Machine Learning (2023) 112:1465–1497
https://doi.org/10.1007/s10994-023-06320-1

αILP: thinking visual scenes as differentiable logic programs

Hikaru Shindo[1] · Viktor Pfanschilling[1] · Devendra Singh Dhami[1,2] · Kristian Kersting[1,2,3]

Received: 7 June 2022 / Revised: 29 November 2022 / Accepted: 15 February 2023 /
Published online: 14 March 2023

Abstract
Deep neural learning has shown remarkable performance at learning representations for visual object categorization. However, deep neural networks such as CNNs do not explicitly encode objects and relations among them. This limits their success on tasks that require a deep logical understanding of visual scenes, such as Kandinsky patterns and Bongard problems. To overcome these limitations, we introduce *αILP*, a novel differentiable inductive logic programming framework that learns to represent scenes as logic programs—intuitively, logical atoms correspond to objects, attributes, and relations, and clauses encode high-level scene information. αILP has an end-to-end reasoning architecture from visual inputs. Using it, αILP performs differentiable inductive logic programming on complex visual scenes, i.e., the logical rules are learned by gradient descent. Our extensive experiments on *Kandinsky patterns* and *CLEVR-Hans* benchmarks demonstrate the accuracy and efficiency of *αILP* in learning complex visual-logical concepts.

Keywords Differentiable reasoning · Inductive logic programming · Object-centric learning · Neuro-symbolic AI

Editors: Alireza Tamaddoni-Nezhad, Alan Bundy, Luc De Raedt, Artur d'Avila Garcez, Sebastijan Dumančić, Cèsar Ferri, Pascal Hitzler, Nikos Katzouris, Denis Mareschal, Stephen Muggleton, Ute Schmid.

✉ Hikaru Shindo
hikaru.shindo@cs.tu-darmstadt.de

Viktor Pfanschilling
viktor.pfanschilling@cs.tu-darmstadt.de

Devendra Singh Dhami
devendra.dhami@cs.tu-darmstadt.de

Kristian Kersting
kersting@cs.tu-darmstadt.de

[1] TU Darmstadt, Darmstadt, Germany

[2] Hessian Center for AI (hessian.AI), Darmstadt, Germany

[3] Centre for Cognitive Science, TU Darmstadt, Darmstadt, Germany

1 Introduction

Understanding visual scenes is a fundamental problem in building an intelligent agent. Deep Neural Networks such as Convolutional Neural Networks (CNNs) have succeeded in many visual-perception benchmarks but produce poor performance in complex visual scenes, where several *objects* appear in an image, and the agent needs to reason and learn about the *attributes* and *relations*. CNN-based models do not explicitly encode objects and relations, and thus often fail to capture the patterns defined in complex visual scenes.

Kandinsky patterns (Holzinger et al., 2019; Müller & Holzinger, 2021; Holzinger et al., 2021) have been proposed to assess the ability of intelligent systems to explain complex visual scenes. In a similar vein, CLEVR-Hans (Stammer et al., 2021) has been proposed to assess the ability of a model to understand confounded visual scenes. CNN-based models cannot produce proper explanations in such cases and can also suffer from the problem of confounding factors. Moreover, they are data-hungry and struggle to learn abstract visual relations (Kim et al., 2018). A natural question thus arises: How can we build an intelligent system avoiding these pitfalls?

To build a system overcoming the shortages of CNN-based models, *Neuro-Symbolic* approaches (Besold et al., 2017; d'Avila Garcez & Lamb, 2020; Tsamoura et al., 2021) have emerged, where symbolic computations are integrated with neural networks. As logic-based neuro-symbolic systems, many frameworks have been proposed, e.g., DeepProblog (Manhaeve et al., 2018, 2021), NeurASP (Yang et al., 2020), and ∂ILP (Evans & Grefenstette, 2018). However, previous studies are not capable of complete structure learning from visual input (Manhaeve et al., 2018, 2021; Yang et al., 2020) or not capable of handling complex rules and visual scenes (Evans & Grefenstette, 2018). Therefore, structure learning on complex scenes such as Kandinsky patterns (Holzinger et al., 2019; Müller & Holzinger, 2021; Holzinger et al., 2021) and CLEVR-Hans (Stammer et al., 2021) problems is difficult, if not impossible, using these frameworks.

To mitigate this issue, we propose *αILP*,[1] a novel differentiable Inductive Logic Programming (ILP) framework that combines object-centric perception with ILP (Muggleton, 1991, 1995; Nienhuys-Cheng et al., 1997; Cropper et al., 2022), establishing one of the first in the 4th type of neuro-symbolic system, i.e., *Neuro:Symbolic→Neuro*, as proposed by Kautz (2022). αILP maps output of neural networks (Neuro) to symbolic representations (Symbolic), then gradient-based learning is performed on top of it (Neuro). αILP performs structure learning, i.e., learns discrete logic programs from complex visual scenes. To this end, our system is an extension of Neuro-Symbolic systems such as DeepProblog (Manhaeve et al., 2018, 2021) and ∂ILP (Evans & Grefenstette, 2018).

αILP has an end-to-end reasoning architecture from visual input, which consists of *three* main components: (i) visual perception module, (ii) facts converter, and (iii) differentiable reasoning module. The facts converter converts the output of the visual perception module into the form of probabilistic facts, which can be fed into the reasoning module. Then, the reasoning module performs differentiable forward-chaining inference from a given set of facts. It computes the set of facts that can be deduced from the given set of facts and weighted logical rules (Evans & Grefenstette, 2018; Shindo et al., 2021). The final prediction can be made based on the result of the forward-chaining inference. αILP learns

[1] The name αILP is inspired by "the eyes of Argus" from Greek mythology since we primarily deal with visual scenes.

logic programs that encode high-level scene information by differentiable ILP techniques (Shindo et al., 2021). It generates candidates of clauses by top-k beam search and learns the weights for the clauses by backpropagation.

Overall, we make a number of key contributions: (1) We propose αILP, a novel framework that performs differentiable ILP from visual scenes. (2) To establish αILP, we propose an end-to-end reasoning architecture from visual inputs. It performs differentiable forward-chaining inference for visual scenes by using perception models and a facts-converting algorithm. (3) We also propose a learning scheme for αILP to perform differentiable ILP for complex visual scenes. It integrates differentiable ILP techniques with the visual domain, i.e., generates clauses efficiently and performs gradient-based optimization from complex visual scenes. (4) We empirically show the following advantages of α ILP: (i) αILP solves ILP problems in visual scenes, i.e., Kandinsky patternss (Holzinger et al., 2019; Müller & Holzinger, 2021; Holzinger et al., 2021) and CLEVR-Hans (Stammer et al., 2021), with high accuracy outperforming neural baseline models. (ii) αILP can generate explanations, i.e., produces a readable solution in the form of logic programs. (iii) αILP is robust to confounding, i.e., avoids being over-fitted to confounding factors. (iv) α ILP is data-efficient, i.e., reports no performance drop even when using 10% of the training data. (v) αILP can perform fast inference. It supports efficient parallelized batch computation on GPUs, therefore, it can classify a large number of instances in a large dataset quickly.

2 Background and related work

We use bold lowercase letters $\mathbf{v}, \mathbf{w}, \ldots$ for vectors. We use bold capital letters $\mathbf{X}, \ldots$ for tensors. We use calligraphic letters $\mathcal{C}, \mathcal{A}, \ldots$ for (ordered) sets and typewriter font $\mathtt{p(X, Y)}$ for terms and predicates in logical expressions.

2.1 Preliminaries on logic and ILP

Language $\mathcal{L}$ is a tuple $(\mathcal{P}, \mathcal{F}, \mathcal{T}, \mathcal{V})$, where $\mathcal{P}$ is a set of predicates, $\mathcal{F}$ is a set of function symbols, $\mathcal{T}$ is a set of constants, and $\mathcal{V}$ is a set of variables. A *term* is a constant, a variable, or an expression $\mathtt{f}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$ where $\mathtt{f}$ is a n-ary function symbol and $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are terms. We denote n-ary predicate $\mathtt{p}$ by $\mathtt{p}/(n, [\mathtt{dt}_1, \ldots, \mathtt{dt}_n])$, where $\mathtt{dt}_i$ is the datatype of the i-th argument. An *atom* is a formula $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n)$, where $\mathtt{p}$ is an n-ary predicate symbol and $\mathtt{t}_1, \ldots, \mathtt{t}_n$ are terms. A *ground atom* or simply a *fact* is an atom with no variables. A *literal* is an atom or its negation. A *positive literal* is just an atom. A *negative literal* is the negation of an atom. A *clause* is a finite disjunction ($\vee$) of literals. A *definite clause* is a clause with exactly one positive literal. If $A, B_1, \ldots, B_n$ are atoms, then $A \vee \neg B_1 \vee \cdots \vee \neg B_n$ is a definite clause. We write definite clauses in the form of A :- $B_1, \ldots, B_n$. Atom A is called the *head*, and the set of negative atoms $\{B_1, \ldots, B_n\}$ is called the *body*. We denote the special constant *true* as $\top$ and *false* as $\bot$. An atom is an atomic *formula*. For formula F and G, $\neg F$, $F \wedge G$, and $F \vee G$ are also formulas. *Interpretation* of language $\mathcal{L}$ is a tuple $(\mathcal{D}, \mathcal{I}_\mathcal{A}, \mathcal{I}_\mathcal{F}, \mathcal{I}_\mathcal{P})$, where $\mathcal{D}$ is the domain, $\mathcal{I}_\mathcal{A}$ is the assignments of an element in $\mathcal{D}$ for each constant $\mathtt{a} \in \mathcal{A}$, $\mathcal{I}_\mathcal{F}$ is the assignments of a function from $\mathcal{D}^n$ to $\mathcal{D}$ for each n-ary function symbol $\mathtt{f} \in \mathcal{F}$, and $\mathcal{I}_\mathcal{P}$ is the assignments of a function from $\mathcal{D}^n$ to $\{\top, \bot\}$ for each n-ary predicate $\mathtt{p} \in \mathcal{P}$. For language $\mathcal{L}$ and formula F, an interpretation $\mathcal{I}$ is a *model* if the truth value of F w.r.t $\mathcal{I}$ is true. Formula F is a *logical consequence* or *logical entailment* of a set of formulas $\mathcal{S}$,

denoted $\mathcal{S} \models F$, if, $\mathcal{I}$ is a model for $\mathcal{S}$ implies that $\mathcal{I}$ is a model for F for every interpretation $\mathcal{I}$ of $\mathcal{L}$.

An ILP problem $\mathcal{Q}$ is a tuple $(\mathcal{E}^+, \mathcal{E}^-, \mathcal{B}, \mathcal{L})$, where $\mathcal{E}^+$ is a set of positive examples, $\mathcal{E}^-$ is a set of negative examples, $\mathcal{B}$ is background knowledge, and $\mathcal{L}$ is a language. Background knowledge can be given in the form of the set of facts or clauses. The solution to an ILP problem is a set of definite clauses $\mathcal{H} \subseteq \mathcal{L}$ that satisfies the following conditions:

$\forall A \in \mathcal{E}^+ \; \mathcal{H} \cup \mathcal{B} \models A$ and $\forall A \in \mathcal{E}^- \; \mathcal{H} \cup \mathcal{B} \not\models A$.

2.2 Related work towards visual ILP

Over 50 years ago, M. M. Bongard, a Russian computer scientist, invented a collection of one hundred human-designed visual recognition tasks (Bongard & Hawkins, 1970), now named the Bongard Problems (BPs), to demonstrate the gap between high-level human cognition and computerized pattern recognition. Inspired by BPs, the Bongard-LOGO (Nie et al., 2020) problem has been proposed as a benchmark for the machine learning community. Kandinsky patterns (Holzinger et al., 2019; Müller & Holzinger, 2021; Holzinger et al., 2021) have been proposed to assess the ability of intelligent systems to explain complex visual scenes. In a similar vein, CLEVR-Hans (Stammer et al., 2021) has been proposed to assess the ability of the model to understand the confounded visual scenes. These benchmarks present a challenge to CNN-based recognition models.

Logic, both propositional and first-order, is an established framework for performing reasoning on machines (Lloyd, 1984; Kowalski, 1988). A pioneering study of inductive inference on logic was done in the early 70 s (Plotkin, 1971). The Model Inference System (MIS) (Shapiro, 1983) has been implemented as an efficient search algorithm for logic programs. Inductive Logic Programming (Muggleton, 1991, 1995; Nienhuys-Cheng et al., 1997; Cropper et al., 2022) has emerged at the intersection of machine learning and logic programming. Many ILP frameworks have been developed, e.g., FOIL (Quinlan, 1990), Progol (Muggleton, 1995), ILASP (Law et al., 2014), Metagol (Cropper & Muggleton, 2016; Cropper et al., 2019), and Popper (Cropper & Morel, 2021). Symbolic ILP systems are dedicated to symbolic inputs. αILP deals with visual inputs by having an end-to-end neuro-symbolic reasoning architecture. αILP employs similar structure-learning techniques which have been developed for probabilistic logic programs (Bellodi & Riguzzi, 2015; Nguembang Fadja & Riguzzi, 2019) and performs learning on complex visual scenes. Different settings of probabilistic ILP approaches have been introduced in De Raedt et al. (2008). αILP is based on the learning from entailment setting, where the logical entailment is computed from probabilistic inputs. αILP computes the logical entailment with probabilistic values for facts and clauses in a differentiable manner.

The integration of symbolic programs and neural networks, which is called *Neuro-Symbolic computation* (Besold et al., 2017; d'Avila Garcez & Lamb, 2020; Tsamoura et al., 2021), has previously been addressed, e.g., DeepProblog (Manhaeve et al., 2018, 2021), NeurASP (Yang et al., 2020), ∂ILP (Evans & Grefenstette, 2018; Jiang & Luo, 2019), NS-CL (Mao et al., 2019), integration with abductive learning (Dai et al., 2019), and differentiable theorem provers (Rocktäschel & Riedel, 2017; Minervini et al., 2020). Kandinsky patterns and CLEVR-Hans cannot be solved easily by these frameworks because they require complete structure learning from complex visual scenes. DeepProblog supports structure learning but is limited for the sketching setting (Solar-Lezama, 2008; Bošnjak et al., 2017). αILP supports object-centric perception models, differentiable forward reasoning, and efficient clause search for solving tasks in complex visual scenes.

Some neuro-symbolic models have been developed for Visual Question Answering (VQA) (Antol et al., 2015; Johnson et al., 2017; Santoro et al., 2018; Mao et al., 2019; Amizadeh et al., 2020). In VQA-based models, the symbolic programs are determined by the natural language sentences that represent questions, but αILP does not have that assumption. Moreover, αILP stands in the line of probabilistic logic programming (De Raedt et al., 2016; Raedt et al., 2020). Therefore, αILP can employ methods for probabilistic logic programming, which have been developed in the community. Similar concepts of some key components of αILP have been investigated in previous studies, e.g., Neural Predicates (Diligenti et al., 2017; Donadello et al., 2017; Badreddine et al., 2022), weighted forward-chaining reasoning (Sourek et al., 2018; Si et al., 2019), and differentiable structure learning (Evans & Grefenstette, 2018; Sourek et al., 2017). αILP is the first that integrates these concepts for the visual object-centric domain as a consistent framework. Logic Tensor Networks (LTNs) (Badreddine et al., 2022) provide a unified differentiable language for first-order logic. LTNs map each term in first-order logic to numerical representations in place of *interpretation*. Then predicates are grounded to functions that take numerical representations of terms and return a truth value in [0, 1]. αILP takes a similar approach to connect the sub-symbolic and symbolic representations.

Object-centric learning is an approach to decomposing an input image into representations in terms of objects (Dittadi et al., 2022). This problem has been widely addressed in the computer vision community. Another approach is the unsupervised approach (Burgess et al., 2019; Engelcke et al., 2020; Locatello et al., 2020), where the models acquire the ability of object-perception without or with fewer annotations. αILP uses these object-centric models as a perception module.

Differentiable solvers for dynammic programming problems have been developed (Cuturi & Blondel, 2017; Mensch & Blondel, 2018). αILP adopts some techniques to achieve differentiable implementations of the discrete operations for first-order logic. Various types of differentiable logical operations have been also investigated (van Krieken et al., 2022; Sen et al., 2022).

3 αILP

We now introduce αILP in the following steps. First, we give an overview of the problem setting and the framework. Second, we explain the reasoning architecture of αILP consisting of (i) the visual-perception module, (ii) *facts converter*, an algorithm to convert object-centric representations into probabilistic facts, and (iii) the differentiable forward-reasoning mechanism. Finally, we describe the learning strategy on αILP to perform differentiable ILP on visual scenes.

What is visual ILP? We address the ILP problem in visual scenes, which is called *visual ILP problem*, where each example is given as an image containing several objects. The classification pattern is defined on high-level concepts such as attributes and relations of objects.

3.1 Architecture overview

Figure 1 illustrates an overview of αILP and consists of a *Reasoning* module and a *Learning* module. We now introduce these in detail.

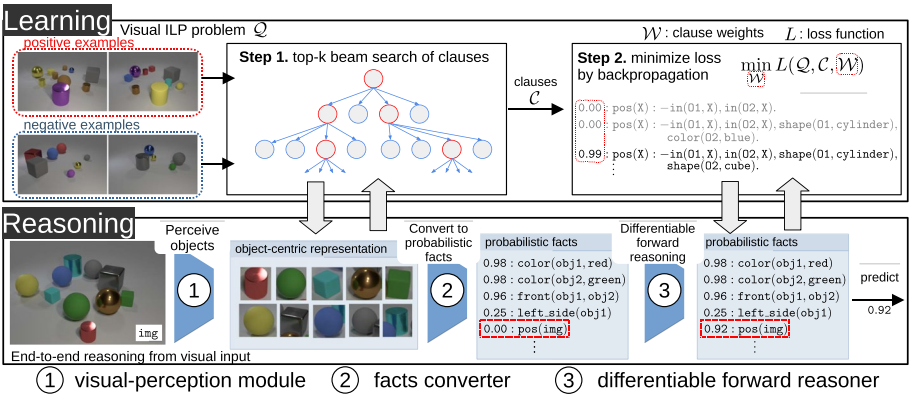


Fig. 1 An overview of αILP. **(Reasoning)**αILP has an end-to-end *reasoning* architecture from visual input based on differentiable forward reasoning. In the reasoning step, **(1)** The raw input images are factorized in terms of objects using the visual-perception model. **(2)** The object-centric representation is converted into a set of probabilistic facts. **(3)** The differentiable forward reasoning is performed using weighted clauses. **(Learning)** To solve the classification problem of visual scenes, we provide positive examples, negative examples, and background knowledge. Each example is given as a visual scene. αILP performs 2-steps learning as follows: **(Step1)** A set of candidates of clauses is generated by top-k beam search. The search is conducted from examples of visual scenes using the end-to-end reasoning architecture. **(Step2)** Then, the weights for the generated clauses are trained to minimize the loss function. By using the end-to-end reasoning architecture, αILP finds a logic program that explains the complex visual scenes by gradient descent

3.1.1 Reasoning

αILP has an end-to-end reasoning architecture, which works as follows: (i) The raw input images are factorized in terms of objects using the visual-perception model. (ii) The object-centric representation is converted into a set of probabilistic facts. (iii) The differentiable forward reasoning is performed using weighted clauses. The bottom row of Fig. 1 illustrates the reasoning architecture in αILP.

3.1.2 Learning

αILP learns logic programs from visual inputs by performing differentiable ILP, i.e., we provide positive examples, negative examples, and background knowledge. Each example is given as a visual scene. The top row of Fig. 1 illustrates the learning pipeline in αILP. Learning with αILP is as follows: **(Step1)** A set of candidates of clauses is generated by top-k beam search. The search is conducted from examples of visual scenes using the end-to-end reasoning architecture. **(Step2)** The weights for the generated clauses are trained to minimize the loss function. By using the end-to-end reasoning architecture, αILP finds a logic program that explains the complex visual scenes by gradient descent. We now describe our architecture in detail.

3.2 Visual perception

We make the minimal assumption that the perception network takes an image and returns a set of object-centric vectors, where each dimension represents an attribute of the object, e.g., colors, shapes, and positions. Thus, any type of neural network that segments the input images into the individual objects present in the image can be utilized. For example, αILP can employ a slot attention model (Locatello et al., 2020) for 3D scenes. However, with natural images, αILP can employ other established object-detection models such as YOLO (Redmon et al., 2016), Faster-RCNN (Ren et al., 2015), and Mask-RCNN (He et al., 2017). The visual-perception module is trained on randomly-generated figures with annotations about each object, i.e., the number of objects and the attributes of the objects are randomly determined.

3.3 Facts converter: lifting to symbolic representation

After the object-centric perception, αILP generates a logical representation, i.e., a set of probabilistic facts. We propose a new type of predicate that can refer to differentiable functions to compute the probability. We also present an algorithm to convert the perception result into probabilistic facts.

3.3.1 Neural predicate

To build a bridge between the sub-symbolic and symbolic representations, we provide a new type of predicate, which we term as *neural predicates*. A neural predicate is associated with a differentiable function, which we call *valuation function*, that produces the probability of the grounded facts.

Definition 1 A neural predicate $\mathtt{p}/(n, [\mathtt{dt}_1, \ldots, \mathtt{dt}_n])$ is a n-ary predicate associated with a *valuation function* $v_\mathtt{p} : \mathbb{R}^{d_1 \times \cdots \times d_n} \to \mathbb{R}$, where $\mathtt{dt}_i$ is the datatype of the i-th argument, $d_i \in \mathbb{N}$ is the dimension of the vector representation of the term whose datatype is $\mathtt{dt}_i$.

Intuitively, we give the first-order logic *interpretation* for neural predicates and terms as follows: (i) each neural predicate is assigned to a function in a vector space, (ii) each term in the arguments of neural predicates is assigned to a vector. The vector can be an output of neural networks, or an encoding of the term, e.g., one-hot encoding of the attributes.

3.3.2 Facts-converting algorithm

The facts converter produces a set of probabilistic facts from the output of the perception module. Let $\mathcal{G}$ be the set of all facts; then the conversion proceeds as follows: For each fact $\mathtt{p}(\mathtt{t}_1, \ldots, \mathtt{t}_n) \in \mathcal{G}$, if it consists of a neural predicate, then the corresponding valuation function $v_\mathtt{p}$ is called to compute the probability of the fact. Otherwise, zero is given as the probability of the fact. If the fact is in the background knowledge, one is given as the probability of the fact. The valuation function maps each term $\mathtt{t}_1, \ldots, \mathtt{t}_n$ to vector representations according to the interpretation. The forward reasoning function

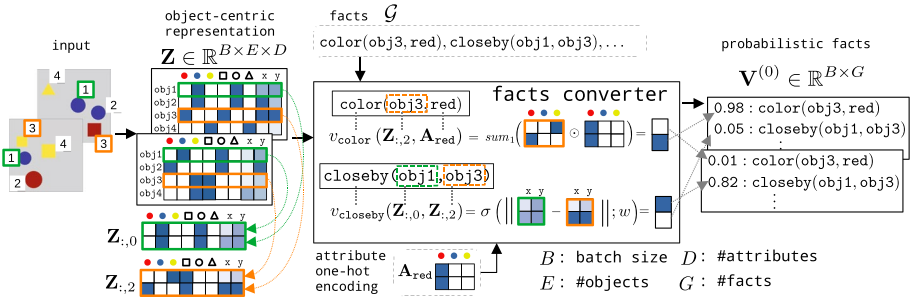


Fig. 2 An illustration of the facts converter in αILP. It decomposes the raw-input images into object-centric representations (left). The valuation functions are called to compute the probability of facts. Each term in the arguments is mapped to a vector representation (middle). The result is converted into the form of probabilistic facts (right)

requires a vector that maps each fact to a probabilistic value to achieve the differentiable computation. Thus, the probabilistic values are computed for all of the facts.

Figure 2 illustrates an example of the implementation of the facts converter. We assume that the perception model produces the probabilities of the attributes (color, shape, position) for each object. (1) For neural predicate `color`/(2, [`object`, `color`]), we compute the probability of fact `color`(`obj3`, `red`) by calling the valuation function $v_{\texttt{color}}$. Term `obj3` is mapped to the output of the perception module, and term `red` is mapped to its one-hot encoding. By using these vector representations of the terms, $v_{\texttt{color}}$ computes the probability of the atom, simply performing the tensor multiplication and summation. (2)For neural predicate `closeby`(2/[`object`, `object`]), we compute the probability of fact `closeby`(`obj1`, `obj3`) by calling the valuation function $v_{\texttt{closeby}}$. Term `obj1` and `obj3` are mapped to the corresponding output of the perception model, respectively. Then the positional information is extracted, and logistic regression is performed on the distance between two data points. By adapting the weights of the linear transformation, the facts converter can learn the concept of `closeby` flexibly. We note that the valuation functions of neural predicates are defined by the user, and parameterized valuation functions are trained before performing structure learning.

3.4 Differentiable forward-chaining inference

Forward-chaining inference is a type of inference in first-order logic to compute logical entailment (Russell & Norvig, 2009). For example, let $\mathcal{C}$ be a set of clauses and $\mathcal{G}$ be a set of all known facts. Then, forward-chaining inference can compute the set of facts $\mathcal{F}$ such that $\mathcal{C} \cup \mathcal{G} \models \mathcal{F}$. Differentiable forward-chaining inference (Evans & Grefenstette, 2018; Shindo et al., 2021) computes the logical entailment in a differentiable manner. We briefly summarize the steps: **(Step 1)** A tensor that holds the relationships between clauses and facts is computed. **(Step 2)** Each clause is compiled into a differentiable function that performs forward reasoning using the tensor. **(Step 3)** A differentiable logic program is composed of the clause functions and their weights. T-time step inference is computed by amalgamating the inference results recursively.

3.4.1 Tensor encoding

Following (Shindo et al., 2021), we build a tensor holding relationships between clauses $\mathcal{C}$ and facts $\mathcal{G}$. We assume that $\mathcal{C}$ and $\mathcal{G}$ are ordered sets, i.e., where every element has its own index. Let L be the maximum body length in $\mathcal{C}$, S be the maximum number of substitutions for existentially quantified variables in clauses $\mathcal{C}$, $C = |\mathcal{C}|$ and $G = |\mathcal{G}|$. *Index tensor* $\mathbf{I} \in \mathbb{N}^{C \times G \times S \times L}$ contains the indices of the facts to compute forward inferences. Intuitively, $\mathbf{I}_{i,j,k,l}$ is the index of the l-th fact (subgoal) in the body of the i-th clause to derive the j-th fact with the k-th substitution for existentially quantified variables.

Example

Let $R_0 = \texttt{pos(X)} \texttt{:-} \texttt{in(O1, X), color(O1, red)} \in \mathcal{C}$ and $F_2 = \texttt{pos(img)} \in \mathcal{G}$, and we assume that terms of objects are $\{\texttt{obj1}, \texttt{obj2}\}$. To compute the subgoals for fact F_2 and clause R_0, F_2 and the head atom can be unified by substitution $\theta = \{\texttt{X} = \texttt{img}\}$. By applying θ to body atoms, we get clause $\texttt{pos(img)} \texttt{:-} \texttt{in(O1, img), color(O1, red)}.$, which has an existentially quantified variable O1. By considering the possible substitutions for O1, namely O1/obj1 and O1/obj2, we have grounded clauses, as shown on top of Table 1. Bottom rows of Table 1 shows elements of tensor $\mathbf{I}_{0,:,0,:}$ and $\mathbf{I}_{0,:,1,:}$. Facts $\mathcal{G}$ and the indices are represented on the upper rows in the table. For example, $\mathbf{I}_{0,2,0,:} = [3, 5]$ because R_0 entails pos(img) with substitution $\theta = \{\texttt{O1} = \texttt{obj1}\}$. Then the subgoal atoms are $\{\texttt{in(obj1, img)}, \texttt{color(obj1, red)}\}$, which have indices [3, 5], respectively. The atoms which have a different predicate, e.g., shape(obj1, square), will never be entailed by clause R_0. Therefore, the corresponding values are filled with 0, which represents the index of the *false* atom.

3.4.2 Valuation

The valuation vector $\mathbf{v}^{(t)} \in \mathbb{R}^G$ maps each fact into a continuous value at each time step t. Each value $\mathbf{v}_i^{(t)}$ represents the probability of fact $F_i \in \mathcal{G}$. The differentiable inference is performed based on valuation vectors. To compute the T-step forward-chaining inference, we compute the sequence of valuation vectors $\mathbf{v}^{(0)}, \ldots, \mathbf{v}^{(T)}$. We denote a batch of valuation vectors at time step t as $\mathbf{V}^{(t)} \in \mathbb{R}^{B \times G}$, where B is the batch size. In logical reasoning, the parallelized batch computation is non-trivial. Thus, we explicitly denote the dimension of the batch in this section.

3.4.3 Clause function

Each clause $R_i \in \mathcal{C}$ is compiled into a clause function. The clause function takes a valuation vector $\mathbf{V}^{(t)}$, and returns a valuation vector $\mathbf{C}_i^{(t)} \in \mathbb{R}^{B \times G}$, which is the result of 1-step forward reasoning using R_i and $\mathbf{V}^{(t)}$. The clause function is computed as follows. Let $\mathbf{I} \in \mathbb{N}^{C \times G \times S \times L}$ be an index tensor. First, tensor $\mathbf{I}_i \in \mathbb{N}^{G \times S \times L}$ is extended for batches, i.e., $\tilde{\mathbf{I}}_i \in \mathbb{N}^{B \times G \times S \times L}$, and $\mathbf{V}^{(t)} \in \mathbb{R}^{B \times G}$ is extended to the same shape, i.e., $\tilde{\mathbf{V}}^{(t)} \in \mathbb{R}^{B \times G \times S \times L}$. Using these tensors, the clause function is computed as:

$$\mathbf{C}_i^{(t)} = softor_2^{\gamma}(prod_3(gather_1(\tilde{\mathbf{V}}^{(t)}, \tilde{\mathbf{I}}_i)), \tag{1}$$

where $gather_1(\mathbf{X}, \mathbf{Y})_{i,j,k,l} = \mathbf{X}_{i,\mathbf{Y}_{i,j,k,l},k,l}$, and $prod_3$ returns the product along dimension 3. $softor_d^{\gamma}$ is a function for taking logical *or* softly along dimension d:

Table 1 Example of grounded clauses (top) and elements in the index tensor (bottom)

$(k = 0)$	`pos(img) :- in(obj1, img), color(obj1, red)`					
$(k = 1)$	`pos(img) :- in(obj2, img), color(obj2, red)`					
j	0	1	2	3		4
G_j	$\bot$	$\top$	`pos(img)`	`in(obj1, img)`		`in(obj2, img)`
$\mathbf{I}_{0,j,0,:}$	[0, 0]	[1, 1]	[3, 5]	[0, 0]		[0, 0]
$\mathbf{I}_{0,j,1,:}$	[0, 0]	[1, 1]	[4, 6]	[0, 0]		[0, 0]
j		5		6	–	
G_j		`color(obj1, red)`		`color(obj2, red)`	–	
$\mathbf{I}_{0,j,0,:}$		[0, 0]		[0, 0]	–	
$\mathbf{I}_{0,j,1,:}$		[0, 0]		[0, 0]	–	

Each fact has its index, and index tensor contains the indices of the facts to compute forward inferences

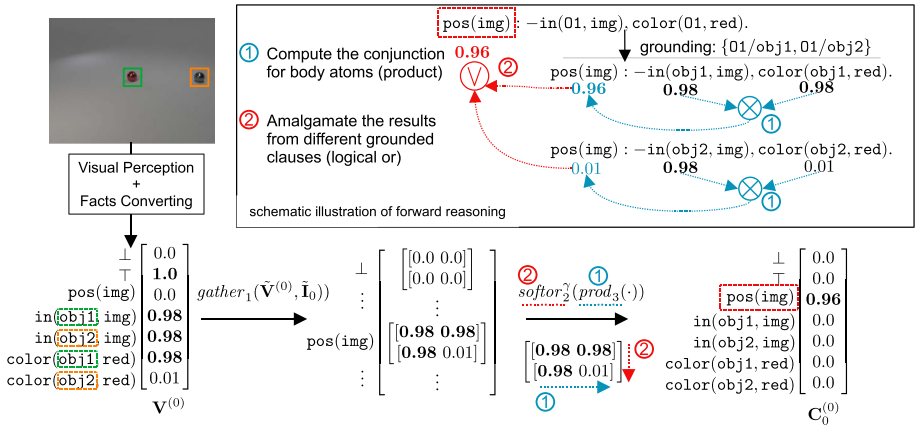


Fig. 3 An illustration of the clause function for clause $R_0 = \texttt{pos(X)}\texttt{:-in(O1, X), color(O1, red)}$. with a schematic illustration of the forward reasoning (top-right). The perception module and facts converter produce initial valuation vector $\mathbf{V}^{(0)}$. For each grounded clause, the probability for the subgoals is extracted by the *gather* function. Then the product for the body atoms is computed, then the logical *or* is computed softly to amalgamate the results from different groundings of clauses. For simplicity, the first dimension for the batch is removed in the figure (Color figure online)

$$softor_d^{\gamma}(\mathbf{X}) = \frac{1}{S}\gamma \log\left(sum_d \exp\left(\mathbf{X}/\gamma\right)\right), \tag{2}$$

where $\gamma > 0$ is a smooth parameter, sum_d is the sum function along dimension d, and $S = max(1.0, max\left(\gamma \log sum_d \exp\left(\mathbf{X}/\gamma\right)\right)$. Normalization term S ensures that the function returns the normalized probabilistic values. More details of the function is in Appendix D. In Eq. 1, applying the $softor_2^{\gamma}$ function corresponds to considering all possible substitutions for existentially quantified variables in the body atoms of the clause and taking *logical or* softly over the results of possible substitutions. The results from each clause is stacked into tensor $\mathbf{C}^{(t)} \in \mathbb{R}^{C\times B\times G}$, i.e., $\mathbf{C}^{(t)} = stack_0(\mathbf{C}_1^{(t)}, \ldots, \mathbf{C}_C^{(t)})$, where $stack_0$ is stack function for tensors along dimension 0.

Figure 3 illustrates the clause function. A clause function computes the forward-chaining inference for a clause. The perception module and facts converter produce an initial valuation vector $\mathbf{V}^{(0)}$. For each grounded clause, the probability for the subgoals is extracted by the *gather* function. Then the product for the body atoms is computed, then the logical *or* is computed softly to amalgamate the results from different groundings of clauses.

3.4.4 Soft (logic) program composition

In αILP, a logic program is represented smoothly as a weighted sum of the clause functions following (Shindo et al., 2021). Intuitively, αILP has M distinct weights for each clause, i.e., $\mathbf{W} \in \mathbb{R}^{M\times C}$. By taking softmax of $\mathbf{W}$ along dimension 1, M clauses are softly chosen from C clauses. The weighted sum of clause functions is computed as follows. First, we take the softmax of the clause weights $\mathbf{W} \in \mathbb{R}^{M\times C}$: $\mathbf{W}^* = softmax_1(\mathbf{W})$ where $softmax_1$ is a softmax function over dimension 1. The clause weights $\mathbf{W}^* \in \mathbb{R}^{M\times C}$ and the output of the clause function $\mathbf{C}^{(t)} \in \mathbb{R}^{C\times B\times G}$ are expanded to the same shape $\tilde{\mathbf{W}}^*, \tilde{\mathbf{C}}^{(t)} \in \mathbb{R}^{M\times C\times B\times G}$. Then we compute tensor $\mathbf{H}^{(t)} \in \mathbb{R}^{M\times B\times G}$: $\mathbf{H}^{(t)} = sum_1(\tilde{\mathbf{W}}^* \odot \tilde{\mathbf{C}}^{(t)})$, where $\odot$ is element-wise multiplication,

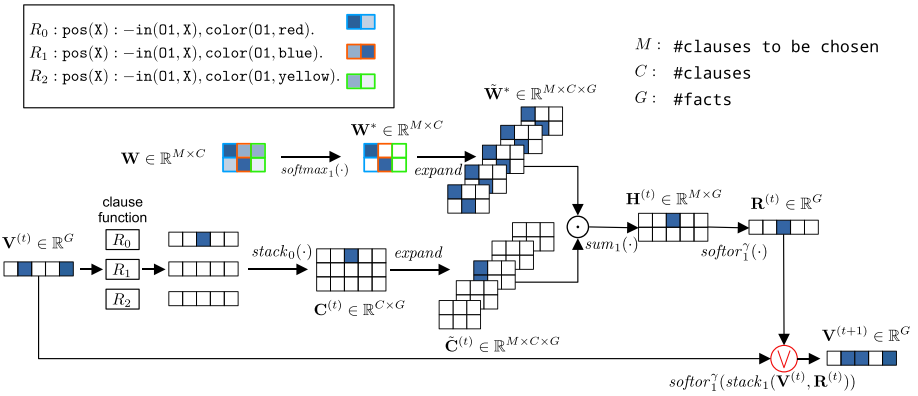


Fig. 4 An illustration of differentiable forward-chaining reasoning. Each clause is compiled into a clause function. Each clause has M distinct weights. The input valuation vector $\mathbf{V}^{(t)}$ is fed to the clause function. By applying tensor operations, the forward-chaining reasoning is computed using weighted clauses. More details are in the main text. For simplicity, the dimension for the batch is removed in the figure

and sum_1 is a summation along dimension 1. Each value $\mathbf{H}^{(t)}_{i,j,k}$ represents the result for the k-th fact using the i-th clause weights for the j-th example in the batch. Finally, we compute tensor $\mathbf{R}^{(t)} \in \mathbb{R}^{B \times G}$ corresponding to the fact that logic program is a set of clauses: $\mathbf{R}^{(t)} = softor_0^\gamma(\mathbf{H}^{(t)})$, taking logical *or* softly over M-chosen clauses. To compute the multi-step reasoning, $\mathbf{V}^{(t+1)}$ is computed as: $\mathbf{V}^{(t+1)} = softor_1^\gamma(stack_1(\mathbf{V}^{(t)}, \mathbf{R}^{(t)}))$. The reasoning process is illustrated in Fig. 4.

3.4.5 Prediction

We assume that language $\mathcal{L}$ has a constant that represents the input image and a predicate to compose an atom representing that the input is positive, e.g., `pos(img)` $\in \mathcal{G}$. For given visual input e, αILP simply extracts the value from the result of the forward reasoning to predict class label $y \in \{0, 1\}$ as follows:

$$p(y \mid e, \mathcal{C}, \mathcal{B}, \mathcal{W}, \Theta_{per}, \Theta_{np}) = \mathbf{v}^{(T)}[I_{\mathcal{G}}(\texttt{pos(img)})], \tag{3}$$

where $\mathcal{C}$ is a set of clauses, $\mathcal{B}$ is background knowledge, $\mathcal{W}$ is a set of clause weights, Θ_{per} is a set of parameters for the visual-perception model, and Θ_{np} is a set of parameters for neural predicates, $I_{\mathcal{G}}(x)$ a function that returns the index of x in $\mathcal{G}$, and $\mathbf{v}[i]$ is the i-th element of $\mathbf{v}$, i.e., $\mathbf{v}_i$. αILP accepts background knowledge as a set of facts and clauses.

3.5 Program induction from visual scenes

αILP learns differentiable logic programs that describe complex visual scenes. We basically follow the differentiable ILP setting (Evans & Grefenstette, 2018; Shindo et al., 2021), where an ILP problem is formulated as an optimization problem that has the following general form:

$$\min_{\mathcal{W}} loss(\mathcal{Q}, \mathcal{C}, \mathcal{W}), \tag{4}$$

where $\mathcal{Q}$ is an ILP problem, $\mathcal{C}$ is a set of candidates of clauses, $\mathcal{W}$ is a set of weights for clauses, and *loss* is a loss function that returns a penalty when training constraints are violated. We note that we solve visual ILP problems, where each positive and negative example is an image containing several objects.

Algorithm 1 Learning in αILP

Input: $\mathcal{D}_{perception}, \mathcal{Q}$

1: $\Theta^*_{per}, \Theta^*_{np} \leftarrow f_{train}(\mathcal{D}_{perception})$ ▷ `train the perception model and neural predicates`

2: $\mathcal{C} \leftarrow f_{top-k\_beam\_search}(\mathcal{Q}, \Theta^*_{per}, \Theta^*_{np})$ ▷ `generate clauses by top-k beam search`

3: $\mathcal{W}^* \leftarrow \arg\min_{\mathcal{W}} loss(\mathcal{Q}, \mathcal{C}, \mathcal{W}, \Theta^*_{per}, \Theta^*_{np})$ ▷ `minimize the loss w.r.t. clause weights`

4: $\mathcal{C}^* \leftarrow f_{get\_clauses}(\mathcal{C}, \mathcal{W}^*)$ ▷ `extract clauses by discretizing the clause weights`

Output: $\mathcal{C}^*$

Algorithm 1 describes the learning process of αILP. **(Line 1)** The perception model is trained using perception dataset $\mathcal{D}_{perception}$, which consists of pattern-free figures. The dataset is annotated for objects, e.g., class labels. Parameterized neural predicates can also be trained by visual input with a trained perception module or by scene graphs. **(Line 2–4)** Finally, the logic program that describes the visual scene is learned by performing differentiable ILP, as illustrated in the top row in Fig. 1. The process mainly consists of *two* steps: (i) clause generation by top-k beam search and (ii) learning of clause weights by backpropagation. We now describe each step in detail.

3.5.1 Top-*k* beam search of clauses

Let $\mathcal{Q}$ be a visual ILP problem. αILP generates promising candidates of clauses using top-k beam search. Promising candidates of clauses for an ILP problem are those that entail a majority of positive examples but few negative examples. Figure 5 illustrates the clause generation steps from visual scenes. We start from given initial clauses and iteratively refine the top-k clauses based on the following evaluation score:

$$eval(R, \mathcal{Q}) = \sum_{e \in \mathcal{E}^+} p(y \mid e, \{R\}, \mathcal{B}, \{\mathbf{1}\}, \Theta_{per}, \Theta_{np}), \tag{5}$$

where $\mathcal{E}^+ \in \mathcal{Q}$ is a set of positive examples and $\mathbf{1}$ is an 1×1 identity matrix. If clause R can entail the majority of positive examples combined with background knowledge, then clause R gets a high evaluation score. In each step, αILP evaluates clauses in parallel using a variant of the reasoning module designed for the evaluation of clauses, i.e., without loops in terms of clauses. The generation of new clauses is conducted using the downward refinement operator (Nienhuys-Cheng et al., 1997), which is a fundamental clause-generation tool in ILP. The downward refinement operator weakens the clauses, i.e., the new clauses that are produced by the operator entail fewer examples with background knowledge than the original clause (Nienhuys-Cheng et al., 1997). Thus fewer negative examples are entailed by the newly generated clauses. Therefore we evaluate clauses only by positive

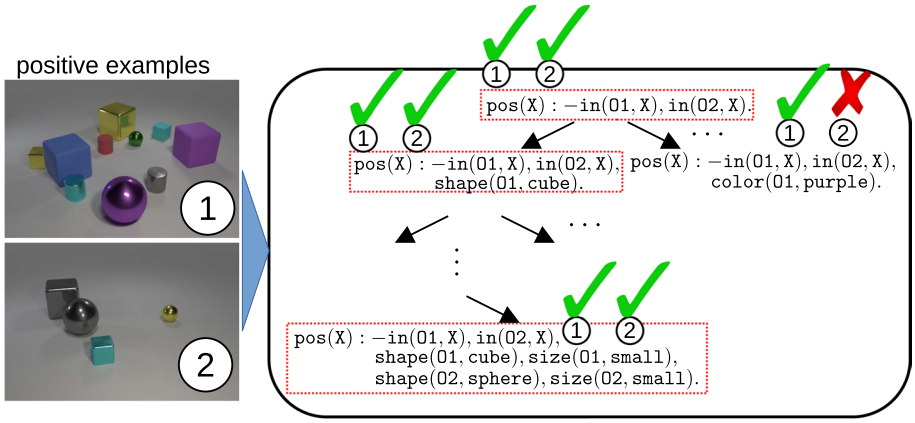


Fig. 5 Clause generation by beam search using input images. In each step, clauses that explain a majority of positive examples are selected (red-dotted rectangles), and refined to generate new clauses. After expanding them for a certain depth, the set of all of the selected clauses in the process is returned to perform differentiable ILP (Color figure online)

examples and repeatedly generate clauses by the downward refinement operator to produce a search space that contains general and specific clauses. During the clause generation, α ILP adopts mode declarations (Muggleton, 1995; Ray & Inoue, 2007) to manage the search space, i.e., the clauses that are inconsistent with mode declarations are pruned. The visual-perception module enables αILP to evaluate each clause using visual input. αILP utilizes symbolic learning techniques while dealing with complex visual scenes.

3.5.2 Learning weights

αILP assigns weights for generated clauses. Clause weights are optimized by gradient descent. Let $\mathcal{Q} = (\mathcal{E}^+, \mathcal{E}^-, \mathcal{B}, \mathcal{L})$ be a visual ILP problem, $\mathcal{C}$ be a set of generated clauses, $\mathcal{W}$ be a set of clause weights, Θ_{per} be the parameters for the perception model, and Θ_{np} be the parameters for the neural predicates. We solve visual ILP problem $\mathcal{Q}$ by minimizing cross-entropy loss with respect to $\mathcal{W}$, defined as:

$$\begin{aligned} loss = -\mathbb{E}_{(e,y)\sim\mathcal{Y}}[y \log p(y \mid e, \mathcal{C}, \mathcal{B}, \mathcal{W}, \Theta_{per}, \Theta_{np}) \\ + (1-y)\log(1 - p(y \mid e, \mathcal{C}, \mathcal{B}, \mathcal{W}, \Theta_{per}, \Theta_{np}))]. \end{aligned} \tag{6}$$

where $\mathcal{Y} = \{(e, 1) \mid e \in \mathcal{E}^+\} \cup \{(e, 0) \mid e \in \mathcal{E}^-\}$, which is a set of tuples of an example and the label indicating positive or negative.

4 Experimental evaluation

We empirically demonstrate the following desired properties of αILP on *two* different datasets: (i) αILP solves ILP problems in visual scenes with high accuracy. (ii) αILP can explain, i.e., it produces a readable solution in the form of logic programs. (iii) αILP is robust to confoundings. (iv) αILP is data-efficient unlike CNNs. (v) αILP performs fast inference.

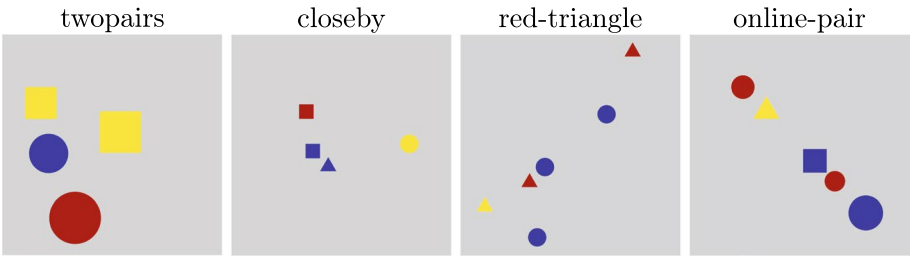


Fig. 6 Positive examples for each dataset in Kandinsky patterns. Each pattern is described as follows: (**twopairs**) *The Kandinsky figure has two pairs of objects with the same shape. In one pair, the objects have the same colors in the other pair different colors. Two pairs are always disjunct, i.e., they do not share objects.* (**closeby**) *The Kandinsky figure has a pair of objects that are close to each other.* (**red-triangle**) *The Kandinsky figure has a pair of objects that are close to each other. And the one object of the pair is a red triangle, and the other object has a different color and different shape.* (**online-pair**) *The Kandinsky figure has five objects that are aligned on a line, and it contains at least one pair of objects that have the same shape and the same color* (Color figure online)

All experiments were performed in the following environment; CPU: AMD EPYC 7742 64-Core Processor, RAM: 2000 GB, GPU: NVIDIA A100-SXM4-40GB GPU with 40 GB of RAM.

4.1 Solving Kandinsky patterns

4.1.1 Dataset

We adopted Kandinsky pattern datasets (Holzinger et al., 2019; Müller & Holzinger, 2021; Holzinger et al., 2021), a relatively new benchmark for object-centric reasoning tasks. Kandinsky-20k contains 10k training examples for each positive and negative class, respectively. Each validation and test split contains 5k examples for each positive and negative class, respectively. In the Kandinsky-2k dataset, we reduced the amount of training data by randomly sampling from the Kandinsky-20k dataset. The training split contains 1k examples for each positive and negative class, respectively. Validation and test split are the same as the Kandinsky-20k dataset.

We use 4 Kandinsky patterns: *twopairs*, *closeby*, *red-triangle*, and *online-pair*. Figure 6 shows a positive example for each pattern. For the clause generation step, we used 500 examples from the validation split for each dataset.

4.1.2 Pre-training

For pre-training of the visual perception module, we generated randomly 15k Kandinsky figures with annotations about each object, i.e., the number of objects and the attributes of the objects are randomly determined. We used YOLO (Redmon et al., 2016) as a perception module. Each object has the class label and the bounding box as an annotation. Neural predicate `closeby` and `online` are trained on the 10k Kandinsky figures that represent the concepts, respectively, e.g., figures that consist of two objects that are close by each other are generated for the positive examples for `closeby`.

Table 2 The mean classification accuracy in the test split in the Kandinsky patterns dataset over 5 random seeds

Model	Twopairs	Closeby	Red-triangle	Online-pair
Kandinsky-20k				
CNN	50.0	52.33	55.0	50.59
YOLO+MLP	99.0	72.93	82.95	80.18
αILP	**100.0**	**100.0**	**100.0**	**100.0**
Kandinsky-2k				
CNN	50.1	51.8	50.7	50.9
YOLO+MLP	91.7	62.3	78.25	75.7
αILP	**100.0**	**100.0**	**100.0**	**100.0**

αILP outperforms the considered baselines. CNNs over-fit while training and perform poorly with testing data

Kandinsky patterns

```
pos(X):-in(O1,X),in(O2,X),in(O3,X),in(O4,X),same_shape_pair(O1,O2),
    same_color_pair(O1,O2),same_shape_pair(O3,O4),diff_color_pair(O3,O4).
pos(X):-in(O1,X),in(O2,X),closeby(O1,O2).
pos(X):-in(O1,X),in(O2,X),closeby(O1,O2),color(O1,red),shape(O1,triangle),
    diff_shape_pair(O1,O2),diff_color_pair(O1,O2).
pos(X):-in(O1,X),in(O2,X),in(O3,X),in(O4,X),in(O5,X),online(O1,O2,O3,O4,O5)
    ,same_shape_pair(O1,O2),same_color_pair(O1,O2).
```

CLEVR-Hans

```
pos(X):-in(O1,X),in(O2,X),size(O1,large),shape(O1,cube),size(O2,large),
    shape(O2,cylinder).
pos(X):-in(O1,X),in(O2,X),size(O1,small),material(O1,metal),shape(O1,cube),
    size(O2,small),shape(O2,sphere).
pos(X):-in(O1,X),in(O2,X),size(O1,large),color(O1,blue),shape(O1,sphere),
    size(O2,small),color(O2,yellow),shape(O2,sphere).
```

Fig. 7 The classification rules discovered by αILP, which are obtained by taking argmax of the rule weights. The top four lines show the classification rules for the four patterns in Kandinsky patterns dataset, respectively. The bottom three lines show the classification rules for the three classes in CLEVR-Hans3 dataset, respectively

4.1.3 Baselines

We adopted ResNet (He et al., 2016) as a CNN-based benchmark and also compared it against YOLO+MLP, where the input figure is fed to the pre-trained YOLO model, and a simple MLP module predicts the class label from the YOLO outputs. The whole network is jointly trained.

4.1.4 Results

Table 2 shows the results for the test split in each Kandinsky dataset. The CNN model overfits while training and thus performs poorly in every Kandinsky pattern. The YOLO+MLP model performs comparatively better and achieves greater than 90% accuracy in *twopairs*. However, in relatively complex patterns of *closeby*, *red-triangle*, and *online-pair*, the performance degrades. On the contrary, αILP outperforms the considered baselines significantly and achieves perfect classification in all of the patterns.

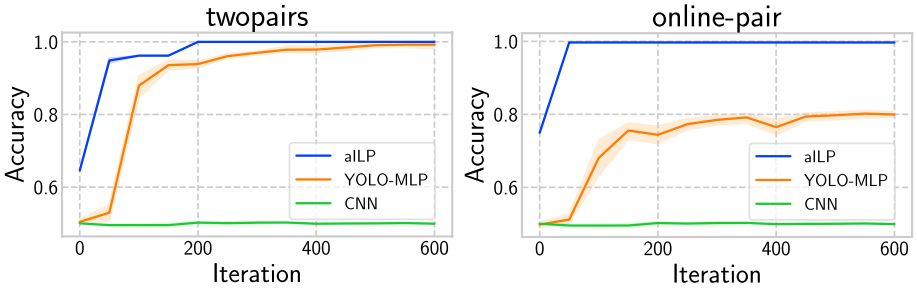


Fig. 8 Accuracy for iterations of weight updates in *twopairs* and *online-pair* dataset in Kandinsky-20k. The line represents the mean, and the shadow represents the standard deviation of 5 trials, respectively. One iteration corresponds to one weight update using a batch of training examples. αILP achieves high classification accuracy with fewer iterations compared to neural-based baselines

In the smaller dataset, Kandinsky-2k, neural-based benchmarks reduce its performance because of the lack of training data to be generalized. On the contrary, αILP still achieves perfect accuracy. This shows the data efficiency of αILP.

Figure 7 shows the classification rules discovered by αILP, which are obtained by taking *argmax* of the rule weights. αILP successfully produced interpretable results in all of the datasets. After the training step, we observed that the distribution of the weights over clauses turned to be sharp, i.e., one clause get nearly 1.0 and others get almost 0.0.

Figure 8 shows the accuracy for the test split in the first 600 iterations in Kandinsky-20k dataset. To have a fair comparison, we used the same learning rate 1e−2 for αILP and YOLO+MLP model and 1e−5 for the CNN baseline[2] to plot the figure. The line represents the mean, and the shadow represents the standard deviation of 5 trials, respectively. One iteration corresponds to one weight update using a batch of training examples. The result shows αILP achieves high classification accuracy with fewer iterations compared to neural-based baselines.

4.2 Solving CLEVR-Hans problems

4.2.1 Dataset

The CLEVR-Hans dataset (Stammer et al., 2021) contains confounded CLEVR (Johnson et al., 2017) images, and each image is associated with a class label. We adopted the CLEVR-Hans3 dataset, which has *three* classes, as shown in Fig. 9. Each class has a corresponding classification rule. For each class, we create an ILP problem, where positive examples are the set of images that belongs to a classification rule, and negative examples are images that belong to other classes. As a result, we have *three* classification problems: *class1*, *class2*, and *class3*. CLEVR-Hans problems involve confounding data. For example, in the training and validation split of *class1*, the large cube in the positive examples always has the color of gray, but in the test split, it has different colors. To achieve good performance in the test split, the model needs to know the exact classification rule without

[2] A large learning rate for the CNN baseline degraded the performance.

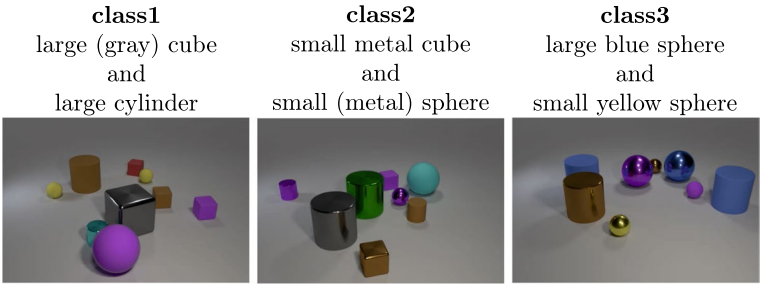


Fig. 9 Examples in CLEVR-Hans3 dataset. The dataset consists of *three* classes. **(class 1)** "*Each figure contains large a cube and a large cylinder. In the training and validation split, the large cube always has the color of gray.*" **(class 2)** "*Each figure has a small metal cube and a small sphere. In the training and validation split, the small sphere always has the material of metal.*" **(class 3)** "*Each figure contains large blue sphere and small yellow sphere.*" (Color figure online)

Table 3 The mean classification accuracy for CLEVR-Hans3 dataset compared to baselines over 5 random seeds

Model	Validation	Test
CNN	99.55 ± 0.10	70.34 ± 0.30
NeSy	98.55 ± 0.27	81.71 ± 3.09
NeSy-XIL	$\mathbf{100.00 \pm 0.00}$	91.31 ± 3.13
αILP	97.58 ± 1.16	$\mathbf{97.52 \pm 0.81}$

For αILP we report the mean over the *three* ILP problems

overfitting. For the clause generation step, we used 500 examples from the validation split for each dataset.

4.2.2 Pre-training

The slot attention model was pre-trained following Locatello et al. (2020) using the set prediction setting on the CLEVR (Johnson et al., 2017) dataset.

4.2.3 Baselines

The considered baselines are the ResNet-based CNN model (He et al., 2016), and the Neuro-Symbolic (NeSy) model (Stammer et al., 2021). The NeSy model has a visual perception module based on slot attention (Locatello et al., 2020) and a reasoning module based on Set Transformer (Lee et al., 2019). The NeSy model was trained in *two* different settings: (1) training using classification rules (NeSy), and (2) the *right for the right reasons* (Ross et al., 2017) setting, i.e., the model is trained using supervision about confounding factors (NeSy-XIL). NeSy-XIL is the SOTA model in the CLEVR-Hans dataset.

4.2.4 Results

Table 3 shows the classification accuracy in the CLEVR-Hans dataset. The results of baselines have been presented in Stammer et al. (2021). αILP achieved more than 97% in each

Table 4 Running time of top-k beam search and number of generated clauses in αILP using 500 examples from the validation split in each dataset

	Clause generation time (s)	#Generated clauses	#Selected clauses
Kandinsky patterns			
Twopairs	626.7	2686	26
Closeby	57.97	274	5
Red-triangle	1007	459	84
Online-pair	779	3475	13
CLEVR-Hans			
Class1	278.8	1685	26
Class2	316.5	1932	73
Class3	338.3	2027	92

The size of the beam in search is 20. #generated clauses is the total number of clauses generated by the refinement operator. #selected clauses is the number of clauses that are chosen as top-k clauses in the search step. αILP successfully identified promising clauses from visual scenes

split. Note that, NeSy-XIL model exploits the supervision of the confounding factors. On the contrary, αILP is unsupervised in terms of confounding factors. This shows that αILP is *robust to confounding*. αILP can control the complexity of the solution (logic programs) by controlling the depth of top-k beam search, i.e., αILP can prevent overfitting by giving a proper depth of top-k beam search, which can be determined by trying from a small number on the validation split. Figure 7 shows the classification rules discovered by αILP, which are obtained by taking *argmax* of the rule weights. After the training step, we observed that the distribution of the weights over clauses turned to be sharp, i.e., one clause gets nearly 1.0, and others get almost 0.0.

4.3 Ablation study

We analyze the efficiency of αILP for the clause generation step, the weight learning step, and the reasoning step, respectively. To this end, we discuss limitations of αILP.

4.3.1 Running time and number of clauses in clause generation

We analyze the clause generation step of αILP in terms of the running time and the number of clauses to be generated. We used 500 examples from the validation split for each dataset. The size of the search beam is 20. The depth of the search is [5, 2, 6, 4] for *twopairs, closeby, red-triangle,* and *online-pair* in Kandinsky Patterns, respectively, and [5, 6, 7] for *class1, class2,* and *class3* in CLEVR-Hans, respectively. Table 4 shows running time of top-k beam search and number of generated clauses in αILP. The clause generation step takes about 58 s in the best case and about 1000 s in the worst case. αILP searched a space that contains several thousands of candidates of clauses and then successfully searched promising clauses from visual scenes. This empirically shows that αILP performed an

Table 5 The running time (sec) of weight learning per epoch in α ILP and baseline models. αILP has a reasoning process in the forward path, thus it takes longer than baselines per epoch

Dataset	CNN	YOLO + MLP	αILP
Twopairs	29.02	28.91	336.4
Closeby	29.46	29.65	51.8
Red-triangle	29.72	29.6	1081
Online-pair	29.59	28.93	312.4

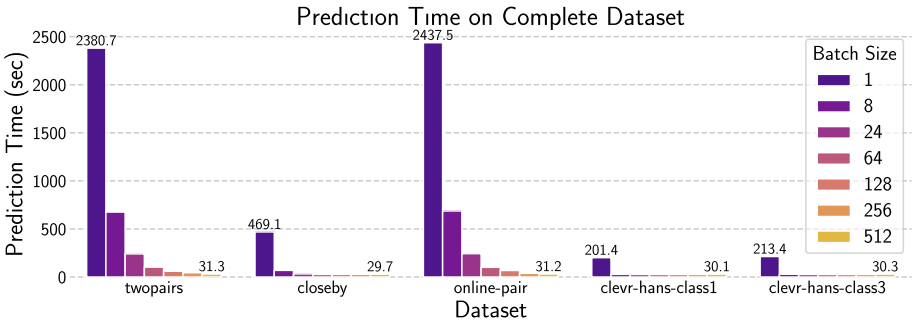


Fig. 10 Prediction time of αILP on the complete dataset. We report the running time to predict all of the training examples for each dataset using the learned rules. The bottom labels specify the datasets. For each dataset, different colors correspond to different batch sizes. We used batch sizes of 1, 8, 24, 64, 128, 256 and 512. Kandinsky patterns have 20k training samples for each pattern, CLEVR-Hans has 9k training samples for each dataset, respectively. αILP predicts quickly for large datasets by parallelized batch computation

efficient search for clauses, which is necessary to solve the visual ILP tasks of complex visual scenes.

4.3.2 Running time of weight learning

We compare the running time of weight learning of αILP with neural baselines. Table 5 shows the running time of weight learning for one epoch in Kandinsky patterns datasets. αILP achieved comparably fast weight-learning iterations for problems with simple rules, e.g., *closeby*. For the problems that have a complex search space, e.g., *red-triangle*, αILP takes longer to compute the gradient and update the clause weights. This is because, as shown in Table 4, αILP deals with a large number of clauses for difficult problems, which require a deeper search of clauses. We note that, as shown in Fig. 8, αILP can achieve high accuracy with fewer iterations compared to neural-based baselines.

4.3.3 Running time of reasoning

We show that αILP can perform fast inference by parallelized GPU-based batch computation. Figure 10 shows the prediction time with different batch sizes in Kandinsky datasets. We measured the inference time for all training examples in *twopairs*, *closeby*, and *online-pair* datasets for Kandinsky patterns, and *class1* and *class3* datasets for CLEVR-Hans. We used different batch sizes of [1, 8, 24, 64, 128, 256, 512]. We extracted the learned clause

after the training of αILP, and fed it to the forward-reasoning module, i.e., the reasoning module handles *one* clause for each dataset, enabling a large batch size on a single GPU.

For each dataset, αILP achieved fast prediction with larger batch sizes. In the *twopairs* dataset, with a batch size of 1, it takes 2380 s to classify all of the 20k training visual examples. However, with the batch size of 512, αILP classified them in 31 s. The empirical result shows that αILP can perform fast reasoning using batch computation, which is an essential function to be tightly coupled with deep neural networks.

4.3.4 Limitations

We discuss limitations of αILP. The approach is memory-intensive because the size of the index tensor is not linear with respect to the number of facts and that of clauses. In contrast to backward reasoning, all of the possible solutions are computed in forward-chaining reasoning. To handle large knowledge bases, a more memory-efficient mechanism is necessary. In the experiments, αILP assumed that the perception model is pre-trained. It also assumed that there exists a set of rules that can perfectly classify the examples in the search space. αILP requires the language bias to limit the search space, e.g., mode declarations. The learning algorithm is a hybrid approach of top-k beam-search and gradient descent. Before performing numerical optimization, the search space, i.e., the set of candidates of rules, needs to be identified by the top-k beam search.

5 Conclusion and future work

We proposed αILP, a novel differentiable ILP framework for visual scenes. αILP learns logic programs that explain complex visual scenes from visual inputs based on object-centric perception and differentiable ILP. In our experiments, αILP outperformed CNN-based baselines in *Kandinsky patterns* and *CLEVR-Hans* datasets, where the classification rules are defined on high-level concepts. αILP provides the following advantages against CNN-based models. Firstly, αILP solves complex patterns in visual scenes such as Kandinsky patterns and CLEVR-Hans datasets, which cannot be solved by CNN-based models. Secondly, αILP produces explicit classification rules as a logic program. Thirdly, αILP is data-efficient, i.e., it can achieve high performance even from a small training set. Lastly, αILP is robust to confounding, i.e., it can be generalized even if some features are confounded in the training dataset. These advantages highlight that αILP can overcome some significant limitations of neural models for complex visual scenes. Moreover, αILP performs fast differentiable inference for a large number of instances of complex visual scenes. This feature is critical for logical reasoning to be tightly coupled with neural networks. To this end, αILP is an extension to the Neuro-Symbolic systems, e.g., DeepProblog (Manhaeve et al., 2018, 2021) and ∂ILP (Evans & Grefenstette, 2018), for structure learning in visual domains.

A common criticism of ILP can be applied to αILP, e.g., hand-crafted background knowledge and language bias are crucial. A promising direction of future research is to develop a neuro-symbolic pipeline to generate proper background knowledge and language bias from data by incorporating ILP techniques such as predicate invention (Cropper et al., 2022). Solving compositional reasoning tasks (Vedantam et al., 2021) will be another direction of future research. The algorithmic supervision setting (Petersen et al., 2021), where neural networks are trained combined with differentiable implementations

Table 6 Mode declarations for Kandinsky patterns

```
modeh(1, pos(−image))
modeb(#obj, in(−object, +image))
modeb(1, color(+object, #color))
modeb(1, shape(+object, #shape))
modeb(2, same_color_pair(+object, +object))
modeb(2, same_shape_pair(+object, +object))
modeb(1, diff_color_pair(+object, +object))
modeb(1, duff_shape_pair(+object, +object))
modeb(1, closeby(+object, +object))
modeb(1, online(+object, ... , +object))
```

Table 7 Datatype and constants in Kandinsky patterns

Datatype	Terms
image	img
object	obj1, obj2, ⋯, obj6
color	red, blue, yellow
shape	square, circle, triangle

of discrete algorithms, is also a promising approach with αILP, because the reasoning of α ILP is compatible with neural networks in terms of the running time. Moreover, differentiable implementations of the top-k operator (Goyal et al., 2018; Xie et al., 2020; Pietruszka et al., 2021) could lead αILP to have an end-to-end learning system.

Appendix A: Experimental setting

In this section, we describe the experimental setting of our main experiments.

A.1 Kandinsky-20k

CNN We trained ResNet18 for 300 epochs with a batch size of 512. We used the Adam optimizer (Kingma & Ba, 2015; Ruder, 2016) with a learning rate of 1e−5.

YOLO+MLP We used MLP with *two* hidden layers. Each hidden layer applies a linear transformation and a non-linearity. The output of the pre-trained YOLO model is reshaped and fed into MLP to predict the class label. We trained the whole YOLO+MLP network jointly for 1000 epochs with a batch size of 512. We used the Adam optimizer (Kingma & Ba, 2015; Ruder, 2016) with a learning rate of 1e−5.

αILP. We trained the αILP model for 100 epochs with a batch size of 64. We used the RMSProp (Ruder, 2016) optimizer with a learning rate of 1e−2. We used 500 positive examples in the validation split to generate clauses by beam search.

Table 8 Predicates in the Kandinsky patterns

Predicate	Explanation
pos/(1, [image])	The image belongs to the Kandinsky pattern
same_shape_pair/(2, [object, object])	The two objects have the same shape
same_color_pair/(2, [object, object])	The two objects have the same color
diff_shape_pair/(2, [object, object])	The two objects have different shapes
diff_color_pair/(2, [object, object])	The two objects have different colors
Neural predicate	**Explanation**
in/(2, [object, image])	The object is in the image
shape/(2, [object, shape])	The object has the shape of the second argument
color/(2, [object, color])	The object has the color of the second argument
closeby/(2, [object, object])	The two objects are located close by each other
online/(5, [object, … , object])	The objects are aligned on a line

Table 9 Hyper parameters of α ILP in Kandinsky patterns

	T_{beam}	N_{beam}	#obj
Twopairs	5	20	4
Closeby	2	20	2
Red-triangle	6	20	2
Online-pair	4	20	5

T_{beam} represents the depth of the search, and N_{beam} represents the size of the beam, i.e., the width of the beam search. #obj is the maximum number of objects that can appear in a clause

Table 10 Background knowledge for Kandinsky patterns

```
same_shape_pair(X, Y) :- shape(X, Z), shape(Y, Z)
same_color_pair(X, Y) :- color(X, Z), color(Y, Z)
diff_shape_pair(X, Y) :- shape(X, Z), shape(Y, W), diff_shape(Z, W)
diff_color_pair(X, Y) :- color(X, Z), color(Y, W), diff_color(Z, W)
diff_color(red, blue), diff_color(blue, red)
diff_color(red, yellow), diff_color(yellow, red)
diff_color(blue, yellow), diff_color(yellow, blue)
diff_shape(circle, square), diff_shape(square, circle)
diff_shape(circle, triangle), diff_shape(triangle, circle)
diff_shape(square, triangle), diff_shape(triangle, square)
```

Mode declarations (Muggleton, 1995; Cropper et al., 2022) we used are shown in Table 6. Table 7 shows the data types and constants, and Table 8 shows the predicates for Kandinsky patterns, respectively. Hyperparameters for the clause generation is shown in Table 9. #obj represents the number of objects to be focused on the classification, which can be identified by trying from the smallest number and evaluating by validation split and increasing if the performance is not enough. We set the initial clause to

Table 11 Mode declarations for CLEVR-Hans

```
modeh(1,pos(−image))
modeb(#obj,in(−object,+image))
modeb(1,color(+object,#color))
modeb(1,shape(+object,#shape))
modeb(1,material(+object,#material))
modeb(1,size(+object,#size))
```

Table 12 Hyper parameters of α ILP in CLEVR-Hans

	T_{beam}	N_{beam}	#obj
class 1	5	20	2
class 2	6	20	2
class 3	7	20	2

T_{beam} represents the depth of the search, and N_{beam} represents the size of the beam, i.e., the width of the beam search. #obj is the maximum number of objects that can appear in a clause

be the root node in the beam search as: `pos(X):- in(O1,X),…,in(On,X)`, where n is the number of objects to be focused, i.e., #obj in Table 9. Background knowledge given in for Kandinsky patterns is shown in Table 10.

For neural predicate `closeby`, we used the following valuation function:

$$v_{\texttt{closeby}}\left(\mathbf{Z}^{(1)},\mathbf{Z}^{(2)}\right)=\sigma\left(f_{linear}\left(f_{norm}\left(\mathbf{Z}^{(1)}_{center}-\mathbf{Z}^{(2)}_{center}\right);\mathbf{w}\right)\right),$$

where $\mathbf{Z}^{(i)}_{center}$ represents the center coordinate of the bounding box for the i-th object, and $\mathbf{w}$ is a parameter to be trained.

For neural predicate `online`, we used the following valuation function

$$v_{\texttt{online}}\left(\mathbf{Z}^{(1)},\ldots,\mathbf{Z}^{(5)}\right)=\sigma\left(f_{linear}\left(f_{reg}\left(\mathbf{Z}^{(1)}_{center},\ldots,\mathbf{Z}^{(5)}_{center}\right);\mathbf{w}\right)\right),$$

where function f_{reg} computes the closed-form solution of the linear regression in batch and returns the error values, and $\mathbf{w}$ is a parameter to be trained.

A.2 Kandinsky-2k

CNN. We trained ResNet18 for 300 epochs with a batch size of 64. We used the Adam optimizer with a learning rate of 1e − 5.

YOLO+MLP. We used MLP with *two* hidden layers with a non-linearity. The output of the pre-trained YOLO model is reshaped and fed into MLP to predict the class label. We trained the whole YOLO+MLP network jointly for 1000 epochs with a batch size of 64. We used the Adam optimizer (Kingma & Ba, 2015; Ruder, 2016) with a learning rate of 1e−5.

α**ILP**. We trained in the same setting as in Kandinsky-20k.

Table 13 Datatype and constants in CLEVR-Hans

Datatype	Terms
`image`	`img`
`object`	`obj0`, `obj1`, ⋯, `obj9`
`color`	`cyan`, `blue`, `yellow`, `purple`, `red`, `green`, `gray`, `brown`
`shape`	`sphere`, `cube`, `cylinder`
`size`	`large`, `small`
`material`	`rubber`, `metal`

Table 14 Predicates in CLEVR-Hans

Predicate	Explanation
`pos`/(1, [`image`])	The image belongs to the CLEVR-Hans pattern
Neural predicate	Explanation
`in`/(2, [`object`, `image`])	The object is in the image
`shape`/(2, [`object`, `shape`])	The object has the shape of the second argument
`color`/(2, [`object`, `color`])	The object has the color of the second argument
`material`/(2, [`object`, `material`])	The object has the material of `material`
`size`/(2, [`object`, `size`])	The object has the size of the second argument

A.3 CLEVR-Hans

We trained the αILP model for 100 epochs with a batch size of 256. We used the RMSProp optimizer (Ruder, 2016) with a learning rate of 1e−2. We used 500 positive examples in the validation split to generate clauses by beam search.

Mode declarations (Muggleton, 1995; Cropper et al., 2022) we used are shown in Table 11. Hyper parameters for the clause generation is shown in Table 12. Table 13 shows the data types and constants, and Table 14 shows the predicates for CLEVR-Hans, respectively. We set the initial clause to be the root node in the beam search as: `pos(X) :- in(O1, X), in(O2, X)`. We did not provide any background knowledge for CLEVR-Hans tasks.

Appendix B: Perception models in experiments

We describe the experimental setting of the pre-training of the perception models in our experiments.

B.1 YOLO for Kandinsky patterns

Model We used YOLOv5[3] model, whose implementation is publicly available. We adopted the YOLOv5s model, which has 7.3 M parameters.

Dataset We generated 15,000 pattern-free figures for training, 5000 figures for validation. The class labels and positions are generated randomly. The original image size is 620×620, and resized into 128×128. The label consists of the class labels and the bounding box for each object. The class label is generated by the combination of the shape and the color of the object, e.g., *red circle* and *blue square*. The number of classes is 9. Each image contains at least 2 objects and at most 10 objects.

Optimization We trained the YOLOv5s model by stochastic gradient descent (SGD) for 400 epochs using the pre-trained weights.[4] We used the loss function that approximates detection performance, presented in Redmon et al. (2016). We set the learning rate to 0.01 and the batch size to 64. The SGD optimizer used the momentum, which is set to 0.937. We set the weight decay as 0.0005. We took 3 warmup epochs for training.

B.2 Slot Attention for CLEVR-Hans

We used the same model and training setup as the pre-training of the slot-attention module in Stammer et al. (2021). In the preprocessing, we downscaled the CLEVR-Hans images to a dimension of 128×128 and normalized the images to lie between -1 and 1. For training the slot-attention module, an object is represented as a vector of binary values for the shape, size, color, and material attributes and continuous values between 0 and 1 for the x, y, and z positions. We trained the slot attention model with the set prediction architecture following Locatello et al. (2020), using the loss function, which is based on the Hungarian algorithm. We refer to Stammer et al. (2021) for more details.

Appendix C: Background knowledge in αILP

αILP accepts background knowledge as set of facts $\mathcal{G}_{bk}$ and clauses $\mathcal{C}_{bk}$. For fact $g_i \in \mathcal{G}_{bk}$, we set the initial valuation value as 1.0, i.e., $\mathbf{V}^{(0)}_{:,i} = 1.0$. For clauses, let $\mathbf{I}_{bk}$ be an index tensor for clauses $\mathcal{C}_{bk}$ in background knowledge. Then we compute the reasoning as $\mathbf{V}^{(t+1)} = softor^{\gamma}_{1}(stack_1(\mathbf{V}^{(t)}, r(\mathbf{V}^{(t)};\mathbf{I}, \mathbf{W}), r(\mathbf{V}^{(t)};\mathbf{I}_{bk}, \mathbf{1})))$, where $\mathbf{1} \in \{0,1\}^{M \times |\mathcal{C}_{bk}|}$ is an identity matrix.

Appendix D: Details on the softor function

In the differentiable inference process, αILP often computes logical *or* for probabilistic values. Taking *max* repeatedly can violate the gradient flow. The $softor^{\gamma}_{d}$ function approximates the *or* computation softly. The key idea is to use the log-sum-exp technique to approximate the *max/min* operation (Cuturi & Blondel, 2017). We define the $softor^{\gamma}_{d}$ function as follows:

[3] https://github.com/ultralytics/yolov5.

[4] https://github.com/ultralytics/yolov5/releases.

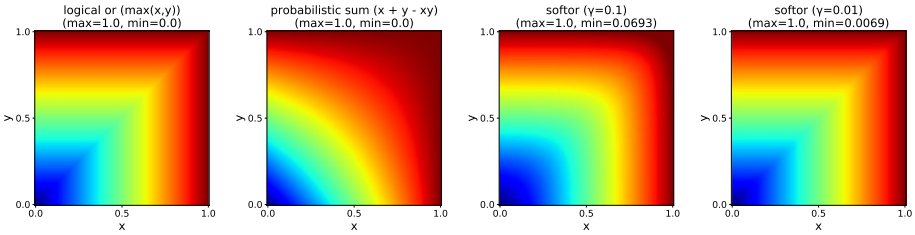


Fig. 11 The visualization of various 4 *or* functions. From left to right, each plot corresponds to *max*, *probabilistic sum*, *softor* with $\gamma = 0.1$, and *softor* with $\gamma = 0.01$, respectively. The maximum and minimum value for each plot are shown on top of each, which are represented by the colors from blue to red. The $softor^{\gamma}_{d}$ function with a sufficiently small smooth parameter approximates well the logical *or* function for probabilistic values (Color figure online)

$$softor^{\gamma}_{d}(\mathbf{X}) = \frac{1}{S}\gamma \log\left(sum_d \exp\left(\mathbf{X}/\gamma\right)\right), \tag{D.1}$$

where $\gamma > 0$ is a smooth parameter, sum_d is the sum function along dimension d, and $S = max(1.0, max\left(\gamma \log sum_d \exp\left(\mathbf{X}/\gamma\right)\right)$. The normalization term ensures that the $softor^{\gamma}_{d}$ function returns a normalized probabilistic values. The dimension d specifies the dimension to be removed.

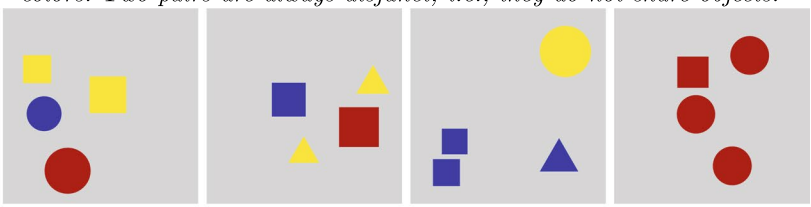
A popular choice is the *probabilistic sum* function: $f_{prob\_sum}(\mathbf{X}, \mathbf{Y}) = \mathbf{X} + \mathbf{Y} - \mathbf{X} \odot \mathbf{Y}$, which was adopted in Evans and Grefenstette (2018) and Jiang and Luo (2019). We plot the various functions for logical *or* in Fig. 11 to compare. From left to right, each plot corresponds to *max*, *probabilistic sum*, *softor* with $\gamma = 0.1$, and *softor* with $\gamma = 0.01$, respectively, with respect to 2-dimensional input $x, y \in [0, 1]$. The maximum and minimum value for each plot are shown on top of each, which are represented by the colors from blue to red. The $softor^{\gamma}_{d}$ function with a sufficiently small smooth parameter approximates well the logical *or* function for probabilistic values.

Appendix E: Mode declaration

Mode Declaration (Muggleton, 1995; Ray & Inoue, 2007) is one of the common language biases. We used mode declaration, which is defined as follows. A mode declaration is either a head declaration $\texttt{modeh}(\texttt{r}, \texttt{p}(\texttt{mdt}_1, \ldots, \texttt{mdt}_\texttt{n}))$ or a body declaration $\texttt{modeb}(\texttt{r}, \texttt{p}(\texttt{mdt}_1, \ldots, \texttt{mdt}_\texttt{n}))$, where $\texttt{r} \in \mathbb{N}$ is an integer, p is a predicate, and $\texttt{mdt}_\texttt{i}$ is a mode datatype. A mode datatype is a tuple $(\texttt{pm}, \texttt{dt})$, where pm is a place-marker and dt is a datatype. A place-marker is either #, which represents constants, or + (resp. −), which represents input (resp. output) variables. r represents the number of the usages of the predicate to compose a solution.

TwoPairs

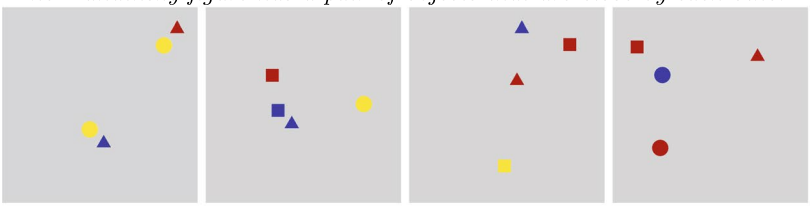
"The Kandinsky figure has two pairs of objects with the same shape. In one pair, the objects have the same colors in the other pair different colors. Two pairs are always disjunct, i.e., they do not share objects."



positive positive negative negative

Closeby

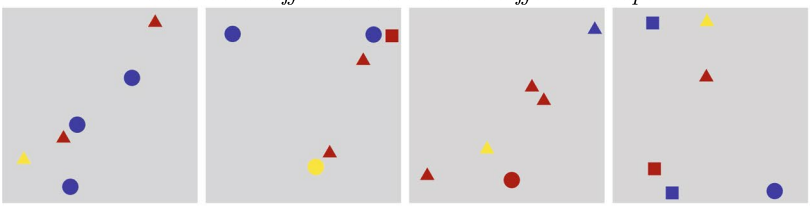
"The Kandinsky figure has a pair of objects that are close by each other."



positive positive negative negative

Red-Triangle

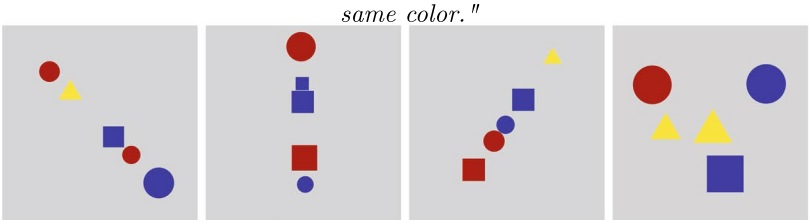
"The Kandinsky figure has a pair of objects that are close by each other, and one object of the pair is a red triangle, and the other object has a different color and different shape. (A red triangle is attacking someone who has a different color and a different shape."



positive positive negative negative

Online/Pair

"The Kandinsky figure has five objects that are aligned on a line, and it contains at least one pair of objects that have the same shape and the same color."



positive positive negative negative

Fig. 12 Examples in each Kandinsky pattern in our experiments. The left two images are positive examples, and the right two images are negative examples

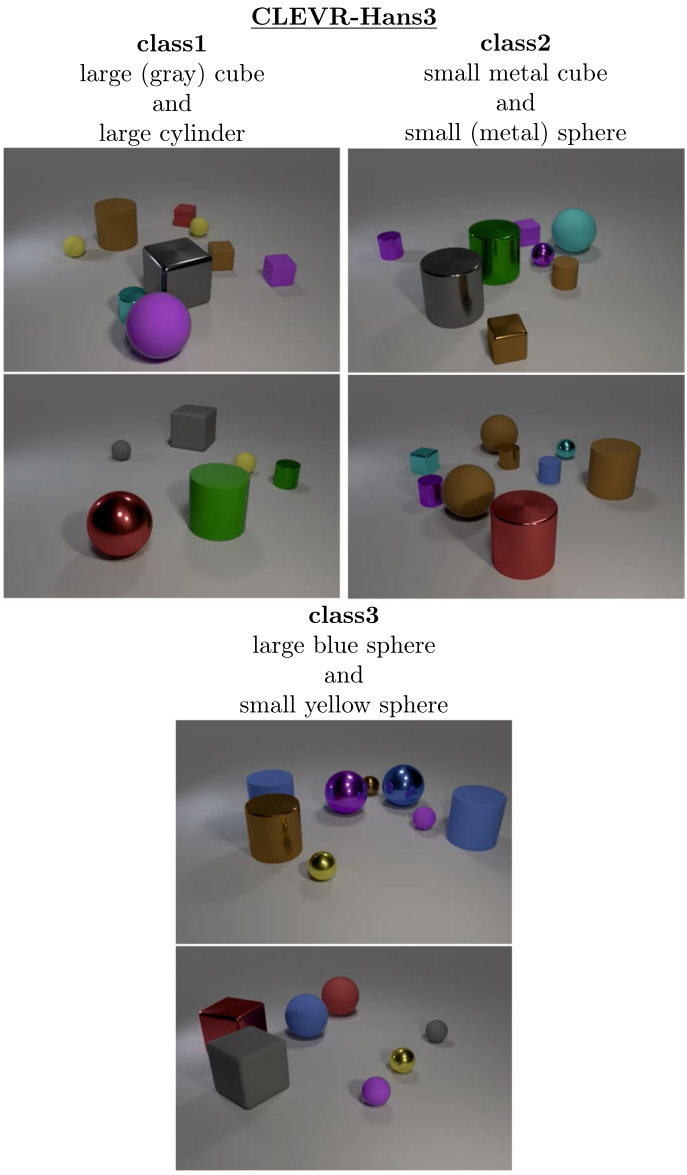


Fig. 13 Examples in CLEVR-Hans3 dataset. The dataset consists of *three* classes. Two images are shown for each class. The text on the top of the images describes the confounded classification rule for each class. For example, images of the first class contain a large cube and a large cylinder. The large cube has the color of gray in every image of the train and validation split. In the test split, the color of the large cube is shuffled randomly (Color figure online)

Appendix F: More examples of Kandinsky patterns and CLEVR-Hans

We show some examples for each pattern we used in Kandinsky patterns in Fig. 12. We also show some examples for each class of CLEVR-Hans3 in Fig. 13.

Author Contributions HS, VP, DD, KK designed the study and developed the initial idea. HS, VP, DD, KK interpreted the data and drafted the manuscript. HS conducted experiments and algorithmic implementation. VP helped with the writing. HS, VP, DD, KK designed the experimental setting. All authors read and approved the final manuscript.

Funding Open Access funding enabled and organized by Projekt DEAL. This work was supported by the AI lighthouse project "SPAICER" (01MK20015E), the EU ICT-48 Network of AI Research Excellence Center "TAILOR" (EU Horizon 2020, GA No 952215), and the Collaboration Lab "AI in Construction" (AICO). The work has also benefited from the Hessian Ministry of Higher Education, Research, Science and the Arts (HMWK) cluster projects "The Third Wave of AI" and "The Adaptive Mind".

Data availability The CLEVR-Hans dataset is available at: https://github.com/ml-research/CLEVR-Hans. The full dataset of Kandinsky patterns used in the experiments will be uploaded and publicly available.

Code availability The public code is available: https://github.com/ml-research/alphailp.

Declarations

Conflict of interest Not Applicable.

Ethics approval Not Applicable.

Consent to participate Not Applicable.

Consent for publication Not Applicable.

References

Amizadeh, S., Palangi, H., Polozov, A., Huang, Y., & Koishida, K. (2020). Neuro-symbolic visual reasoning: Disentangling visual from reasoning. *Proceedings of the 37th international conference on machine learning (ICML)* (Vol. 119, pp. 279–290).

Antol, S., Agrawal, A., Lu, J., Mitchell, M., Batra, D., Zitnick, C. L., & Parikh, D. (2015). Vqa: Visual question answering. In *Proceedings of the IEEE international conference on computer vision (ICCV)*.

Badreddine, S., d'Avila Garcez, A., Serafini, L., & Spranger, M. (2022). Logic tensor networks. *Artificial Intelligence, 303*, 103649.

Bellodi, E., & Riguzzi, F. (2015). Structure learning of probabilistic logic programs by searching the clause space. *Theory and Practice of Logic Programming, 15*(2), 169–212.

Besold, T. R., d'Avila Garcez, A. S., Bader, S., Bowman, H., Domingos, P. M., Hitzler, P., Kühnberger, K., Lamb, L. C., Lowd, D., Lima, P. M. V., de Penning, L., Pinkas, G., Poon, H., & Zaverucha, G. (2017). Neural-symbolic learning and reasoning: A survey and interpretation. In *CoRR*arXiv:1711.03902.

Bongard, M. M., & Hawkins, J. K. (1970). *Pattern recognition*. New York: Spartan Books.

Bošnjak, M., Rocktäschel, T., Naradowsky, J., & Riedel, S. (2017). Programming with a differentiable forth interpreter. In *Proceedings of the 34th international conference on machine learning (ICML)* (Vol. 70, pp. 547–556).

Burgess, C. P., Matthey, L., Watters, N., Kabra, R., Higgins, I., Botvinick, M. M., & Lerchner, A. (2019). Monet: Unsupervised scene decomposition and representation. CoRR arXiv:1901.11390.
Cropper, A., & Muggleton, S. H. (2016). Metagol system. https://github.com/metagol/metagol.
Cropper, A., Dumancic, S., Evans, R., & Muggleton, S. H. (2022). Inductive logic programming at 30. *Machine Learning, 111*(1), 147–172.
Cropper, A., & Morel, R. (2021). Learning programs by learning from failures. *Machine Learning, 110*(4), 801–856.
Cropper, A., Morel, R., & Muggleton, S. (2019). Learning higher-order logic programs. *Machine Learning, 109*, 1289–1322.
Cuturi, M., & Blondel, M. (2017). Soft-DTW: A differentiable loss function for time-series. In *Proceedings of the 34th international conference on machine learning (ICML)* (Vol. 70, pp. 894–903).
Dai, W.-Z., Xu, Q., Yu, Y., & Zhou, Z.-H. (2019). Bridging machine learning and logical reasoning by abductive learning. In *Proceedings of the advances in neural information processing systems (NeurIPS)* (Vol. 32).
d'Avila Garcez, A., & Lamb, L. C. (2020). Neurosymbolic AI: The 3rd wave. In *CoRR*arXiv:2012.05876.
De Raedt, L., Frasconi, P., Kersting, K., & Muggleton, S. H. (Eds.) (2008). *Probabilistic inductive logic programming—theory and applications. Lecture Notes in Computer Science* (Vol. 4911). Berlin: Springer.
De Raedt, L., Kersting, K., Natarajan, S., & Poole, D. (2016). Statistical relational artificial intelligence: Logic, probability, and computation. In *Synthesis lectures on artificial intelligence and machine learning* (Vol. 32). San Rafael, CA: Morgan & Claypool.
Diligenti, M., Gori, M., & Saccà, C. (2017). Semantic-based regularization for learning and inference. *Artificial Intelligence, 244*, 143–165.
Dittadi, A., Papa, S., De Vita, M., Schölkopf, B., Winther, O., & Locatello, F. (2022). Generalization and robustness implications in object-centric learning. In *Proceedings of the 39th international conference on machine learning (ICML)*.
Donadello, I., Serafini, L., & d'Avila Garcez, A. (2017). Logic tensor networks for semantic image interpretation. In *Proceedings of the 26th international joint conference on artificial intelligence (IJCAI)* (pp. 1596–1602).
Engelcke, M., Kosiorek, A. R., Jones, O. P., & Posner, I. (2020). Genesis: Generative scene inference and sampling with object-centric latent representations. In *Proceedings of the 8th international conference on learning representations (ICLR)*.
Evans, R., & Grefenstette, E. (2018). Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research, 61*, 1–64.
Goyal, K., Neubig, G., Dyer, C., & Berg-Kirkpatrick, T. (2018). A continuous relaxation of beam search for end-to-end training of neural sequence models. In *Proceedings of the 32th AAAI conference on artificial intelligence (AAAI)* (Vol. 32, No 1).
He, K., Gkioxari, G., Dollár, P., & Girshick, R. B. (2017). Mask r-cnn. In *Proceedings of the IEEE international conference on computer vision (ICCV)* (pp. 2980–2988).
He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)* (pp. 770–778).
Holzinger, A., Kickmeier-Rust, M., & Müller, H. (2019). Kandinsky patterns as IQ-test for machine learning. In *Proceedings of the 3rd international cross-domain conference for machine learning and knowledge extraction (CD-MAKE)* (pp. 1–14).
Holzinger, A., Saranti, A., & Müller, H. (2021). Kandinsky patterns: An experimental exploration environment for pattern analysis and machine intelligence. In *CoRR*arXiv:2103.00519.
Jiang, Z., & Luo, S. (2019). Neural logic reinforcement learning. In *Proceedings of the 36th international conference on machine learning (ICML)* (Vol. 97, pp. 3110–3119).
Johnson, J., Hariharan, B., van der Maaten, L., Fei-Fei, L., Zitnick, C. L., & Girshick, R. B. (2017). Clevr: A diagnostic dataset for compositional language and elementary visual reasoning. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)* (pp. 1988–1997).
Kautz, H. (2022). The third AI summer: AAAI Robert S. Engelmore memorial lecture. *AI Magazine, 43*(1), 93–104.
Kim, J., Ricci, M., & Serre, T. (2018). Not-So-CLEVR: Learning same-different relations strains feedforward neural networks. *Interface Focus, 8*, 20180011.
Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of the 3rd international conference on learning representation (ICLR)*.
Kowalski, R. A. (1988). The early years of logic programming. *Communications of the ACM, 31*(1), 38–43.
Law, M., Russo, A., & Broda, K. (2014). Inductive learning of answer set programs. In E. Fermé, J. Leite (Eds.), *Logics in artificial intelligence—14th European Conference (JELIA). Lecture Notes in Computer Science* (Vol. 8761, pp. 311–325).

Lee, J., Lee, Y., Kim, J., Kosiorek, A., Choi, S., & Teh, Y. W. (2019). Set transformer: A framework for attention-based permutation-invariant neural networks. In *Proceedings of the 36th international conference on machine learning (ICML)* (Vol. 97, pp. 3744–3753).

Lloyd, J. W. (1984). *Foundations of logic programming*.

Locatello, F., Weissenborn, D., Unterthiner, T., Mahendran, A., Heigold, G., Uszkoreit, J., Dosovitskiy, A., & Kipf, T. (2020). Object-centric learning with slot attention. *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), 33*, 11525–11538.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., & De Raedt, L. (2018). Deepproblog: Neural probabilistic logic programming. In *Proceedings of the advances in neural information processing systems (NeurIPS)* (Vol. 31).

Manhaeve, R., Dumančić, S., Kimmig, A., Demeester, T., & De Raedt, L. (2021). Neural probabilistic logic programming in DeepProbLog. *Artificial Intelligence, 298*, 103504.

Mao, J., Gan, C., Kohli, P., Tenenbaum, J. B., & Wu, J. (2019). The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *Proceedings of the 7th international conference on learning representations (ICLR)*.

Mensch, A., & Blondel, M. (2018). Differentiable dynamic programming for structured prediction and attention. In *Proceedings of the 35th international conference on machine learning (ICML)* (Vol. 80, pp. 3462–3471).

Minervini, P., Riedel, S., Stenetorp, P., Grefenstette, E., & Rocktäschel, T. (2020). Learning reasoning strategies in end-to-end differentiable proving. In *Proceedings of the 37th international conference on machine learning (ICML)*.

Muggleton, S. H. (1991). Inductive logic programming. *New Generation Computing, 8*(4), 295–318.

Muggleton, S. (1995). Inverse Entailment and Progol. *New Generation Computing, Special issue on Inductive Logic Programming, 13*(3–4), 245–286.

Müller, H., & Holzinger, A. (2021). Kandinsky patterns. *Artificial Intelligence, 300*, 103546.

Nguembang Fadja, A., & Riguzzi, F. (2019). Lifted discriminative learning of probabilistic logic programs. *Machine Learning, 108*(7), 1111–1135.

Nienhuys-Cheng, S.-H., Wolf, R. D., Siekmann, J., & Carbonell, J. G. (1997). *Foundations of inductive logic programming*.

Nie, W., Yu, Z., Mao, L., Patel, A. B., Zhu, Y., & Anandkumar, A. (2020). Bongard-logo: A new benchmark for human-level concept learning and reasoning. *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), 33*, 16468–16480.

Petersen, F., Borgelt, C., Kuehne, H., & Deussen, O. (2021). Learning with algorithmic supervision via continuous relaxations. *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), 34*, 16520–16531.

Pietruszka, M., Borchmann, L., & Gralinski, F. (2021). Successive halving top-k operator. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence (AAAI)* (Vol. 35, No. 18, pp. 15869–15870).

Plotkin, G. (1971). A further note on inductive generalization. In *Machine intelligence* (Vol. 6).

Quinlan, J. R. (1990). Learning logical definitions from relations. *Machine Learning, 5*, 239–266.

Raedt, L. d., Dumančić, S., Manhaeve, R., & Marra, G. (2020). From statistical relational to neuro-symbolic artificial intelligence. In *Proceedings of the 29th international joint conference on artificial intelligence (IJCAI)* (pp. 4943–4950).

Ray, O., & Inoue, K. (2007). Mode-directed inverse entailment for full clausal theories. In *Proceedings of the 17th international conference on inductive logic programming (ILP). Lecture notes in computer science* (Vol. 4894, pp. 225–238).

Redmon, J., Divvala, S. K., Girshick, R. B., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)* (pp. 779–788).

Ren, S., He, K., Girshick, R., & Sun, J. (2015). Faster r-cnn: Towards real-time object detection with region proposal networks. In *Proceedings of the advances in neural information processing systems (NeurIPS)* (Vol. 28).

Rocktäschel, T., & Riedel, S. (2017). End-to-end differentiable proving. In *Proceedings of the advances in neural information processing systems (NeurIPS)* (Vol. 30).

Ross, A. S., Hughes, M. C., & Doshi-Velez, F. (2017). Right for the right reasons: Training differentiable models by constraining their explanations. In *Proceedings of the 26 international joint conference on artificial intelligence (IJCAI)* (pp. 2662–2670).

Ruder, S. (2016). An overview of gradient descent optimization algorithms. In *CoRR*arXiv:1609.04747.

Russell, S., & Norvig, P. (2009). *Artificial intelligence: A modern approach* (3rd ed.). Hoboken, NJ: Prentice Hall Press.

Santoro, A., Faulkner, R., Raposo, D., Rae, J., Chrzanowski, M., Weber, T., Wierstra, D., Vinyals, O., Pascanu, R., & Lillicrap, T. (2018). Relational recurrent neural networks. In: *Proceedings of the advances in neural information processing systems (NeurIPS)* (Vol. 31).

Sen, P., Carvalho, B. W. S. R. D., Riegel, R., & Gray, A. (2022). Neuro-symbolic inductive logic programming with logical neural networks. In *Proceedings of the AAAI conference on artificial intelligence (AAAI)* (Vol. 36, No 8, pp. 8212–8219).

Shapiro, E. Y. (1983). *Algorithmic program debugging*. Cambridge: MIT Press.

Shindo, H., Nishino, M., & Yamamoto, A. (2021). Differentiable inductive logic programming for structured examples. In *Proceedings of the 35th AAAI conference on artificial intelligence (AAAI)* (pp. 5034–5041).

Si, X., Raghothaman, M., Heo, K., & Naik, M. (2019). Synthesizing datalog programs using numerical relaxation. In *Proceedings of the 28th international joint conference on artificial intelligence (IJCAI)* (pp. 6117–6124).

Solar-Lezama, A. (2008). *Program synthesis by sketching*. Ph.D. Thesis.

Sourek, G., Svatos, M., Zelezný, F., Schockaert, S., & Kuzelka, O. (2017). Stacked structure learning for lifted relational neural networks. In N. Lachiche, C. Vrain (Eds.), *Proceedings of the 27th international conference on inductive logic programming. Lecture notes in computer science* (Vol. 10759, pp. 140–151).

Sourek, G., Aschenbrenner, V., Zelezný, F., Schockaert, S., & Kuzelka, O. (2018). Lifted relational neural networks: Efficient learning of latent relational structures. *Journal of Artificial Intelligence Research, 62*, 69–100.

Stammer, W., Schramowski, P., & Kersting, K. (2021). Right for the right concept: Revising neuro-symbolic concepts by interacting with their explanations. In *Proceedings of the IEEE conference on computer vision and pattern recognition (CVPR)* (pp. 3619–3629).

Tsamoura, E., Hospedales, T. M., & Michael, L. (2021). Neural-symbolic integration: A compositional perspective. In *Proceedings of the 35th AAAI conference on artificial intelligence (AAAI)* (pp. 5051–5060).

van Krieken, E., Acar, E., & van Harmelen, F. (2022). Analyzing differentiable fuzzy logic operators. *Artificial Intelligence, 302*, 103602.

Vedantam, R., Szlam, A., Nickel, M., Morcos, A., & Lake, B. M. (2021). Curi: A benchmark for productive concept learning under uncertainty. In *Proceedings of the 38th international conference on machine learning (ICML)* (Vol. 139, pp. 10519–10529).

Xie, Y., Dai, H., Chen, M., Dai, B., Zhao, T., Zha, H., Wei, W., & Pfister, T. (2020). Differentiable top-k with optimal transport. *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS), 33*, 20520–20531.

Yang, Z., Ishay, A., & Lee, J. (2020). Neurasp: Embracing neural networks into answer set programming. In *Proceedings of the 29th international joint conference on artificial intelligence (IJCAI)* (pp. 1755–1762).