text{otherwise.} \end{cases}$$

The encoding of the roles is the main reason why the resulting transformer is restricted to a fixed number of objects, as the number of vectors needed to represent the roles grows linearly with the number of objects. The input contains in total $k + (l+1) \cdot n$ boolean vectors of length n. Tables 5.1 and 5.2 show how each of the description logic operators of the $\mathcal{L}$ logic can be expressed in B-RASP. (Refer back to Figures 2.1 and 2.2 for the semantics of these operators).

Table 5.1: Translation of description logic concept operators into B-RASP. Assume given concepts C, C1, C2 and roles R, R1, R2.
let $\top_n := [1, \ldots, 1] \in \{0,1\}^n$ and $e_a \in \{0,1\}^n$ with $e_{a,i} := \begin{cases} 1 \text{ if } i = a \\ 0 \text{ else} \end{cases}$ be the a-th unit vector.

Concept operator	B-RASP translation
$some(R, C)$	$P_a[i] := \blacktriangleright_j[\top_n, e_a[j]]P_C(j) : \top_n[i]$, for $a \in [n]$ $P[i] := \bigvee_{a=1}^{n} R_a[i] \wedge P_a[i]$
$all(R, C)$	$P_0[i] := \neg C[i]$ $P_1[i] := some(R, P_0)[i]$ $P[i] := \neg P_1[i]$
$and(C_1, C_2)$	$P[i] := P_{C_1}[i] \wedge P_{C_2}[i]$
bot_c	$P[i] := \bot[i]$
$diff(C_1, C_2)$	$P[i] := P_{C_1}[i] \wedge \neg P_{C_2}[i]$
$eq(R_1, R_2)$	$P[i] := \bigwedge_{a=1}^{n} \big(R_{1,a}[i] \iff R_{2,a}[i]\big)$
$not(C)$	$P[i] := \neg P_C[i]$
$or(C_1, C_2)$	$P[i] := P_{C_1}[i] \vee P_{C_2}[i]$
$primitive(C)$	$P[i] := P_C[i]$, where C is cell, color, target
$projection(R)$	$P[i] := \bigvee_{a=1}^{n} R_a[i]$
$subset(R_1, R_2)$	$P[i] := \bigwedge_{a=1}^{n} \big(R_{1,a}[i] \implies R_{2,a}[i]\big)$
top_c	$P[i] := \top[i]$

Some example explanations of certain table entries are given next.

For $some(R, C)$, the construction is split into two steps. First, for each target object a, the selector expression computes intermediate vectors P_a that evaluate whether a satisfies $C[j]$. Second, these object-wise results are combined with the role rows to obtain a single concept

Table 5.2: Translation of description logic role operators into B-RASP. Assume given concepts C, C1, C2 and roles R, R1, R2.
let $\top_n := [1, \dots, 1] \in \{0,1\}^n$ and $e_a \in \{0,1\}^n$ with $e_{a,i} := \begin{cases} 1 \text{ if } i = a \\ 0 \text{ else} \end{cases}$ be the a-th unit vector.

Role-operator	B-RASP translation
	For $a \in [n]$:
$and(R_1, R_2)$	$P_a[i] := P_{R_1,a}[i] \wedge P_{R_2,a}[i]$
$compose(R_1, R_2)$	$P_{a,b}[i] := \blacktriangleright_j[\top_n, e_a[j]]P_{R_1,b}[j] : \bot[i]$ $P_a[i] := \bigvee_{b=1}^{n} P_{a,b}[i] \wedge P_{R_2,b}[i]$
$diff(R_1, R_2)$	$P_a[i] := P_{R_1,a}[i] \wedge \neg P_{R_2,a}[i]$
$id(C)$	$P_a[i] := P_C[i] \wedge e_a[i]$
$inverse(R)$	$P_{a,b}[i] := \blacktriangleright_j[\top_n, e_a[j]]P_{R,b}[j] : \bot[i]$ $P_a[i] := \bigvee_{b=1}^{n} e_b[i] \wedge P_{b,a}[i]$
$not(R)$	$P_a[i] := \neg P_{R,a}[i]$
$or(R_1, R_2)$	$P_a[i] := P_{R_1,a}[i] \vee P_{R_2,a}[i]$
$restrict(R, C)$	$P_a[i] := P_{R,a}[i] \wedge P_C[i]$
$top(R)$	$P_a[i] := \top[i]$
$transitive\_closure(R, C)$	–

vector P, which evaluates for a source object i whether there is an edge in R such that the target a satisfies C. This is exactly the semantics of $some(R, C)$.

The row for $all(R, C)$ uses the standard duality with existential quantification:

$$all(R, C) \equiv \neg some(R, \neg C).$$

Hence, the translation first computes $P_0 := \neg C$, then applies $some(R, P_0)$, and finally negates again.

For $compose(R_1, R_2)$, the intermediate terms $P_{a,b}$ represent the edges (a, b) of R_1 in all entries. Then the second step combines these by evaluating whether there is some intermediate node b such that there is an edge from a to b in R_1 and an edge from b to i in R_2. This realizes the semantics of role composition.

The translation of $inverse(R)$ follows the same idea: it creates intermediate vectors $P_{a,b}$, indicating the entries of P_R, and then the final result is obtained by assigning the i-th entry of P_a the value 1 if there is some b such that $P_{b,a}[i]$ is 1, which means that there is an edge from b to a in R.

The operator $projection(R)$ maps a role back to a concept by taking, for each position, a disjunction over source objects a. Intuitively, an object is selected iff it participates in at least one R-edge.

One can then construct a B-RASP program for any DL-formula by recursively translating each operator in the formula to its corresponding B-RASP code using the constructions in Table 5.2.

However, this construction has a drawback. The resulting transformer is restricted to a fixed number of objects and role operations require number of objects many steps, as for each object a new vector needs to be constructed to represent the new adjacency matrix row. Moreover, the repeated application of identical operations across objects is unlikely to be learned efficiently and would require extensive training data, potentially including all object permutations and would using the construction from Yang et al.[20] lead to $O(n^2)$ layers for a single role composition operation.

Remark: Sarrof et al.[16] investigate a similar problem, where they deal with operating on a variable vocabulary size in planning problems. The alphabet is split into a fixed finite set, capturing the symbols constant in the domain, and a countable set used to identify objects. They then proceed by reformulating the transformer architecture and derive a symbolic programming language capturing the expressivity of the resulting architecture.

As these are quite recent results, their implications could not be fully investigated in the context of this thesis, but they suggest that it might be possible to obtain a transformer architecture that is able to represent DL-formulas without being restricted to a fixed number of objects.

To address this limitation, the repetitive operations and role representation can be captured explicitly. This motivates a different approach, the next section introduces a dedicated differentiable model designed to capture description-logic operations more directly.

5.3 Neural 2: Neural Description Logic Machines

This section introduces Neural Description Logic Machines (NDLM), a neural architecture that is in principle capable of representing arbitrary description logic formulas in $\mathcal{L}$. The sections first presents the architecture design, including input formulation, layer structure, and internal operations and the training procedure. Then the expressivity of the proposed architecture is analyzed in detail.

We again start with a description logic signature $N_C = \{C_1, \dots, C_c\}$ and $N_R = \{R_1, \dots, R_r\}$. Then, to define how an interpretation is encoded as the input, let $I = (\Delta, \cdot^I)$ be an interpretation, where Δ is a *finite* non-empty domain of n elements. Again, the objects in Δ are enumerated as $\Delta = \{\mathbf{obj_1}, \dots, \mathbf{obj_n}\}$. Note that the size of the domain is not fixed ad-hoc, but the architecture can handle varying domain sizes.

The main idea is to split the data into two parts: one to represent the concepts and one to represent the roles. For the concepts, a tensor $C \in \mathbb{R}^{c \times n}$ is created such that concept $C_i \in N_C$ for $i \in [c]$ is encoded in the i-th row of the matrix:

$$C_{i,j} = \begin{cases} 1 & \text{if } \mathbf{obj_j} \in C_i^I, \\ 0 & \text{otherwise.} \end{cases}$$

The roles are encoded in a tensor $R \in \mathbb{R}^{r \times n \times n}$. For $i \in [r]$, the i-th role $R_i \in N_R$ is represented by the i-th slice of the tensor, which is an $n \times n$ matrix encoding the adjacency matrix of the role:

$$R_{i,j,k} = \begin{cases} 1 & \text{if } (\mathbf{obj_j}, \mathbf{obj_k}) \in R_i^I, \\ 0 & \text{otherwise.} \end{cases}$$

In the following $enc(I) = (R, C)$ refers to this encoding for a description logic interpretation I. The entries of the tensors are in $\mathbb{R}$ instead of $\{0, 1\}$, as before, to allow the manipulations of the data to encode the description logic operations differentiably. This separation between concepts represented as vectors and roles represented as matrices is preserved throughout the architecture, and the output preserves it as well, such that the network is able to represent multiple concept and role formulas in parallel.

A *NDLM* network consists of a number of layers $k \in \mathbb{N}$, each transforming the concepts and roles from the previous layer into new concepts and roles. Each layer has its own set of learnable weight matrices and bias terms for transforming concepts and roles. The depth of the network is defined as the number of layers k. The output of the network is determined by applying each layer sequentially. Each layer predicts a fixed number of hidden concepts and roles. In the experiments, the number of hidden concepts and roles is equal and is referred to as the width of the network. The final layer produces some desired number of output concepts and roles depending on the specific task.

Formally, a *NDLM* model N is a function $N : \mathbb{R}^{c_{in} \times n} \times \mathbb{R}^{r_{out} \times n \times n} \to \mathbb{R}^{c_{out} \times n} \times \mathbb{R}^{r_{out} \times n \times n}$. It consists of multiple layers that are applied sequentially:

$N(C, R) = (Layer_k \circ Layer_{k-1} \circ \ldots \circ Layer_1)(C, R)$

The layers are parameterized by weight and bias matrices. For hidden dimensions c_{hidden} and r_{hidden}, each layer is parameterized by its own set of weight matrices and bias terms. The first layer is a function

$$Layer_{1;W_c,W_r,b_c,b_r} : \mathbb{R}^{c_{in} \times n} \times \mathbb{R}^{r_{in} \times n \times n} \to \mathbb{R}^{c_{hidden} \times n} \times \mathbb{R}^{r_{hidden} \times n \times n}$$

with weight and bias terms $W_c \in \mathbb{R}^{c_{in} \times c_{hidden}}$, $W_r \in \mathbb{R}^{r_{in} \times r_{hidden}}$, $b_c \in \mathbb{R}^{c_{hidden}}$ and $b_r \in \mathbb{R}^{r_{hidden}}$.

The layers $i = 2$ to $k - 1$ are functions

$$Layer_{i;W_c,W_r,b_c,b_r} : \mathbb{R}^{c_{hidden} \times n} \times \mathbb{R}^{r_{hidden} \times n \times n} \to \mathbb{R}^{c_{hidden} \times n} \times \mathbb{R}^{r_{hidden} \times n \times n}$$

with weight and bias terms $W_c \in \mathbb{R}^{c_{hidden} \times c_{hidden}}$, $W_r \in \mathbb{R}^{r_{hidden} \times r_{hidden}}$, $b_c \in \mathbb{R}^{c_{hidden}}$ and $b_r \in \mathbb{R}^{r_{hidden}}$.

The last layer is a function

$$Layer_{k;W_c,W_r,b_c,b_r} : \mathbb{R}^{c_{hidden} \times n} \times \mathbb{R}^{r_{hidden} \times n \times n} \to \mathbb{R}^{c_{out} \times n} \times \mathbb{R}^{r_{out} \times n \times n}$$

with weight and bias terms $W_c \in \mathbb{R}^{c_{hidden} \times c_{out}}$, $W_r \in \mathbb{R}^{r_{hidden} \times r_{out}}$, $b_c \in \mathbb{R}^{c_{out}}$ and $b_r \in \mathbb{R}^{r_{out}}$.

5.3.1 Structure of a Single Layer

Algorithm 2 Pseudocode of a single layer. $\circ$ indicates concatenation along the first axis.

```
Single Layer of the Network Layer_i(C, R; W_c, W_r, b_c, b_r):
Given: concepts: C ∈ ℝ^{r×n},
roles: R ∈ ℝ^{c×n×n},
weight matrices: W_c ∈ ℝ^{c,c'}, W_r ∈ ℝ^{r,r'}
bias terms: b_c ∈ ℝ^{c'}, b_r ∈ ℝ^{r'}

    C_1 ← ConceptRoleAttention(C, R)
    R_1 ← RoleRoleAttention(R)

    C_2 ← RoleAggregation(R)
    R_2 ← ConceptExtension(C)

    C_intermediate ← C ∘ C_1 ∘ C_2
    R_intermediate ← R ∘ R_1 ∘ R_2

    C_out ← ConceptFFN(C_intermediate, W_c, b_c)
    R_out ← RoleFFN(R_intermediate, W_r, b_r)

Output: C_out, R_out
```

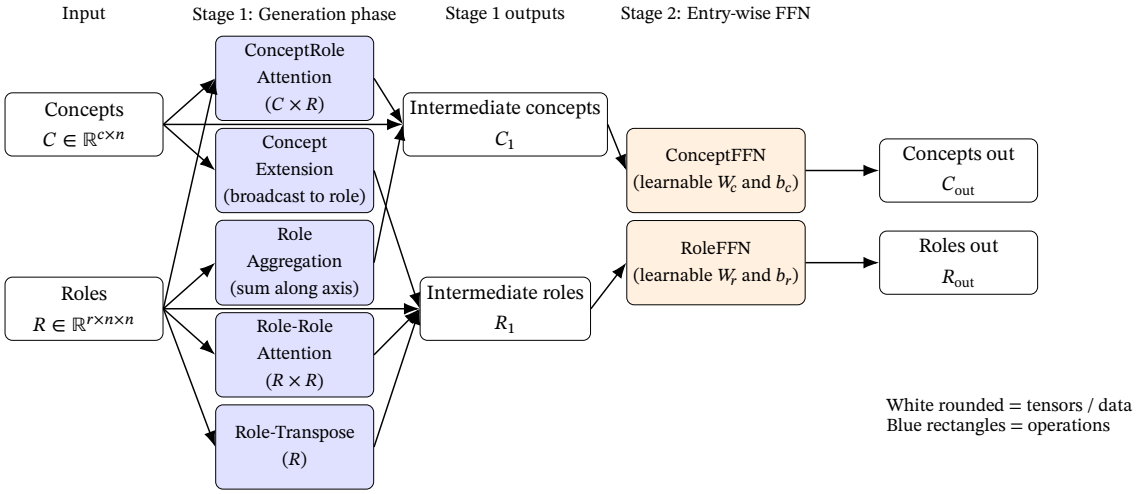


Figure 5.1: Information flow through a single layer

Each layer can be divided into two stages, which are visualized in Figure 5.1. The first combines concepts and roles to construct new ones. This uses a set of fixed and differentiable operations, such that the gradient can pass through them. In this stage, new concepts are derived by the operations *ConceptRoleAttention*, which combines concepts and roles to produce new concepts, and *RoleAggregation*, where a role is aggregated into a concept. Additionally, new roles are derived by the operations *RoleRoleAttention*, which computes an approximation of role composition, the *ConceptExtension* operation, which generates roles that indicate whether the target object of an edge is in a given concept, and finally *RoleTranspose*, which inverts/transposes of the current roles. All these operations are then concatenated with the original concepts and roles, such that the resulting collection of concepts and roles contains the original ones as well as the newly derived ones. The second stage applies a single layer of entrywise feedforward neural networks (*ConceptFFN* and *RoleFFN*) to both the concepts and the roles, mapping the collection of intermediate concepts and roles to a smaller number of concepts and roles. Both stages are discussed in detail in the following sections. See Listing 2 for pseudocode of a single layer and Figure 5.1 for a graphical illustration of the layer structure.

We consider two variants of *NDLM*, called relaxed mode ($NDLM^r$) and strict mode ($NDLM^s$). Both use the same input and output representation, layer structure, and entrywise feedforward networks. They differ in the fixed set of operations that combine concepts and roles in the first stage. These are the operations performed in lines 8-12 in Algorithm 2. In the following, superscripts ^{r} and ^{s} indicate the relaxed and strict variants of operations, respectively. In relaxed mode, standard arithmetic operations are used, including sums and matrix products. Strict mode more closely resembles the logical operations by utilizing min/max-based operations.

ConceptRoleAttention:

In the relaxed mode, this operation computes matrix-vector products for all pairs of roles and concepts. This can be interpreted as moving all objects in a concept along the edges defined by a role and then summing all incoming edges, weighted by the entries in the role matrices. This is similar to the message-passing operation in graph neural networks, with the distinction that the underlying graph on which the network operates is not static but learned, and the messages are weighted by real-valued edge weights.

Given c and r as the number of concepts and roles,

$$\begin{aligned} ConceptRoleAttention^r &: \mathbb{R}^{c\times n} \times \mathbb{R}^{r\times n\times n} \to \mathbb{R}^{cr\times n}, \\ (C,R) &\mapsto \left(R_i \cdot \left(C_j\right)^T\right)_{1\le ic+j\le c\cdot r} \end{aligned}$$

Here $\left(R_i \cdot \left(C_j\right)^T\right)$ is a matrix-vector product and $\left(C_j\right)^T$ is the transpose of vector containing the j-th concept. R_i is the $n \times n$ adjacency matrix of the i-th role.

In the strict mode, the matrix product is modified such that there is a min or max aggregation of the pairwise terms instead of a sum. This more closely corresponds to the quantification along roles used in description logics, such that max corresponds to existential quantification and min to universal quantification. The network computes both min and max operations as intermediate concepts. The successive application of the ConceptFFN can learn to ignore either of them if they are not useful for solving the task.

$$\begin{aligned} ConceptRoleAttention^s &: \mathbb{R}^{c\times n} \times \mathbb{R}^{r\times n\times n} \to \mathbb{R}^{2cr\times n}, \\ (C,R) &\mapsto \left(\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,p}\right)_{p\in[n]}\right)_{1\le(i-1)c+j\le c\cdot r} \circ \left(\left(\max_{k\in[n]} C_{j,k}\cdot R_{i,k,p}\right)_{p\in[n]}\right)_{1\le(i-1)c+j\le c\cdot r} \end{aligned}$$

Again C_j is the vector of length n containing the j-th concept. R_i is the $n \times n$ adjacency matrix of the i-th role. $\circ$ indicates a concatenation along the first dimension.

RoleRoleAttention:

In the relaxed mode, the matrix products for all pairs of roles are computed. In the strict mode, the sum is replaced by either a max or a min operation, such that the resulting edge weight from p to q corresponds to the maximum or minimum weight of any path from p to q via an intermediate node k. Both versions correspond to differentiable role concatenation operations on real-valued connectivities.

For the relaxed mode this is defined by

$$\begin{aligned} RoleRoleAttention^r &: \mathbb{R}^{r\times n\times n} \to \mathbb{R}^{r^2\times n\times n}, \\ R &\mapsto \left(R_i \cdot R_j\right)_{1\le ir+j\le r^2} \end{aligned}$$

and in the strict mode by

$$\begin{aligned} &RoleRoleAttention^s : \mathbb{R}^{r\times n\times n} \to \mathbb{R}^{2r^2\times n\times n}, \\ R \mapsto &\left(\left(\min_{k\in[n]} R_{i,p,k}\cdot R_{j,k,q}\right)_{p,q\in[n]}\right)_{1\le(i-1)r+j\le r^2} \circ \left(\left(\max_{k\in[n]} R_{i,p,k}\cdot R_{j,k,q}\right)_{p,q\in[n]}\right)_{1\le(i-1)r+j\le r^2}. \end{aligned}$$

RoleTranspose:

This is the same in both modes, all this operation does is to compute the transpose matrix for each of the roles. This corresponds to inverting the direction of all edges in the role. It is defined by

$$\begin{aligned} RoleTranspose &: \mathbb{R}^{r\times n\times n} \to \mathbb{R}^{r\times n\times n}, \\ R &\mapsto \left(R_i^T\right)_{i\in[r]}. \end{aligned}$$

RoleAggregation:

In this operation, a concept is derived from each role matrix by either summing along the second dimension in the relaxed mode or aggregating along the second dimension using min or max in the strict mode.

In the relaxed mode this counts the number of incoming edges weighted by their edge weights (entries in the role matrix). In the strict mode, two concepts are derived, identifying the minimum and maximum incoming edge weights.

RoleAggregation is defined in the relaxed mode as

$$\begin{aligned} RoleAggregation^r &: \mathbb{R}^{r\times n\times n} \to \mathbb{R}^{r\times n}, \\ R &\mapsto \left(\sum_{k=1}^{n} R_{i,k,j}\right)_{\substack{i\in[r]\\ j\in[n]}}. \end{aligned}$$

and in the strict mode, for each role two new concepts are generated, leading to the following definition of the operation:

$$RoleAggregation^s : \mathbb{R}^{r\times n\times n} \to \mathbb{R}^{2r\times n},$$
$$R \mapsto \left(\min_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\ j\in[n]}} \circ \left(\max_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\ j\in[n]}}.$$

ConceptExtension:

In this operation, concepts are turned into roles by repeating the vector of the concept n times. This corresponds to a role that has an edge (p, q) if the node p is contained in the concept. In a continuous formulation, the edge (p, q) is weighted by the membership weight of p in the concept. In both the relaxed and the strict mode, the operation is the same. It is defined by

$$ConceptExtension : \mathbb{R}^{c\times n} \to \mathbb{R}^{c\times n\times n},$$
$$C \mapsto E, \text{ such that } E_{i,p,q} = C_{i,p} \text{ for all } i \in [c], p, q \in [n]$$

This concludes all the operations of the first stage. All these operations contribute new concepts and roles to the input and all of them are collected in single tensors for intermediate concepts and intermediate roles. These are then passed forward to the second stage of the layer. This stage operates only entrywise and combines the information gathered in the first stage into a smaller number of concepts and roles, which are then passed to the next layer.

Entrywise FFN:

Second stage processes the intermediate concepts and roles by applying feedforward neural networks entrywise. For the concepts, a single feedforward neural network is applied to each object individually in parallel, such that the input is a vector containing the entries of that object in the intermediate concepts. For the roles, similarly, a single feedforward neural network is applied to each pair of objects individually in parallel, such that the input is a vector containing the entries of the pair of objects in the intermediate roles. The output of the feedforward neural network is then again a vector of potentially smaller length. The number of output concepts and roles is fixed as a hyperparameter. In principle, any feedforward neural network architecture can be used here, but in the experiments a simple architecture with just a single layer is used. The activation function σ used in this layer, is considered a hyperparameter as well. This stage is the same for both the relaxed and strict mode. The operation deriving the output concepts of a layer is called *ConceptFFN* and is defined by

$$\begin{aligned} ConceptFFN &: \mathbb{R}^{c \times n} \times \mathbb{R}^{c' \times c} \times \mathbb{R}^{c'} \to \mathbb{R}^{c' \times n}, \\ (C, W, b) &\mapsto \sigma\Big(\big(W \cdot C_{\_,j} + b\big)_{j \in [n]}\Big) \end{aligned}$$

Here, $C_{\_,j}$ is the column vector of the j-th column of C; this corresponds to the j-th entry of each of the c concepts in C. $C_{\_,j}$ is therefore the vector of length c storing the memberships of the j-th object in the concepts. The activation function σ is applied element-wise.

The output roles of the layer are constructed by the *RoleFFN*, which is defined by

$$\begin{aligned} RoleFFN &: \mathbb{R}^{r \times n \times n} \times \mathbb{R}^{r' \times r} \times \mathbb{R}^{r'} \to \mathbb{R}^{r' \times n \times n}, \\ (R, W, b) &\mapsto \sigma\left(\big(W \cdot R_{\_,i,j} + b\big)_{\substack{i \in [n] \\ j \in [n]}}\right) \end{aligned}$$

This is also applied entrywise, with r' being the number of output roles and $R_{\_,i,j}$ being the vector of length r storing the memberships of the pair of objects (i, j) in the roles.

This construction allows the network to compute simple operations like conjunctions and disjunctions of the intermediate concepts and roles, but also more complex operations like weighted sums of the intermediate concepts and roles. Still, the range of functions that can be computed by a single layer is relatively limited, which aims to support better generalization of the network.

Extensions: Transitive Closure

One possible way to extend the network is to add an additional operation in the first phase of each layer. This operation computes an approximation of the transitive closure for each of the roles in the input. The definition of transitive closure here depends on the mode the network is operating in. In the relaxed mode, the transitive closure of a given role is computed by

$$TC^r(R) = (R + I)^{n-1}.$$

For role-matrices with entries in $[0, 1]$ this corresponds to a continuous version of the transitive closure, where the weight of an edge (p, q) in the transitive closure corresponds to the sum of the weights of all paths from p to q, where the weight of a path is given by the product of the weights of the edges along the path.

Since this operation performs n matrix multiplications in sequence, it yields a time overhead of $O\big(n_r \cdot n^4\big)$ per layer, assuming that the layer has n_r input roles and n objects.

If the network operates in the strict mode, an entry for (p, q) in the transitive closure should correspond to the maximum path weight for any path from p to q, where a path weight is given by the minimum weight of any edge along the path. For an input of n objects, this can be computed iteratively as

$$TC^s(R) = \max_{i \in [n-1]} TC^s_i(R)$$

with $TC^s_0(R)_{i,j} = I_{i,j}$ and $TC^s_i(R) = \max_{p \in [n]}\{\min\left(TC^s_i(R)_{i,p}, R_{p,j}\right)\}$ for $i \in [k]$. TC^s_i encodes the reachability for paths of fixed length i.

This operation corresponds to a matrix product over a different algebraic structure (with min and max operations) and hence again has a time overhead of $O\left(n_r \cdot n^4\right)$ per layer, assuming that the layer has n_r input roles and n objects. The gain of expressivity from adding the transitive closure operator is investigated empirically in 6.3.

5.3.2 Training

The described approach is implemented in Python using the PyTorch library[15]. The model is optimized using stochastic gradient descent with the Adam optimizer[11]. The batch size and learning rate introduce additional hyperparameters. The loss function *BCEWithLogitsLoss* is used, consisting of a sigmoid application and a binary cross-entropy loss. This loss is computed for both the output roles and concepts and then combined.

In the experiments the training is performed for a fixed number of epochs on a training set and then tested on a dedicated test set. Since torch operates on tensors, to evaluate a batch in a single pass, all elements in the batch need to have the same number of objects. To make it possible to train on data that has a varying number of objects, the training data is split into groups of problems with equal size. Batches are then sampled only within the groups.

5.3.3 Expressivity of the Network

This section investigates the expressivity of the presented NDLM architecture variants. The first three theorems characterize lower bounds on the expressivity of the *NDLM* models. Then, an upper bound for the $NDLM^s$ models is provided. Finally, the number of layers of a model is related to the formula depth of the corresponding formulas.

The expressivity of $NDLM^s$ and $NDLM^r$, can be bounded below by $\mathcal{L}$ in the following sense:
Theorem 1. *The following statements hold:*

1. *For any concept formula in $F \in \mathcal{L}$, there is both a NDLM^s model N^s and a NDLM^r model N^r, with a single output concept, such that for any finite interpretation I:*

$$\begin{aligned} F^I =& \{o \in \Delta : N^s(enc(I))_{0,0,o} = 1\} \\ =& \Delta \backslash \{o \in \Delta : N^s(enc(I))_{0,0,o} = 0\}, \end{aligned}$$

$$\begin{aligned} F^I =& \{o \in \Delta : N^r(enc(I))_{0,0,o} = 1\} \\ =& \Delta \backslash \{o \in \Delta : N^r(enc(I))_{0,0,o} = 0\} \end{aligned}$$

2. *For any role formula in $F \in \mathcal{L}$, there is both a NDLM^s model N^s and a NDLM^r model N^r, with a single output role, such that for any finite interpretation I:*

$$\begin{aligned} F^I =& \{(o_1, o_2) \in \Delta^2 : N^s(enc(I))_{1,0,o_1,o_2} = 1\} \\ =& \Delta^2 \backslash \{(o_1, o_2) \in \Delta^2 : N^s(enc(I))_{1,0,o_1,o_2} = 0\}, \end{aligned}$$

$$\begin{aligned} F^I =& \{(o_1, o_2) \in \Delta^2 : N^r(enc(I))_{1,0,o_1,o_2} = 1\} \\ =& \Delta^2 \backslash \{(o_1, o_2) \in \Delta^2 : N^r(enc(I))_{1,0,o_1,o_2} = 0\} \end{aligned}$$

Furthermore, if the network is extended with the transitive closure operation, the expressivity of both $NDLM^{s+}$ and $NDLM^{r+}$ can be lower-bounded by $\mathcal{L}^+$:

Theorem 2. *The following statements hold:*

1. *For any concept-$\mathcal{L}^+$-formula F, there is both a NDLM^{s+} model N^{s+} and a NDLM^{r+} model N^{r+} with a single output concept, such that for any finite interpretation I:*

$$\begin{aligned} F^I =& \{o \in \Delta : N^{s+}(enc(I))_{0,o} = 1\} \\ =& \Delta \backslash \{o \in \Delta : N^{s+}(enc(I))_{0,0,o} = 0\}, \end{aligned}$$

$$\begin{aligned} F^I =& \{o \in \Delta : N^{r+}(enc(I))_{0,o} = 1\} \\ =& \Delta \backslash \{o \in \Delta : N^{r+}(enc(I))_{0,0,o} = 0\} \end{aligned}$$

2. *For any role-$\mathcal{L}^+$-formula F, there is both a NDLM^{s+} model N^{s+} and a NDLM^{r+} model N^{r+}, with a single output role, such that for any finite interpretation I:*

$$\begin{aligned} F^I =& \{(o_1, o_2) \in \Delta^2 : N^{s+}(enc(I))_{0,o_1,o_2} = 1\} \\ =& \Delta^2 \backslash \{(o_1, o_2) \in \Delta^2 : N^{s+}(enc(I))_{1,0,o_1,o_2} = 0\}, \end{aligned}$$

$$\begin{aligned} F^I =& \{(o_1, o_2) \in \Delta^2 : N^{r+}(enc(I))_{0,o_1,o_2} = 1\} \\ =& \Delta^2 \backslash \{(o_1, o_2) \in \Delta^2 : N^{r+}(enc(I))_{1,0,o_1,o_2} = 0\} \end{aligned}$$

Proof of Theorems 1 and 2. Every *NDLM* layer uses seven operations to combine concepts and roles to derive new ones. These are *ConceptRoleAttention*, *RoleRoleAttention*, *RoleTranspose*, *RoleAggregation*, *ConceptExtension*, *ConceptFFN*, *RoleFFN*, (*TransitiveClosure*).

All $\mathcal{L}$-operators (introduced in Figures 2.1 and 2.2) can be implemented quite directly using these operations. Table 5.3 shows the translations of $\mathcal{L}^+$-operators with network operators in the strict mode. The translations for the relaxed mode operations is provided in Table B.1.

By structural induction on the formula, an equivalent *NDLM* can be constructed extending the network with new layers to account for additional new logic operators. The number of hidden concepts and roles needs to be chosen appropriately large. □

The above theorems imply the following:
Corollary 2.1. *For any description logic interpretation $I = (\Delta, \cdot^I)$, the following statements hold:*

1. *For all $o_1, o_2 \in \Delta$:*
   There exists a concept-formula $F \in \mathcal{L}$, with

   $$o_1 \in F^I \wedge o_2 \notin F^I$$

   ⇒ There exists a NDLM^s model N^s with a single output concept, such that

   $$N^s(enc(I))_{0,0,o_1} \neq N^s(enc(I))_{0,0,o_2}.$$

2. *For all $(o_1, o_2), (o_3, o_4) \in \Delta^2$:*
   There exists a role-formula $F \in \mathcal{L}$ with

   $$(o_1, o_2) \in F^I \wedge (o_3, o_4) \notin F^I$$

   ⇒ There exists a NDLM^s model N^s with a single output role, such that

   $$N^s(enc(I))_{1,0,o_1,o_2} \neq N^s(enc(I))_{1,0,o_3,o_4}.$$

Proof. Given a description logic interpretation $I = (\Delta, \cdot^I)$.

Table 5.3: Translation of DL-operators to network operators.

Concept Operators	Implementation in the Network	Explanation
$all(R_1, C_1)$	$ConceptRoleAttention([C_1], [R_1])$	*ConceptRoleAttention* directly computes this, using *min* aggregation.
$some(R_1, C_1)$	$ConceptRoleAttention([C_1], [R_1])$	Using *max* aggregation *ConceptRoleAttention* computes the expected output.
$and(C_1, C_2)$	$ConceptFFN([C_1, C_2])$	*FFN* computes $C_1 \wedge C_2$
$diff(C_1, C_2)$	$ConceptFFN([C_1, C_2])$	*FFN* computes $C_1 \wedge \neg C_2$
$eq(R_1, R_2)$	$RoleAggregation(RoleFFN([R_1, R_2]))$	*FFN* computes $R_1 \leftrightarrow R_2$, *RoleAggregation* (min) aggregates this.
$not(C_1)$	$ConceptFFN([C_1])$	*FFN* computes $\neg C_1$
$or(C_1, C_2)$	$ConceptFFN([C_1, C_2])$	*FFN* computes $C_1 \vee C_2$
$projection(R)$	$RoleAggregation([R])$	*RoleAggregation* with *max* aggregation determines the values.
$subset(R_1, R_2)$	$RoleAggregation(RoleFFN([R_1, R_2]))$	*FFN* computes $R_1 \rightarrow R_2$, *RoleAggregation* with *min* aggregation determines the values.
top	$ConceptFFN([])$	*FFN* computes 1
bot	$ConceptFFN([])$	*FFN* computes 0
$primitive$	Encoded as Input	
Role Operators	**Implementation in the Network**	**Explanation**
$and(R_1, R_2)$	$RoleFFN([R_1, R_2])$	*FFN* computes $R_1 \wedge R_2$.
$compose(R_1, R_2)$	$RoleRoleAttention([R_1, R_2])$	*RoleRoleAttention* computes composition directly (*min* aggregation)
$diff(R_1, R_2)$	$RoleFFN([R_1, R_2])$	*FFN* computes $R_1 \wedge \neg R_2$.
id	Encoded as input role	The identity matrix is provided as one of the input roles.
$inverse(R_1)$	$RoleTranspose([R_1])$	*RoleTranspose* directly computes the adjacency matrix of the inverted role.
$not(R_1)$	$RoleFFN([R_1])$	*FFN* computes $\neg R_1$
$or(R_1, R_2)$	$RoleFFN([R_1, R_2])$	*FFN* computes $R_1 \vee R_2$.
$restrict(R_1, C_1)$	$RoleFFN([ConceptExtension([C_1], R_1)])$	*FFN* computes $R_{1,i,j} \wedge C_j$ where C_j is accessed by converting it to a role first.
top	$RoleFFN([])$	*FFN* predicts 1
bot	$RoleFFN([])$	*FFN* predicts 0
$primitive$	Encoded as Input	
$TC(R)$	$TC([R])$	*TC* evaluates to 0 if there is no connection between two nodes and 1 else.

1. Let $o_1, o_2 \in \Delta$ and let $F \in \mathcal{L}$ be a concept-formula. By Theorem 1 there exists a $NDLM^s$ model N^s, such that $1 = N^s(enc(I))_{0,0,o_1}$ and $0 = N^s(enc(I))_{0,0,o_2}$. Hence,

$$1 = N^s(enc(I))_{0,0,o_1} \neq N^s(enc(I))_{0,0,o_2} = 0.$$

2. Let $(o_1, o_2), (o_3, o_4) \in \Delta^2$ and let $F \in \mathcal{L}$ be a role-formula. By Theorem 1 there exists a $NDLM^s$ model N^s, such that $1 = N^s(enc(I))_{1,0,o_1,o_2}$ and $0 = N^s(enc(I))_{1,0,o_3,o_4}$. Hence,

$$1 = N^s(enc(I))_{1,0,o_1,o_2} \neq N^s(enc(I))_{1,0,o_3,o_4} = 0.$$

□

The theorems state, that for each concept or role formula in $\mathcal{L}$ there is a $NDLM^s$ and a $NDLM^r$ model that can distinguish exactly the same objects or object pairs as the formula, and analogously, for each concept or role formula in $\mathcal{L}^+$, there is a $NDLM^{s+}$ and a $NDLM^{r+}$ model that can distinguish exactly the same objects or object pairs as the formula. Hence, the expressivity of both $NDLM^s$ and $NDLM^r$ is at least as high as $\mathcal{L}$, and the expressivity of both $NDLM^{s+}$ and $NDLM^{r+}$ is at least as high as $\mathcal{L}^+$. The theorems do not imply that this lower bound is tight. For models operating in the relaxed mode, this is not the case.

$NDLM^r$ models can express counting existential quantifiers of the form $C(y) := \exists^{\geq k} x.q(x, y)$. For a binary predicate/role q this statement can be computed in the network using the *RoleAggregation* operator combined with the *ConceptFFN*, such that the
RoleAggregation sums the number of objects that are connected to a given y by a q. Then, the *FFN* assigns it the value 1 if this sum is greater than some k and 0 else. In the strict mode this construction is not possible since the *RoleAggregation* does not sum the number of incoming edges but rather computes the min and max of the incoming edge weights.

Counting quantifiers are not expressible in $\mathcal{L}$ (or $\mathcal{L}^+$). Hence, the expressivity of $NDLM^r$ is strictly higher than $\mathcal{L}$ and $\mathcal{L}^+$. However, the expressivity of $NDLM^s$ models is upper-bounded by $\mathcal{L}$ in the following sense:

Theorem 3. *For any description logic interpretation $I = (\Delta, \cdot^I)$, the following statements hold:*

1. *For all $o_1, o_2 \in \Delta$:*
   There exists a $NDLM^s$ model N^s with a single output concept, such that

$$N^s(enc(I))_{0,0,o_1} \neq N^s(enc(I))_{0,0,o_2}$$

   $\Rightarrow$ There exists a concept-formula $F \in \mathcal{L}$, with

$$o_1 \in F^I \wedge o_2 \notin F^I.$$

2. *For all $(o_1, o_2), (o_3, o_4) \in \Delta^2$:*

There exists a NDLM^s model N^s with a single output role, such that

$$N^s(enc(I))_{1,0,o_1,o_2} \neq N^s(enc(I))_{1,0,o_3,o_4}$$

$\Rightarrow$ *There exists a role-formula $F \in \mathcal{L}$ with*

$$(o_1, o_2) \in F^I \wedge (o_3, o_4) \notin F^I.$$

Proof. Assume a given description logic interpretation $I = \left(\Delta, \cdot^I\right)$ over a signature of primitive concept names N_C and primitive role names N_R and objects $o_1, o_2 \in \Delta$ with $o_1 \neq o_2$.

Let N be a *NDLM^s* model and assume that N consists of k layers $L_1, \ldots, L_k$, such that $N = L_k \circ L_{k-1} \circ \ldots \circ L_1$.

To observe which objects can potentially be distinguished at a certain point in the execution of N, a notion of equivalence between nodes and node pairs is introduced. The equivalence relation is refined through each layer, such that in each of the k layers this equivalence is an upper bound on which nodes or pairs of nodes can be distinguished from each other.

For a set A let $\mathcal{E}(A)$ be the set containing all equivalence relations over A. For each layer two equivalence relations are defined. $U_i \in \mathcal{E}(\Delta), i \in [k]$ captures the equivalence between objects (unary), and $B_i \in \mathcal{E}\left(\Delta^2\right), i \in [k]$ captures the equivalence between object pairs (binary). Notationally we refer to the equivalence class of an object in layer i as $[o]_i \in U_i$, and equivalence between two objects o, o' under the relation is denoted as $o \sim_i o' :\Leftrightarrow [o]_i = [o']_i$. Likewise, for object pairs, the equivalence class of a pair of objects (o_1, o_2) in layer i is denoted as $[(o_1, o_2)]_i \in B_i$, and the equivalence between two pairs of objects (o_1, o_2) and (o_3, o_4) is denoted as $(o_1, o_2) \sim_i (o_3, o_4) :\Leftrightarrow [(o_1, o_2)]_i = [(o_3, o_4)]_i$.

The initial equivalence relations are obtained by observing the primitive roles and concepts. If an object is contained in some concept while another is not, then these nodes can be distinguished. Likewise, for object pairs, if one object pair is included in a role while another is not, then they can be distinguished.

The initial unary equivalence relation is defined by for $o, o' \in \Delta$ by

$$o \sim_0 o' :\Leftrightarrow \forall C \in N_C : o \in C^I \leftrightarrow o' \in C^I$$

and object pair equivalence is defined for two object pairs $(x, x'), (y, y') \in \Delta^2$ as

$$(x, x') \sim_0 (y, y') :\Leftrightarrow \forall R \in N_R : (x, x') \in R^I \leftrightarrow (y, y') \in R^I.$$

We will now consider a fixed layer i. Assume the network has c hidden concepts and r hidden roles and has computed concepts $C_i \in \mathbb{R}^{c \times n}$ and roles $R_i \in \mathbb{R}^{r \times n \times n}$ in layer i, based on the initial input encoding of the interpretation I.

We will now observe how the equivalence relations are refined through the layer.

Formally we want to ensure: For $x, y \in \Delta$: $x \sim_i y \Rightarrow C_{i,x} = C_{i,y}$ and for $x, x', y, y' \in \Delta$: $(x, x') \sim_i (y, y') \Rightarrow R_{i,x,x'} = R_{i,y,y'}$.

To account for all internal operations of the $NDLM^s$ networks, refinement operators are introduced to capture how each operator can distinguish additional objects. Each layer uses the following internal operations: *ConceptRoleAttention*, *RoleRoleAttention*, *RoleTranspose*, *RoleAggregation*, *ConceptExtension*, *ConceptFFN*, *RoleFFN*. These are then combined to describe the layer as a whole.

However, the *ConceptFFN* and the *RoleFFN* do not contribute to equivalence refinement, as they operate entrywise and if presented with the same input produce the same output. The following defines the refinement operators for each of the remaining operations. These are then used to describe how each equivalence relation is transformed by the layer.

The refinement operators for the network operations are defined as follows:

- RoleAggregation

$$
\begin{aligned}
&Ref_{AGG} : \mathcal{E}(\Delta) \times \mathcal{E}(\Delta^2) \to \mathcal{E}(\Delta), \text{ such that} \\
&\qquad ([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{AGG}, \text{ with} \\
&\qquad [o]_{AGG} := \left\{ o' \in \Delta \mid \quad \{[(k,o)]_R \mid k \in \Delta\} = \{[(k,o')]_R \mid k \in \Delta\} \right\}
\end{aligned}
$$

- ConceptRoleAttention

$$
\begin{aligned}
&Ref_{CRA} : \mathcal{E}(\Delta) \times \mathcal{E}(\Delta^2) \to \mathcal{E}(\Delta), \text{ such that} \\
&\qquad [\cdot]_C, [\cdot]_R \mapsto [\cdot]_{CRA}, \text{ with} \\
&\qquad [o]_{CRA} := \left\{ o' \in \Delta \mid \quad \{([q]_C, [(q,o)]_R) \mid q \in \Delta\} = \{([q]_C, [(q,o')]_R) \mid q \in \Delta\} \right\}
\end{aligned}
$$

- ConceptExtension

$$
\begin{aligned}
&Ref_{EXT} : \mathcal{E}(\Delta) \times \mathcal{E}(\Delta^2) \to \mathcal{E}(\Delta^2), \text{ such that} \\
&\qquad ([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{EXT}, \text{ with} \\
&\qquad [(o_1, o_2)]_{EXT} := \left\{ (o_1', o_2') \in \Delta^2 \mid \quad [o_1]_C = [o_1']_C \right\}
\end{aligned}
$$

- RoleTranspose

$$
\begin{aligned}
&Ref_{TR} : \mathcal{E}(\Delta) \times \mathcal{E}(\Delta^2) \to \mathcal{E}(\Delta^2), \text{ such that} \\
&\quad ([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{TR}, \text{ with} \\
&\quad [(o_1, o_2)]_{TR} := \left\{ (o_1', o_2') \in \Delta^2 \mid \quad [(o_1, o_2)]_R = [(o_2', o_1')]_R \right\}
\end{aligned}
$$

- RoleRoleAttention

$$
\begin{aligned}
&Ref_{RRA} : \mathcal{E}(\Delta) \times \mathcal{E}(\Delta^2) \to \mathcal{E}(\Delta^2), \text{ such that} \\
&\quad ([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{RRA}, \text{ with} \\
&\quad [(o_1, o_2)]_{RRA} := \Big\{ (o_1, o_2) \in \Delta^2 \mid \\
&\qquad \{ \left([(o_1, q)]_R, [(q, o_2)]_R \right) \mid q \in \Delta \} = \{ \left([(o_1', q)]_R, [(q, o_2')]_R \right) \mid q \in \Delta \} \Big\}
\end{aligned}
$$

The claim is that these refinement operators capture the network operations correctly. For the intermediate tensors

$C_{AGG} := RoleAggregation(R_i)$,
$C_{CRA} := ConceptRoleAttention(C_i, R_i)$,
$R_{EXT} := ConceptExtension(C_i)$,
$R_{TR} := RoleTranspose(R_i)$ and
$R_{RRA} := RoleRoleAttention(R_i)$

and equivalences

$[\cdot]_{AGG} := Ref_{AGG}(U_i, B_i)$
$[\cdot]_{CRA} := Ref_{CRA}(U_i, B_i)$
$[\cdot]_{EXT} := Ref_{EXT}(U_i, B_i)$
$[\cdot]_{TR} := Ref_{TR}(U_i, B_i)$
$[\cdot]_{RRA} := Ref_{RRA}(U_i, B_i)$

following relations hold:

For all $x, y \in \Delta$:

$C_{AGG,x} \neq C_{AGG,y} \Rightarrow x \nsim_{AGG} y$
$C_{CRA,x} \neq C_{CRA,y} \Rightarrow x \nsim_{CRA} y$,

For all $(x, x'), (y, y') \in \Delta$:

$R_{EXT,x,x'} \neq R_{EXT,y,y'} \Rightarrow (x, x') \nsim_{EXT} (y, y')$
$R_{TR,x,x'} \neq R_{TR,y,y'} \Rightarrow (x, x') \nsim_{TR} (y, y')$
$R_{RRA,x,x'} \neq R_{RRA,y,y'} \Rightarrow (x, x') \nsim_{RRA} (y, y')$

Detailed derivations of these statements are provided in the Appendix in Section B.3.

Having established the connection between the refinement operators and the network operations, it remains to combine them. As the network operations in the *NDLM* architecture are concatenated, it is sufficient for any one of the operations to distinguish two entries. Then the equivalence relation of the next layer can be defined by intersecting the refined classes of the previous layer in the following way.

For each $x \in \Delta$ let $\quad [x]_{i+1} := [x]_i \cap [x]_{AGG} \cap [x]_{CRA}$,

and for all $x, y \in \Delta$ let $\quad [(x,y)]_{i+1} := [(x,y)]_i \cap [(x,y)]_{EXT} \cap [(x,y)]_{TR} \cap [(x,y)]_{RRA}$

With this definition, the equivalence relations capture the network's ability to distinguish objects and object pairs in each layer:

For all $x, y \in \Delta$:
$C_{i+1,x} \neq C_{i+1,y} \Rightarrow x \nsim_{i+1} y$

and for all $(x, x'), (y, y') \in \Delta$:
$R_{i+1,x,x'} \neq R_{i+1,y,y'} \Rightarrow (x, x') \nsim_{i+1} (y, y')$.

In order to establish a relation to the logic $\mathcal{L}$, we show inductively that for each equivalence class a formula can be constructed to recognize this class.

Each of the initial equivalence classes in U_0 and B_0 can be represented by a $\mathcal{L}$ concept or role formula by propositionally describing exactly the signature of evaluations of primitive concepts and roles on this object or object pair. A formula represents an equivalence class if it evaluates exactly to the set equal to the equivalence class under interpretation I.

The initial equivalence relations U_0 and B_0 are represented by the following formulas. For each $x \in \Delta$,

$$F^0_{[x]_0} := \bigwedge_{C \in N_C, x \in C^I} C \wedge \bigwedge_{C \in N_C, x \notin C^I} \neg C$$

And for each object pair $(x, y) \in \Delta^2$ the role formula is defined as

$$F^0_{[(x,y)]_0} := \bigwedge_{R \in N_R, (x,y) \in R^I} R \wedge \bigwedge_{R \in N_R, (x,y) \notin R^I} \neg R$$

This establishes the base case for the induction. Next assume that up to layer i each equivalence class in U_i and B_i can be represented by a formula.

For all objects $x \in \Delta$ let $F_{[x]_i} \in \mathcal{L}$ be the concept formula, such that $F^I_{[x]_i} = [x]_i$, and for all object pairs $(x, y) \in \Delta^2$ let $F_{[(x,y)]_i} \in \mathcal{L}$ be a role formula, such that $F^I_{[(x,y)]_i} = [(x, y)]_i$, by induction assumption.

Then the refinements of the refinement operators can be captured by the following formulas.

$$F_{[(x,y)]_{TR}} = \left(F_{[(x,y)]}\right)^{-1}$$

$$F_{[(x,y)]_{EXT}} = \left(\top|_{F_{[x]_C}}\right)^{-1}$$

where the $\left(\top|_{F_{[o_1]_C}}\right)^{-1}$ is a role satisfied by all elements of $[o_1]_C \times \Delta$.

The aggregation refinement R_{AGG} changes only the unary equivalences while keeping the binary ones unchanged. For each unary equivalence class $[(x)]_{AGG}$ for all $x \in \Delta$ there is a concept formula

$$F_{[x]_{AGG}} = \bigwedge_{y \in \Delta} \exists F_{[(x,y)]_R}.\top \wedge \neg\exists\left(\bigwedge_{y \in \Delta} \neg F_{[(x,y)]_R}\right).\top$$

The ConceptRoleAttention refinement R_{CRA} changes the unary equivalences, where the changes can be captured by following formulas. For any equivalence class $[(o)]_{CRA}$ for all $o \in \Delta$, the following formula represents it.

$$F^{CRA}_{[o]_{CRA}} = \bigwedge_{o' \in \Delta} \exists F_{[(o,o')]_R}.F_{[o']_C} \wedge \neg\exists\left(\bigwedge_{o' \in \Delta} \neg\left(F_{[(o,o')]_R} \wedge \top|_{F_{[o']_C}}\right)\right).\top$$

The RoleRoleAttention refinement R_{RRA} changes the binary equivalences, where the changes can be captured by following formulas. For any equivalence class $[(o_1, o_2)]_{RRA}$ for all $o_1, o_2 \in \Delta^2$, the following formula represents it.

$$F^{RRA}_{[(o_1,o_2)]_{RRA}} = \bigwedge_{o' \in \Delta} \left(F_{[(o_1,o')]_R} \circ F_{[(o',o_2)]_R}\right) \wedge \neg\left(\bigwedge_{o' \in \Delta} \neg F_{[(o_1,o')]_R}\right) \circ \left(\bigwedge_{o' \in \Delta} \neg F_{[(o',o_2)]_R}\right)$$

For each of the refinement operators, the claim is that the corresponding formulas correctly represent the output equivalence classes.

For all $x \in \Delta$:
$[x]_{AGG} = F^I_{[x]_{AGG}}$
$[x]_{CRA} = F^I_{[x]_{CRA}}$

and for all $(x, y) \in \Delta^2$:
$[(x,y)]_{EXT} = F^I_{[(x,y)]_{EXT}}$
$[(x,y)]_{TR} = F^I_{[(x,y)]_{TR}}$
$[(x,y)]_{RRA} = F^I_{[(x,y)]_{RRA}}$

Detailed derivations of the statements are provided in the Appendix in Section B.2.

Then, U_{i+1} and B_{i+1} can be represented by the following formulas:

For all $x \in \Delta$:

$$F_{[x]_{i+1}} := F_{[x]_{AGG}} \wedge F_{[x]_{CRA}} \wedge F_{[x]_i}$$

with $F^I_{[x]_{i+1}} = [x]_{i+1}$.

and for all $(x, y) \in \Delta^2$:

$$F_{[(x,y)]_{i+1}} := F_{[(x,y)]_{EXT}} \wedge F_{[(x,y)]_{TR}} \wedge F_{[(x,y)]_{RRA}} \wedge F_{[(x,y)]_i}$$

with $F^I_{[(x,y)]_{i+1}} = [(x,y)]_{i+1}$.

By induction this shows

1. For $x, y \in \Delta$, there is the formula $F^* := F_{[x]_k}$, such that
   $N(enc(I))_{0,0,x} \neq N(enc(I))_{0,0,y} \Rightarrow x \nsim_k y \Rightarrow x \in (F^*)^I \wedge y \notin (F^*)^I$.
2. For $(x, x'), (y, y') \in \Delta^2$, there is the formula $F^* := F_{[(x,x')]_k}$, such that
   $N(enc(I))_{1,0,x,x'} \neq N(enc(I))_{1,0,y,y'} \Rightarrow (x, x') \nsim_k (y, y') \Rightarrow (x, x') \in (F^*)^I \wedge (y, y') \notin (F^*)^I$.

□

Corollary 3.1. *For a given description logic interpretation $(\Delta, \cdot^I)$ and $o_1, o_2 \in \Delta$, the following statements are equivalent:*

1. *There exists a NDLM^s model N^s with a single output concept, such that*

$$N^s(enc(I))_{0,0,o_1} \neq N^s(enc(I))_{0,0,o_2}$$

2. *There exists a concept-formula $F \in \mathcal{L}$, with*

$$o_1 \in F^I \wedge o_2 \notin F^I$$

And for $(o_1, o_2), (o_3, o_4) \in \Delta^2$, the following statements are equivalent:

1. *There exists a NDLM^s model N^s with a single output role, such that*

$$N^s(enc(I))_{1,0,o_1,o_2} \neq N^s(enc(I))_{1,0,o_3,o_4}$$

2. *There exists a role-formula $F \in \mathcal{L}$ with*

$$(o_1, o_2) \in F^I \wedge (o_3, o_4) \notin F^I$$

Proof. follows directly from Corollary 2.1 and Theorem 3. □

The statement of Theorem 3 can be further generalized by showing that a formula can be constructed that not only distinguishes the two objects or object pairs, but also correctly captures the output of the network for all objects or object pairs and arbitrary finite interpretations:
Theorem 4. *The following statements hold:*

1. *For any NDLM^s model N, with a single output concept, there is a concept-formula $F \in \mathcal{L}$, such that for any finite interpretation I:*

$$\{o \in \Delta : N(enc(I))_{0,0,o} >= 0.5\} = F^I$$

2. *For each NDLM^s model N with a single output role, there is a role formula $F \in \mathcal{L}$, such that for any finite interpretation I:*

$$\{(o_1, o_2) \in \Delta^2 : N(enc(I))_{1,0,o_1,o_2} >= 0.5\} = F^I$$

(The threshold of 0.5 is chosen arbitrarily. The statement holds for any decision mapping the output of the network to a binary value.)

Proof. Let N be a *NDLM^s* model and assume that N consists of k layers $L_1, \ldots, L_k$, such that $N = L_k \circ L_{k-1} \circ \ldots \circ L_1$.

To show the statement we generalize the equivalence relation from the proof of Theorem 3 to arbitrary finite interpretations. It then suffices to show that the number of possible equivalence classes in the final layer over all interpretations is finite. Then, each of these classes can be represented by a formula as shown in the proof of Theorem 3. All objects in an equivalence class are evaluated to the same output value, such that by combining formulas of the equivalence classes that are evaluated to a value $>= 0.5$ by the network, a formula can be constructed that captures exactly the output decision of the network.

Let N_C be the set of primitive concept names and N_R be the set of primitive role names, both are finite. Now we consider interpretations over the signature of N_C and N_R. It suffices to only consider interpretations with a domain $\Delta = [n]$ for some $n \in \mathbb{N}$, since for each finite interpretation there is an isomorphic interpretation with domain $[n]$ for some $n \in \mathbb{N}$. Let $\mathcal{I}$ be the set of all interpretations over the signature of N_C and N_R with domain $[n]$ for some $n \in \mathbb{N}$.

$\mathcal{I}$ is countable, since for a fixed $n \in \mathbb{N}$ there are only finitely many interpretations over the signature of N_C and N_R with domain $[n]$, and there are countably many choices for n.

To now define how the network is able to distinguish objects or object pairs across all interpretations, we construct equivalence relations over all objects in all interpretations. To identify each object with its interpretation, associate each object with its domain. For an interpretation $I = \left(\Delta, \cdot^I\right) \in \mathcal{I}$ and an object $o \in \Delta$, we write $o_I = (I, o)$. Likewise, for an object pair $(o_1, o_2) \in \Delta^2$ we refer to the pair as $(o_1, o_2)_I = (I, (o_1, o_2))$.

Let $\Delta_\mathcal{I} := \{o_I | I = \left(\Delta, \cdot^I\right) \in \mathcal{I}, o \in \Delta\}$ be the set of all objects in all interpretations, and $\Delta^2_\mathcal{I} := \{(o_1, o_2)_I | I = \left(\Delta, \cdot^I\right) \in \mathcal{I}, o_1, o_2 \in \Delta\}$ be the set of all object pairs in all interpretations. Then, we again consider the set of all equivalence relations over $\Delta_\mathcal{I}$ and $\Delta^2_\mathcal{I}$, denoted as $\mathcal{E}(\Delta_\mathcal{I})$ and $\mathcal{E}\left(\Delta^2_\mathcal{I}\right)$, respectively.

With this we can again construct equivalence relations $U_i \in \mathcal{E}(\Delta_\mathcal{I})$ and $B_i \in \mathcal{E}\left(\Delta^2_\mathcal{I}\right)$ for each layer i. The initial equivalence relations are defined as follows: For $x_{I_1}, y_{I_2} \in \Delta_\mathcal{I}$

$$x_{I_1} \sim_0 y_{I_2} :\Leftrightarrow \forall C \in N_C : x \in C^{I_1} \leftrightarrow y \in C^{I_2}$$

and for $(x, x')_{I_1} \in \Delta^2_\mathcal{I}$ and $(y, y')_{I_2} \in \Delta^2_\mathcal{I}$:

$$(x, x')_{I_1} \sim_0 (y, y')_{I_2} :\Leftrightarrow \forall R \in N_R : (x, x') \in R^{I_1} \leftrightarrow (y, y') \in R^{I_2}$$

In these relations there are $2^{|N_C|}$ equivalence classes for objects and $2^{|N_R|}$ possible equivalence classes for object pairs, as each concept or role can either include or exclude an object or object pair.

We proceed by induction to show that for each layer $i \in [k]$, the number of equivalence classes in U_i and B_i is finite. The base case for $i = 0$ holds, as initially there are only finitely many equivalence classes, as shown above.

For the inductive step, we fix some $i \in [k]$ and assume that there are only finitely many equivalence classes in U_i and B_i. Then, we show that there are only finitely many equivalence classes in U_{i+1} and B_{i+1}.

Towards this, we generalize the intermediate equivalence relations defined by the refinement operators to $\Delta_\mathcal{I}$ and $\Delta^2_\mathcal{I}$. The domain of the refinement operators is then $\mathcal{E}(\Delta_\mathcal{I}) \times \mathcal{E}\left(\Delta^2_\mathcal{I}\right)$, and the codomain is $\mathcal{E}(\Delta_\mathcal{I})$ for Ref_{AGG} and Ref_{CRA}, and $\mathcal{E}\left(\Delta^2_\mathcal{I}\right)$ for Ref_{EXT}, Ref_{TR} and Ref_{RRA}. The definitions can be adapted in a straightforward way by evaluating each object or object pair according to its own interpretation:

- RoleAggregation

$$\widetilde{Ref}_{AGG} : \mathcal{E}(\Delta_\mathcal{I}) \times \mathcal{E}(\Delta_\mathcal{I}^2) \to \mathcal{E}(\Delta_\mathcal{I}), \text{ such that}$$
$$([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{AGG}, \text{ with}$$
$$\left[o_{I_1}\right]_{AGG} := \left\{o'_{I_2} \in \Delta_\mathcal{I} \mid \quad \{\left[(k,o)_{I_1}\right]_R \mid k_{I_1} \in \Delta_\mathcal{I}\} = \{\left[(k,o')_{I_2}\right]_R \mid k_{I_2} \in \Delta_\mathcal{I}\}\right\}$$

- ConceptRoleAttention

$$\widetilde{Ref}_{CRA} : \mathcal{E}(\Delta_\mathcal{I}) \times \mathcal{E}(\Delta_\mathcal{I}^2) \to \mathcal{E}(\Delta_\mathcal{I}), \text{ such that}$$
$$([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{CRA}, \text{ with}$$
$$\left[o_{I_1}\right]_{CRA} := \left\{o'_{I_2} \in \Delta_\mathcal{I} \mid \right.$$
$$\left. \{\left(\left[q_{I_1}\right]_C, \left[(q,o)_{I_1}\right]_R\right) \mid q_{I_1} \in \Delta_\mathcal{I}\} = \{\left(\left[q_{I_2}\right]_C, \left[(q,o')_{I_2}\right]_R\right) \mid q_{I_2} \in \Delta_\mathcal{I}\}\right\}$$

- ConceptExtension

$$\widetilde{Ref}_{EXT} : \mathcal{E}(\Delta_\mathcal{I}) \times \mathcal{E}(\Delta_\mathcal{I}^2) \to \mathcal{E}(\Delta_\mathcal{I}^2), \text{ such that}$$
$$([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{EXT}, \text{ with}$$
$$\left[(o_1,o_2)_{I_1}\right]_{EXT} := \left\{(o'_1,o'_2)_{I_2} \in \Delta_\mathcal{I}^2 \mid \quad \left[o_{1,I_1}\right]_C = \left[o'_{1,I_2}\right]_C\right\}$$

- RoleTranspose

$$\widetilde{Ref}_{TR} : \mathcal{E}(\Delta_\mathcal{I}) \times \mathcal{E}(\Delta_\mathcal{I}^2) \to \mathcal{E}(\Delta_\mathcal{I}^2), \text{ such that}$$
$$([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{TR}, \text{ with}$$
$$\left[(o_1,o_2)_{I_1}\right]_{TR} := \left\{(o'_1,o'_2)_{I_2} \in \Delta_\mathcal{I}^2 \mid \quad \left[(o_1,o_2)_{I_1}\right]_R = \left[(o'_2,o'_1)_{I_2}\right]_R\right\}$$

- RoleRoleAttention

$$\widetilde{Ref}_{RRA} : \mathcal{E}(\Delta_\mathcal{I}) \times \mathcal{E}(\Delta_\mathcal{I}^2) \to \mathcal{E}(\Delta_\mathcal{I}^2), \text{ such that}$$
$$([\cdot]_C, [\cdot]_R) \mapsto [\cdot]_{RRA}, \text{ with}$$
$$\left[(o_1,o_2)_{I_1}\right]_{RRA} := \left\{(o'_1,o'_2)_{I_2} \in \Delta_\mathcal{I}^2 \right.$$
$$\left. \{\left(\left[(o_1,q)_{I_1}\right]_R, \left[(q,o_2)_{I_1}\right]_R\right) \mid q \in \Delta_\mathcal{I}\} = \{\left(\left[(o'_1,q)_{I_2}\right]_R, \left[(q,o'_2)_{I_2}\right]_R\right) \mid q \in \Delta_\mathcal{I}\}\right\}$$

The following investigates how many equivalence classes can be generated by applying the refinement operators. Applying these refinement operators to U_i and B_i leads to intermediate equivalence relations

$[\cdot]_{AGG} := \widetilde{Ref}_{AGG}(U_i, B_i)$,
$[\cdot]_{CRA} := \widetilde{Ref}_{CRA}(U_i, B_i)$,

$[\cdot]_{EXT} := \widetilde{Ref}_{EXT}(U_i, B_i)$,
$[\cdot]_{TR} := \widetilde{Ref}_{TR}(U_i, B_i)$,
$[\cdot]_{RRA} := \widetilde{Ref}_{RRA}(U_i, B_i)$.

- **RoleAggregation:** Each resulting class $[o]_{AGG}$ is defined by the set of classes $\left[(k_{I_1}, o_{I_1})\right]_i$ for all $k_{I_1} \in \Delta_\mathcal{J}$. There are at most $|B_i|$ many classes $\left[(k_{I_1}, o_{I_1})\right]_i$, and thus at most $2^{|B_i|}$ many different sets of classes. This leads to at most $2^{|B_i|}$ many different classes $[o]_{AGG}$.
- **ConceptRoleAttention:** Each resulting class $[o]_{CRA}$ is defined by the set of pairs $\left(\left[q_{I_1}\right]_C, \left[(q_{I_1}, o_{I_1})\right]_R\right)$ for all $q_{I_1} \in \Delta_\mathcal{J}$. There are at most $|U_i|$ many classes $\left[q_{I_1}\right]_C$ and at most $|B_i|$ many classes $\left[(q_{I_1}, o_{I_1})\right]_R$, leading to at most $2^{|U_i|\cdot|B_i|}$ many different sets of pairs and thus at most $2^{|U_i|\cdot|B_i|}$ many different classes $[o]_{CRA}$.
- **ConceptExtension:** Each resulting class $\left[(o_{I_1}, o_{I_2})\right]_{EXT}$ is defined by the class $\left[o_{I_1}\right]_C$. There are at most $|U_i|$ many classes $\left[o_{I_1}\right]_C$, leading to at most $|U_i|$ many different classes $\left[(o_{I_1}, o_{I_2})\right]_{EXT}$.
- **RoleTranspose:** Each resulting class $\left[(o_{I_1}, o_{I_2})\right]_{TR}$ is defined by the class $\left[(o_{I_2}, o_{I_1})\right]_R$. There are at most $|B_i|$ many classes $\left[(o_{I_2}, o_{I_1})\right]_R$, leading to at most $|B_i|$ many different classes $\left[(o_{I_1}, o_{I_2})\right]_{TR}$.
- **RoleRoleAttention:** Each resulting class $\left[(o_{I_1}, o_{I_2})\right]_{RRA}$ is defined by the set of pairs $\left(\left[(o_{I_1}, q)\right]_R, \left[(q, o_{I_2})\right]_R\right)$ for all $q \in \Delta_\mathcal{J}$. There are at most $|B_i|$ many classes $\left[(o_{I_1}, q)\right]_R$ and at most $|B_i|$ many classes $\left[(q, o_{I_2})\right]_R$, leading to at most $2^{|B_i|\cdot|B_i|}$ many different sets of pairs and thus at most $2^{|B_i|\cdot|B_i|}$ many different classes $\left[(o_{I_1}, o_{I_2})\right]_{RRA}$.

We again combine the intermediate equivalence relations by intersecting the resulting classes, leading to $[x_I]_{i+1} := [x_I]_i \cap [x_I]_{AGG} \cap [x_I]_{CRA}$ for all $x_I \in \Delta_\mathcal{J}$,

and $\left[(x,y)_I\right]_{i+1} := \left[(x,y)_I\right]_i \cap \left[(x,y)_I\right]_{EXT} \cap \left[(x,y)_I\right]_{TR} \cap \left[(x,y)_I\right]_{RRA}$ for $(x,y)_I \in \Delta_\mathcal{J}^2$

Intersecting the relations leads to at most $|U_i| \cdot |[\cdot]_{AGG}| \cdot |[\cdot]_{CRA}| = |U_i| \cdot 2^{|B_i|} \cdot 2^{|U_i|\cdot|B_i|}$ many classes in U_{i+1} and at most $|B_i| \cdot |[\cdot]_{EXT}| \cdot |[\cdot]_{TR}| \cdot |[\cdot]_{RRA}| = |B_i| \cdot |U_i| \cdot |B_i| \cdot 2^{|B_i|\cdot|B_i|}$ many classes in B_{i+1}. As by induction assumption there are only finitely many classes in U_i and B_i, there are only finitely many classes in U_{i+1} and B_{i+1}.

By induction, it follows that for each layer $i \in [k]$, there are only finitely many equivalence classes in U_i and B_i and in particular in the final layer k.

Then, by calculations similar to those in Section B.2, one can show the following statements:

For all $x_{I_1}, y_{I_2} \in \Delta_\mathcal{J}$:

$$\left[x_{I_1}\right]_k = \left[y_{I_2}\right]_k \Rightarrow N(enc(I_1))_{0,0,x} = N(enc(I_2))_{0,0,y}$$

and for all $(x, x')_{I_1} \in \Delta_{\mathcal{J}}^2$ and $(y, y')_{I_2} \in \Delta_{\mathcal{J}}^2$:

$$\left[(x, x')_{I_1}\right]_k = \left[(y, y')_{I_2}\right]_k \Rightarrow N(enc(I_1))_{1,0,x,x'} = N(enc(I_2))_{1,0,y,y'}$$

Further, the same formulas as in the proof of Theorem 3 can be used to represent each of the new equivalence classes. The derivations are analogous to those in Section B.3, adapted to objects from different interpretations. For each equivalence class $\left[x_{I_1}\right]_k$, there is hence a corresponding concept formula $F_{\left[x_{I_1}\right]_k} \in \mathcal{L}$, such that for all $y_{I_2} \in \left[x_{I_1}\right]_k$ with $I_2 = \left(\Delta_{I_2}, \cdot^{I_2}\right)$:

$$\left[y_{I_2}\right]_k \cap \Delta_{I_2} = F^{I_2}_{\left[x_{I_1}\right]_k}.$$

For each equivalence class $\left[(x, x')_{I_1}\right]_k$, there is a corresponding role formula $F_{[(x,x')]_k} \in \mathcal{L}$, such that for all $(y, y')_{I_2} \in \left[(x, x')_{I_1}\right]_k$ with $I_2 = \left(\Delta_{I_2}, \cdot^{I_2}\right)$:

$$\left[(y, y')_{I_2}\right]_k \cap \Delta^2_{I_2} = F^{I_2}_{[(x,x')]_k}.$$

1. If N computes a single output concept. Then, let $\left[o_{1,I_1}\right]_k, \ldots, \left[o_{m,I_m}\right]_k \in U_k$ be the finite number of equivalence classes where each object is evaluated to a value >= 0.5 by the network (i.e. $N(enc(I_i))_{0,0,x}$ >= 0.5 for all $x \in \left[o_{i,I_i}\right]_k$ for all $i \in [m]$). For each of these, let $F_{\left[o_{i,I_i}\right]_k}$ be the formula representing the class.
   Constructing the formula
   $$F_N := \bigvee_{i \in [m]} F_{\left[o_{i,I_i}\right]_k},$$
   it follows that for any interpretation $I = \left(\Delta, \cdot^I\right) \in \mathcal{I}$ and any object $x \in \Delta_I$:
   $$\begin{aligned} & x \in F_N^I \Leftrightarrow \exists i \in [m] : x \in F^I_{\left[o_{i,I_i}\right]_k} \\ \Rightarrow & x \in F_N^I \Leftrightarrow \exists i \in [m] : x \in \left[o_{i,I_i}\right]_k \\ \Rightarrow & x \in F_N^I \Leftrightarrow N(enc(I))_{0,0,x} >= 0.5 \\ \Rightarrow & F_N^I = \{x \in \Delta : N(enc(I))_{0,0,x} >= 0.5\} \end{aligned}$$

2. If N computes a single output role. Then, let $\left[(o, o')_{1,I_1}\right]_k, \ldots, \left[(o, o')_{m,I_m}\right]_k \in B_k$ be the finite number of equivalence classes where each object pair is evaluated to a value >= 0.5 by the network (i.e. $N(enc(I_i))_{1,0,x,x'}$ >= 0.5 for all $(x, x') \in \left[(o, o')_{i,I_i}\right]_k$ for all $i \in [m]$). For each of these, let $F_{\left[(o,o')_{i,I_i}\right]_k}$ be the formula representing the class.
   Constructing the formula
   $$F_N := \bigvee_{i \in [m]} F_{\left[(o,o')_{i,I_i}\right]_k},$$

it follows that for any interpretation $I = \left(\Delta, \cdot^I\right) \in \mathcal{I}$ and any object pair $(x, x') \in \Delta_I^2$:

$$
\begin{aligned}
&(x, x') \in F_N^I \Leftrightarrow \exists i \in [m] : (x, x') \in F^I_{\left[(o,o')_{i,I_i}\right]_k} \\
\Rightarrow &(x, x') \in F_N^I \Leftrightarrow \exists i \in [m] : (x, x') \in \left[(o, o')_{i,I_i}\right]_k \\
\Rightarrow &(x, x') \in F_N^I \Leftrightarrow N(enc(I))_{1,0,x,x'} >= 0.5 \\
\Rightarrow &F_N^I = \{(x, x') \in \Delta^2 : N(enc(I))_{1,0,x,x'} >= 0.5\}
\end{aligned}
$$

□

A further observation is that the complexity of representable formulas can be related to the number of layers.

Theorem 5. *For every $\mathcal{L}$ formula, there is a NDLM^s network satisfying the conditions of Theorem 1 with at most twice the maximum operator depth many layers.*

Proof. Each operator can be implemented in at most two layers. Since all subformulas can be computed in parallel in earlier layers, it follows by induction on the formula depth that the network needs at most twice the maximum operator depth many layers. □

Section 6.3 empirically investigates the logical limitations of the different network configurations on a collection of small graph problems.

5.4 Relations to other Learning Architectures

This section compares the NDLM architecture to the Graph Neural Network (GNN) (introduced in Section 4.2) and the Neural Logic Machine (NLM) (introduced in Section 4.1) architectures. Both are structurally similar to the NDLM architecture.

5.4.1 Relation to Graph Neural Networks

The basic formulation of Graph Neural Networks using sum aggregation can be implemented by the *NDLM^r* architecture by using a single fixed role to represent the edges of the graph and using the *ConceptRoleAttention* to aggregate information from neighboring nodes. The *ConceptFFN* can be used to update the node representations and apply the weight matrices accordingly.

To show this for a single layer t of a GNN, let $G = (V, E)$ be a graph and $f^{(t-1)}(v) \in \mathbb{R}^d$ be the node features at layer $t-1$. Let the node features at layer t be computed as follows:

$$f^{(t)}(v) = \sigma\left(W_1^{(t)} \cdot f^{(t-1)}(v) + \sum_{u \in \mathcal{N}(v)} W_2^{(t)} \cdot f^{(t-1)}(u)\right)$$

where $f^{(t)}(v) \in \mathbb{R}^e$ is the representation of node v at layer t, $\mathcal{N}(v)$ is the set of neighbors of node v, $W_1^{(t)} \in \mathbb{R}^{e \times d}$ and $W_2^{(t)} \in \mathbb{R}^{e \times d}$ are some weight matrices, and σ is a (non-linear) activation function.

We now construct one layer of a $NDLM^r$ network to compute the same node representations. Assume that the input for the $NDLM^r$ layer consists of a single role $R \in \mathbb{R}^{1 \times n \times n}$ encoding the edges of the graph and d concepts $C \in \mathbb{R}^{d \times n}$, such that $C_{i,v} = f^{(t-1)}(v)_i$. The *ConceptRoleAttention* computes the sum aggregation of neighboring node representations as $C'_{i,v} = \sum_{u \in V} R_{u,v} \cdot C_{i,v} = \sum_{u \in \mathcal{N}(v)} C_{i,v} = \sum_{u \in \mathcal{N}(v)} f^{(t-1)}(v)_i$.

The *ConceptFFN* in the second stage then applies the weight matrices to both $C_{,v}$, the representation of a node itself, and $C'_{,v}$, the aggregated representation of its neighbors. These can be combined and passed through the activation function to compute the new node representation.

The following weight matrix can be used to achieve the correct mapping:

$$W_c^{(t)} = \begin{bmatrix} W_1^{(t)} & W_2^{(t)} & 0_{e \times (1)} \end{bmatrix} \in \mathbb{R}^{e \times (2d+1)}.$$

Using σ as activation function and zero bias, this layer computes:

$$\begin{aligned} C_{out,\_,v} &= \sigma\left(W_1^{(t)} \cdot C_{\_,v} + W_2^{(t)} \cdot C'_{\_,v}\right) \\ &= \sigma\left(W_1^{(t)} \cdot C_{\_,v} + W_2^{(t)} \cdot \sum_{u \in \mathcal{N}(v)} C_{\_,u}\right) \\ &= \sigma\left(W_1^{(t)} \cdot f^{(t-1)}(v) + \sum_{u \in \mathcal{N}(v)} W_2^{(t)} \cdot f^{(t-1)}(u)\right) \\ &= f^{(t)}(v) \end{aligned}$$

The *RoleFFN* can be specified to ignore the other operations and keep the role representation unchanged:

$$W_r^{(t)} = \begin{bmatrix} 1 & 0_{1 \times d+2} \end{bmatrix} \in \mathbb{R}^{1 \times (1+d)}.$$

This shows that the $NDLM^r$ architecture can implement the same computations as a GNN. By stacking multiple layers, the $NDLM^r$ architecture can implement GNNs of arbitrary depth. Only the *ConceptRoleAttention* and the *ConceptFFN* are required to implement the GNN computations, while the other operations are ignored.

The $NDLM^s$ architecture corresponds to a variant of the GNN architecture where min or max aggregation is used instead of sum aggregation. The construction of the layer cannot be directly adapted, as it uses the linear property of sum aggregation to first compute the sum aggregation and then apply the weight matrix to the aggregated representation. However, the same computations can be implemented by a two-layer network, where the first layer applies the weights, which are then aggregated in the second layer.

5.4.2 Relation to NLM

The NLM architecture is structurally similar to the $NDLM^s$ architecture and is also based on computing intermediate concept and role representations and using attention mechanisms to combine them. The NLM, however, is more general as it allows for arbitrary arities of predicates. The operations *RoleAggregation*, *ConceptExtension*, *RoleTranspose*, *ConceptFFN*, and *RoleFFN* have direct counterparts in the NLM architecture, while the *ConceptRoleAttention* and *RoleRoleAttention* do not have direct counterparts. *ConceptRoleAttention* can be implemented in an NLM by first extending a unary predicate, corresponding to the concept, to a binary one. Then, combining this new binary predicate with the binary predicate corresponding to the role in the *FFN*. Finally, by aggregating the resulting binary predicate, the same result as *ConceptRoleAttention* can be achieved. RoleRoleAttention can be implemented by extending both roles to ternary predicates, then combining them in the *FFN* and finally aggregating the resulting ternary predicate. However, in both cases the *FFN* would need to learn multiplication of the two predicates, which is not guaranteed to be learnable.

We make three observations regarding the relation between the NLM and the NDLM architecture.

1. The $NDLM^s$ architecture can be implemented by a NLM architecture of maximal arity 3.
2. The $NDLM^s$ architecture cannot be implemented by a NLM architecture of maximal arity 2.
3. The $NDLM^s$ architecture can implement the NLM architecture of maximal arity 2, treating nullary predicates as unary ones with same evaluation everywhere.

The $NDLM^s$ architecture lies, expressivity-wise, somewhere in between the NLM architecture of maximal arity 2 and the NLM architecture of maximal arity 3. The advantage of using the $NDLM^s$ architecture over the NLM architecture of maximal arity 3 is that the NLM model needs to store full ternary predicates leading to space demand of $O(n^3)$ (for a fixed number of layers and intermediate predicate count), while the $NDLM^s$ architecture stores at most binary predicates leading to space demand of $O(n^2)$. The *RoleRoleAttention* allows some reasoning about triplets of nodes, although not everything that can be expressed by a full ternary predicate.

The advantage of using the $NDLM^s$ architecture over NLM with maximal arity 2, is the expressivity gain due to the *RoleRoleAttention*, which is not expressible in the NLM architecture

of maximal arity 2. This however comes at the cost of increased computational demand, as the *RoleRoleAttention* requires computing a sort of matrix product of the role representations, leading to a computational demand of $O(n^3)$ (for a fixed number of layers and intermediate predicate count), while the NLM architecture of maximal arity 2 only requires $O(n^2)$ computations

6 Results

The previously presented approaches were implemented, and multiple experiments were conducted to evaluate their performance in different scenarios.

This chapter first presents the results on the 1D-ARC dataset (Section 6.1), then discusses the results on action model learning (Section 6.2). Next, it presents handcrafted problems to further investigate the capabilities of different NDLM configurations (Section 6.3). Finally, it presents the results on maze-solving tasks (Section 6.4).

6.1 1D-ARC

Both the description-logic-based approach (presented in Section Section 5.1) and the implementation of the *NDLM* networks (presented in Section 5.3) were evaluated on the 1D-ARC dataset. The results are presented next.

All experiments in this section share common parameters: 1000 epochs, learning rate 0.001, 4 layers, and 10 intermediate concepts and roles. Each configuration had a time limit of 1 hour per task; if the limit was reached, the last model to solve all training examples was selected for evaluation. Not correctly learning the training examples was considered a failure.

The first experiment investigated the ability of both approaches to learn from three training examples and a single test example. For the NDLM architecture, all variants ($NDLM^s$, $NDLM^{s+}$, $NDLM^r$, and $NDLM^{r+}$) were evaluated, using both sigmoid and identity activation functions and batch sizes of 1 and 10. Table Figure 6.1 shows an overview of the results across the tasks. For each configuration, the number of solved tasks is presented. A task is considered solved if the model correctly learns all three training examples and the single test example. If only the training examples are learned but not the test example, this is indicated separately.

This experiment highlights the capability of the NDLM architecture to learn the correct transformations for most of the tasks, even with a small training set. The best performing configuration was $NDLM^s$ with identity activations and batch size 10, which solved 13 out of 17 tasks. The description logic based approach slightly outperformed this by solving 14 out of 16 tasks. In general using Sigmoid activations led to worse performance, while the batch size improved performance in multiple settings. The $NDLM^r$ and $NDLM^{r+}$ variants performed significantly worse than the $NDLM^s$ and $NDLM^{s+}$ variants. $NDLM^{r+}$ reached the timelimit in all tasks without converging to a solution. Futher analysis of the executions of the models revealed that

Figure 6.1: Results of the experiments on the 1D-ARC dataset. The models are trained on a **single** version of the task. A/B indicates that for A tasks the test transformation was correctly found, while in B tasks the training examples could be recreated. Bold indicates the best performing configuration of the NDLM architecture.

Act.	Batch size	$NDLM^r$	$NDLM^{r+}$	$NDLM^s$	$NDLM^{s+}$	Enumerative
Identity	1	2/7	0/0	9/17	7/17	-
Sigmoid	1	0/1	0/0	9/9	8/9	-
Identity	10	10/17	0/0	**13/17**	12/16	-
Sigmoid	10	0/0	0/0	10/14	6/7	-
-	-	-	-	-	-	14/16

the transitive closure component in relaxed mode is numerically instable and often diverges to very large values, which additionally hinders convergence to a solution.

Additionally, the NDLM models were evaluated on a dataset consisting of all 50 versions of the training and test datasets, again across all variants with different batch sizes and activation functions. The results are shown in Table 6.1.

Table 6.1: Results of the experiments on the 1D-ARC dataset. The models are trained on **all** versions of the tasks. A/B indicates that for A tasks the transformations were solved, while in B tasks the training examples were learned. Bold indicates the best performing configuration of the NDLM architecture.

Act.	Batch size	$NDLM^r$	$NDLM^{r+}$	$NDLM^s$	$NDLM^{s+}$
Identity	1	10/11	0/0	**11/12**	**11/12**
Sigmoid	1	2/2	0/0	9/9	5/5
Identity	10	10/10	0/0	9/10	0/0
Sigmoid	10	0/0	0/0	8/8	0/0

The results indicate that the performance of the NDLM architecture slightly decreases when trained on all versions of the tasks, compared to training on a single version. This can be attributed to the fact that, in multiple cases, the time limit was reached before the model could converge properly. However, generalization to the test tasks improved: more training data prevented overfitting to the training tasks and ensured that, when a solution was found, it generalized. Again, configurations with identity activations performed better than those with sigmoid activations, and the $NDLM^s$ and $NDLM^{s+}$ variants outperformed the $NDLM^r$ and $NDLM^{r+}$ variants.

Finally, Table 6.2 reports the results of the best-performing models from the previous experiments by task. Tasks that could not be solved by the DL-based approach were also challenging for the NDLM architectures. For the four tasks where the DL-based approach failed to find a generalizing solution, none of the best-performing NDLM architectures found a solution.

Full results of all configurations and tasks are included in the Appendix in Table C.1.

Table 6.2: Results of the experiments on the 1D-ARC dataset. ✓ indicates that all training examples and the single test task were correctly learned. ✗ indicates that the task was not solved. (✓) indicates that only the training tasks were solved but not the test task. *FIRST* is an $NDLM^s$ model with identity activations and batch size 10, trained and tested only on the first version of each task. *ALL* is an $NDLM^s$ model with identity activations and batch size 1, while ALL^+ is an $NDLM^{s+}$ model with identity activations and batch size 1. Both *ALL* and ALL^+ were trained and tested on all 50 versions.

Task	**DL-based**	*FIRST*	*ALL*	ALL^+
1d_denoising_1c	✓	✓	✓	✓
1d_denoising_mc	✓	✓	✓	✓
1d_fill	✓	✓	(✓)	✓
1d_flip	✓	✓	✓	✓
1d_hollow	✓	✓	✓	✓
1d_mirror	✓	✓	✗	✗
1d_move_1p	✓	✓	✓	✓
1d_move_2p	✓	✓	✓	✓
1d_move_2p_dp	✓	✓	✓	✓
1d_move_3p	✓	✓	✓	✓
1d_move_dp	(✓)	(✓)	✗	✗
1d_padded_fill	✗	✗	✗	✗
1d_pcopy_1c	✓	✓	✓	✓
1d_pcopy_mc	✓	✓	✓	(✓)
1d_recolor_cmp	(✓)	(✓)	✗	✗
1d_recolor_cnt	✓	✓	✗	✗
1d_recolor_oe	✗	(✓)	✗	✗
1d_scale_dp	✓	(✓)	✓	✓
Sum	**14/16**	**13/17**	**11/12**	**11/12**

All approaches failed on the 1d_move_dp task. This task involves moving a block of pixels towards some indicated position. The model therefore needs to count the number of pixels in the block in order to verify the newly created block is of the same size.

The 1d_padded_fill task was also not solved by any of the approaches. This task involves filling every second space between isolated pixels. The model would need to identify every second pixel and fill the space after it, which is a complex parity-based task.

None of the approaches could correctly solve the 1d_recolor_cmp task, here blocks were recolored based on their size in relation to other block sizes. This task therefore needs counting and comparison of block sizes, which is a complex task for the models.

Finally, the 1d_recolor_oe task was not solved by any of the approaches. This task involves recoloring blocks of odd size. Again, the model would need to count the number of pixels in a block and determine whether this number is odd.

6.2 Action Model Learning

This section explores the capabilities of the proposed architecture in an action model learning setting, where the task is to learn the preconditions and effects of action schemas. Section 3.3 introduces the task setting and the proposed architecture for this task. The following presents the results of the experiments conducted in this setting and discusses their implications.

The tests were conducted on the STRIPS domains *blocks world*, *logistics*, *delivery*, and *c-puzzle*. In all of these, some action arguments were removed, such that the models needed to predict the next state and action applicability with only a subset of action arguments. The removal of action arguments followed Jansen et al.[9], where some arguments can be removed, transforming the domain into a $STRIPS^+$ domain. For example, in delivery all considered instances involve only a single truck, hence the truck argument can be removed from all actions without changing the behaviour. Table 6.3 gives an overview of the removed arguments for each domain and action.

Table 6.3: Overview of removed action arguments, _ indicates that the argument was removed.

Domain	Action Name	Original Arguments	Simplified Arguments
blocks world	pickup	(?block)	(?block)
blocks world	putdown	(?block)	(_)
blocks world	stack	(?block1, ?block2)	(_, ?block2)
blocks world	unstack	(?block1, ?block2)	(_, ?block2)
c-puzzle	up	(?tile, ?from, ?cell)	(_,_,_)
c-puzzle	down	(?tile, ?from, ?cell)	(_,_,_)
c-puzzle	left	(?tile, ?from, ?cell)	(_,_,_)
c-puzzle	right	(?tile, ?from, ?cell)	(_,_,_)
delivery	pick-package	(?truck, ?package, ?cell)	(_, ?package,_)
delivery	drop-package	(?truck, ?package, ?cell)	(_, ?package,_)
delivery	move	(?truck, ?from, ?to)	(_, _, ?to)

In addition to the removal of action arguments, some predicates are not necessary to determine action behaviour and can be removed as well. In *blocks world*, the evaluation of the *clear* and *handempty* predicates can be recovered from other predicates. Hence, they can be removed without losing information about the state. In the *c-puzzle* domain, the *blank* predicate can be removed, as it is satisfied by the unique cell that is not occupied by any tile. The domains with removed predicates are marked with $^-$ in the following.

Tables 6.4 and 6.5 illustrate the results of the experiments, precondition and effect networks are trained separately. The shown experiment uses the $NDLM^s$ architecture with identity activation functions, trained with removed action arguments.

The instances and the split between applicable and non-applicable transitions for precondition learning for both the training and testing datasets are given in the appendix (Section C.2 and Table C.2). Additionally, the experiment was conducted with other NDLM configurations, operating in strict and relaxed mode, with and without transitive closure, and with sigmoid and identity activation functions. The results of these experiments are included in the appendix (Tables C.3 to C.6). The $NDLM^s$ architecture with identity activation functions was overall the best performing configuration.

Table 6.4: Results for action model **effect** learning experiments. Domains with $^-$ indicate that some predicates were removed from the state descriptions. T is the number of transitions in the training dataset.

Domain	Action Name	TRAIN	TEST	$NDLM^s$; Id
blocks world	pickup	26	8	✓
blocks world	putdown	11	18	✓
blocks world	stack	29	36	✓
blocks world	unstack	14	25	✓
delivery	pick-package	7	4	✓
delivery	drop-package	12	10	✓
delivery	move	82	172	✓
logistics	load	16	270	✓
logistics	unload	7	99	(✓)
logistics	drive	34	516	✓
logistics	fly	34	1001	(✓)
c-puzzle	up	9	8	✓
c-puzzle	down	12	11	✓
c-puzzle	left	14	15	✓
c-puzzle	right	14	14	✓
blocks world$^-$	pickup	26	8	✓
blocks world$^-$	putdown	11	18	✓
blocks world$^-$	stack	29	36	✓
blocks world$^-$	unstack	14	25	✓
c-puzzle$^-$	up	9	8	✓
c-puzzle$^-$	down	12	11	(✓)
c-puzzle$^-$	left	14	15	✓
c-puzzle$^-$	right	14	14	✓

Table 6.5: Results for action model **precondition** learning experiments. Domains with $^-$ indicate that some predicates were removed from the state descriptions. TRAIN is the number of transitions in the training dataset, TEST is the number of transitions in the test dataset.

Domain	Action Name	TRAIN	TEST	$NDLM^s$; Id
blocks world	pickup	129	390	✓
blocks world	putdown	101	228	✓
blocks world	stack	232	383	✓
blocks world	unstack	227	397	✓
delivery	pick-package	280	591	✓
delivery	drop-package	192	440	✓
delivery	move	317	652	✓
logistics	load	186	1001	(✓)
logistics	unload	170	1001	✓
logistics	drive	160	1001	✓
logistics	fly	204	1001	✓
c-puzzle	move-up	129	118	✓
c-puzzle	move-down	102	91	✓
c-puzzle	move-left	84	55	✓
c-puzzle	move-right	84	64	✓
blocks world$^-$	pickup	129	390	✓
blocks world$^-$	putdown	101	228	✓
blocks world$^-$	stack	232	383	✓
blocks world$^-$	unstack	227	397	✓
c-puzzle$^-$	move-up	129	118	✓
c-puzzle$^-$	move-down	102	91	✓
c-puzzle$^-$	move-left	84	55	✓
c-puzzle$^-$	move-right	84	64	✓

The results indicate that the model is able to learn the correct preconditions for the domains *blocks world*, *delivery*, and *c-puzzle*, including domains with removed predicates. The single failure in the effects of the *c-puzzle*$^-$ domain may be attributed to the stochastic nature of the training process and little training data, as the model was able to learn the correct effects for the other actions, which behave similarly.

Only for the *logistics* domain did the model fail to generalize to the test instances in mulitple cases. This could potentially be attributed to the fact that the training instance consists of only two objects of each type, while the test instance contains three objects of each type. This could

potentially allow formulas that are too specific to the training instance to be learned, which do not generalize to the test instance.

6.3 Hard Graphs

In this section, three graph reasoning problems are presented to highlight the expressivity capabilities and shortcomings of different NDLM configurations.

6.3.1 Cycles

The first pair of graphs is a standard example of two non-isomorphic graphs that cannot be distinguished by the color refinement algorithm (1-WL) and hence cannot be distinguished by regular message-passing GNN architectures [14]. The first graph is a 6-cycle, while the second graph consists of two disjoint 3-cycles.

The 1-WL fails because all nodes have degree two and are initially assigned the same color. In each iteration of 1-WL, every node receives the same multiset of neighbor colors, and thus all nodes remain in the same color class throughout all refinement steps. Consequently, the algorithm converges to identical colorings for both graphs.

The NDLM architecture, however, is not limited to the expressivity of 1-WL. Particularly, the *RoleRoleAttention* operation allows reasoning over triplets of nodes, which is beyond message passing over edges.

See Figure 6.2 for an illustration of the two graphs. The task of distinguishing the nodes of the 6-cycle from the 3-cycle is formulated as deriving a single concept, that contains all nodes of the 6-cycle but none of the two 3-cycles.

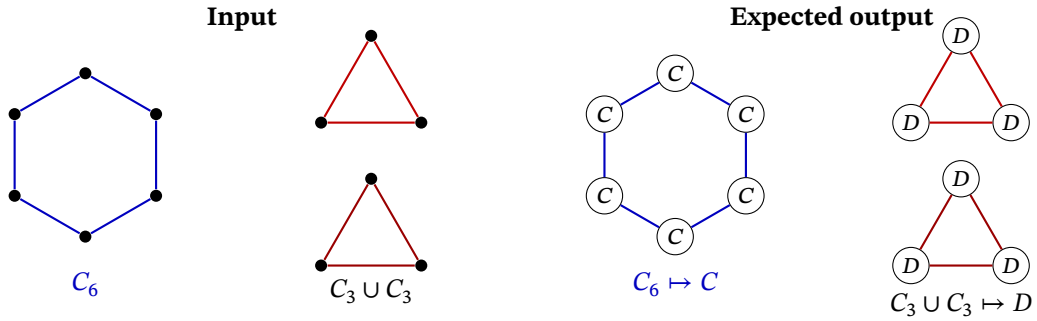


Figure 6.2: C_6 and $C_3 \cup C_3$: input and expected labeling output.

Each configuration of the *NDLM* architecture is trained on the tasks with 6 layers and 10 intermediate concepts and roles. The tested configurations include strict and relaxed mode, with and without transitive closure, and with sigmoid and identity activation functions. The results shown in Table 6.6 indicate that strict mode solves the task in all cases, while in relaxed

mode with identity activation all components are linear and the model fails to solve the task. Additionally, the transitive closure component in relaxed mode appears to hinder convergence to a solution.

Table 6.6: Results for the cycle task. ✓indicates that the task was learned, while ✗indicates that the task could not be solved.

Configuration	$NDLM^{s}$	$NDLM^{s+}$	$NDLM^{r}$	$NDLM^{r+}$
Identity	✓	✓	✗	✗
Sigmoid	✓	✓	✓	✗

6.3.2 Stars

The next task investigates the ability of the different configurations to count the number of neighbors and make a decision based on this information. To evaluate this, the model was trained to distinguish between a 4-star graph and a 3-star graph. The 4-star graph consists of a central node surrounded by 4 nodes, while the 3-star consists of a central node and 3 surrounding nodes. The model was only provided with the adjacency relations of both graphs, and the task was to derive a concept that identifies all nodes of the 4-star, while containing none of the 3-star nodes.

Figure 6.3 illustrates the two graphs and the target concepts.

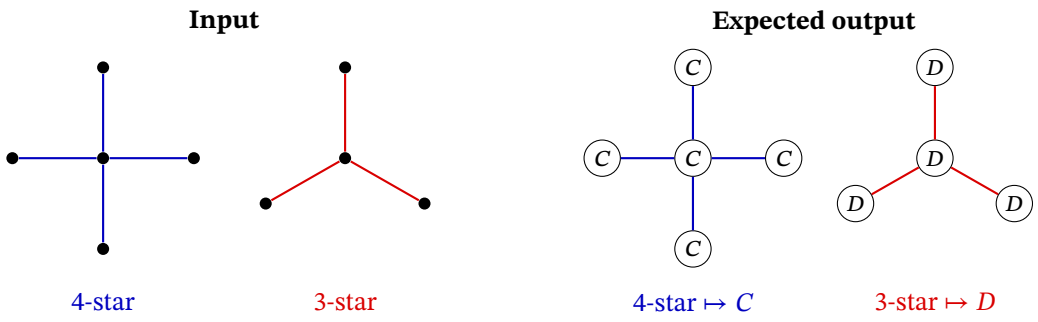


Figure 6.3: 4-star and 3-star graphs: input and expected center labeling output.

Again multiple configurations were evaluated. The configurations consisted of 3 Layers with 10 intermediate concepts and roles each. The model in the strict mode fails the task in all configurations as it lacks counting ability. The model in the relaxed mode solves the task in all cases.

Table 6.7: Results for the star task. ✓ indicates that the task was solved, while ✗ indicates that the task was not solved.

Problem	Activation Function	$NDLM^s$	$NDLM^{s+}$	$NDLM^r$	$NDLM^{r+}$
4-star vs. 3-star	Sigmoid	✗	✗	✓	✓
4-star vs. 3-star	Identity	✗	✗	✓	✓

6.3.3 Line

The last experiment of this section investigates limitations due to the receptive field and how adding the transitive closure operator can help overcome them.

Towards this end, a family of graphs is constructed. For each $k \in \mathbb{N}$, let G_k and H_k be graphs consisting of a single path of length k each. The first node of G_k is labeled with concept C, while the first node of H_k is labeled with concept D. The task is, given the encoding of $G_k \cup H_k$ as input, to assign all nodes of G_k to concept C_{out} and all nodes of H_k to concept D_{out}. See Figure 6.4 for an illustration of the construction and the problem.

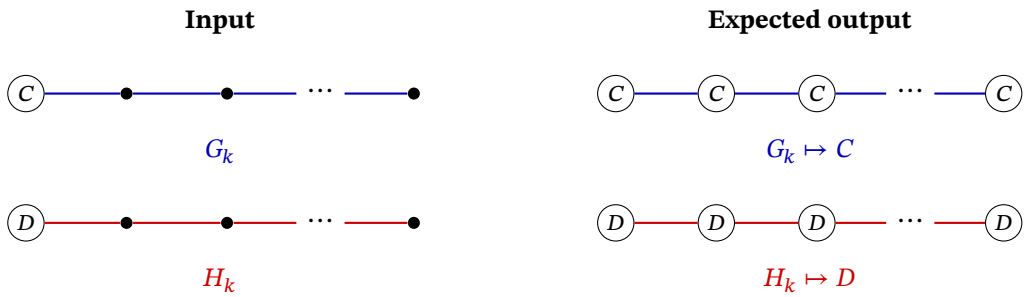


Figure 6.4: Input and expected output for the two path graphs of variable length $k \in \mathbb{N}$.

All experiments are performed under the same conditions with the same hyperparameters, including 1000 epochs and a learning rate of 0.001, but varying model depths (layer count). The number of intermediate concepts and roles is 10 throughout. For this experiment, the architecture operates in strict mode and the results are illustrated in Figure 6.5.

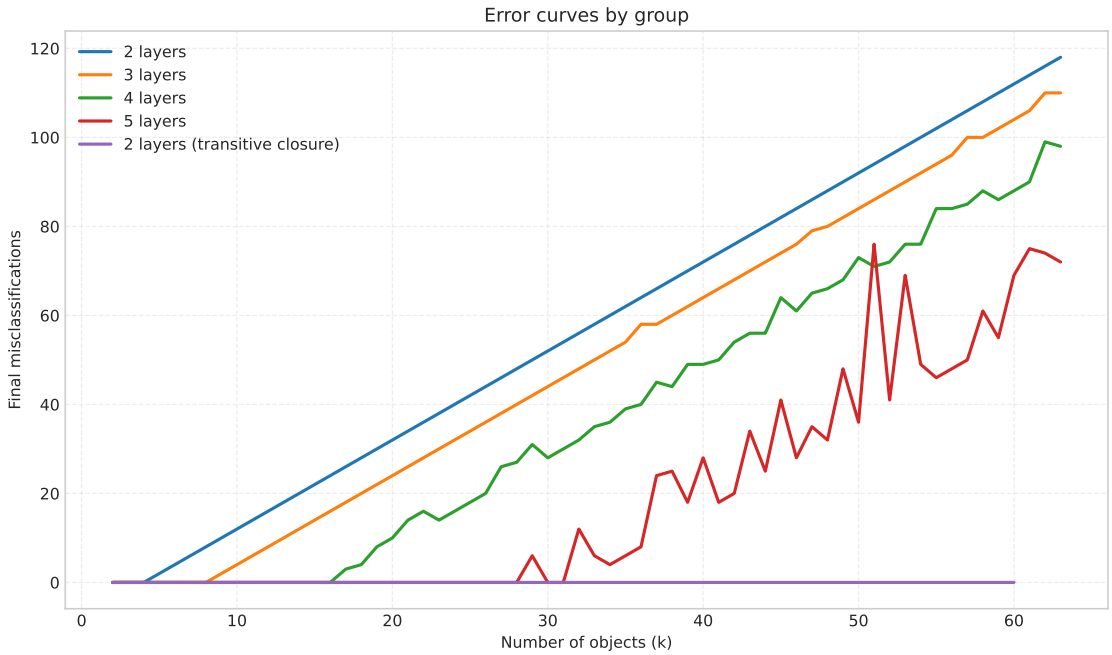


Figure 6.5: Number of misclassified nodes after training versus number of objects k for experiments with different model depths. In the *2 layers (transitive closure)* configuration, the transitive closure operator is applied. For each k, the model is trained on the corresponding dataset for 1000 epochs.

The results nicely demonstrate how the network doubles its receptive field with each layer via the role-concatenation operation, reaching a receptive field of size 2^l after l layers. By contrast, adding the transitive closure operator yields an unbounded receptive field, allowing the model to solve the task for arbitrarily large k with only 2 layers. However, the transitive closure configuration exceeded resource limits for $k > 60$, while the configurations without it could be trained beyond this point.

Next the speed of convergence of all eight different configurations was tested. For this, we will look at the number of misclassified nodes over the course of the training procedure. This will be performed for a fixed $k = 8$ and 3 layers in all configurations. The results are illustrated in Figure 6.6.

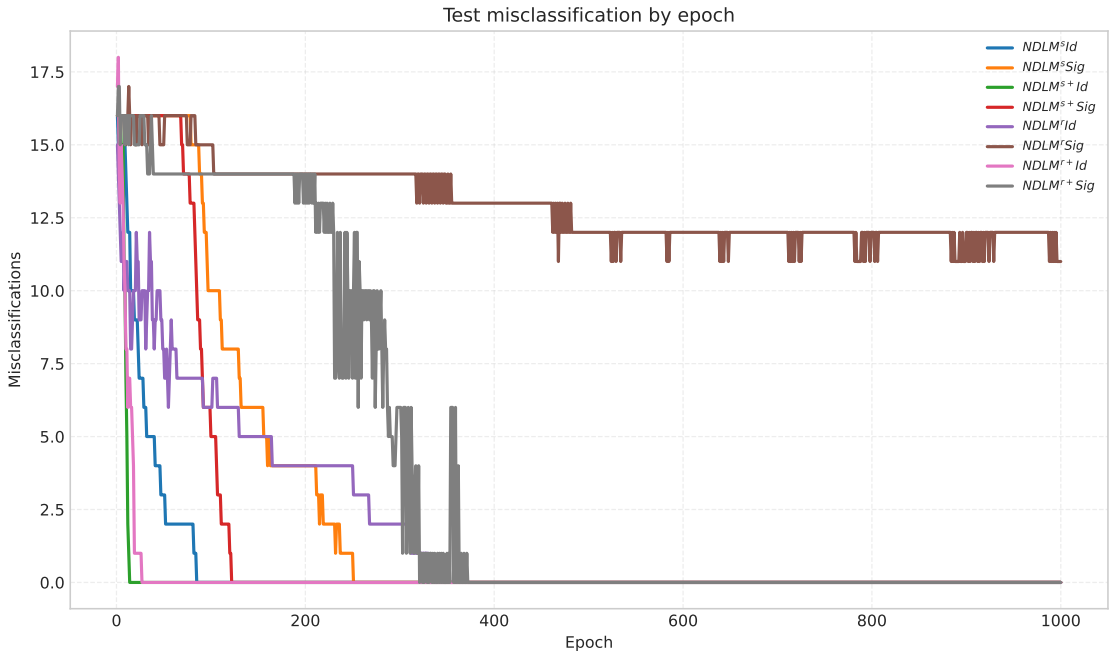


Figure 6.6: Number of misclassified nodes during training for the different configurations. The task is the same as in the previous experiment with $k = 8$ and 3 layers in all configurations.

Consistently the strict mode converged faster to a solution than the same configuration in the relaxed mode. Additionally the sigmoid activations seem to hinder convergence compared to identity activation functions. The network in the relaxed mode with sigmoid activations even fails to learn the task in the given 1000 epochs. Transitive closure consistently improves convergence in this task.

6.4 Mazes

The general setup of this experiment is presented in Section 3.2. The task is to learn the transformations required to solve maze tasks of increasing size. The training and testing datasets consist of 100 examples each, where the transition from the initial state to the goal state is represented as a sequence of actions. The model is trained on mazes of size 5×5 and 6×6, with path lengths between 1 and 16. The experiments investigate the ability of the architecture to learn the transformations required to solve maze tasks of increasing size. The path lengths of the test examples are the same as those of the training examples, while the mazes differ.

In all experiments, the $NDLM^r$ architecture with identity activations was used and evaluated with both 3 and 4 layers. The results indicate that the model is able to learn the transformations

up to a certain path length, but fails to generalize to longer paths. The results are illustrated in Table 6.8.

Table 6.8: Results of the maze experiments. For each path length, the number of misclassified tensor entries is indicated.

Path Length	5×5; 3 Layer	5×5; 4 Layer	6×6; 3 Layer	6×6; 4 Layer
3	0	0	0	0
4	0	0	0	0
5	0	0	0	0
6	0	0	0	0
7	0	0	0	0
8	0	0	0	0
9	3	0	0	0
10	33	0	12	0
11	23	0	20	0
12	44	0	28	0
13	40	0	44	0
14	49	0	38	0
15	53	0	51	0
16	56	0	69	0

The results follow the expected pattern, where the model is able to learn transformations up to a certain path length, but fails to generalize to longer paths. As observed in Section 6.3.3, the maximum path length scales approximately exponentially with the number of layers. The model with 4 layers is able to learn transformations up to path length 16, while the model with 3 layers fails to learn transformations for path lengths greater than 9.

6.4.1 Generalizing to Larger Mazes

Next, the ability of the model to generalize to **a)** longer paths, **b)** larger mazes, and **c)** both longer paths in larger mazes is investigated. For this, a model trained on 5×5 mazes with path length 8 is evaluated on mazes of size 5×5 and 6×6 with path lengths between 8 and 16. The results are illustrated in Tables 6.9 to 6.11. In this experiment, both the $NDLM^s$ and $NDLM^{s+}$ setups are evaluated. $NDLM^s$ is the strict mode, and $NDLM^{s+}$ is strict mode with the transitive-closure extension enabled. Additionally, both configurations are evaluated using sigmoid or identity activations, and the number of layers is fixed to four in all cases.

a) For this task the model trained on 5×5 mazes with path length 8 is evaluated on 5×5 mazes with path lengths between 8 and 16. Table 6.9 illustrates the results. The results indicate that the model is able to solve the training task in all cases, but fails to generalize to longer paths if the transitive-closure extension is not enabled. With transitive closure enabled, the model

Table 6.9: a) Generalization to longer paths in 5×5 mazes. For each path length, it is indicated whether the network solved all mazes. *Sig* indicates that a sigmoid activation function is used and *Id* indicates that an identity activation function is used.

Path Length	Maze size	$NDLM^s$; *Sig*	$NDLM^s$; *Id*	$NDLM^{s+}$; *Sig*	$NDLM^{s+}$; *Id*
Training (8)	5×5	✓	✓	✓	✓
8	5×5	✓	✓	✓	✓
9	5×5	✗	✗	✓	✓
10	5×5	✗	✗	✓	✓
11	5×5	✗	✗	✓	✓
12	5×5	✗	✗	✓	✓
13	5×5	✗	✗	✓	✓
14	5×5	✗	✗	✓	✓
15	5×5	✗	✗	✓	✓
16	5×5	✗	✗	✓	✓

learns a general solution that solves all training problems and generalizes to longer paths as well.

b) Next the model trained on 5×5 mazes with path length 8 is evaluated on 6×6 mazes with path lengths between 2 and 8. The results, illustrated in Table 6.10, show that all configurations are able to adapt to the larger mazes and solve the tasks correctly.

Table 6.10: b) Generalization from 5×5 to 6×6 mazes. For each path length, it is indicated if the network solved all mazes. *Sig* indicates that a sigmoid activation function is used and *Id* indicates that an identity activation function is used.

Path Length	Maze size	$NDLM^s$; *Sig*	$NDLM^s$; *Id*	$NDLM^{s+}$; *Sig*	$NDLM^{s+}$; *Id*
Training (8)	6×6	✓	✓	✓	✓
2	6×6	✓	✓	✓	✓
3	6×6	✓	✓	✓	✓
4	6×6	✓	✓	✓	✓
5	6×6	✓	✓	✓	✓
6	6×6	✓	✓	✓	✓
7	6×6	✓	✓	✓	✓
8	6×6	✓	✓	✓	✓

c) Finally, the ability of the models to find longer paths in larger mazes is evaluated. For this, the model is again trained on paths of length 8 and evaluated on 6×6 mazes with path lengths between 8 and 16. The results in Table Table 6.11 follow the pattern of a): models with transitive closure are able to learn arbitrarily long paths, while those without transitive closure fail to generalize to longer paths.

Table 6.11: c) Generalization to longer paths in 6×6 mazes. For each path length, it is indicated whether the network solved all mazes. *Sig* indicates that a sigmoid activation function is used and *Id* indicates that an identity activation function is used.

Path Length	Maze size	$NDLM^s$; *Sig*	$NDLM^s$; *Id*	$NDLM^{s+}$; *Sig*	$NDLM^{s+}$; *Id*
Training (8)	6 × 6	✓	✓	✓	✓
8	6 × 6	✓	✓	✓	✓
9	6 × 6	✗	✗	✓	✓
10	6 × 6	✗	✗	✓	✓
11	6 × 6	✗	✗	✓	✓
12	6 × 6	✗	✗	✓	✓
13	6 × 6	✗	✗	✓	✓
14	6 × 6	✗	✗	✓	✓
15	6 × 6	✗	✗	✓	✓
16	6 × 6	✗	✗	✓	✓

7 Conclusion

This thesis investigated approaches to learning description logic formulas from data. Starting with a target description logic covering a wide range of operators, three approaches were developed and two of them were evaluated: a symbolic enumerative approach, a B-RASP encoding, which was discontinued after identifying limitations, and a novel neural architecture, the Neural Description Logic Machine (NDLM) architecture. This chapter summarizes the main findings, discusses limitations, and outlines potential directions for future work.

7.1 Summary

The first approach was a symbolic method that enumeratively generates candidate formulas and combines them into a decision list. The method was able to learn solutions for 16 out of 18 1D-ARC tasks, of which 14 generalized to the test problems. The approach could, in practice, only cover a limited fragment of the proposed description logic $\mathcal{L}$. Combinatorial growth of the number of candidate features prohibited the use of the full expressivity, as the necessary formula depth for solving the 1D-ARC tasks could not be reached within the given resource limits.

The second approach was to translate DL-operators into B-RASP code, with the goal of training corresponding transformers. This approach was discontinued after observing limitations of the encoding, such as the restriction of the model to a fixed number of objects.

Finally, a more general architecture was proposed – the *NDLM* architecture. It is explicitly designed to capture the expressivity of the targeted description logic. The architecture is based on a set of operators that are designed to capture the semantics of the description logic operators, while also containing learnable parameters to combine the outputs of the operators in a flexible way. The architecture was evaluated in multiple different scenarios, showing promising results in all applications. In the 1D-ARC tasks, the best model learned transformations for 17 out of 18 tasks, while generalizing to 13 of these.

Further, the problem of learning action models from state-action traces was formulated as a description logic learning problem. The *NDLM* architecture was applied to this problem, showing promising results in learning action models in multiple domains.

Additional experiments were conducted on graph reasoning tasks, highlighting some limitations of the architecture. Finally, the architecture was evaluated on a maze-solving task, further demonstrating the ability of the architecture to reason and generalize on complex tasks.

The expressivity of the *NDLM* architecture was analyzed in detail, showing that certain configurations exactly capture the expressivity of the targeted description logic.

Regarding related architectures, the presented neural approach is closely related to Neural Logic Machines (NLMs) [6], as both operate on structured relational representations with arity-aware operations. The relaxed mode was developed entirely independently and without prior knowledge of NLMs. The min/max aggregations in the strict mode are the only component explicitly inspired by the NLM architecture.

7.2 Limitations

The first approach, the symbolic enumerative method, suffered from a trade-off between expressivity and computational demands. In a restricted fragment, the method was able to reliably learn solutions for the majority of the 1D-ARC tasks, but allowing the full expressivity of the proposed description logic turned out to be computationally infeasible, as the number of candidate features grows exponentially and the required formula depth for solving 1D-ARC tasks could not be reached within the given resource limits.

The B-RASP encoding was discontinued after identifying limitations of the encoding, tying the model to fixed domain sizes and an inefficient encoding of the DL-operators.

Some limitations of the *NDLM* architecture were observed throughout the experiments. In the 1D-ARC task the model tended to overfit more than the symbolic approach, when presented with the same training data. As the 1D-ARC is specifically designed to challenge the generalization ability of the model, this is not surprising.

The model also showed some limitations in the graph reasoning tasks, where the receptive field and counting ability of the model were identified as bottlenecks in certain configurations.

Scalability of the proposed architecture to large instance sizes is one of the main limitations. All binary relations are represented as $n \times n$ matrices, where n is the number of objects in the domain. This leads to a quadratic growth of the input size with the number of objects, and operators involving these have a time complexity of up to $O(n^3)$. This is prohibitive for larger numbers of objects. Additionally, the number of intermediate representations grows with how many intermediate concepts and roles are used in the formula, which can further increase the computational demands of the model.

The extension of the model with the transitive closure operator showed promising results by allowing to reason about many steps along the relations. It increases the computational demands of the model significantly, adding a time complexity of $O(n^4)$ per role, limiting the applicability of the model to small instance sizes.

Furthermore, the model showed some sensitivity to the choice of hyperparameters: while some configurations performed well on some tasks, they performed worse on others.

The action model learning approach is inherently limited to domains containing only nullary, unary, and binary predicates. Higher-arity predicates are not supported. The model was able to reproduce all training transitions in all experiments, but in some cases failed to generalize to unseen transitions in other instances. This may be due to limited training data or insufficient regularization.

7.3 Outlook

Future work could focus on improving the scalability of the proposed architecture, e.g. by using sparse representations for the binary relations.

Another interesting direction for future work is to decompile a learned model into a symbolic description logic formula, which would allow one to extract human-interpretable knowledge from the model.

As future evaluation perspectives, the architecture could be evaluated for action model learning on domains with conditional effects, which require higher expressivity than the domains considered in this thesis. Additionally, the architecture could be applied to learning general policies, which can be framed as a similar formula synthesis problem.

More broadly, this work suggests that neural architectures grounded in formal logic are a promising direction for sample-efficient and interpretable learning on structured relational data.

A Blocksworld PDDL

Listing A.1: Example of a Blocksworld domain with 4 actions in PPDL syntax.

```
(define (domain BLOCKS)
   (:requirements :strips)
   (:predicates (on ?x ?y)
           (ontable ?x)
           (clear ?x)
           (handempty)
           (holding ?x)
           )

   (:action pickup
           :parameters (?x)
           :precondition (and (clear ?x) (ontable ?x) (handempty))
           :effect
           (and (not (ontable ?x))
           (not (clear ?x))
           (not (handempty))
           (holding ?x)))

   (:action putdown
           :parameters (?x)
           :precondition (holding ?x)
           :effect
           (and (not (holding ?x))
           (clear ?x)
           (handempty)
           (ontable ?x)))
   (:action stack
           :parameters (?x ?y)
           :precondition (and (holding ?x) (clear ?y))
           :effect
           (and (not (holding ?x))
           (not (clear ?y))
           (clear ?x)
           (handempty)
           (on ?x ?y)))
   (:action unstack
           :parameters (?x ?y)
           :precondition (and (on ?x ?y) (clear ?x) (handempty))
           :effect
```

```
40              (and (holding ?x)
41              (clear ?y)
42              (not (clear ?x))
43              (not (handempty))
44              (not (on ?x ?y)))))
```

Listing A.2: Example of a Blocksworld instance with 5 blocks in PPDL syntax.

```
1  (define (problem BLOCKS-5-CLEAR)
2      (:domain BLOCKS)
3      (:objects A B C D E)
4      (:init
5          (ONTABLE A)
6          (ONTABLE B)
7          (ONTABLE C)
8          (ONTABLE D)
9          (ONTABLE E)
10         (CLEAR A)
11         (CLEAR B)
12         (CLEAR C)
13         (CLEAR D)
14         (CLEAR E)
15         (HANDEMPTY))
16     (:goal (AND (CLEAR A))))
```

B Proof details

B.1 Translation $NDLM^r$

The translation of the operators into the $NDLM^r$ architecture, need some more steps, where the output of the operators is mapped to 0 or 1 correspondingly. The following table illustrates these translations.

Table B.1: Translation of DL-operators to network operators in relaxed mode.

Concept Operators	Implementation in the Network	Explanation
$all(R_1, C_1)$	$ConceptFFN(ConceptRoleAttention($ $ConceptFFN([C_1]), [R_1]))$	Firstly FFN computes $\neg C_1$ then ConceptRoleAttention computes for each object if there is some edge that is directed into $\neg C_1$, FFN then maps to 0 or 1 correspondingly.
$some(R_1, C_1)$	$ConceptFFN(ConceptRoleAttention([C_1], [R_1]))$	
$and(C_1, C_2)$	$ConceptFFN([C_1, C_2])$	FFN computes $C_1 \wedge C_2$
$diff(C_1, C_2)$	$ConceptFFN([C_1, C_2])$	FFN computes $C_1 \wedge \neg C_2$
$eq(R_1, R_2)$	$ConceptFFN(RoleAggregation(RoleFFN([R_1, R_2])))$	FFN computes $\neg(R_1 \leftrightarrow R_2)$, RoleAggregation (sum or min) aggregates this and FFN remaps it to 0 or 1
$not(C_1)$	$ConceptFFN([C_1])$	FFN computes $\neg C_1$
$or(C_1, C_2)$	$ConceptFFN([C_1, C_2])$	FFN computes $C_1 \vee C_2$
$projection(R)$	$ConceptFFN(RoleAggregation([R]))$	RoleAggregation(sum or max) evaluates if some object has an outgoing edge The FFN then maps the result to 0 or 1.
$subset(R_1, R_2)$	$ConceptFFN(RoleAggregation(RoleFFN([R_1, R_2])))$	FFN computes $\neg(R_1 \rightarrow R_2)$, R, RoleAggregation (sum or min) aggregates this and FFN remaps it to 0 or 1.
top	$ConceptFFN([])$	FFN computes 1
bot	$ConceptFFN([])$	FFN computes 0
Role Operators	**Implementation in the Network**	**Explanation**
$and(R_1, R_2)$	$RoleFFN([R_1, R_2])$	FFN computes $R_1 \wedge R_2$.
$compose(R_1, R_2)$	$ConceptFFN(RoleRoleAttention([R_1, R_2]))$	RoleRoleAttention computes composition, FFN maps to 0 or 1.
$diff(R_1, R_2)$	$RoleFFN([R_1, R_2])$	FFN computes $R_1 \wedge \neg R_2$.
id	Encoded as input role	The identity matrix is provided as one of the input roles.
$inverse(R_1)$	$RoleTranspose([R_1])$	RoleTranspose directly computes the adjacency matrix of the inverted role.
$not(R_1)$	$RoleFFN([R_1])$	FFN computes $\neg R_1$
$or(R_1, R_2)$	$RoleFFN([R_1, R_2])$	FFN computes $R_1 \vee R_2$.
$restrict(R_1, C_1)$	$RoleFFN([ConceptExtension([C_1], R_1)])$	FFN computes $R_{1,i,j} \wedge C_j$ where C_j is accessed by converting it to a role first.
top	$RoleFFN([])$	FFN predicts 1
$TC(R)$	$RoleFFN(TC([R]))$	TC evaluates to 0 if there is no connection between two nodes and ≥ 1 else, FFN then maps this to 0/1.

B.2 Relation between Network Operations and Refinement Operators

Assuming the context in the proof of Theorem 3:

RoleAggregation:

For all $x, y \in \Delta$:

To show: $C_{AGG,x} \neq C_{AGG,y} \Rightarrow x \nsim_{AGG} y$

By definition: $C_{AGG,x} = \left(\min_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}} \circ \left(\max_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}}$
$C_{AGG,y} = \left(\min_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}} \circ \left(\max_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}}$

$$
\begin{aligned}
& C_{AGG,x} \neq C_{AGG,y} \\
\Rightarrow & \left(\left(\min_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}} \circ \left(\max_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}}\right) \neq \left(C_{AGG,y} = \left(\min_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}} \circ \left(\max_{k\in[n]} R_{i,k,j}\right)_{\substack{i\in[r]\\j\in[n]}}\right) \\
\Rightarrow & \bigvee_{i\in[r]} \left(\min_{k\in[n]} R_{i,k,x} \neq \min_{k\in[n]} R_{i,k,y} \vee \max_{k\in[n]} R_{i,k,x} \neq \max_{k\in[n]} R_{i,k,y}\right) \\
\Rightarrow & \bigvee_{i\in[r]} \left(\min_{k\in[n]} R_{i,k,x} < \min_{k\in[n]} R_{i,k,y} \vee \min_{k\in[n]} R_{i,k,x} > \min_{k\in[n]} R_{i,k,y}\right. \\
& \left. \vee \max_{k\in[n]} R_{i,k,x} < \max_{k\in[n]} R_{i,k,y} \vee \max_{k\in[n]} R_{i,k,x} > \max_{k\in[n]} R_{i,k,y}\right) \\
\Rightarrow & \bigvee_{i\in[r]} \left(\bigvee_{k\in[n]} \bigwedge_{k'\in[n]} R_{i,k,x} < R_{i,k',y} \vee \bigvee_{k'\in[n]} \bigwedge_{k'\in[n]} R_{i,k,x} > R_{i,k',y}\right. \\
& \left. \vee \bigvee_{k\in[n]} \bigwedge_{k'\in[n]} R_{i,k,x} > R_{i,k',y} \vee \bigvee_{k'\in[n]} \bigwedge_{k'\in[n]} R_{i,k,x} < R_{i,k',y}\right) \\
\Rightarrow & \bigvee_{i\in[r]} \left(\bigvee_{k\in[n]} \bigwedge_{k'\in[n]} R_{i,k,x} \neq R_{i,k',y} \vee \bigvee_{k'\in[n]} \bigwedge_{k'\in[n]} R_{i,k,x} \neq R_{i,k',y}\right) \\
\Rightarrow & \bigvee_{i\in[r]} \left(\bigvee_{k\in[n]} \bigwedge_{k'\in[n]} [(k,x)]_i \neq [(k',y)]_i \vee \bigvee_{k'\in[n]} \bigwedge_{k'\in[n]} [(k,x)]_i \neq [(k',y)]_i\right) \\
\Rightarrow & \bigvee_{k\in[n]} \left(\bigwedge_{k'\in[n]} [(k,x)]_i \neq [(k',y)]_i \vee \bigvee_{k'\in[n]} \bigwedge_{k'\in[n]} [(k,x)]_i \neq [(k',y)]_i\right) \\
\Rightarrow & \{[(k,x)] | k \in \Delta\} \neq \{[(k,y)] | k \in \Delta\} \\
\Rightarrow & [x]_{AGG} \neq [y]_{AGG}
\end{aligned}
$$

ConceptRoleAttention:

For all $x, y \in \Delta$:

To show: $C_{CRA,x} \neq C_{CRA,y} \Rightarrow x \nsim_{CRA} y$

By definition:

$$C_{CRA,x} = \left(\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,x}\right)\right)_{1\le(i-1)c+j\le c\cdot r} \circ \left(\left(\max_{k\in[n]} C_{j,k}\cdot R_{i,k,x}\right)\right)_{1\le(i-1)c+j\le c\cdot r}$$

$$C_{CRA,y} = \left(\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right)\right)_{1\le(i-1)c+j\le c\cdot r} \circ \left(\left(\max_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right)\right)_{1\le(i-1)c+j\le c\cdot r}$$

$$\begin{aligned}
&C_{CRA,x} \neq C_{CRA,y}\\
\Rightarrow&\left(\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,x}\right)_{1\le(i-1)c+j\le c\cdot r} \circ \left(\max_{k\in[n]} C_{j,k}\cdot R_{i,k,x}\right)_{1\le(i-1)c+j\le c\cdot r}\right)\\
&\quad\neq\left(\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right)_{1\le(i-1)c+j\le c\cdot r} \circ \left(\max_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right)_{1\le(i-1)c+j\le c\cdot r}\right)\\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[c]}\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,x} \circ \max_{k\in[n]} C_{j,k}\cdot R_{i,k,x}\right) \neq \left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,y} \circ \max_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right)\\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[c]}\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,x} \neq \min_{k\in[n]} C_{j,k}\cdot R_{i,k,y} \vee \max_{k\in[n]} C_{j,k}\cdot R_{i,k,x} \neq \max_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right)\\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[c]}\left(\min_{k\in[n]} C_{j,k}\cdot R_{i,k,x} < \min_{k\in[n]} C_{j,k}\cdot R_{i,k,y} \vee \min_{k\in[n]} C_{j,k}\cdot R_{i,k,x} > \min_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right.\\
&\quad quad\vee\left.\max_{k\in[n]} C_{j,k}\cdot R_{i,k,x} < \max_{k\in[n]} C_{j,k}\cdot R_{i,k,y} \vee \max_{k\in[n]} C_{j,k}\cdot R_{i,k,x} > \max_{k\in[n]} C_{j,k}\cdot R_{i,k,y}\right)\\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[c]}\left(\left(\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} C_{j,k}\cdot R_{i,k,x} < C_{j,k'}\cdot R_{i,k',y}\right) \vee \left(\bigvee_{k'\in[n]}\bigwedge_{k\in[n]} C_{j,k}\cdot R_{i,k,x} > C_{j,k'}\cdot R_{i,k',y}\right)\right.\\
&\quad quad\vee\left.\left(\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} C_{j,k}\cdot R_{i,k,x} < C_{j,k'}\cdot R_{i,k',y}\right) \vee \left(\bigvee_{k'\in[n]}\bigwedge_{k\in[n]} C_{j,k}\cdot R_{i,k,x} > C_{j,k'}\cdot R_{i,k',y}\right)\right)\\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[c]}\left(\left(\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} C_{j,k}\cdot R_{i,k,x} \neq C_{j,k'}\cdot R_{i,k',y}\right) \vee \left(\bigvee_{k'\in[n]}\bigwedge_{k\in[n]} C_{j,k}\cdot R_{i,k,x} \neq C_{j,k'}\cdot R_{i,k',y}\right)\right)\\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[c]}\left(\left(\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} (C_{j,k} \neq C_{j,k'} \vee R_{i,k,x} \neq R_{i,k',y})\right) \vee \left(\bigvee_{k'\in[n]}\bigwedge_{k\in[n]} (C_{j,k} \neq C_{j,k'} \vee R_{i,k,x} \neq R_{i,k',y})\right)\right)\\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[c]}\left(\left(\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} ([k]_i \neq [k']_i \vee [(k,x)]_i \neq [(k',y)]_i) \vee \left(\bigvee_{k'\in[n]}\bigwedge_{k\in[n]} [k]_i \neq [k']_i \vee [(k,x)]_i \neq [(k',y)]_i\right)\right)\right)\\
\Rightarrow&\{([k]_i, [(k,x)]_i) | k \in \Delta\} \neq \{([k]_i, [(k,y)]_i) | k \in \Delta\}\\
\Rightarrow&[x]_{CRA} \neq [y]_{CRA}
\end{aligned}$$

ConceptExtension:

For $(x,x'),(y,y') \in \Delta$:

To show: $C_{EXT,} \neq C_{EXT,y,y'} \Rightarrow (x,x') \nsim_{EXT} (y,y')$

By definition:
$R_{EXT,x,x'} = (C_{i,x})_{i\in[c]}$
$R_{EXT,y,y'} = (C_{i,y})_{i\in[c]}$

$$
\begin{aligned}
& R_{EXT,x,x'} \neq R_{EXT,y,y'} \\
\Rightarrow & (C_{i,x})_{i\in[c]} \neq (C_{i,y})_{i\in[c]} \\
\Rightarrow & \bigvee_{i\in[c]} (C_{i,x}) \neq (C_{i,y}) \\
\Rightarrow & \bigvee_{i\in[c]} [x]_i \neq [y]_i \\
\Rightarrow & [x]_i \neq [y]_i \\
\Rightarrow & [(x,x')]_{EXT} \neq [y,y']_{EXT}
\end{aligned}
$$

RoleTranspose:

For $(x,x'),(y,y') \in \Delta$:

To show: $C_{TR,x,x'} \neq C_{TR,y,y'} \Rightarrow (x,x') \nsim_{TR} (y,y')$

By definition:
$R_{TR,x,x'} = (R_{i,x',x})_{i\in[r]}$
$R_{TR,y,y'} = (R_{i,y',y})_{i\in[r]}$

$$
\begin{aligned}
& R_{TR,x,x'} \neq R_{TR,y,y'} \\
\Rightarrow & (R_{i,x',x})_{i\in[r]} \neq (R_{i,y',y})_{i\in[r]} \\
\Rightarrow & \bigvee_{i\in[r]} R_{i,x',x} \neq R^T_{i,y',y} \\
\Rightarrow & \bigvee_{i\in[r]} [(x',x)]_i \neq [(y',y)]_i \\
\Rightarrow & [(x',x)]_i \neq [(y',y)]_i \\
\Rightarrow & [(x,x')]_{TR} \neq [y,y']_{TR}
\end{aligned}
$$

RoleRoleAttention:

For $(x,x'),(y,y') \in \Delta$:

To show: $C_{RRA,x,x'} \neq C_{RRA,y,y'} \Rightarrow (x,x') \nsim_{RRA} (y,y')$

By definition:

$$R_{RRA,x,x'} = \left(\left(\min_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'}\right)\right)_{1\leq(i-1)r+j\leq r^2} \circ \left(\left(\max_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'}\right)_{p,q\in[n]}\right)_{1\leq(i-1)r+j\leq r^2}$$

$$R_{RRA,y,y'} = \left(\left(\min_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right)\right)_{1\leq(i-1)r+j\leq r^2} \circ \left(\left(\max_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right)_{p,q\in[n]}\right)_{1\leq(i-1)r+j\leq r^2}$$

$$\begin{aligned}
&R_{RRA,x,x'} \neq R_{RRA,y,y'} \\
\Rightarrow&\left(\left(\min_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'}\right)\right)_{1\leq(i-1)r+j\leq r^2} \circ \left(\max_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'}\right)_{1\leq(i-1)r+j\leq r^2} \\
&\quad \neq \left(\left(\min_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right)\right)_{1\leq(i-1)r+j\leq r^2} \circ \left(\max_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right)_{1\leq(i-1)r+j\leq r^2} \\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[r]}\left(\left(\min_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'}\right) \neq \left(\min_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right)\right. \\
&\quad \left.\vee\left(\max_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'}\right) \neq \left(\max_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right)\right) \\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[r]}\left(\min_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'} < \min_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'} \vee \min_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'} > \min_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right. \\
&\quad \left.\vee \max_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'} < \max_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'} \vee \max_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'} > \max_{k\in[n]} R_{i,y,k} \cdot R_{j,k,y'}\right) \\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[r]}\left(\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} R_{i,x,k} \cdot R_{j,k,x'} < R_{i,y,k'} \cdot R_{j,k',y'} \vee \bigvee_{k'\in[n]}\bigwedge_{k\in[n]} R_{i,x,k} \cdot R_{j,k,x'} > R_{i,y,k'} \cdot R_{j,k',y'}\right. \\
&\quad \left.\vee \bigvee_{k'\in[n]}\bigwedge_{k\in[n]} \left(R_{i,x,k} \cdot R_{j,k,x'}\right) < R_{i,y,k'} \cdot R_{j,k',y'} \vee \bigvee_{k\in[n]}\bigwedge_{k'\in[n]} R_{i,x,k} \cdot R_{j,k,x'} > R_{i,y,k'} \cdot R_{j,k',y'}\right) \\
\Rightarrow&\bigvee_{i\in[r]}\bigvee_{j\in[r]}\left(\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} \left(R_{i,x,k} \neq R_{i,y,k'} \vee R_{j,k,x'} \neq R_{j,k',y'}\right)\right. \\
&\quad \left.\vee \bigvee_{k'\in[n]}\bigwedge_{k\in[n]} \left(R_{i,x,k} \neq R_{i,y,k'} \vee R_{j,k,x'} \neq R_{j,k',y'}\right)\right) \\
\Rightarrow&\bigvee_{k\in[n]}\bigwedge_{k'\in[n]} \left([(x,k)]_i \neq [y,k']_i \vee [(k,x')]_i \neq [(k',y')]_i\right) \\
&\quad \vee \bigvee_{k'\in[n]}\bigwedge_{k\in[n]} \left([(x,k)]_i \neq [(y,k')]_i \vee [(k,x')]_i \neq [(k',y')]_i\right) \\
\Rightarrow&\{\left([(x,k)]_i, [(k,x')]_i\right) | k \in \Delta\} \neq \{\left([(y,k)]_i, [(k,y')]_i\right) | k \in \Delta\} \\
\Rightarrow&[(x,x')]_{RRA} \neq [(y,y')]_{RRA}
\end{aligned}$$

B.3 Equivalence between Refinement Operators and Formulas

ConceptExtension:

For all $o \in \Delta$:
To show: $[o]_{AGG} = \left(F^{AGG}_{[o]_{AGG}}\right)^I$

$$
\begin{aligned}
\left(F^{AGG}_{[o]_{AGG}}\right)^I =& \left\{ x \in \Delta | \bigwedge_{o' \in \Delta} \left(x \in \exists F^{-1}_{[(o',o)]_i}.\top \right) \wedge \neg x \in \exists \left(\bigwedge_{o' \in \Delta} \neg F^{-1}_{[(o',o)]_i} \right).\top \right\} \\
=& \left\{ x \in \Delta | \bigwedge_{o' \in \Delta} \exists k \in \Delta : (x,k) \in \left(F^{-1}_{[(o',o)]_i}\right)^I \wedge \neg \exists k : \bigwedge_{o' \in \Delta} \neg (x,k) \in \left(F^{-1}_{[(o,o')]_i}\right)^I \right\} \\
=& \left\{ x \in \Delta | \bigwedge_{o' \in \Delta} \exists k \in \Delta : (k,x) \in [(o',o)]^I_i \wedge \neg \exists k : \bigwedge_{o' \in \Delta} \neg (k,x) \in [(o,o')]_i \right\} \\
=& \left\{ x \in \Delta | \bigwedge_{o' \in \Delta} \bigvee_{k \in \Delta} (x,k) \in [(o,o')]^I_i \wedge \bigwedge_{k \in \Delta} \bigvee_{o' \in \Delta} (x,k) \in [(o,o')]_i \right\} \\
=& \left\{ x \in \Delta | \bigwedge_{o' \in \Delta} \bigvee_{k \in \Delta} [(x,k)]_i = [(o,o')]^I_i \wedge \bigwedge_{k \in \Delta} \bigvee_{o' \in \Delta} [(x,k)]_i = [(o,o')]_i \right\} \\
=& \{ x \in \Delta | \{ [(k,x)] | k \in \Delta \} = \{ [(o',o)] | o' \in \Delta \} \} \\
=& [o]_{AGG}
\end{aligned}
$$

ConceptRoleAttention

For all $o \in \Delta$:
To show: $[o]_{CRA} = \left(F^{CRA}_{[o]_{CRA}}\right)^I$

$$\begin{aligned}
\left(F^{CRA}_{[o]_{CRA}}\right)^I =& \left\{x \in \Delta | x \in \left(\bigwedge_{o' \in \Delta} \exists F_{[(o,o')]_i}.F_{[o']_i}\right)^I \wedge \neg x \in \left(\exists\left(\bigwedge_{o' \in \Delta} \neg\left(F_{[(o,o')]_i} \wedge \top|_{F_{[o']_i}}\right)\right).\top\right)^I\right\} \\
=& \left\{x \in \Delta | \bigwedge_{o' \in \Delta} \exists k \in \Delta : \left((x,k) \in \left(F_{[(o,o')]_i}\right)^I \wedge k \in \left(F_{[o']_i}\right)^I\right)\right. \\
& \left. \wedge \neg \exists k \in \Delta : \left(\bigwedge_{o' \in \Delta} \neg (x,k) \in \left(F_{[(o,o')]_i} \wedge \top|_{F_{[o']_i}}\right)\right)^I \wedge k \in \top^I\right\} \\
=& \left\{x \in \Delta | \bigwedge_{o' \in \Delta} \bigvee_{k \in \Delta} \left((x,k) \in [(o,o')]_i \wedge k \in [o']_i\right)\right. \\
& \left. \wedge \bigwedge_{o' \in \Delta o' \in \Delta} \bigvee \left((x,k) \in [(o,o')]_i \wedge \left((x,k) \in \top^I \wedge k \in \left(F_{[o']_i}\right)\right)^I \wedge k \in \top^I\right)\right\} \\
=& \left\{x \in \Delta | \bigwedge_{o' \in \Delta} \bigvee_{k \in \Delta} \left([(x,k)]_i = [(o,o')]_i \wedge [k]_i = [o']_i\right)\right. \\
& \left. \wedge \bigwedge_{o' \in \Delta o' \in \Delta} \bigvee \left([(x,k)]_i = [(o,o')]_i \wedge k \in \left(F_{[o']_i}\right)^I\right)\right\} \\
=& \left\{x \in \Delta | \bigwedge_{o' \in \Delta} \bigvee_{k \in \Delta} \left([(x,k)]_i = [(o,o')]_i \wedge [k]_i = [o']_i\right)\right. \\
& \left. \wedge \bigwedge_{o' \in \Delta o' \in \Delta} \bigvee \left([(x,k)]_i = [(o,o')]_i \wedge [k]_i = [o']_i\right)\right\} \\
=& \left\{x \in \Delta | \{\left([x,q]_i, [q]_i,\right) | q \in \Delta\} = \{\left([q]_i, [x,q]_i\right) | q \in \Delta\}\right\} \\
=& [o]_{CRA}
\end{aligned}$$

ConceptExtension:

For all $(o_1, o_2) \in \Delta^2$:
To show: $[(o_1,o_2)]_{EXT} = \left(F^{EXT}_{[(o_1,o_2)]_{EXT}}\right)^I$

$$\begin{aligned}
\left(F^{EXT}_{[(o_1,o_2)]_{Ext}}\right)^I =& \left\{(x_1,x_2) \in \Delta^2 | x \in \left(\left(\top|_{F_{[o_1]_i}}\right)^{-1}\right)^I\right\} \\
=& \left\{(x_1,x_2) \in \Delta^2 | \wedge (x_2,x_1) \in \left(\top|_{F_{[o_1]_i}}\right)^I\right\} \\
=& \left\{(x_1,x_2) \in \Delta^2 | x_1 \in F^I_{[o_1]_i}\right\} \\
=& \{(x_1,x_2) \in \Delta^2 | x_1 \in [o_1]_i\} \\
=& \left\{(x_1,x_2) \in \Delta^2 | [x_1]_i = [o_1]_i\right\} \\
=& [(o_1,o_2)]_{EXT}
\end{aligned}$$

RoleTranspose

For all $(o_1, o_2) \in \Delta^2$:
To show: $[(o_1, o_2)]_{TR} = \left(F^{TR}_{[(o_1,o_2)]_{TR}}\right)^I$

$$\begin{aligned}
\left(F^{TR}_{[(o_1,o_2)]_{TR}}\right)^I &= \left\{(x_1, x_2) \in \Delta^2 | (x_1, x_2) \in \left(F^{-}_{[(o_1,o_2)]_i} 1\right)^I\right\} \\
&= \left\{(x_1, x_2) \in \Delta^2 | (x_2, x_1) \in \left(F_{[(o_1,o_2)]_i}\right)^I\right\} \\
&= \left\{(x_1, x_2) \in \Delta^2 | (x_2, x_1) \in [(o_1, o_2)]_i\right\} \\
&= \left\{(x_1, x_2) \in \Delta^2 | [(x_2, x_1)]_i = [(o_1, o_2)]_i\right\} \\
&= [(o_1, o_2)]_{TR}
\end{aligned}$$

RoleRoleAttention

For all $(o_1, o_2) \in \Delta^2$:
To show: $[(o_1, o_2)]_{RRA} = \left(F^{RRA}_{[(o_1,o_2)]_{RRA}}\right)^I$

$$\begin{aligned}
\left(F^{RRA}_{[(o_1,o_2)]_{RRA}}\right)^I = \Bigg\{ & (x_1, x_2) \in \Delta^2 \mid (x_1, x_2) \in \left(\bigwedge_{o' \in \Delta} \left(F_{[(o_1,o')]_i} \circ F_{[(o',o_2)]_i} \right) \right)^I \\
& \wedge \neg (x_1, x_2) \in \left(\left(\bigwedge_{o' \in \Delta} \neg F_{[(o_1,o')]_i} \right) \circ \left(\bigwedge_{o' \in \Delta} \neg F_{[(o',o_2)]_i} \right) \right)^I \Bigg\} \\
= \Bigg\{ & (x_1, x_2) \in \Delta^2 \mid \bigwedge_{o' \in \Delta} (x_1, x_2) \in \left(F_{[(o_1,o')]_i} \circ F_{[(o',o_2)]_i} \right)^I \\
& \wedge \neg \exists b \in \Delta : \left((x_1, b) \in \left(\bigwedge_{o' \in \Delta} \neg F_{[(o_1,o')]_i} \right)^I \wedge (b, x_2) \in \left(\bigwedge_{o' \in \Delta} \neg F_{[(o',o_2)]_i} \right)^I \right) \Bigg\} \\
= \Bigg\{ & (x_1, x_2) \in \Delta^2 \mid \bigwedge_{o' \in \Delta} \exists b \in \Delta : \left((x_1, b) \in F^I_{[(o_1,o')]_i} \wedge (b, x_2) \in F^I_{[(o',o_2)]_i} \right) \\
& \wedge \neg \exists b \in \Delta : \bigwedge_{o' \in \Delta} \left(\neg (x_1, b) \in F^I_{[(o_1,o')]_i} \wedge \neg (b, x_2) \in F^I_{[(o',o_2)]_i} \right) \Bigg\} \\
= \Bigg\{ & (x_1, x_2) \in \Delta^2 \mid \bigwedge_{o' \in \Delta} \exists b \in \Delta : [(x_1, b)]_i = [(o_1, o')]_i \wedge [(b, x_2)]_i = [(o', o_2)]_i \\
& \wedge \neg \exists b \in \Delta : \bigwedge_{o' \in \Delta} \left(\neg (x_1, b) \in [(o_1, o')]_i \wedge \neg (b, x_2) \in [(o', o_2)]_i \right) \Bigg\} \\
= \Bigg\{ & (x_1, x_2) \in \Delta^2 \mid \bigwedge_{o' \in \Delta} \bigvee_{b \in \Delta} [(x_1, b)]_i = [(o_1, o')]_i \wedge [(b, x_2)]_i = [(o', o_2)]_i \\
& \wedge \bigwedge_{b \in \Delta} \bigvee_{o' \in \Delta} \left((x_1, b) \in [(o_1, o')]_i \wedge (b, x_2) \in [(o', o_2)]_i \right) \Bigg\} \\
= \Bigg\{ & (x_1, x_2) \in \Delta^2 \mid \bigwedge_{o' \in \Delta} \bigvee_{b \in \Delta} [(x_1, b)]_i = [(o_1, o')]_i \wedge [(b, x_2)]_i = [(o', o_2)]_i \\
& \wedge \bigwedge_{b \in \Delta} \bigvee_{o' \in \Delta} \left([(x_1, b)]_i = [(o_1, o')]_i \wedge [(b, x_2)]_i = [(o', o_2)]_i \right) \Bigg\} \\
= \Big\{ & (o_1', o_2') \in \Delta^2 \mid \{ \left([o_1', q]_i, [q, o_2']_i \right) \mid q \in \Delta \} = \{ \left([o_1, q]_i, [q, o_2]_i \right) \mid q \in \Delta \} \\
& \wedge [(o_1', o_2')]_i = [(o_1, o_2)]_i \Big\} \\
= & [(o_1, o_2)]_{RRA}
\end{aligned}$$

C Additional Results

C.1 1D-ARC

Table C.1 summarizes the 1D-ARC results by task and configuration. Here, ✓means that both training and test data were learned, (✓) means that only the training data was learned, and ✗means that the training data was not learned. - indicates that the model did not produce any results due to compute limitations.

Table C.1: 1D-ARC results by configuration and task. Rightmost column reports A/B with $A = \#(✓)$ and $B = \#(✓ \text{ or } (✓))$.

Model	Act.	Batch	1d_denoising_1c	1d_denoising_mc	1d_fill	1d_flip	1d_hollow	1d_mirror	1d_move_1p	1d_move_2p	1d_move_2p_dp	1d_move_3p	1d_move_dp	1d_padded_fill	1d_pcopy_1c	1d_pcopy_mc	1d_recolor_cmp	1d_recolor_cnt	1d_recolor_oe	1d_scale_dp	**A/B**
First Version																					
$NDLM^r$	id	B10	✓	✓	(✓)	✓	✓	(✓)	✓	✓	✓	✓	(✓)	✗	✓	✓	(✓)	(✓)	(✓)	(✓)	10/17
$NDLM^{r+}$	id	B10	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	id	B10	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	(✓)	✗	✓	✓	(✓)	✓	(✓)	(✓)	13/17
$NDLM^{s+}$	id	B10	✓	(✓)	✓	✓	✓	✗	✓	✓	✓	✓	(✓)	✗	✓	✓	✓	✓	(✓)	(✓)	12/16
$NDLM^r$	sig	B10	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0/0
$NDLM^{r+}$	sig	B10	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	sig	B10	✓	✓	✓	(✓)	✓	✗	✓	✓	✓	✓	✗	✗	(✓)	✓	(✓)	✓	✗	(✓)	10/14
$NDLM^{s+}$	sig	B10	✓	(✓)	✓	✗	✓	✗	✓	✓	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	6/7
$NDLM^r$	id	B1	(✓)	✓	✗	✗	✗	✗	✓	(✓)	(✓)	✗	✗	✗	✗	(✓)	✗	✗	✗	(✓)	2/7
$NDLM^{r+}$	id	B1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	id	B1	✓	✓	(✓)	(✓)	✓	(✓)	✓	✓	✓	✓	(✓)	✗	(✓)	✓	(✓)	✓	(✓)	(✓)	9/17
$NDLM^{s+}$	id	B1	✓	(✓)	✓	(✓)	✓	(✓)	(✓)	✓	✓	✓	(✓)	-	(✓)	✓	(✓)	(✓)	(✓)	(✓)	7/17
$NDLM^r$	sig	B1	✗	(✓)	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0/1
$NDLM^{r+}$	sig	B1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	sig	B1	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	✗	✗	✓	✗	✗	✗	✗	9/9
$NDLM^{s+}$	sig	B1	✓	(✓)	✓	✓	✓	✗	✓	✓	✓	✓	✗	-	✗	✗	✗	✗	✗	✗	8/9
All Versions																					
$NDLM^r$	id	B10	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	-	✓	✓	✗	✗	✗	✗	10/10
$NDLM^{r+}$	id	B10	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	id	B10	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	-	✗	✓	✗	✗	✗	(✓)	9/10
$NDLM^{s+}$	id	B10	-	-	✗	✗	✗	-	✗	✗	✗	✗	✗	-	-	-	-	-	-	✗	0/0
$NDLM^r$	sig	B10	✗	✓	✗	✗	✗	✗	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	0/0
$NDLM^{r+}$	sig	B10	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	sig	B10	✓	✗	✗	✗	✓	✗	✓	✓	✓	✓	✗	-	✓	✓	✗	✗	✗	✗	8/8
$NDLM^{s+}$	sig	B10	-	-	✗	-	✗	-	✗	✗	✗	✗	✗	-	-	-	-	-	-	✗	0/0
$NDLM^r$	id	B1	✓	✓	✗	✓	✓	✗	✓	✓	✓	✓	✗	-	✓	✓	✗	✗	✗	(✓)	10/11
$NDLM^{r+}$	id	B1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	id	B1	✓	✓	(✓)	✓	✓	✗	✓	✓	✓	✓	✗	-	✓	✓	✗	✗	✗	✓	11/12
$NDLM^{s+}$	id	B1	✓	✓	✓	✓	✓	✗	✓	✓	✓	✓	✗	-	✓	(✓)	✗	✗	✗	✓	11/12
$NDLM^r$	sig	B1	✗	✓	✗	✗	✗	✗	✓	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	2/2
$NDLM^{r+}$	sig	B1	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	✗	-	✗	✗	✗	✗	✗	✗	0/0
$NDLM^s$	sig	B1	✓	✗	✗	✓	✓	✗	✓	✓	✓	✓	✗	-	✓	✓	✗	✗	✗	✗	9/9
$NDLM^{s+}$	sig	B1	✓	✗	✓	✗	✗	✗	✓	✓	✗	✓	✗	-	✗	✗	✗	✗	✗	✗	5/5

C.2 Action Model Learning

In this section the results for the action model learning experiments are presented in more detail. The results for the effect learning and precondition learning experiments are shown in the tables below.

Table C.2: Overview of the number of applicable and non-applicable training and test transitions by task and action. *train_app* is number of applicable transitions in the training set,
train_non is number of non-applicable transitions in the training set,
train_percent is percentage of applicable transitions in the training set,
test_app is number of applicable transitions in the test set,
test_non is number of non-applicable transitions in the test set,
test_percent is percentage of applicable transitions in the test set.

action	train_app	train_non	train_percent	test_app	test_non	test_percent
blocks world						
pickup	26	103	20.16%	8	382	2.05%
putdown	11	70	13.58%	18	210	7.89%
stack	29	185	13.55%	36	346	9.42%
unstack	14	214	6.14%	35	353	9.02%
delivery						
drop-package	12	180	6.25%	10	430	2.27%
move	82	235	25.87%	172	470	26.79%
pick-package	7	273	2.50%	4	512	0.78%
logistics						
drive	34	144	19.10%	273	728	27.27%
fly	34	170	16.67%	378	623	37.76%
load	16	170	8.60%	123	878	12.29%
unload	7	170	3.95%	47	954	4.70%
c-puzzle						
move-down	12	90	11.76%	11	80	12.09%
move-left	14	70	16.67%	15	40	27.27%
move-right	14	70	16.67%	14	50	21.88%
move-up	9	120	6.98%	8	110	6.78%

Following instance sizes were used for sampling the training and testing transitions:

- blocksworld:
  - Train instance 1: 5 blocks
  - Train instance 2: 6 blocks
  - Test instance 1: 10 blocks
  - Test instance 2: 10 blocks
- delivery:
  - Train instance 1: 4x4 grid, 1 package, 1 truck
  - Train instance 2: 3x3 grid, 3 packages, 1 truck
  - Test instance 1: 5x5 grid, 3 packages, 1 truck

  - Test instance 2: 7x7 grid, 2 packages, 1 truck
- logistics:
  - Train instance 1: 2 cities, 1 airport and 1 office per city, 2 trucks, 2 planes, 2 packages
  - Test instance 1: 3 cities, 1 airport and 1 office per city, 3 trucks, 3 planes, 3 packages
- c-puzzle:
  - Train instance 1: 3x2 grid
  - Test instance 1: 4x2 grid

Table C.3: Results for action model learning in **blocksworld** domain. ✓:both training and test data were learned; (✓): only train data learned; ✗: training data was not learned.

Configuration	pickup	putdown	stack	unstack
Effects				
$NDLM^r$; Id	(✓)	(✓)	(✓)	✗
$NDLM^r$; Sig	✓	✓	✓	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^s$; Id	✓	✓	✓	✓
$NDLM^s$; Sig	✓	✓	✓	✗
$NDLM^{s+}$; Id	✓	✓	✓	✓
$NDLM^{s+}$; Sig	✓	✓	✓	✓
Preconditions				
$NDLM^r$; Id	(✓)	✓	(✓)	(✓)
$NDLM^r$; Sig	(✓)	✗	✓	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✓	✗	✗
$NDLM^s$; Id	✓	✓	✓	✓
$NDLM^s$; Sig	✓	✓	✓	✓
$NDLM^{s+}$; Id	✓	✓	✓	✓
$NDLM^{s+}$; Sig	✓	✓	✓	✓

Table C.4: Results for action model learning in **delivery** domain. ✓:both training and test data were learned; (✓): only train data learned; ✗: training data was not learned.

Configuration	pick-package	drop-package	move
Effects			
$NDLM^r$; Id	✗	✗	✗
$NDLM^r$; Sig	✗	✗	(✓)
$NDLM^{r+}$; Id	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗
$NDLM^s$; Id	✓	✓	✓
$NDLM^s$; Sig	✗	✗	✗
$NDLM^{s+}$; Id	✓	✓	✓
$NDLM^{s+}$; Sig	✗	✗	✗
Preconditions			
$NDLM^r$; Id	(✓)	(✓)	(✓)
$NDLM^r$; Sig	✗	✗	✗
$NDLM^{r+}$; Id	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗
$NDLM^s$; Id	✓	✓	✓
$NDLM^s$; Sig	✓	✓	✓
$NDLM^{s+}$; Id	✓	✓	✓
$NDLM^{s+}$; Sig	✗	✓	✓

Table C.5: Results for action model learning in **logistics** domain. ✓:both training and test data were learned; (✓): only train data learned; ✗: training data was not learned.

Configuration	load	unload	drive	fly
Effects				
$NDLM^r$; Id	✗	✗	✗	✗
$NDLM^r$; Sig	✗	✗	✗	(✓)
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^s$; Id	✓	(✓)	✓	(✓)
$NDLM^s$; Sig	✗	✗	✗	✗
$NDLM^{s+}$; Id	✓	(✓)	✓	✓
$NDLM^{s+}$; Sig	✗	✗	✗	✗
Preconditions				
$NDLM^r$; Id	(✓)	✓	(✓)	(✓)
$NDLM^r$; Sig	✗	✗	✗	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^s$; Id	(✓)	✓	✓	✓
$NDLM^s$; Sig	✗	✓	✓	(✓)
$NDLM^{s+}$; Id	✓	✓	✓	(✓)
$NDLM^{s+}$; Sig	(✓)	✗	(✓)	✓

Table C.6: Results for action model learning in **c-puzzle** domain. ✓:both training and test data were learned; (✓): only train data learned; ✗: training data was not learned.

configuration	move-up	move-down	move-left	move-right
Effects				
$NDLM^r$; Id	✗	✗	✗	✗
$NDLM^r$; Sig	✗	✗	✗	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^s$; Id	✓	✓	✓	✓
$NDLM^s$; Sig	✗	✗	✗	✗
$NDLM^{s+}$; Id	✓	✓	✓	✓
$NDLM^{s+}$; Sig	✗	✗	✗	✗
Preconditions				
$NDLM^r$; Id	(✓)	(✓)	(✓)	(✓)
$NDLM^r$; Sig	✗	✗	✗	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^s$; Id	✓	✓	✓	✓
$NDLM^s$; Sig	✓	✓	✓	✓
$NDLM^{s+}$; Id	✓	✓	✓	✓
$NDLM^{s+}$; Sig	✓	✓	✓	✓

Table C.7: Results for action model learning in **blocks world**$^-$ domain. ✓:both training and test data were learned; (✓): only train data learned; ✗: training data was not learned.

configuration	move-up	move-down	move-left	move-right
Effects				
$NDLM^r$; Id	(✓)	(✓)	(✓)	(✓)
$NDLM^r$; Sig	✓	✓	✓	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^s$; Id	✓	✓	✓	✓
$NDLM^s$; Sig	✓	✓	✓	✗
$NDLM^{s+}$; Id	✓	✓	✓	✓
$NDLM^{s+}$; Sig	✓	✓	✓	✓
Preconditions				
$NDLM^r$; Id	✗	(✓)	(✓)	(✓)
$NDLM^r$; Sig	✓	✓	✗	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	(✓)	✓	(✓)	(✓)
$NDLM^s$; Id	✓	✓	✓	✓
$NDLM^s$; Sig	✓	✓	✓	✓
$NDLM^{s+}$; Id	✓	✓	✓	✓
$NDLM^{s+}$; Sig	✓	✓	✓	✓

Table C.8: Results for action model learning in **c-puzzle**$^{-}$ domain. ✓:both training and test data were learned; (✓): only train data learned; ✗: training data was not learned.

configuration	move-up	move-down	move-left	move-right
Effects				
$NDLM^{r}$; Id	✗	✗	✗	✗
$NDLM^{r}$; Sig	✗	✗	✗	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^{s}$; Id	✓	✓	✓	✓
$NDLM^{s}$; Sig	✗	✗	✗	✗
$NDLM^{s+}$; Id	✓	✓	✓	(✓)
$NDLM^{s+}$; Sig	✗	✗	✗	✗
Preconditions				
$NDLM^{r}$; Id	(✓)	(✓)	(✓)	(✓)
$NDLM^{r}$; Sig	✗	✗	✗	✗
$NDLM^{r+}$; Id	✗	✗	✗	✗
$NDLM^{r+}$; Sig	✗	✗	✗	✗
$NDLM^{s}$; Id	✓	✓	✓	✓
$NDLM^{s}$; Sig	✗	✓	✓	✗
$NDLM^{s+}$; Id	✓	✓	✓	✓
$NDLM^{s+}$; Sig	✓	✓	✓	✗

List of Acronyms

B-RASP Boolean-Restricted Access Sequence Processing Language

DL Description Logic

LTL Linear Temporal Logic

List of Symbols

Mathematical Notation

$\mathbb{N}$	set of natural numbers
$\mathbb{R}$	set of real numbers
$[n]$	set of natural numbers $\{1, \cdot, n\}$
$\circ$	tensor concatenation along first dimension or function composition

B-RASP

Σ	finite alphabet
Q_σ	boolean vector indicating for each position of a word if $\sigma \in \Sigma$ appears in this position
$P[i]$	i-th entry of boolean vector P

Description Logic

N_c	set of primitive concept names
N_r	set of primitive role names
I	description logic interpretation $I = (\Delta, \cdot^I)$
Δ	(finite) domain of the interpretation I
$\mathcal{L}$	description logic induced by operators in Figures 2.1 and 2.2
$\mathcal{L}^+$	description logic induced by operators in Figures 2.1 and 2.2 with transitive closure
F^I	evaluation of a dl-formula by an interpretation I

NDLM

$enc(I)$	encoding of a description logic interpretation into tensors (C,R)
$NDLM^s$	NDLM model in strict mode using min and max aggregations
$NDLM^r$	NDLM model in relaxed mode using sum aggregations
$NDLM^{s+}$	NDLM model in strict mode with transitive closure
$NDLM^{r+}$	NDLM model in relaxed mode with transitive closure
U_i, B_i	equivalence relations after layer i
$[o]_i$	equivalence class of element o after layer i
$[o_1, o_2]_i$	equivalence class of elements (o_1, o_2) after layer i

List of Figures

List of Tables

List of Algorithms

List of Listings

List of References

[1] F. Baader, "Description Logic Terminology," in *The Description Logic Handbook: Theory, Implementation, and Applications*, F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, Eds., Cambridge University Press, 2003, pp. 485–495.

[2] B. Bonet and H. Geffner, "General Policies, Representations, and Planning Width," en, *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 35, no. 13, pp. 11 764–11 773, May 2021, ISSN: 2374-3468. DOI: 10.1609/aaai.v35i13.17398 [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/17398

[3] J.-Y. Cai, M. Fürer, and N. Immerman, "An optimal lower bound on the number of variables for graph identification," en, *Combinatorica*, vol. 12, no. 4, pp. 389–410, Dec. 1992, ISSN: 1439-6912. DOI: 10.1007/BF01305232 [Online]. Available: https://doi.org/10.1007/BF01305232

[4] F. Chollet, *On the Measure of Intelligence*, arXiv:1911.01547 [cs], Nov. 2019. DOI: 10.48550/arXiv.1911.01547 [Online]. Available: http://arxiv.org/abs/1911.01547

[5] K. van Deemter, A. Gatt, I. van der Sluis, and R. Power, "Generation of referring expressions: Assessing the Incremental Algorithm," eng, *Cognitive Science*, vol. 36, no. 5, pp. 799–836, Jul. 2012, ISSN: 1551-6709. DOI: 10.1111/j.1551-6709.2011.01205.x

[6] H. Dong, J. Mao, T. Lin, C. Wang, L. Li, and D. Zhou, "Neural logic machines," *CoRR*, vol. abs/1904.11694, 2019. arXiv: 1904.11694. [Online]. Available: http://arxiv.org/abs/1904.11694

[7] D. Drexler, G. Francès, and J. Seipp, *Description Logics State Features for Planning (DLPlan)*, Jan. 2022. DOI: 10.5281/ZENODO.5826139 [Online]. Available: https://zenodo.org/record/5826139

[8] R. E. Fikes and N. J. Nilsson, "Strips: A new approach to the application of theorem proving to problem solving," en, *Artificial Intelligence*, vol. 2, no. 3-4, pp. 189–208, Dec. 1971, ISSN: 00043702. DOI: 10.1016/0004-3702(71)90010-5 [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0004370271900105

[9] N. Jansen, J. Gösgens, and H. Geffner, *Learning Lifted Action Models From Traces of Incomplete Actions and States*, arXiv:2508.21449 [cs], Aug. 2025. DOI: 10.48550/arXiv.2508.21449 [Online]. Available: http://arxiv.org/abs/2508.21449

[10] B. Khemani, S. Patil, K. Kotecha, and S. Tanwar, "A review of graph neural networks: Concepts, architectures, techniques, challenges, datasets, applications, and future directions," en, *Journal of Big Data*, vol. 11, no. 1, p. 18, Jan. 2024, ISSN: 2196-1115. DOI: 10.1186/s40537-023-00876-4 [Online]. Available: https://doi.org/10.1186/s40537-023-00876-4

[11] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, arXiv:1412.6980 [cs], Jan. 2017. DOI: 10.48550/arXiv.1412.6980 [Online]. Available: http://arxiv.org/abs/1412.6980

[12] T. N. Kipf and M. Welling, *Semi-Supervised Classification with Graph Convolutional Networks*, en, arXiv:1609.02907 [cs.LG], Feb. 2017. DOI: 10.48550/arXiv.1609.02907 [Online]. Available: http://arxiv.org/abs/1609.02907

[13] M. Martín and H. Geffner, "Learning Generalized Policies from Planning Examples Using Concept Languages," en, *Applied Intelligence*, vol. 20, no. 1, pp. 9–19, Jan. 2004, ISSN: 1573-7497. DOI: 10.1023/B:APIN.0000011138.20292.dd [Online]. Available: https://doi.org/10.1023/B:APIN.0000011138.20292.dd

[14] C. Morris, M. Ritzert, M. Fey, W. L. Hamilton, J. E. Lenssen, G. Rattan, and M. Grohe, "Weisfeiler and leman go neural: Higher-order graph neural networks," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 33, 2019, pp. 4602–4609. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/4384

[15] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, *PyTorch: An Imperative Style, High-Performance Deep Learning Library*, arXiv:1912.01703 [cs], Dec. 2019. DOI: 10.48550/arXiv.1912.01703 [Online]. Available: http://arxiv.org/abs/1912.01703

[16] Y. Sarrof, Y. Du, K. Stein, A. Koller, S. Thiébaux, and M. Hahn, *On the Ability of Transformers to Verify Plans*, arXiv:2603.19954 [cs], Mar. 2026. DOI: 10.48550/arXiv.2603.19954 [Online]. Available: http://arxiv.org/abs/2603.19954

[17] A. F. Spies, W. Edwards, M. Ivanitskiy, A. Skapars, T. Rauker, K. Inoue, A. Russo, and M. Shanahan, "TRANSFORMERS USE CAUSAL WORLD MODELS IN MAZE-SOLVING TASKS," en, 2025.

[18] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph Attention Networks," en, Feb. 2018. [Online]. Available: https://openreview.net/forum?id=rJXMpikCZ

[19] Y. Xu, W. Li, P. Vaezipoor, S. Sanner, and E. B. Khalil, *LLMs and the Abstraction and Reasoning Corpus: Successes, Failures, and the Importance of Object-based Representations*, arXiv:2305.18354 [cs], Feb. 2024. DOI: 10.48550/arXiv.2305.18354 [Online]. Available: http://arxiv.org/abs/2305.18354

[20] A. Yang, D. Chiang, and D. Angluin, *Masked Hard-Attention Transformers Recognize Exactly the Star-Free Languages*, arXiv:2310.13897 [cs], Oct. 2024. DOI: 10.48550/arXiv.2310.13897 [Online]. Available: http://arxiv.org/abs/2310.13897

[21] M. Zimmer, X. Feng, C. Glanois, Z. Jiang, J. Zhang, P. Weng, D. Li, J. Hao, and W. Liu, *Differentiable Logic Machines*, arXiv:2102.11529 [cs], Jul. 2023. DOI: 10.48550/arXiv.2102.11529 [Online]. Available: http://arxiv.org/abs/2102.11529