A domain-specific language for NeSy focussing on symbolic knowledge injection

Mattia Matteini[a], **Giovanni Ciatto**[a,*], **Matteo Magnini**[a], **Emre Kuru**[b], **Reyhan Aydoğan**[b] and **Andrea Omicini**[a]

[a]Department of Computer Science and Engineering, Alma Mater Studiorum—Università di Bologna, Bologna, Italy
[b]Özyeğin University: Istanbul, İstanbul, Turkey
ORCID (Mattia Matteini): https://orcid.org/0009-0007-4474-8431, ORCID (Giovanni Ciatto): https://orcid.org/0000-0002-1841-8996, ORCID (Matteo Magnini): https://orcid.org/0000-0001-9990-420X, ORCID (Emre Kuru): https://orcid.org/0009-0007-6130-6272, ORCID (Reyhan Aydoğan): https://orcid.org/0000-0002-5260-9999, ORCID (Andrea Omicini): https://orcid.org/0000-0002-6655-3869

Abstract. In neuro-symbolic AI (NeSy), integrating symbolic languages – typically subsets of first-order logic (FOL) –, with neural networks (NNs) serves goals like enhancing symbolic processing, extending reasoning with pattern recognition, and guiding neural learning with symbolic knowledge—a.k.a. symbolic knowledge injection (SKI). Despite its utility, FOL's expressiveness poses challenges to SKI algorithms, and its general-purpose nature complicates use for non-experts. We propose SKI-lang, a domain-specific language for SKI that balances practicality, clear semantics, and expressiveness–tractability trade-offs. SKI-lang simplifies symbolic specification, serves as a unified interface for diverse SKI approaches, and allows for automating benchmarks from NeSy literature. We discuss the design choices behind SKI-lang and its implementation, and demonstrate its effectiveness and versatility through a few case studies.

1 Introduction

In the context of neuro-symbolic AI (NeSy) [4], many approaches have been proposed to integrate (some sort) for symbolic language – most commonly, a subset of first-order logic (FOL) – with neural networks (NNs) [5], to pursue disparate goals, including, but not limited to: *(i)* speeding up symbolic processing via neural computation, *(ii)* extending symbolic reasoning with pattern-recognition capabilities, or *(iii)* controlling the learning process of NNs with symbolic knowledge.

The latter goal in particular is also known into the literature as symbolic knowledge injection (SKI) [1]. SKI has been addressed by several works, proposing as many algorithms, each one focussing on a different subset of FOL, ranging from propositional logic to Datalog [2] and beyond—up to the full power of FOL itself.

However, the very choice of the FOL *syntax* (and its subsets) as the target symbolic language has never been questioned, despite posing several challenges to SKI algorithms, because of its expressiveness. To complicate the matter, we observe that writing a symbolic specification to be injected via general-purpose and expressive languages like FOL, Prolog [10], or Datalog, may often be cumbersome for non-experts in symbolic reasoning. In fact, a modelling effort is required to translate the domain knowledge into the target symbolic

language, and we argue that such modelling effort could be reduced by using a domain-specific language (DSL).

Accordingly, in this paper, we propose SKI-lang, a domain-specific language for SKI, which aims at being *practical* for non-experts in symbolic reasoning, while still retaining a clear semantics and a good expressiveness–tractability trade-off [11]. In particular, we discuss the engineering choices behind the design of SKI-lang, and we show how it can act as a common interface for different benchmarks from the NeSy literature. Most notably, the main goal of this paper is to motivate the need for an *ad-hoc* language for SKI – complementary to FOL – and to propose one particular syntactical reification for such a language, tailored on the current state of practice in machine learning (ML) and NeSy. The implementation is preliminary, and we do not claim completeness or generality. Instead, the paper reports about our proof-of-concept implementation of SKI-lang, and provides a roadmap for future work.

2 Background

NeSy has emerged as a significant area of research within artificial intelligence (AI), aiming to integrate symbolic reasoning with NNs learning to leverage the strengths of both symbolic and connectionist approaches. Typically, symbolic – most commonly, logic – languages are integrated to enhance NNs by enabling structured reasoning, interpretability, and explicit knowledge representation.

Information representation within these systems often combines localist (symbol-based) and distributed (sub-symbolic/neural-based) approaches [23], providing flexible and effective knowledge representation and data processing capabilities. There, training processes integrate neural-based inductive learning from data with symbolic reasoning approaches, allowing systems to learn effectively even from smaller datasets due to explicit symbolic knowledge representation.

From symbolic AI, NeSy approaches may inherit various reasoning capabilities, such as: deductive, inductive, abductive, commonsense, and combinatorial reasoning [4]. Yet, decision-making processes in these systems combine intuitive, heuristic neural processing with deliberate symbolic reasoning, closely mimicking human cognitive patterns.

---

* Corresponding Author. Email: giovanni.ciatto@unibo.it

Lastly, logic utilised in NeSy ranges from propositional to higher-order logic, offering extensive capabilities for knowledge representation and reasoning by embedding logical frameworks directly within neural architectures, thus enabling complex logical inferences.

2.1 Symbolic knowledge injection (SKI)

SKI refers to the injection of symbolic knowledge – expressed in formal logic – into sub-symbolic predictors like NNs. As defined by [5], SKI is "any algorithmic procedure affecting how sub-symbolic predictors draw their inferences in such a way that predictions are either computed as a function of, or made consistent with, some given symbolic knowledge".

SKI aims to improve model interpretability, robustness, and controllability by incorporating structured, human-intelligible prior knowledge into the learning process. Three major SKI strategies are recognised in the literature: structuring, guided learning, and embedding [5]. *Structuring*: the architecture of the predictor is built or modified to reflect the symbolic knowledge structure, e.g., via encoding rules directly as modules in a NN. *Guided learning* (a.k.a. *Constraining*): symbolic constraints are added to the loss function, guiding learning via soft penalties or hard constraints. *Embedding:* symbolic knowledge is converted into continuous representations that are fed into the predictor as part of the input.

SKI methods vary widely with respect to the kind of logic language they support. These range from propositional logic, to FOL (used in LTN [3] and LNN [21]), to probabilistic or Horn logics (used in DeepProbLog [16], NTP [20]), to Datalog (as in Scallop [12]).

Related works. Neural theorem proving (NTP) [20] implements a differentiable version of backward chaining using Horn logic. Variables are grounded via soft unification in embedding space, blending guided learning with embedding strategies. DeepProbLog [16] extends probabilistic Prolog with neural predicates and performs SKI via structuring. The logic program controls the inference pipeline, and variables are grounded via probabilistic backward chaining over dataset constants. Neuro-symbolic forward chaining (NSFR) [22] performs forward-chaining symbolic reasoning over probabilistic ground atoms. It structures symbolic inference within neural computation, and performs grounding by enumerating atoms and chaining over them. Logic tensor networks (LTN) [3] uses fuzzy FOL and injects rules via differentiable real-valued semantics that act as soft constraints in the loss function of NNs which are structured to reflect the symbolic knowledge. Therefore, this method combines the 'constraining' and 'structuring' strategies. Knowledge-enhanced neural networks (KENN) [6] works in propositional logic and injects knowledge through a knowledge enhancement layer added to a neural classifier. Rules are translated into differentiable adjustments applied post-hoc to network outputs, realising guided learning. Logic neural networks (LNN) [21] encodes weighted FOL directly in network structure. Symbolic formulas are embedded into the network via confidence scores and differentiable logic gates. This combines structuring with constrained learning. Hierarchical rule induction (HRI) [9] learns logic programs from data using meta-rules, combining inductive learning with differentiable logic. Grounding is achieved via substitution over datasets and similarity in neural embedding space. Knowledge Injection via Lambda Layer (KILL) [14] regularises NN training with symbolic knowledge in stratified Datalog with negation. Knowledge Injection via Network Structuring (KINS) [15] injects logic formulas, expressed in stratified Datalog with negation, into NN by structuring additional layers that mimic the symbolic

knowledge. Scallop [12], supports Datalog-style programs over tensors, offering differentiable symbolic reasoning. Grounding is done via batched comprehension over input data, and symbolic rules are enforced structurally. DeepLogic [7] uses structured neural logic operators over tree-based FOL expressions. It applies structuring and guided learning to learn logical forms jointly with perception.

2.2 About SKI Languages

In the realm of NeSy, the interplay between symbolic logic languages and NN architectures introduces crucial considerations around expressivity, computational tractability, and usability—particularly within ML workflows.

This discussion aims to reflect on these issues, emphasising the challenges posed by different logic languages to SKI, the various solutions adopted by existing methods, and ultimately addressing the question of how these complexities relate practically to the everyday tasks of data scientists and ML practitioners.

Expressivity vs. Tractability: A Spectrum. Symbolic languages vary significantly in expressivity and computational complexity, fundamentally influencing their usability and suitability in ML contexts.

At the lower end of the expressivity spectrum lies propositional logic, which provides simplicity and computational tractability. It allows for straightforward symbolic-to-sub-symbolic translation and integration into ML workflows, as seen in early and simpler approaches. However, propositional logic is limited to representing flat, atomic statements about domain entities and lacks the capability to generalise across instances using variables or quantifiers, limiting its practical utility in more complex ML applications.

Conversely, FOL stands at the upper end of the expressivity spectrum, empowered by variables, quantifiers, unification, and logical inference mechanisms such as resolution. This language allows for compact, intentional, and relational representations of knowledge, which are powerful within symbolic reasoning frameworks. Nonetheless, when integrating FOL into neural models – where datasets form strict subsets of the Herbrand universe – complexities emerge due to grounding (instantiation of variables), interpretation, and computational overhead.

In particular, FOL's expressive nature demands significant computational resources and sophisticated tricks to effectively use it for SKI. Common methods include: *soft grounding*, as utilised in differentiable neural logic frameworks (e.g., LTN [3]); *probabilistic backward chaining*, exemplified by systems like DeepProbLog [16]; and *embedding-based grounding*, such as the soft unification in NTP [20]. These methods essentially mitigate the complexity by translating symbolic structures into computationally manageable, differentiable forms.

Intermediate languages such as Horn clauses and Datalog – including their stratified and negation-free variants –, strike a middle ground [13]. They simplify symbolic formulations through syntactical constraints, enhancing computational tractability. However, they still present challenges, particularly when recursive definitions are involved, as recursion may lead to non-terminating grounding processes—unless managed through smart techniques like: *batched grounding*, as in Scallop's (cf. [12]) implementation for Datalog programs; or *forward chaining with probabilistic atoms*, such as in NSFR [22]. These "tricks" prevent infinite computations by limiting recursive expansions or bounding inference procedures through practical heuristics.

Practical Usability. From a purely symbolic perspective, these technical challenges offer deep theoretical interest, particularly around the foundational study of neuro-symbolic integration. Nevertheless, the critical question remains: are such complexities genuinely necessary for practical ML tasks?

Practically speaking, data scientists typically resort to SKI when raw data alone is insufficient for training robust ML models. This insufficiency arises due to data scarcity, uneven data distribution, or dataset bias—scenarios common in real-world applications such as healthcare diagnostics, fairness-aware AI, or structured data interpretation.

In these cases, symbolic knowledge becomes a complementary resource, enhancing model performance through structured constraints and prior domain knowledge. Specifically, symbols in these contexts are not referencing abstract entities but rather direct representations of data instances, their features, and relational knowledge explicitly linked to dataset columns and rows.

Along this line, SKI practitioners may just need a language that allows them to express symbolic constraints or relations over the domain of the dataset at hand, simplifying the expression of declarative statements which involve the dataset's instances (and their components), and features—rather than arbitrary Herbrand terms.

Moreover, as the *(i)* specification of the knowledge to be injected and *(ii)* the hyperparameters of the learning and *(iii)* injections processes are deeply intertwined, with a lot of back-and-forth between the two, it is crucial to provide SKI practitioners with a unified language to express both symbolic knowledge and SKI/ML workflows. This would represent a significant advantage in terms of usability and experimental setup time.

To address these concerns, in the next sections, we present SKI-lang, and its design rationale, and we attempt to demonstrate its effectiveness and usability in ordinary ML tasks where SKI may apply.

2.3 Running Examples and Benchmarks

In the remainder of this paper, we rely on running examples taken from three distinct SKI benchmarks, tailored onto as many application domains, namely: handwritten digit recognition, fairness-aware income prediction, and SKI-enhanced Poker hand classification. Each benchmark demonstrates a different way to exploit SKI and lets us showcase some feature of SKI-lang in real-world scenarios.

The benchmarks differ in data format, learning objective, and symbolic knowledge to be injected. We present them here as they will be referenced multiple times in the following section.

Sum of MNIST Digits. This benchmark is based on the well-known MNIST dataset[1] of handwritten digits. The dataset consists of 70,000 gray-scale 28×28 pixels images labelled with one of 10 digit classes. The goal is to train a neural classifier that predicts the digit class of an image, but with an additional symbolic constraint: when images are grouped in pairs, the sum of the true classes of each pair must equal the sum of their predicted classes.

The symbolic constraint thus involves a global consistency condition across *two* independent input instances. This setting highlights a key foundational challenge for SKI, namely how to enforce *relational* constraints across multiple inputs, especially when symbolic information is not local to a single instance.

Fair Income Prediction. Based on the Adult (a.k.a. Census Income) dataset[2], this benchmark addresses the problem of learning a binary income predictor (above or below $50k) from demographic and employment data. The dataset contains 48,842 tabular records with 14 features including `age`, `education`, `occupation`, and `race`. The learning task is binary classification over the `income` field.

The symbolic knowledge to be injected encodes a fairness constraint: the predicted `income` should be independent of the sensitive attribute `race`. This is commonly formalised as a *statistical parity* requirement (a.k.a. *demographic* parity, cf. [8]). From a foundational SKI perspective, this example highlights the challenge of injecting *distributional constraints*—not over individual predictions but over group-level statistics. The benchmark is also interesting because it combines numerical and categorical features, making it a test case for symbolic reasoning over mixed-type structured data.

SKI-enhanced Poker-Hand Classification. This benchmark involves training a sub-symbolic classifier over a highly imbalanced dataset of poker hands[3]. Each data instance encodes 5 playing cards through 10 attributes (5 suits and 5 ranks), and is assigned one of 10 class labels corresponding to the type of hand (e.g., pair, flush, full house, etc.).

The available symbolic knowledge consists of a rich set of *crisp* logic rules that fully characterise each class. This makes the benchmark suitable for stress-testing SKI under conditions where symbolic information is both precise and essential, due to the extreme class imbalance and low data coverage. From a foundational standpoint, this example poses challenges in terms of combining rule-based logic (e.g., multiple conditions with dependencies and precedence) with neural learning, and allows research on prioritised rule injection and expressiveness–tractability trade-offs.

3 SKI-lang: A Practical Language for SKI

Here we introduce SKI-lang, a DSL for NeSy that is specifically designed to make SKI practical for data scientists. SKI-lang is a declarative language that allows users to express symbolic knowledge in a way that is both intuitive and concise, tailoring that knowledge to the data-related task at hand – for which a dataset is supposed to be available –, and training ML predictors accordingly—in such a way that, at the end of training, they comply with the aforementioned symbolic knowledge expressed in SKI-lang.

Accordingly, in this section, we first discuss the abstract design criteria that guided our design of SKI-lang, and then we provide a brief overview of its concrete syntax and the intended semantics.

3.1 Design Criteria

SKI-lang is designed to serve the purposes of a data scientist working on some supervised ML task of interest, for which a *dataset* is available via a clear *schema*, as well as some *background knowledge* that may be worth keeping into account when training ML predictors for the task.

For the sake of simplicity – yet without loss of generality –, we describe the dataset as a table-like structure, where each row represents an instance of the domain at hand, and each column represents a feature of the dataset. We assume that features come with mnemonic

---

[1] 'MNIST' Dataset on UCI Repository: https://doi.org/10.24432/C53K8Q

[2] 'Adult' Dataset on UCI: https://doi.org/10.24432/C5XW20
[3] 'Poker Hand' Dataset on UCI: https://doi.org/10.24432/C5KW38

names, whereas instances are represented by their row number. Finally, we assume that one feature is marked as the target feature w.r.t. the supervised ML task at hand.

Under these assumptions, the core goals of SKI-lang are to provide the data scientist with (**G1**) a convenient, concise, and expressive syntax for expressing their background knowledge; as well as (**G2**) a declarative syntax for specifying the ML workflow – including the dataset schema, the predictors to be trained, and their hyperparameters –, in such a way that SKI-lang is the *only* entry point for any SKI-enhanced ML pipeline.

To address these goals, we design SKI-lang to satisfy the following requirements, enumerated by **R***i*. The discussion is deliberately abstract: please refer to section 3.2 for concrete syntactical examples.

Expressing the knowledge (G1). Firstly, and most importantly, (**R1**) SKI-lang should allow expressing knowledge about the dataset in symbolic form. More specifically, it should be possible to express declarative statements involving: *(i)* references to one or more instances from the dataset, *(ii)* references to one or more features from the dataset, *(iii)* references to one or more features of the same instance *(iv)* arbitrary constants; *(v)* named logic predicate definitions over the items above; *(vi)* any algebraic or logical combination of the items above. Such statements constitute the symbolic knowledge specification to be injected.

Furthermore, to simplify the specification of common logic statements, (**R2**) SKI-lang should support the import and usage of built-in functions and predicates, aimed at keeping the symbolic specification concise and declarative.

Finally, (**R3**) the language should be agnostic w.r.t. the particular sort of SKI approach and algorithm being used. In other words, SKI-lang should be able to express symbolic knowledge in a way that is independent to how it will be injected. In practice, this means that SKI-lang should allow for *(i)* *structuring* the architecture of the predictor out of the symbolic knowledge, *(ii)* *constraining* the loss function of the predictor with symbolic knowledge, *(iii)* *embedding* the symbolic knowledge into vectors that are fed into the predictor, or *(iv)* any combination of the above; while *(v)* requiring minimal or no changes to the specification.

Declaring the workflow (G2). To account for the declaration of end-to-end SKI workflows where all relevant aspects of the process are specified in a single place, SKI-lang should also support (**R4**) the declaration or import of the ML predictor(s) subject to SKI, and of the (**R5**) the dataset and data-schema to be used for training and testing the predictors. Similarly, it should support (**R6**) the selection of the SKI algorithm to be used, and (**R7**) the customisation of any aspect related to the SKI-aware ML pipeline.

More precisely, requirement **R4** prescribes that the predictor undergoing SKI – be it a predictor to be loaded from a file, or a new one to be trained from scratch –, should be declared in SKI-lang. Declarations should specify any modelling aspect, there including: the predictor family of choice (e.g., NNs, decision trees, etc.), its actual hyperparameters (e.g., number of layers, the number of neurons per layer, the activation functions, etc.), and its mapping to symbolic predicates. The latter, in particular, aims at declaring the interface (name + arity) the ML predictor is offering to the symbolic realm. This would allow SKI-lang users to reference the predictor in the symbolic knowledge specification, as if it were a logic symbol[4].

Similarly, requirement **R5** prescribes that the dataset(s) being used for training and testing the predictors – as well as the schema of the

data therein contained –, should be declared in SKI-lang too. This includes the dataset's name, the names of the features, the classification of features as target or non-target, and the data type of each feature—aside from the actual URLs or paths to the dataset(s) files.

Finally, requirement **R6** prescribes that SKI-lang should let the user select the SKI algorithm to be used for injecting the symbolic knowledge into the ML predictors. This implies that a few more facilities should be available to SKI-lang users, namely: *(i)* some syntactical construct to select the SKI algorithm to adopt, and *(ii)* multiple, ad-hoc parsers for adapting SKI-lang's syntax as many SKI algorithms. Requirement **R7** complements such customisability by allowing the user to customize details such as: the fuzzification and grounding strategies, the learning rate, the number of epochs, the random seeds, etc.—possibly including safe defaults.

3.2 Syntax By Examples

SKI-lang adopts a YAML-like syntax as its foundational design choice. YAML[5] is a popular and intuitive configuration language, widely adopted in the data science and software engineering communities for its clean readability and shallow nesting. Its ability to support explicit sectioning, hierarchical definitions, as well as anchors and references makes it ideal for the kind of structured yet flexible specification required in neuro-symbolic workflows. Moreover, the existence of robust and mature parsing libraries across several programming languages ensures seamless integration of SKI-lang into modern ML programming frameworks.

Each SKI-lang script is a YAML file composed of five primary sections: `data`, `optimization`, `learnables`, `knowledge`, and `constraints`. The `data` section declares the dataset(s) to be used and defines their schema, thereby addressing requirement **R5**. The `optimization` section specifies all tunable hyperparameters and SKI-related configuration options, addressing requirements **R6** and **R7**. The `learnables` section declares the structure, I/O types, and hyperparameters of the learnable sub-symbolic predictors to be trained, covering requirements **R4** and **R3** (as far as structuring is concerned). The `constraints` section encodes declarative statements that represent symbolic knowledge to be injected as constraints—hence addressing requirements **R1**, **R2**, and **R3** (as far as constraining is concerned). Finally, the `knowledge` section allows users to declare auxiliary symbolic definitions, reusable logic predicates, and domain knowledge to be referenced in the aforementioned sections—thus supporting requirements **R1**, **R2**, and **R3** (as far as embedding is concerned).

Below, we explain the intended purpose of each section, as well as the key features of SKI-lang's syntax, via a few incremental examples tailored on the benchmarks from section 2.3.

3.2.1 MNIST Example

Here we present a minimal example of SKI-lang applied to the 'sum of MNIST digits' benchmark, where a single rule involving pairs of MNIST digits is injected via constraining. Refer to listing 1 for the features described here.

Pythonic formulas. Symbolic formulas appearing in the `constraints`, `knowledge`, and `learnables` sections are expressed using a compact, Python-like syntax. These formulas consist of algebraic and logical expressions over variables and constants,

---

[4] For instance, a binary classifier may be mapped onto a logic unary predicate, where the predicate's name is the name of the predicted class.

```
1  data:
2    MNIST:
3      instances: [x, y]
4      features:
5        - {name: image, type: tensor2d(28, 28)}
6      targets:
7        - {name: value, values: 0..9}
8  learnables:
9    digit:
10     inputs: [MNIST.features]
11     outputs:
12       - {source: MNIST.targets, transform: ohe}
13     structure:
14       type: neural_network
15       layers:
16         - {type: dense, size: 128, activation: relu}
17         - {type: dense, size: 64, activation: relu}
18         - {type: dense, size: 10, activation: softmax}
19 constraints:
20   - always: digit(x)+digit(y) == x.value+y.value
```

Listing 1: SKI-lang example: Sum of MNIST Digits benchmark

```
1  data:
2    adult:
3      instances: [person]
4      features:
5        - {name: age, type: int}
6        - name: race
7          values: [White, Black, ...]
8          # ... other features here ...
9      targets:
10       - {name: income, values: ['<=50K', '>50K']}
11 learnables:
12   over50k:
13     inputs: [adult.features]
14     outputs:
15       - {source: adult.targets, transform: ordinal}
16     structure:
17       type: neural_network
18       layers:
19         - {type: dense, size: 128, activation: relu}
20         - {type: dense, size: 1, activation: sigmoid}
21 optimization:
22   batch_size: 256
23 constraints:
24   - always: over50k(adult.features) == adult.income
25   - global: SP(adult.race,over50k(adult.features))<=0.1
```

Listing 2: SKI-lang example: Fair Income Prediction

with symbols either implicitly or explicitly declared in the `data` section to ensure consistency with the dataset schema. In this way, SKI-lang allows logic constraints to refer directly to instance-level values, features, or model predictions, using intuitive dot-notation and functional application.

For example, to express the constraint that, for any pair of MNIST digits `x` and `y`, the sum of the predicted classes must equal the sum of the ground-truth classes, one can write a formula as simple as: `digit(x)+digit(y) == x.value+y.value` Unpacking the minimal example from listing 1, we can observe several key elements of SKI-lang in action, and understand how the formula above is interpreted.

The `constraints` section can be filled with a list of formulas, each one expressing a symbolic constraint to be injected. Each constraint is expressed in a natural and concise manner, with a Pythonic syntax which is familiar to most data scientists. Constraints should be prefixed by a keyword specifying their applicability scope (e.g. `always`) over the declared instance variables (e.g. `x` and `y`), which are introduced in the `data` section as independent draws from the dataset(s) therein declared. In the particular case of listing 1, the `always` keyword indicates that the constraint should be re-evaluated for every pair of instances `x` and `y` drawn from the dataset.

The rest of the `data` section declares the structure of the dataset at hand—i.e., what features and targets it contains, and of what types. In the MNIST case, each instance includes an `image` feature (represented as a 28×28 tensor, i.e. a gray-scale picture depicting a handwritten digit), and a target feature `value` ranging from 0 to 9 (representing the digit class). Hence, expressions like `x.value` and `y.value` are evaluated as the ground-truth class labels of `x` and `y`.

Learnables as the link between realms. The `learnables` section hosts *named* declarations for trainable ML predictors, and their hyperparameters, possibly expressed in terms of the dataset schema. The MNIST example from listing 1 introduces a model named `digit`, i.e.: a neural classifier aimed to predict the class of a digit, given its image. The declaration includes the model's name, along with an architectural specification (e.g., two layers with ReLU activations and a softmax output layer). Notice that the input layer is not explicitly declared, as it is automatically inferred from the dataset schema, while the output layer must be explicitly declared with an activation function which is adequate for the task at hand (here: `softmax` for multi-class classification).

Importantly, in SKI-lang, once a predictor is named (here: `digit`), it becomes callable as a logic function in symbolic expressions. Thus, `digit(x)` represents the predicted class of instance `x`, and the whole constraint can be interpreted as a symbolic equality

between predicted and ground-truth sums.

Speaking of `learnables`, it is worth focussing on the `inputs` (resp. `outputs`) sub-section: this is where the model's input and output types are declared, hence allowing for computing the shapes of its input and output layers. Such declarations may reference the dataset's attribute names, as defined in the `data` section, using the dataset's name as a global variable, and feature names as attributes (e.g., `MNIST.features`). If transformations are needed – such as one-hot encoding (OHE) for categorical features – these should be declared here too, as they must be kept into account when shaping the model's structure, yet they should be transparent to the symbolic knowledge specification—meaning that symbolic formulas would keep referring to the original features instead of the encoded ones.

Multiple instances at a time. Most notably, SKI-lang assumes that the specific variable names being used to refer to instances of a dataset are declared in the `data` section too, explicitly, under the `instances` sub-sub-section (cf. `x` and `y` in listing 1). This is not just a readability feature, but rather a crucial design choice that allows SKI-lang to declare when a SKI process is considering to multiple instances at a time.

In fact, there could be scenarios where the symbolic knowledge to be injected involves multiple instances at once. The simplest example is the 'sum of MNIST digits', where the formula to inject considers two digits at a time, and it subtends a universal quantification over all pairs of instances. Generalizing on this point, SKI-lang allows for the declaration of multiple instance variables, hence allowing for the injection of rules that involve (at maximum) all of them at once.

Declaring the multiple instances explicitly, in turns, enables SKI-lang parsers to configure data-loaders and batching strategies accordingly, ensuring that the target amount of instances are loaded altogether during training, injection, and inference.

3.2.2 Census Income Example

Here we present a minimal example of SKI-lang applied to the 'fair income prediction' benchmark, where a column-wise fairness constraint is injected via constraining into an ordinary binary classifier trained via supervised learning. We focus only on new syntactical aspects which are not already covered by the MNIST example. Refer to listing 2 for the features described here.

Column-wise expressions. Let us consider the case where fairness is computed by means of the statistical parity criterion, which

requires that the predicted `income` is independent of some sensitive attribute (say, `race`). To compute statistical parity, one needs to compare the distribution of the predicted `income` across different values of the `race` column—a dataset-wise operation, be it the training- or test-set, or just a batch.

In SKI-lang, column-wise expressions are supported by the `<dataset>.<column>` syntax, where `<dataset>` is the name of the dataset declared in the `data` section, and `<column>` is the name of some column as declared in the same section. Expressions of this form are evaluated as column tensors, allowing the application of tensor operations across the entire column. Hence, assuming that a built-in function `SP` is available to compute statistical parity among two column-tensors, the fairness constraint can be expressed as in listing 2. Another hidden assumption in there is that applying the learnable function `over50k` to a multi-dimensional tensor containing the training instances input features (e.g., `adult.features`) would yield a column-tensor containing the *predicted* `income` for each instance.

In this example, there are then two constraints being declared for injection: an ordinary supervision constraint (i.e., the *predicted* `income` should match the *expected* one) to be computed instance-wise, and a fairness constraint (i.e., the statistical parity between the *predicted* `income` across `races` should be below threshold) to be computed column-wise—hence, `globally`, i.e. once per dataset.

Optimization parameters. A common trick to implement column-wise constraints during gradient-descent-based training processes is to compute those constraints over the entire *batch*—leading to wider batches to be preferrable. To account for this and other similar cases, SKI-lang supports the specification of custom *optimization parameters* in the outer `optimization` section of the YAML configuration. In listing 2, for instance, the `batch_size` parameter is set to 256.

In general, other optimization-related parameters must be specified here, such as: *(i)* the number of training epochs, *(ii)* the learning rate, *(iii)* the random seed, *(iv)* the optimizer and *(v)* the injection algorithm to be used, etc.

Built-in functions. To simplify the specification of common symbolic constraints, SKI-lang supports referencing built-in functions. This is the case, for instance, of the `SP` function in listing 2, which computes the statistical parity between two column tensors. More generally, these are ordinary Python functions involving tensors as arguments, and returning tensors as results. These can be provided as built-in symbols upon calling the SKI-lang parser, and allow for a more concise specification of common symbolic constraints.

Despite their simplicity, the possibility to plug additional functions to simplify the expression of symbolic logic is a key engineering feature of our approach. This is where SKI-lang becomes the substratum upon which SKI algorithms engineers can build re-usable functions for expressing symbolic knowledge to be reused.

3.2.3 Poker Hand Example

Here we present a minimal example of SKI-lang applied to the 'poker hand classification' benchmark, where the learning process may greatly benefit from symbolic knowledge, which is in turn quite complex to express. We focus only on new syntactical aspects which are not already covered by previous examples. Refer to listing 3 for the features described here.

Test-set separation. SKI-lang naturally supports the separation of training- and test-sets, by allowing the user to declare them in sep-

```yaml
data:
  PokerHand:
    train: &dataset
      file: https://site.com/poker-hand.zip
      unpack: path/to/training.data
      type: csv(',')
    test: {<<: *dataset, file: path/to/test.data}
    instance_name: hand
    features:
      - &suits
        name: suit1
        values: {hearts: 1, ..., clubs: 4}
        group: suits
      - &ranks
        name: rank1
        values: {ace: 1, '2': 2, ..., king: 13}
        group: ranks
      # ... other features here ...
      - {<<: *suits, name: suit5}
      - {<<: *ranks, name: rank5}
    target:
      name: Class
      values: [nothing, one_pair, ..., royal_flush]
learnable:
  poker_hand:
    # NN specification here
knowledge:
  Pair:
    args: [hand]
    clause: min_repetitions(2, hand.ranks)
  TwoPairs:
    args: [hand]
    clause:
      2 <= sum(r1==r2 for r1,r2 in combs(hand.ranks, 2))
  # ... other rules here ...
  RoyalFlush:
    args: [hand]
    clause: Flush(hand) & MaximumStraight(hand)
constraints:
  - always: poker_hand(hand.features) == hand.Class,
    weight: 1
  - if: RoyalFlush(hand)
    then: poker_hand(hand.features) == royal_flush
    weight: 9
  # ... other constraints here ...
  - if: Pair(hand)
    then: poker_hand(hand.features) == one_pair
    weight: 1
```

Listing 3: SKI-lang example: Poker Hand Classification

arate subsections of the `data` section. As exemplified in listing 3, the `train` (resp. `test`) subsection declares the training-set (resp. test-set), and they both allow for the indication of a file – possibly remote, possibly to be unpacked from an archive – and the data format (e.g. CSV) of the file contents. They also allow for selecting different samples from the same dataset file, indicating the split percentage.

Handy features from YAML. Being YAML-based, SKI-lang naturally supports the use of[6] anchors (`&name`), aliases (`*name`), and merge keys (`<<: *name`) to avoid code duplication and promote reusability. In this way, repetitive information can be declared once and reused across the script. Choosing meaningful anchor names may help in retaining the declarativeness of the code.

In listing 3, for instance, this feature is used to avoid repeating details between the training- and test-set declarations, as well as to shorten the dataset features' declaration considerably (extensive lists of values for categorical/ordinal features must be written only once).

Background Knowledge. The `knowledge` section allows for the declaration of reusable logic predicates, which can be referenced in the `constraints` and `learnable` sections, in order to keep the constraining or structuring specifications concise and declarative, and to avoid code duplication.

As exemplified in listing 3, the `knowledge` section declares a set of logic predicates, indexed by their name (to avoid name clashes). Each predicate comes with a list of *formal* argument names (`args`), – which can be considered either as logic variables or as references to unknown tensors, depending on the mindset – and a `clause`, which is a Pythonic formula that defines when the predicate holds

---

[6] cf. https://archive.ph/PobLI

as a function of its arguments. Technically speaking, the body of the clause is a Python expression which should return a boolean value – when interpreted as a logic formula – or a scalar tensor in the range $[0, 1]$—when interpreted numerically.

Handy features from Python. Being Pythonic, SKI-lang also supports the use of *comprehensions* to enumerate over multiple items at once. This is particularly useful to make complex rules more concise and declarative.

Consider for instance the case of the `TwoPairs` rule in listing 3. This rule states how to compute whether a numeric tensor representing a poker hand – namely, a vector of the form `[suit1, rank1, ..., suit5, rank5]` – is a two-pairs hand. Computationally, the rule considers only the rank-related features of the input tensor (i.e., `[rank1, ..., rank5]`), and all possible 2-sized combinations of them (via an ad-hoc built-in function `combs`); counting how many combinations are composed by equals ranks. If the count is greater than 2, then the hand is classified as a two-pairs hand. Thanks to generator comprehensions, the rule specification is concise and readable, and it matches the Python implementation directly.

Weighted & Guarded Constraints. Finally, SKI-lang allows for the specification of *weighted* constraints, possibly marked by a *guard* condition which describes when the constraint should be enforced. These features are particularly useful when rules in the `constraints` section are *not* mutually exclusive, like in the case of the Poker hand classification benchmark. For instance, in that benchmark, the `TwoPairs` rule is not mutually exclusive with the `Pair` one: when the first is satisfied, the second is certainly satisfied too. When this is the case, SKI-lang allows for (cf. listing 3) the specification of *(i)* a guard – prefixed by the `if` keyword – which describes when (i.e. for which instances in the dataset) the rule prefixed by `then` should be enforced, and, optionally, *(ii)* a `weight` value, which is a scalar value in the range $\mathbb{R}_{\geq 0}$ defining the relative importance of the constraint w.r.t. to other constraints in the same section.

Of course, both guards and constraints can refer to predicates defined in the `knowledge` section, and can be combined with other constraints via logical operators. Furthermore, despite the particular interpretation of weights is up to the SKI algorithm being used, but they are guaranteed to be normalized w.r.t. the total sum of all weights, and they are commonly implemented as multiplicative factors when constraints are turned into penalties in loss functions.

4 Implementation Status and Roadmap

Technologically speaking, SKI-lang is a working prototype, implemented in Python 3.10, and built on top of well-known ML libraries such as `PyTorch` [18], `SciKit-Learn` [19], and `Pandas` [17]. The source code is available on Anonymous4Science[7], for public inspectability and reproducibility.

At the current stage of development, the implementation acts as a parser for SKI-lang scripts, whose content is then exploited to automate: *(i)* the loading of the training and test datasets as `Pandas` data frames, *(ii)* their preprocessing (e.g., normalization, encoding, etc.), via `SciKit-Learn`'s application programming interfaces (APIs), *(iii)* the instantiation of `PyTorch` modules to represent the learnable predictors, *(iv)* the instantiation of `PyTorch` data-loaders to load the datasets in batches, *(v)* the configuration of `PyTorch` optimizers to train the aforementioned predictors, *(vi)* the fuzzification of symbolic

---

[7] cf. https://anonymous.4open.science/r/skilang-68AC

Figure 1: Results of applying SKI in the Fair Income Prediction benchmark (cf. 2.3) via SKI-lang.
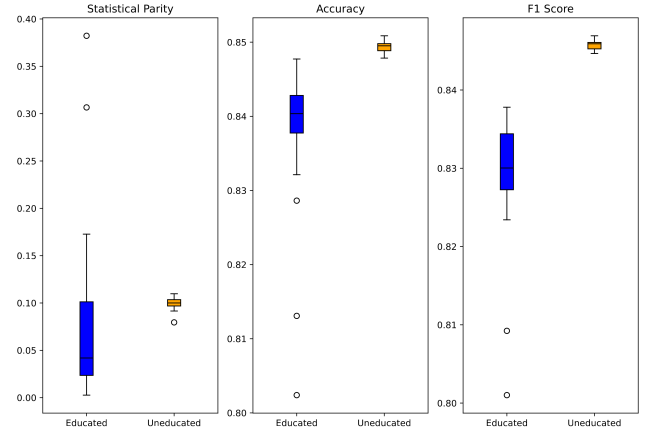
knowledge into the predictors' loss functions via `PyTorch`'s API, and, finally, *(vii)* the training of the predictors, again via `PyTorch`.

It is worth mentioning that the current implementation assumes that training is performed via stochastic gradient descent (SGD) optimizers, in turns relaying on batches of instances being loaded from the training set. The batching here is particularly important because logic constraints are evaluated over batches, rather than over the entire training set, which is a common practice in SKI literature making the "batch size" a crucial hyperparameter to be tuned.

Limitations and Future Interventions. While the current architecture is stable, the implementation is still a work in progress, and some features are still under development, while others are already ready for use. In particular, requirements from **R1** to **R7** are already supported, despite with some minor limitations. Details about the current limitations and our plans to address them are following.

While **R1** is fully satisfied at the syntactical level, meaning that all sorts of expressions prescribed by the requirement can be expressed in SKI-lang, the implementation currently lacks support for injecting expressions involving two or more instances at once. In fact, expressions of such sorts would require custom data-loaders to be implemented, sampling the Cartesian power of training sets—and a general (supporting N instances at once, with parametric N) solution is still under development.

Requirement **R3** is partially satisfied, as the current implementation only supports the injection of symbolic knowledge as *constraints*, while the *structuring* of predictors from symbolic expressions is ignored by the parser. Again, filling this gap is a work in progress, requiring ad-hoc converters from Pythonic formulæ to neural structures to be implemented – similarly, to what happens in [15] – on top of `PyTorch`'s API, with minimal or no changes to the SKI-lang syntax.

Finally, despite allowing for the customisation of learning parameters such as the learning rate, the number of epochs, etc., requirements **R6** and **R7** are still mostly unsupported, as the current implementation relies on a single SKI *structuring* algorithm, and a single fuzzification strategy. In fact, despite the framework is designed to let developers plug in new SKI algorithms and fuzzification strategies – by providing abstract APIs that implementers can extend and override – the implementation currently only ships no alternative algorithms or strategies. This is a deliberate implementation choice: we wanted to stabilise the syntax and the software architecture first, while leaving the door open for future contributions – from either the community or ourselves – to implement additional algorithms and

strategies. The rationale here is straightforward: widening the coverage of SKI-lang's supported algorithms and strategies takes time, effort, and care, hence this goal should be pursued incrementally.

Demonstrative Experiments. To demonstrate the functionality of SKI-lang in its current implementation, we report experiments on the Fair Income Prediction benchmark, as described in sections 2.3, and 3.2.2. An overview of the results is shown in section 4. As the reader can notice, SKI-lang is effective in injecting symbolic knowledge into the predictors, leading to significant improvements in the fairness of the predictions at the expense of a slight decrease in predictive performance (both accuracy and F1-score). The phenomenon is expected, and it is known in the literature as the *accuracy–fairness trade-off*.

5   Conclusions

In this paper, we introduced SKI-lang, a DSL designed to make SKI practical and accessible for data scientists. We discussed its design rationale, provided concrete syntax examples, and demonstrated its applicability to well-known neuro-symbolic benchmarks. Our preliminary implementation shows that SKI-lang can effectively streamline SKI workflows and facilitate the integration of symbolic knowledge into ML pipelines.

Future work will focus on extending algorithm support, improving customisability, and further evaluating SKI-lang across diverse application domains.

Acknowledgements

References

[1] A. Agiollo, A. Rafanelli, M. Magnini, G. Ciatto, and A. Omicini. Symbolic knowledge injection meets intelligent agents: Qos metrics and experiments. *Autonomous Agents and Multi-Agent Systems*, 37(2), June 2023. ISSN 1573-7454. doi: 10.1007/s10458-023-09609-6. URL http://dx.doi.org/10.1007/s10458-023-09609-6.

[2] M. Ajtai and Y. Gurevich. Datalog vs first-order logic. *Journal of Computer and System Sciences*, 49(3):562–588, Dec. 1994. ISSN 0022-0000. doi: 10.1016/s0022-0000(05)80071-6. URL http://dx.doi.org/10.1016/S0022-0000(05)80071-6.

[3] S. Badreddine, A. S. d'Avila Garcez, L. Serafini, and M. Spranger. Logic tensor networks. *Artif. Intell.*, 303:103649, 2022. doi: 10.1016/J.ARTINT.2021.103649. URL https://doi.org/10.1016/j.artint.2021.103649.

[4] B. P. Bhuyan, A. Ramdane-Cherif, R. Tomar, and T. P. Singh. Neuro-symbolic artificial intelligence: a survey. *Neural Computing and Applications*, 36(21):12809–12844, June 2024. ISSN 1433-3058. doi: 10.1007/s00521-024-09960-z. URL http://dx.doi.org/10.1007/s00521-024-09960-z.

[5] G. Ciatto, F. Sabbatini, A. Agiollo, M. Magnini, and A. Omicini. Symbolic knowledge extraction and injection with sub-symbolic predictors: A systematic literature review. *ACM Computing Surveys*, 56(6):161:1–161:35, June 2024. ISSN 0360-0300. doi: 10.1145/3645103. URL https://dl.acm.org/doi/10.1145/3645103.

[6] A. Daniele and L. Serafini. Knowledge enhanced neural networks for relational domains. In A. Dovier, A. Montanari, and A. Orlandini, editors, *AIxIA 2022 - Advances in Artificial Intelligence - XXIst International Conference of the Italian Association for Artificial Intelligence, AIxIA 2022, Udine, Italy, November 28 - December 2, 2022, Proceedings*, volume 13796 of *Lecture Notes in Computer Science*, pages 91–109. Springer, 2022. doi: 10.1007/978-3-031-27181-6_7. URL https://doi.org/10.1007/978-3-031-27181-6_7.

[7] X. Duan, X. Wang, P. Zhao, G. Shen, and W. Zhu. Deeplogic: Joint learning of neural perception and logical reasoning. *IEEE Trans. Pattern Anal. Mach. Intell.*, 45(4):4321–4334, 2023. doi: 10.1109/TPAMI.2022.3191093. URL https://doi.org/10.1109/TPAMI.2022.3191093.

[8] C. Dwork, M. Hardt, T. Pitassi, O. Reingold, and R. S. Zemel. Fairness through awareness. In S. Goldwasser, editor, *Innovations in Theoretical Computer Science 2012, Cambridge, MA, USA, January 8-10, 2012*, pages 214–226. ACM, 2012. doi: 10.1145/2090236.2090255. URL https://doi.org/10.1145/2090236.2090255.

[9] C. Glanois, Z. Jiang, X. Feng, P. Weng, M. Zimmer, D. Li, W. Liu, and J. Hao. Neuro-symbolic hierarchical rule induction. In K. Chaudhuri, S. Jegelka, L. Song, C. Szepesvári, G. Niu, and S. Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 7583–7615. PMLR, 2022. URL https://proceedings.mlr.press/v162/glanois22a.html.

[10] P. KÖRNER, M. LEUSCHEL, J. BARBOSA, V. S. COSTA, V. DAHL, M. V. HERMENEGILDO, J. F. MORALES, J. WIELEMAKER, D. DIAZ, S. ABREU, and G. CIATTO. Fifty years of prolog and beyond. *Theory and Practice of Logic Programming*, 22(6):776–858, May 2022. ISSN 1475-3081. doi: 10.1017/s1471068422000102. URL http://dx.doi.org/10.1017/S1471068422000102.

[11] H. J. Levesque and R. J. Brachman. Expressiveness and tractability in knowledge representation and reasoning. *Computational Intelligence*, 3:78–93, 1987. doi: 10.1111/j.1467-8640.1987.tb00176.x. URL https://doi.org/10.1111/j.1467-8640.1987.tb00176.x.

[12] Z. Li, J. Huang, and M. Naik. Scallop: A language for neurosymbolic programming. *Proc. ACM Program. Lang.*, 7(PLDI):1463–1487, 2023. doi: 10.1145/3591280. URL https://doi.org/10.1145/3591280.

[13] M. Magnini, G. Ciatto, and A. Omicini. On the design of psyki: A platform for symbolic knowledge injection into sub-symbolic predictors. In D. Calvaresi, A. Najjar, M. Winikoff, and K. Främling, editors, *Explainable and Transparent AI and Multi-Agent Systems - 4th International Workshop, EXTRAAMAS 2022, Virtual Event, May 9-10, 2022, Revised Selected Papers*, volume 13283 of *Lecture Notes in Computer Science*, pages 90–108. Springer, 2022. doi: 10.1007/978-3-031-15565-9_6. URL https://doi.org/10.1007/978-3-031-15565-9_6.

[14] M. Magnini, G. Ciatto, and A. Omicini. A view to a KILL: Knowledge injection via lambda layer. In A. Ferrando and V. Mascardi, editors, *WOA 2022 – 23rd Workshop "From Objects to Agents"*, volume 3261 of *CEUR Workshop Proceedings*, pages 61–76. CEUR-WS.org, Genova, Italy, Nov. 2022. URL http://ceur-ws.org/Vol-3261/paper5.pdf.

[15] M. Magnini, G. Ciatto, and A. Omicini. KINS: Knowledge injection via network structuring. In R. Calegari, G. Ciatto, and A. Omicini, editors, *CILC 2022 – Italian Conference on Computational Logic*, volume 3204 of *CEUR Workshop Proceedings*, pages 254–267, Bologna, Italy, 2022. CEUR-WS.org. URL http://ceur-ws.org/Vol-3204/paper_25.pdf.

[16] R. Manhaeve, S. Dumancic, A. Kimmig, T. Demeester, and L. D. Raedt. Neural probabilistic logic programming in deepproblog. *Artif. Intell.*, 298:103504, 2021. doi: 10.1016/J.ARTINT.2021.103504. URL https://doi.org/10.1016/j.artint.2021.103504.

[17] W. McKinney. Data structures for statistical computing in python. In S. van der Walt and J. Millman, editors, *Proceedings of the 9th Python in Science Conference (SciPy 2010), Austin, Texas, June 28 - July 3, 2010*, pages 56–61. scipy.org, 2010. doi: 10.25080/MAJORA-92BF1922-00A. URL https://doi.org/10.25080/Majora-92bf1922-00a.

[18] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Z. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. M. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. B. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 8024–8035, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/bdbca288fee7f92f2bfa9f7012727740-Abstract.html.

[19] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. VanderPlas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in python. *J. Mach. Learn. Res.*, 12:2825–2830, 2011. doi: 10.5555/1953048.2078195. URL https://dl.acm.org/doi/10.5555/1953048.2078195.

[20] T. Rocktäschel and S. Riedel. End-to-end differentiable proving. In I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information*

Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA, pages 3788–3800, 2017. URL https://proceedings.neurips.cc/paper/2017/hash/b2ab001909a8a6f04b51920306046ce5-Abstract.html.

[21] P. Sen, B. W. S. R. de Carvalho, R. Riegel, and A. G. Gray. Neuro-symbolic inductive logic programming with logical neural networks. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pages 8212–8219. AAAI Press, 2022. doi: 10.1609/AAAI. V36I8.20795. URL https://doi.org/10.1609/aaai.v36i8.20795.

[22] H. Shindo, D. S. Dhami, and K. Kersting. Neuro-symbolic forward reasoning. *CoRR*, abs/2110.09383, 2021. URL https://arxiv.org/abs/2110.09383.

[23] T. van Gelder. *Why Distributed Representation is Inherently Non-Symbolic*, pages 58–66. Springer Berlin Heidelberg, 1990. ISBN 9783642760709. doi: 10.1007/978-3-642-76070-9_6. URL http://dx.doi.org/10.1007/978-3-642-76070-9_6.