Journal of Artificial Intelligence Research 78 (2023) 579-617 Submitted 5/2023; published 11/2023

Scalable Neural-Probabilistic Answer Set Programming

Arseny Skryagin ARSENY.SKRYAGIN@CS.TU-DARMSTADT.DE
Daniel Ochs DANIEL.OCHS@CS.TU-DARMSTADT.DE
Devendra Singh Dhami DEVENDRA.DHAMI@CS.TU-DARMSTADT.DE
Computer Science Department, TU Darmstadt
Darmstadt, Germany

Kristian Kersting KERSTING@CS.TU-DARMSTADT.DE
Computer Science Department, TU Darmstadt
German Research Center for Artificial Intelligence (DFKI)
Darmstadt, Germany

Abstract

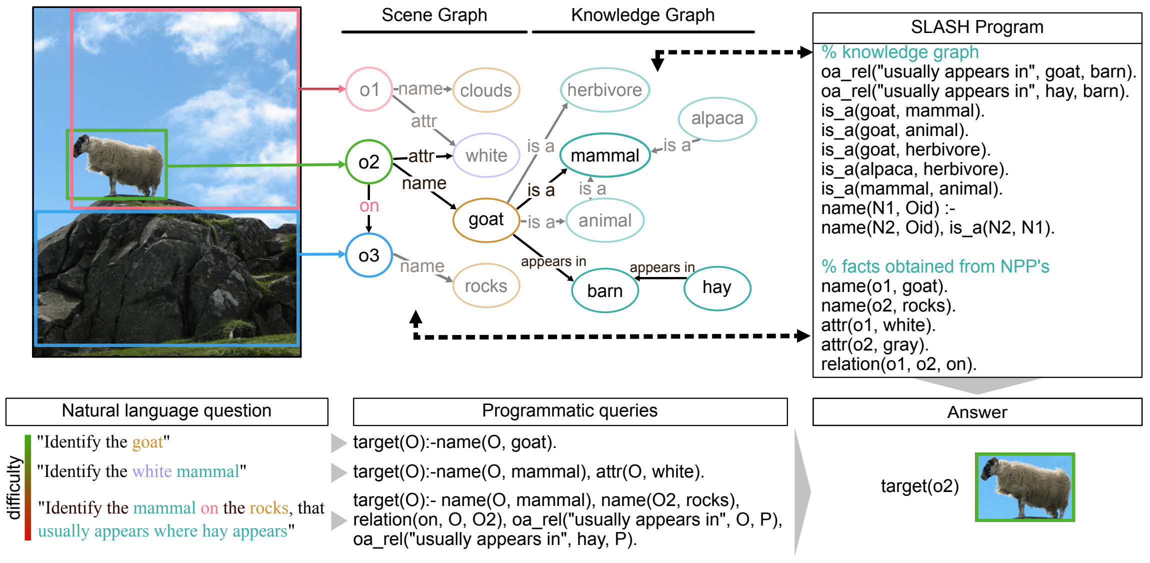
The goal of combining the robustness of neural networks and the expressiveness of symbolic methods has rekindled the interest in Neuro-Symbolic AI. Deep Probabilistic Programming Languages (DPPLs) have been developed for probabilistic logic programming to be carried out via the probability estimations of deep neural networks (DNNs). However, recent SOTA DPPL approaches allow only for limited conditional probabilistic queries and do not offer the power of true joint probability estimation. In our work, we propose an easy integration of tractable probabilistic inference within a DPPL. To this end, we introduce SLASH, a novel DPPL that consists of Neural-Probabilistic Predicates (NPPs) and a logic program, united via answer set programming (ASP). NPPs are a novel design principle allowing for combining all deep model types and combinations thereof to be represented as a single probabilistic predicate. In this context, we introduce a novel $+/-$ notation for answering various types of probabilistic queries by adjusting the atom notations of a predicate. To scale well, we show how to prune the stochastically insignificant parts of the (ground) program, speeding up reasoning without sacrificing the predictive performance. We evaluate SLASH on various tasks, including the benchmark task of MNIST addition and Visual Question Answering (VQA).

1. Introduction

Neuro-symbolic AI approaches to learning (Hudson & Manning, 2019; d'Avila Garcez et al., 2019; Jiang & Ahn, 2020; d'Avila Garcez & Lamb, 2023) are on the rise. They integrate low-level perception with high-level reasoning by combining data-driven neural modules with logic-based symbolic modules. This combination of sub-symbolic and symbolic systems has shown many advantages for various tasks such as visual question answering (VQA) and reasoning (Yi et al., 2018), concept learning (Mao et al., 2019) and improved properties for explainable and revisable models (Ciravegna et al., 2020; Stammer et al., 2021).

Rather than designing specifically tailored neuro-symbolic architectures, where the neural and symbolic modules are often disjoint and trained independently (Yi et al., 2018; Mao et al., 2019; Stammer et al., 2021), deep probabilistic programming languages (DPPLs) provide an exciting alternative (Bingham et al., 2019; Tran et al., 2017; Manhaeve et al., 2018; Yang et al., 2020; Huang et al., 2021). Specifically, DPPLs integrate neural and symbolic modules via a unifying programming framework with probability estimates acting

©2023 The Authors. Published by AI Access Foundation under Creative Commons Attribution License CC BY 4.0.



SLASH Program

```
% knowledge graph
oa_rel("usually appears in", goat, barn).
oa_rel("usually appears in", hay, barn).
is_a(goat, mammal).
is_a(goat, animal).
is_a(goat, herbivore).
is_a(alpaca, herbivore).
is_a(mammal, animal).
name(N1, Oid) :-
name(N2, Oid), is_a(N2, N1).

% facts obtained from NPP's
name(o1, goat).
name(o2, rocks).
attr(o1, white).
attr(o2, gray).
relation(o1, o2, on).
```

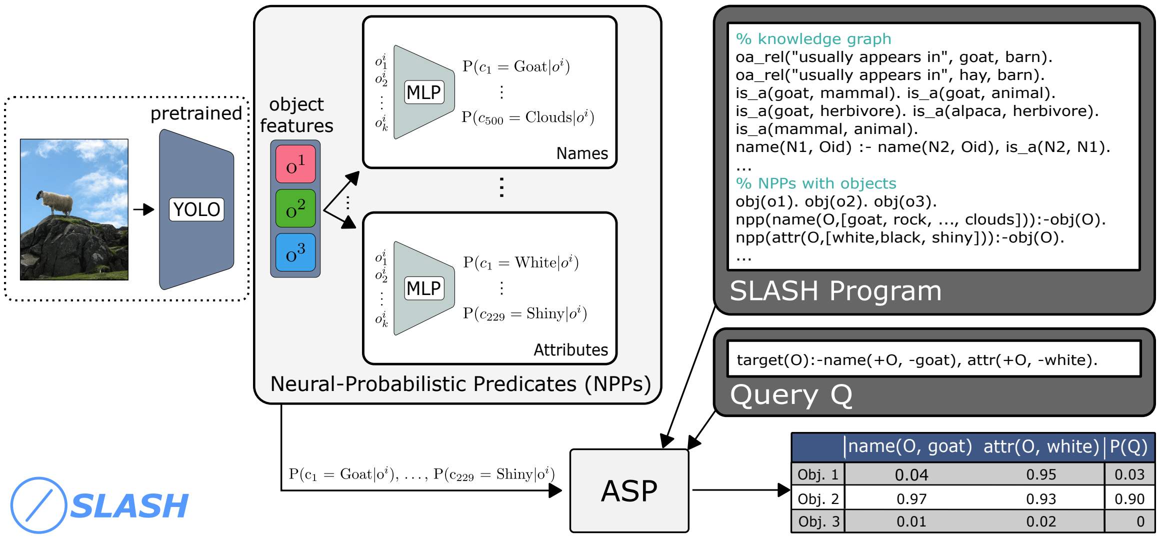
```
target(O):-name(O, goat).
target(O):-name(O, mammal), attr(O, white).
target(O):- name(O, mammal), name(O2, rocks),
relation(on, O, O2), oa_rel("usually appears in", O, P),
oa_rel("usually appears in", hay, P).
```

Figure 1: **The VQA task:** Proposed by Huang et al. (2021), given the features and bounding boxes of objects in an image, the goal is to answer a question requiring multi-hop reasoning. A model is learned that predicts a scene graph consisting of names, attributes, and relations. Additionally, a fixed knowledge graph is given, extending the scene graph with commonsense knowledge. Questions are provided as queries in programmatic form and can vary in complexity with the clause length of the query. Together, the knowledge and scene graph are used to infer the correct answer for the query.

as the *"glue"* between separate modules, thus allowing for reasoning over noisy, uncertain data and, importantly, for joint training of the modules. Additionally, prior knowledge and biases in the form of logic rules can easily and explicitly be added to the learning process with DPPLs. This stands in contrast to specifically tailored, implicit architectural biases of, e.g., purely subsymbolic deep learning approaches. Ultimately, DPPLs thereby allow the easy integration of neural networks (NNs) into downstream logical reasoning tasks.

Recent state-of-the-art DPPLs, such as DeepProbLog (Manhaeve et al., 2018), NeurASP (Yang et al., 2020) and Scallop (Huang et al., 2021) allow for conditional class probability estimates as all three works base their probability estimates on neural predicates. We argue that it is necessary to integrate joint probability estimates into DPPLs, to allow for solving a broader range of tasks. The world is uncertain, and it is necessary to reason in settings in which variables of an observation might be missing or even manipulated. Furthermore, scalability is a central problem for DPPLs. In many applications such as VQA, the solution spaces necessarily grow exponentially with the size of the search spaces, rendering inference computationally infeasible.

Hence, we make the following contributions in this work, addressing limitations in expressing the full range of probabilistic inference types and scalability of DPPLs. First, we propose a novel form of predicates termed Neural Probabilistic Predicates (NPPs, cf. Fig. 2), that allow for task-specific probability queries. NPPs consist of neural and/or probabilistic circuit (PC) modules and act as a unifying term, encompassing the neural predicates of


pretrained
YOLO
object features
o^1
o^2
o^3
o^i_1
o^i_2
o^i_k
MLP
$P(c_1 = \mathrm{Goat}|o^i)$
$P(c_{500} = \mathrm{Clouds}|o^i)$
Names
MLP
$P(c_1 = \mathrm{White}|o^i)$
$P(c_{229} = \mathrm{Shiny}|o^i)$
Attributes
Neural-Probabilistic Predicates (NPPs)
$P(c_1 = \mathrm{Goat}|o^i), \ldots, P(c_{229} = \mathrm{Shiny}|o^i)$
ASP
SLASH
SLASH Program
Query Q

```
% knowledge graph
oa_rel("usually appears in", goat, barn).
oa_rel("usually appears in", hay, barn).
is_a(goat, mammal). is_a(goat, animal).
is_a(goat, herbivore). is_a(alpaca, herbivore).
is_a(mammal, animal).
name(N1, Oid) :- name(N2, Oid), is_a(N2, N1).
...
% NPPs with objects
obj(o1). obj(o2). obj(o3).
npp(name(O,[goat, rock, ..., clouds])):-obj(O).
npp(attr(O,[white,black, shiny])):-obj(O).
...
```

```
target(O):-name(+O, -goat), attr(+O, -white).
```

	name(O, goat)	attr(O, white)	P(Q)
Obj. 1	0.04	0.95	0.03
Obj. 2	0.97	0.93	0.90
Obj. 3	0.01	0.02	0

Figure 2: **VQA task with SLASH:** NPPs consist of neural and/or probabilistic circuit modules and can produce task-specific probability estimates. A YOLO Network and MLPs form the Neural-Probabilistic Predicates for the VQA task. In our novel DPPL, SLASH, NPPs are integrated with a logic program via an ASP module to answer logical queries about data samples. Each MLP computes the conditional distribution for classes c_i given the YOLO feature encodings z_i shared across all NPPs, such as names or attributes. The relation's NPP is omitted for simplicity. One gets task-related probabilities by sending queries to the NPPs, e.g., conditional probabilities for visual reasoning tasks.

DeepProbLog, NeurASP and Scallop, as well as purely probabilistic predicates. Further, we introduce a much more powerful *"flavor"* of NPPs that consists jointly of neural and PC modules, taking advantage of the power of neural computations together with true density estimation of PCs via tractable probabilistic inference.

Second, having introduced NPPs, we construct SLASH[1], a novel DPPL, which efficiently combines NPPs with logic programming. Similar to the punctuation symbol, this can be used to efficiently combine several paradigms into one. Specifically, SLASH represents for the first time an efficient programming language that seamlessly integrates probabilistic logic programming with neural representations and tractable probabilistic estimations. This allows for the integration of all forms of probability estimations, not just class conditionals, thus extending the important works of Manhaeve et al. (2018), Yang et al. (2020) and Huang et al. (2021).

Third, as NPPs become more complex, navigating the solution space becomes more time-consuming. To speed up inference, Scallop (Huang et al., 2021) used top-k to prune unlikely paths in their proof tree using output probabilities of deep neural networks (DNNs). With PCs (as NPP) it is, however, difficult to select the correct k and instead, we go for top-k%. It is based on the observation that for each query there are many possible solutions, but only few of them are plausible. And during the training from these few only one Solution thAt MatchEs the data (SAME). That is, SAME keeps k% of SLASH's solutions to compute (probabilistic) answers. This greatly speeds up inference, as illustrated in Fig. 3.

1. Code is available at https://github.com/ml-research/SLASH

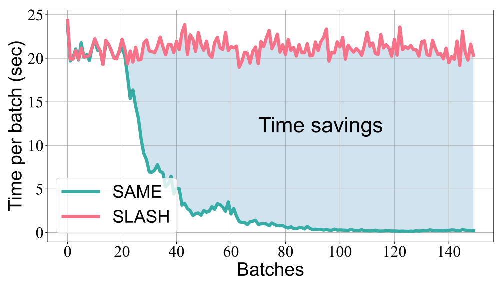


Figure 3: **SAME helps SLASH to reduce training time:** E.g., on the MNIST T3 task per batch, SAME prunes unlikely outcomes, which reduces the training time of SLASH. The more batches we have seen during the training, the more we learn which outcomes are unlikely and can be pruned.

Moreover, SAME allows SLASH to scale to VQA, as implemented in SLASH in Fig. 2. Here, every NPP gets object-detection outputs, in this case from the YOLO network (Redmon et al., 2016), as inputs and produces class conditionals for names, attributes, and relations. A user defines a set of statements and rules in the form of a logic program. Finally, given the query as in Fig. 2, SLASH gives the expected answer.

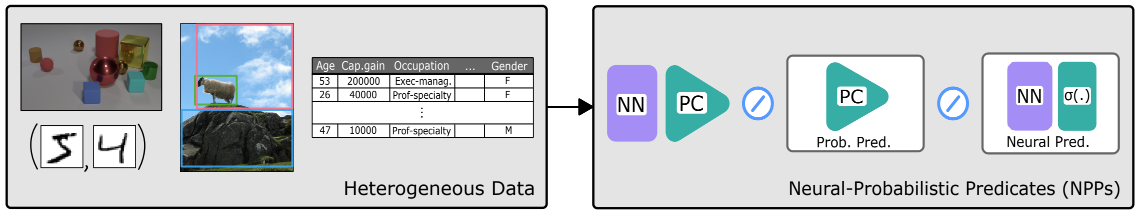
The present paper is a significant extension of a previously published conference paper (Skryagin et al., 2022) and presents SAME and how to use it to scale SLASH to the VQA. Further, we extend this previous work with a detailed ablation study: Empirical results of the set prediction task carried out on the CLEVR dataset (Johnson et al., 2017), the benchmark task of MNIST-Addition (Manhaeve et al., 2018), and Sudoku (Yang et al., 2020) present the advantages coming with SAME.

In summary, we make the following contributions:

- introduce neural-probabilistic predicates,
- efficiently integrate answer set programming (ASP) with probabilistic inference via NPPs within our novel DPPL, SLASH,
- introduce SAME to dynamically prune unlikely NPP outcomes, thus allowing a reduction in the complexity of computing potential solutions,
- effectively train neural, probabilistic and logic modules within SLASH for complex data structures end-to-end via a simple, single loss term,
- show that the integration of NPPs in SLASH provides various advantages across various tasks and data sets compared to state-of-the-art DPPLs and neural models.

These contributions demonstrate the advantage of probabilistic density estimation via NPPs and the benefits of a “one system – two approaches” (Bengio, 2019) framework that can successfully be used for performing various tasks and on many data types.

We proceed as follows. First, we introduce NPPs and how they can be queried via the $+/-$ notation. Next, SLASH programs are presented with the corresponding semantics and parameter learning. Afterward, we discover SAME using top-k%. Before concluding, we support our findings with experimental evaluation.



Age	Cap.gain	Occupation	...	Gender
53	200000	Exec-manag.		F
26	40000	Prof-specialty		F
		⋮		
47	10000	Prof-specialty		M

Figure 4: **NPPs come in various flavors:** Depending on the data set and underlying task, SLASH requires a suitable Neural-Probabilistic Predicate (NPP) to compute query-dependent probability estimates. NPPs can be composed of neural and probabilistic modules, or (depicted via the slash symbol) only one of these two.

2. SLASH through NPPs and Vice Versa

We begin this section by first introducing the novel neural probabilistic predicates (NPPs) framework. After this, we introduce our DPPL, SLASH, which easily integrates NPPs via ASP with logic programming and end this section with the learning procedure in SLASH, allowing us to train all modules via a joint loss term.

2.1 Neural-Probabilistic Predicates and Rules

Previous DPPLs, DeepProbLog (Manhaeve et al., 2018) and NeurASP (Yang et al., 2020), introduced the *Neural Predicate* as an annotated-disjunction or as a propositional atom, respectively, in order to obtain conditional class probabilities, $P(C|X)$, from the softmax function at the output of an arbitrary NN. As mentioned in the introduction, this approach only allows for $P(C|X)$ to be computed, but not $P(X|C)$, $P(C)$ or $P(C, X)$. To overcome this limitation, we introduce *Neural-Probabilisitic Predicates* (NPPs).

Formally, we denote with

$$npp\,(h(x), [v_1, \dots, v_n]) \tag{1}$$

a Neural-Probabilistic Predicate h. Where (i) *npp* is a reserved word to label a NPP, and (ii) h a symbolic name of either a PC, NN or a joint of a PC and NN. Fig. 4 (right) depicts all three variants. Fig. 2 (right, 'SLASH Program'-block) uses *name* for h. Additionally, (iii) x denotes a "term" and (iv) $v_1, \dots, v_n$ are the n possible outcomes of h. Following the example of Fig. 2, the outcomes for *name* are (*goat, rocks, . . . , clouds*). A NPP abbreviates a rule of the form $c = v$ with $c \in \{h(x)\}$ and $v \in \{v_1, \dots, v_n\}$. Furthermore, we denote with Π^{npp} a set of NPPs of the form stated in (Eq. (1)) and r^{npp} the set of all rules $c = v$ of one NPP, which denotes the possible outcomes, obtained from a NPP in Π^{npp}, e.g., $r^{name} = \{c = \mathit{Goat}, c = \mathit{Rocks}, \dots, c = \mathit{Clouds}\}$ for the example depicted in Fig. 2. Rules in the following form

$$npp\,(h(x), [v_1, \dots, v_n]) \leftarrow \mathit{Body} \tag{2}$$

are used as an abbreviation for application to multiple entities, e.g., multiple object features plus bounding boxes for the VQA task (cf. Fig. 2). Here, the *Body* of the rule is identified by $\top$ (tautology, true) or $\bot$ (contradiction, false) during grounding. Rules of the form *Head* $\leftarrow$ *Body* with r^{npp} appearing in *Head* are prohibited for Π^{npp}.

In this work, we use NPPs that contain PCs, which allow for tractable density estimation and modelling of joint probabilities. The term PC (Choi et al., 2020) represents a unifying framework encompassing all computational graphs that encode probability distributions and guarantee tractable probabilistic modelling. These include Sum-Product Networks (SPN) (Poon & Domingos, 2011), which are deep mixture models represented via a rooted directed acyclic graph with a recursively defined structure. In this way, with PCs it is possible to answer a much richer set of probabilistic queries, e.g. $P(X, C)$, $P(X|C)$, $P(C|X)$ and $P(C)$.

In addition to NPPs based purely on PCs, we introduce the arguably more interesting type of NPP that combines a neural module with a PC. Here, the neural module learns to map the raw input data into an optimal latent representation. The PC, in turn, learns to model the joint distribution of these latent variables and produces the final probability estimates. This type of NPP nicely combines the representational power of neural networks with the advantages of PCs in probability estimation and query flexibility. These combined NPPs can be partially pretrained or trained end-to-end. In the VQA example, we utilize a pretrained YOLO network with an MLP predicting class conditional probabilities. In object-centric learning, we train a slot-attention module and PCs over the latent representations end-to-end (see Sec. 4.4).

To make the different probabilistic queries distinguishable in a SLASH program, we follow the mode declarations used in inductive logic programming (ILP), and denote the input variable with + and the output variable with −. E.g., within the example of VQA (cf. Fig. 2, 'Query Q' (right)), with the query $name(+X, -C)$ one is asking for $P(C|X)$ with C being the class and X the object features. If we chose a PC as the underlying network (c.f. Sec. 4.4 and 4.2) we can model the joint distribution $P(X, C)$. Similarly, with $name(-X, +C)$ one is asking for $P(X|C)$ and, finally, with $name(+X, +C)$ for $P(X, C)$. In the case where no data is available, i.e, $name(-X, -C)$, we are querying for the prior $P(C)$.

To summarize, a NPP consists of neural and/or probabilistic modules and produces query-dependent probability estimates. Due to the flexibility of its definition, the term NPP contains the predicates of previous works (Manhaeve et al., 2018; Yang et al., 2020), but also the more interesting predicates discussed above. The specific *flavor* of a NPP should be chosen depending on what type of probability estimation is required (cf. Fig 4).

2.2 SLASH: a Novel DPPL for Integrating NPPs

Now we have everything together to introduce SLASH, a novel DPPL which efficiently integrates NPPs with logic programming.

2.2.1 SLASH Language and Semantics

We continue in the pipeline, Fig. 2, with the question of how the probability estimates of NPPs may be used for answering logical queries, and begin by formally defining a **SLASH program**.

A SLASH program Π is the union of Π^{asp}, Π^{npp}. Where, Π^{asp} is the set of propositional rules (standard rules from ASP-Core-2 (Calimeri et al., 2020)), and Π^{npp} is a set of Neural-Probabilistic Predicates of the form stated in Eq. (1).

Similar to NeurASP, SLASH requires ASP and, as such, adopts its **syntax** for the most part, which includes neural probabilistic rules as defined in Eq. (2). Compared to Prolog, ASP rarely goes into an infinite loop during solving and is therefore preferable as a backbone. For example, the program $p(X)$:-$q(X)$. $q(X)$:-$p(X)$. $query(p(a))$. would not terminate in Prolog, due to the solver trying to unroll endlessly, whereas ASP would result in unsatisfiability. To illustrate, let us revisit the example of VQA as in Fig. 1. A YOLO network detected three objects *o1*, *o2* and *o3* in the image. The task is to *name* each of the objects as either *goat*, *rock*, ... or *clouds*. The overall *target* here is to find an object *goat*:

$$\begin{aligned}&\textit{obj}(\textit{o1}).\quad \textit{obj}(\textit{o2}).\quad \textit{obj}(\textit{o3}).\\ &\textit{npp}(\textit{name}(O, [\textit{goat}, \textit{rock}, \ldots, \textit{clouds}])) \leftarrow \textit{obj}(O).\\ &\textit{target}(O) \leftarrow \textit{name}(+O, \textit{-goat}).\end{aligned}$$

Fig. 1 presents one further SLASH program for the task of VQA, exemplifying a set of propositional rules and neural predicates.

Now, let us define the **semantics** of SLASH. To this end, we show how to integrate NPPs into an ASP-compatible form to obtain the **success probability for a query** given all potential solutions, i.e., stable models. A **query** is an ASP constraint of the form $\leftarrow$ *Body*, i.e., it is a headless rule. To translate the program Π, the rules (Eq. (2)) will be rewritten as follows:

$$1\{h(x) = v_1; \ldots; h(x) = v_n\}1. \tag{3}$$

The ASP-solver should understand this as "Pick exactly one rule from the set". After the translation is done, we can ask an ASP-solver for the solutions for Π.

Next, let us assume that we have a query Q for which we want to compute the probability; keep in mind that NPPs introduce random choices. Since all the potential solutions $I \models Q$ (Q is true in I) for the query Q are mutually exclusive, there are possible worlds, the probability $P_\Pi(Q)$ of Q is the sum of probabilities $P_\Pi(I)$ of each single solution, i.e., stable model of Q:

$$P_\Pi(Q) := \sum_{I \models Q} P_\Pi(I)\ . \tag{4}$$

So, we are left with computing the probability $P_\Pi(I)$ of a single solution I. Here, only the NPPs are contributing to the probability; all other atoms are simply true and have the probability 1. The (ground) NPPs, however, are also independent of each other. Consequently, for each object c and random choice v, we can multiply together the probabilities of $c = v$ and normalize by the number of objects c:

$$P_\Pi(I) = \begin{cases} \frac{\prod_{c=v \in I|_{r^{npp}}} P_\Pi(c=v)}{|I|_{r^{npp}}|}, & \text{if } I \text{ is pot. sol. of } \Pi,\\ 0, & \text{otherwise.}\end{cases} \tag{5}$$

where $I|_{r^{npp}}$ is the subset of ground NPP, r^{npp} in the solution I, $r^{npp} \subseteq I$.

With the success probability $P_\Pi(Q)$ of a single query at hand, the success probability of a set of queries $\mathbf{Q}$ can naturally be written as

$$P_\Pi(\mathbf{Q}) := \prod_{i=1}^{l} P_\Pi(Q_i) = \prod_{i=1}^{l} \sum_{I \models Q_i} P_\Pi(I)\ , \tag{6}$$

since they are independent of each other. With the semantics specified, we are ready to learn the parameters of SLASH programs.

2.2.2 Parameter Learning in SLASH

To estimate the parameters $\boldsymbol{\theta}$ of a SLASH program $\Pi(\boldsymbol{\theta})$, we are following the *learning from entailment* setting, as also used for DeepProbLog (Manhaeve et al., 2018). That is, we estimate $\boldsymbol{\theta}$ from a set $\mathbf{Q}$ of positive examples only, i.e., each training example is a logical query that is known to be true in the SLASH program $\Pi(\boldsymbol{\theta})$. Thereby, $\Pi(\boldsymbol{\theta}) = \Pi^{asp}(\boldsymbol{\theta}) \cup \Pi^{npp}(\boldsymbol{\theta})$ holds. Since $\Pi^{asp}(\boldsymbol{\theta})$ has no weighted rules, i.e., $P_{\Pi^{asp}(\boldsymbol{\theta})} = 1$, we want to find optimal parameters $\boldsymbol{\theta}$ for r^{npp}, i.e., the optimal NPP parameters. The reader will recall that learning symbolic modules is ambiguous. E.g, in inductive logic programming (ILP), see Cropper et al. (2022), the term stands for finding the rules best describing the query. Hereafter, we are using the term in the sense of finding potential solutions satisfying the given query (cf. previous subsection).

To achieve parameter learning in SLASH, we employ an additive loss function. The first part is the *entailment* loss, i.e., the NPPs are fixed, and we maximize the success probability of the query set $\boldsymbol{Q}$. The second part concerns the NPPs (neural networks/ probabilistic circuits) only. So, we want to maximize the probability given the data while the “logical” part is fixed. Thus, the loss function takes the following form

$$L_{SLASH} = L_{ENT} + L_{NPP} \ , \tag{7}$$

and we seek to minimize the loss, e.g., by running coordinate descent. Let us begin with the NPP loss.

NPP loss – The aim of this loss function is to maximize the joint probability of $P_\theta^{(X_\mathbf{Q},C)}(x_\mathbf{Q})$. To omit possibly vanishing values, we apply $\log(\cdot)$ instead and define

$$L_{NPP} := -\log\left(P_\theta^{(X_\mathbf{Q},C)}(x_\mathbf{Q})\right) = -\sum_{i=1}^{l} \log(P_\theta^{(X_\mathbf{Q},C)}(x_{Q_i})) \ , \tag{8}$$

where

- $X_\mathbf{Q}$ are the random variables modeling the training set X associated with the set of the queries $\mathbf{Q}$,
- $x_\mathbf{Q}$ are realizations of $X_\mathbf{Q}$ associated with $\mathbf{Q}$,
- $P^{(X_\mathbf{Q},C)}(x_\mathbf{Q})$ is the probability of the realizations $x_\mathbf{Q}$ estimated by the NPP modelling the joint distribution over the set $X_\mathbf{Q}$ and C – the set of classes (the domain of the NPP cf. Eq. (1)),
- and θ is the parameter set associated with the NPP.

Additionally, we derive the derivative of the NPP loss function, which will be called upon during training with coordinate descent. Formally, we write

$$\frac{\partial}{\partial\theta} L_{NPP} = -\sum_{i=1}^{l} \frac{1}{P_\theta^{(X_\mathbf{Q},C)}(x_{Q_i})} \cdot \frac{\partial}{\partial\theta}\left(P_\theta^{(X_\mathbf{Q},C)}(x_{Q_i})\right) \ . \tag{9}$$

Now, we proceed with the entailment loss.

Entailment loss – We begin with Eq. (6). Dealing with probabilities, we might end up with vanishingly small values due to the product. To resolve this, we apply $\log(\cdot)$ to both sides of the equation and obtain

$$\log(P_{\Pi(\boldsymbol{\theta})}(\mathbf{Q}) = \sum_{i=1}^{l} \log\left(\sum_{I \models Q_i} P_{\Pi(\boldsymbol{\theta})}(I)\right). \tag{10}$$

Since our goal is to give *feedback* from the success log-probability Eq. (10) to our NPPs, we multiply it with the log-probabilities of the NPPs, so that the result lands in the same space

$$\log(P_{\Pi(\theta)}(\mathbf{Q})) \cdot \log\left(P^{(X_{\mathbf{Q}},C)}(x_{\mathbf{Q}})\right), \tag{11}$$

which will turn out to be mathematically convenient later on in the proof of Thm. 2. More precisely, we want Eq. (11) to resonate with every class encoded as a possible outcome v_j as defined in Eq. (1) and with every query Q_i from $\mathbf{Q}$

$$\begin{aligned} &\sum_{i=1}^{l}\sum_{j=1}^{n} \log(P_{\Pi(\theta)}(Q_i)) \cdot \log\left(P^{(\mathbf{Q},C_j)}(x_{Q_i})\right) = \\ &\log LH\left(\log(P_{\Pi(\theta)}(\mathbf{Q})), P^{(X_{\mathbf{Q}},C)}(x_{\mathbf{Q}})\right) =: L_{ENT}. \end{aligned} \tag{12}$$

In the above, we used the definition of the cross-entropy loss to compound every single query i and outcome j to the single term of the *entailment loss.* We remark that this definition of the loss function is valid regardless of the NPP's form (NN with Softmax, PC or PC jointly with NN). The only difference will be the second term, e.g., $P^{(C|X_{\mathbf{Q}})}(x_{\mathbf{Q}})$ or $P^{(X_{\mathbf{Q}}|C)}(x_{\mathbf{Q}}))$ depending on the NPP and task. This loss function aims at maximizing the estimated success probability for a set of Queries. However, for NPPs to notice the *feedback* Eq. (11) we must make Eq. (10) compatible with the log-probabilities of NPPs.

Gradients of the entailment loss – We denote the vector $\log\left(P^{(X_{\mathbf{Q}},C)}(x_{\mathbf{Q}})\right)$ as $\mathbf{p}$ and consider the derivative $\frac{\partial \log\left(P_{\Pi(\theta)}(\mathbf{Q})\right)}{\partial \mathbf{p}}$. As we will later on, this will serve as the communication bridge between $\log(P_{\Pi(\boldsymbol{\theta})}(\mathbf{Q}))$ and $\mathbf{p}$. So, we write,

$$\sum_{i=1}^{l} \frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q_i)\right)}{\partial \mathbf{p}} \times \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}} = \sum_{i=1}^{l} \frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q_i)\right)}{\partial \boldsymbol{\theta}}, \tag{13}$$

reminding ourselves that $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ can be computed as usual via backward propagation through the NPPs. If within the SLASH program, $\Pi(\boldsymbol{\theta})$, the NPP passes the data tensor through a NN first, i.e., the NPP models a joint over the NN's output variables by a PC, then we rewrite Eq. (13) to

$$\sum_{i=1}^{l} \frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q_i)\right)}{\partial \mathbf{p}} \times \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}} \times \frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{\kappa}} = \sum_{i=1}^{l} \frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q_i)\right)}{\partial \boldsymbol{\theta}}. \tag{14}$$

Where $\boldsymbol{\kappa}$ is the set of the NN's parameters and, again, we compute $\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{\kappa}}$ via backward propagation.

Algorithm 1 Gradient computation

```
Input: $P_\Pi(c = v_j), j = 1, \dots, n$, set of $I \models Q$
 1: $P_\Pi(I) \leftarrow$ compute_pot_sol_prob($I$)  # cf. Eq. (5)
 2: $P_\Pi(Q) \leftarrow$ compute_query_prob($P_\Pi(I)$)  # ($:= \kappa$) Normalization, cf. Eq (4)
 3: grads $\leftarrow \emptyset$
 4: for every $c = v_j$ do # cf. Eq. (3)
 5:     grad $\leftarrow 0$
 6:     for every pot. sol. $I$ do
 7:         if $I \models c = v$ then
 8:             grad $\leftarrow$ grad $+ (P_\Pi(I)/P_\Pi(c = v))$  # ($:= \alpha$) Reward
 9:         else
10:             grad $\leftarrow$ grad $- (P_\Pi(I)/P_\Pi(c = v'))$ # ($:= \beta$) Penalty
11:         end if
12:     end for
13:     grads $\leftarrow$ append(grads, grad$/P_\Pi(Q)$)
14: end for
15: return grads
```

Now, $\frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \mathbf{p}}$ is left to be determined. Thus, following the definition from NeurASP (Yang et al., 2020), we write

$$\frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \mathbf{p}} := \left(\overbrace{\sum_{\substack{I:I\models Q,\\ I\models c=v}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v)}}^{:=\alpha} - \overbrace{\sum_{\substack{I,v':I\models Q,\\ I\models c=v',v\neq v'}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v')}}^{:=\beta} \right) \cdot \overbrace{\frac{1}{P_\Pi(Q)}}^{:=\gamma} . \quad (15)$$

Reading the right-hand side of this definition we recognize the three terms: Inside the parentheses, from the reward α is the penalty β subtracted, and the result is normalized with the probability of the query γ, cf. Eq. (4). As can be seen from the definition, running inference is sufficient. And so, having defined the gradients in Eq. (15), we will examine them. The following theorem shows the limit of the gradient vector.

Theorem 1 (Gradients' Limit). *Let $\Pi(\boldsymbol{\theta})$ be a fixed program with a given query Q. Further, m denotes a training iteration, then the following holds for $\frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \mathbf{p}}$ as defined in* (15)*:*

$$\left(\frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \mathbf{p}}\right)_m \xrightarrow{m\to\infty} \left(-1, \dots, -1, \underbrace{1}_{j}, -1, \dots, -1\right).$$

Thereby, the index j corresponds to $c = v$ and any other to $c = v', v \neq v'$.

Proof. W.l.o.g, we assume the program $\Pi(\theta)$ to entail a single NPP, and it can be called upon more than once in a single rule r_{npp}. Besides, a NPP can converge "perfectly", i.e., $P_{\Pi(\boldsymbol{\theta})}(c = v) = 1$ and $P_{\Pi(\boldsymbol{\theta})}(c = v') = 0$ for $v \neq v'$. To answer the question of how such limit values are possible in the first place, we make the observation on the right-hand side of (15),

$$\sum_{\substack{I:I\models Q\\ I\models c=v}} \tfrac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v)} \cdot P_{\Pi(\boldsymbol{\theta})}(c=v) + \sum_{\substack{I,v':I\models Q\\ I\models c=v',v\neq v'}} \tfrac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v')} \cdot P_{\Pi(\boldsymbol{\theta})}(c=v') =$$

$$\sum_{\substack{I:I\models Q\\ I\models c=v}} P_{\Pi(\boldsymbol{\theta})}(I) + \sum_{\substack{I,v':I\models Q\\ I\models c=v',v\neq v'}} P_{\Pi(\boldsymbol{\theta})}(I) = \sum_{I:I\models Q} P_{\Pi(\boldsymbol{\theta})}(I) = P_{\Pi}(Q).$$

As reward, penalty, and normalization constant are defined before the theorem:

$$\alpha := \sum_{\substack{I:I\models Q\\ I\models c=v}} \tfrac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v)}, \quad \beta := \sum_{\substack{I,v':I\models Q\\ I\models c=v',v\neq v'}} \tfrac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v')}, \quad \gamma := \sum_{I:I\models Q} P_{\Pi(\boldsymbol{\theta})}(I),$$

we conclude that

$$\gamma \geq \alpha - \beta \qquad \text{and} \qquad \alpha + \beta \geq \gamma. \tag{*}$$

Now, we consider the following case discrimination based on the training iteration k:

(i) *For $m = 0$:* At the start of the training, the probabilities of n outcomes are either uniformly distributed (the probability of each outcome $P_{\Pi(\boldsymbol{\theta})}(c = v_j), j \in 1, \dots, n$ is the same) or there are small numerical differences. Here, we consider the first possibility and the latter is identical to (ii). Since the probability for each outcome is the same value, we conclude due to (*) that $\alpha = \beta$ and $\frac{\alpha-\beta}{\gamma} = 0$ for the index j. In case that the same NPP being called upon multiple times, an ASP solver will derive potential solutions without consideration of symmetries. Consequently, we have to swap the numerical values obtained in the previous case for α and β. Nonetheless, we obtain the same gradient value for such a case, i.e., 0. For the rest of the indices, $\alpha = 0$ and all values being pulled to β. Hence, we obtain the negative gradient value of $-\frac{2\beta}{\gamma}$ for the rest of the indices.

(ii) *For any $1 \leq m < \infty$:* At the index j - since $0 < \gamma \leq 1$ holds, and due to (*) we get

$$\alpha - \beta \leq \gamma \mid : \gamma \quad \Longleftrightarrow \quad \frac{\alpha - \beta}{\gamma} \leq 1. \tag{**}$$

In case that the same NPP is called upon multiple times, we multiply (**) with -1 and obtain

$$-1 \leq \frac{\beta - \alpha}{\gamma}.$$

For all other indices, we have $-\frac{\beta}{\gamma}$. If $|\beta| > \gamma$, then $-\frac{\beta}{\gamma} < -1$ occurs as well.

(iii) *For $m = \infty$*, if the NPP fully converged, then we have two cases to distinguish: The index j and all other entries of the gradient's vector. We know from Eq. (4) and (5) that γ is equal to 1. Thus, we can focus entirely on $\alpha - \beta$. We therefore conclude that the entries of the gradient's vector are $1 - 0 = 1$ for the index j and $0 - 1 = -1$ otherwise.

□

Following the theorem, the training is done by the principle *winner takes all* if there are more than two NPP's outcomes, and *zero-sum game* otherwise. Hence, we are left with the sign function of the gradient vector, and the convergence in itself, can be thought of as a gradient clipping. The results presented by Yang et al. (2020) show that this works on some problems with little or no loss of accuracy, cf. Seide et al. (2014). Extrapolating from the gradient's vector limit, we see only one outcome to be rewarded, and so only one of the set of all potential solutions matching the data per NPP's call. This observation is the heart of the next section and will be discussed in detail.

Now, it is of great interest to derive the gradients of the *entailment loss* L_{ENT} (12) so that the expression $\frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q_i)\right)}{\partial \mathbf{p}} \times \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ from the left-hand side of Eq. (13) becomes amenable to back-propagation. For this purpose, we formulate the

Theorem 2 (Gradient with respect to entailment loss). *The average derivative of the logical entailment loss function L_{ENT} defined in Eq. (12) can be estimated as follows*

$$\frac{1}{l}\frac{\partial}{\partial \mathbf{p}} L_{ENT} \geq \frac{1}{l}\sum_{i=1}^{l} \frac{\partial \log(P_{\Pi(\theta)}(Q_i))}{\partial \mathbf{p}} \cdot \log(P^{(X_{\mathbf{Q}},C)}(x_{Q_i})).$$

Proof. We begin with the definition of the **cross-entropy** for two vectors y_i and $\hat{y}_i$:

$$-H(y_i,\hat{y}_i) := -\sum_{j=1}^{n} y_{ij}\cdot\log\left(\frac{1}{\hat{y}_{ij}}\right) = -\sum_{j=1}^{n}\left(y_{ij}\cdot\underbrace{\log(1)}_{=0} - y_{ij}\cdot\log(\hat{y}_{ij})\right) = \sum_{j=1}^{n} y_{ij}\cdot\log(\hat{y}_{ij}).$$

Hereafter, we substitute

$$y_i = \log(P_{\Pi(\theta)}(Q_i)) \qquad \text{and} \qquad \hat{y}_i = P^{(X_{\mathbf{Q}},C)}(x_{Q_i}),$$

and thus obtain $-H(y_i,\hat{y}_i) =$

$$-H\left(\log(P_{\Pi(\theta)}(Q_i)), P^{(X_{\mathbf{Q}},C)}(x_{Q_i})\right) = \sum_{j=1}^{n} \log(P_{\Pi(\theta)}(Q_i))\cdot\log\left(P^{(X_{\mathbf{Q}},C_j)}(x_{Q_i})\right). \tag{16}$$

We remark that n represent the number of classes defined in the domain of an NPP. Now, we differentiate the equation (16) with the respect to $\mathbf{p}$ depicted as in Eq. (15) to be the label of the probability of an atom $c = v_j$ in r^{npp}, denoting $P_{\Pi(\boldsymbol{\theta})}(c = v_j)$. Since differentiation is a linear operation, the product rule is applicable directly:

$$-\frac{\partial}{\partial \mathbf{p}} H(y_i,\hat{y}_i) = \sum_{j=1}^{n}\left[\frac{\partial \log(P_{\Pi(\theta)}(Q_i))}{\partial \mathbf{p}}\cdot\log\left(P^{(X_{\mathbf{Q}},C_j)}(x_{Q_i})\right) + \log(P_{\Pi(\theta)}(Q_i))\cdot\frac{\partial \log\left(P^{(X_{\mathbf{Q}},C_j)}(x_{Q_i})\right)}{\partial \mathbf{p}}\right]. \tag{17}$$

We want to avoid considering the latter term of $\log(P_{\Pi(\theta)}(Q_i))\cdot\frac{\partial \log\left(P^{(X_{\mathbf{Q}},C_j)}(x_{Q_i})\right)}{\partial \mathbf{p}}$ because it represents the rescaling $(\log(P_{\Pi(\theta)}(Q_i))\cdot\mathbf{1})$ and to keep the first since SLASH procure

$\frac{\partial \log(P_{\Pi(\theta)}(Q_i))}{\partial \mathbf{p}}$ following Eq. (15). To achieve this, we derive the following lower bound of the equation from above:

$$-\frac{\partial}{\partial \mathbf{p}} H\left(y_i, \hat{y}_i\right) \geq \sum_{j=1}^{n} \frac{\partial \log(P_{\Pi(\theta)}(Q_i))}{\partial \mathbf{p}} \cdot \log\left(P^{(X_{\mathbf{Q}}, C_j)}(x_{Q_i})\right). \tag{18}$$

Furthermore, under i.i.d assumption we obtain from the definition of likelihood

$$LH(y, \hat{y}) = \prod_{i=1}^{l} LH(y_i, \hat{y}_i),$$

and from this negative likelihood coupled with the knowledge that the log-likelihood of y_i is the log of a particular entry of $\hat{y}_i$

$$\begin{aligned} L_{ENT} = \log LH(y, \hat{y}) &= \sum_{i=1}^{l} \log LH(y_i, \hat{y}_i) = \sum_{i=1}^{l} \sum_{j=1}^{n} y_{ij} \cdot \log(\hat{y}_{ij}) \\ &= \sum_{i=1}^{l} \left[\sum_{j=1}^{n} y_{ij} \cdot \log(\hat{y}_{ij}) \right] = -\sum_{i=1}^{l} H(y_i, \hat{y}_i). \end{aligned}$$

Finally, applying inequality (18), we obtain the following estimate

$$\frac{1}{l} \frac{\partial}{\partial \mathbf{p}} L_{ENT} = -\frac{1}{l} \sum_{i=1}^{l} \frac{\partial}{\partial \mathbf{p}} H(y_i, \hat{y}_i) \geq \frac{1}{l} \sum_{i=1}^{l} \frac{\partial \log(P_{\Pi(\theta)}(Q_i))}{\partial \mathbf{p}} \cdot \log\left(P^{(X_{\mathbf{Q}}, C)}(x_{Q_i})\right).$$

Also, we note that the mathematical transformations listed above hold for any type of NPP and task dependent queries (NN with Softmax, PC or PC jointly with NN). The only difference will be the second term, i.e., $\log(P^{(C|X_{\mathbf{Q}})}(x_{Q_i}))$ or $\log(P^{(X_{\mathbf{Q}}|C)}(x_{Q_i}))$ depending on the NPP and task. An NPP in the form of a single PC modeling the joint distribution over $X_{\mathbf{Q}}$ and C was depicted in the example. □

In summary, we have covered the parameter learning within SLASH since the gradients for both L_{NPP} and L_{ENT} have been derived, and thus, know the gradients of L_{SLASH}. Importantly, with the learning schema described above, it is now possible, with SLASH, to simply incorporate specific task and data requirements into the logic program. And we do not require a novel loss function for each individual task and data set. The training loss, however, remains the same.

3. Scaling SLASH with SAME

In the following, we focus on the potential solutions $I \models Q$. Already, according to Eq. (4), we know that the probability of a query is the sum of the probabilities of all potential solutions. However, the question of how many of them match the data x belonging to the query Q remains. Discussing Thm. 1 (Gradients' Limit), we saw that gradients converge to reward only one outcome v_j.

We examine this observation on the digit addition task as it was originally proposed by Manhaeve et al. (2018). The goal is to train an NPP to recognize digits given the sum of the two. For example, consider the query *sum2*(, ,10). The potential solutions for this query are:

$$sum2(1,9,10), sum2(2,8,10), sum2(3,7,10), sum2(4,6,10), sum2(5,5,10),$$
$$sum2(6,4,10), sum2(7,3,10), sum2(8,2,10), sum2(9,1,10).$$

From the above, only *sum2*(3,7,10) corresponds to the given data. This means we always generate all potential solutions for the given query, although only one corresponds to the data assigned to the query. In the following, we formulate SAME (Solution thAt MatchEs the data), a technique to focus only on such potential solutions over time and dynamically reduces the computation time spent deriving all potential solutions. In the following, we abbreviate with SAME the usage of SAME within SLASH.

For every query Q SLASH answers, it produces a set of all potential solutions $\mathcal{I}$. With the growing size of NPP's domain n, $\mathcal{I}$ grows exponentially. Having multiple NPPs with considerable domain size, we might end-up with a computationally infeasible set $\mathcal{I}$ to obtain.

During training, we observe that the probability distribution $P_\Pi(c = v_i)$ as defined below Eq. (1) becomes skewed independent of the chosen inference type through $+/-$ notation for every data entry x assigned to the query Q. I.e., with the progressing training's iteration, fewer and fewer NPP's outcomes v_i contain the vast majority of the critical mass, or more formally

$$\sum_j P_\Pi(c = v_j) \leq t. \tag{19}$$

Thereby, t represents some preset threshold of, e.g., 99%. Furthermore, we know that at the beginning of training $P_\Pi(c = v_j) = \frac{1}{n}$ applies for all v_i, $1 \leq j \leq n$. Thus, the disjunction in Eq. (3) consists of n elements and $\mathcal{I}_0 = \mathcal{I}$. Repeatedly applying Eq. (19), we expect the aforementioned disjunction to entail fewer elements with every further training iteration. I.e., there exists an order such that

$$\sum_e P_\Pi(c = v_e) < \sum_j P_\Pi(c = v_j) \quad \text{with} \quad j \neq e, 1 \leq j, e \leq n \tag{20}$$

We refer to the Algorithm 2 of SAME in pseudocode form as a summary of the considerations made. It depicts how SAME is used when computing all potential solutions. Consequently,

$$\mathcal{I}_0 \supseteq \mathcal{I}_1 \supseteq \mathcal{I}_2 \supseteq \ldots \supseteq \mathcal{I}_m \tag{21}$$

is a formal description of our expectations, and $|\mathcal{I}_m| = 1$ for $m \to \infty$. I.e., among all potential solutions, there exists only one potential solution aligning the data with the query. Together with Eq. (7) and (19) we formulate the following theorem.

Theorem 3 (Convergence of SAME). *Eq.* (21) *holds.*

Proof. We follow the principal of contraposition. W.l.o.g., there exists $m \in \mathbb{N}$ such that $\mathcal{I}_m \subseteq \mathcal{I}_{m+1}$ holds and not in contrary $\mathcal{I}_m \supseteq \mathcal{I}_{m+1}$. I.e., the set of the potential solutions in an $m+1$ iteration entails more elements than the set in the previous iteration, or more formally

Algorithm 2 Potential Solutions with SAME

```
Input: $P_\Pi(c = v_j), j = 1, \dots, n,\ t,\ \Pi^{asp}$
 1: $\Pi^{npp} \leftarrow \emptyset$   # initialize the set of NPP, cf. Eq. (3)
 2: for every $c = v_j$ do
 3:     $\text{prob}_{\text{sort}} \leftarrow \text{sort}(P_\Pi(c = v_j))$
 4:     # add indices of outcomes $v_j$ until $\sum_j P_\Pi(c = v_j) \leq t$ with SAME
 5:     $\text{idx} \leftarrow \text{get\_idx}(\text{prob}_{\text{sort}}, t)$
 6:     # then truncate disjunction $1\{h(x) = v_1; \dots; h(x) = v_n\}1.$ accordingly
 7:     $\Pi^{npp} \leftarrow \text{extend}(\Pi^{npp}, \text{get\_disj}(\text{idx}))$
 8: end for
 9: $\Pi = \Pi^{npp} \cup \Pi^{asp}$
10: $I \leftarrow \text{asp\_solver}(\Pi)$
11: return $I$
```

$|\mathcal{I}_{m+1}| \geq |\mathcal{I}_m|$. Furthermore, if this tendency remains to be true for every subsequent iteration, we obtain

$$\mathcal{I}_m \subseteq \mathcal{I}_{m+1} \subseteq \mathcal{I}_{m+2} \subseteq \dots \subseteq \mathcal{I}_{m+s} \quad \text{with} \quad s \in \mathbb{N}.$$

Since any $\mathcal{I}_{m+s}$ cannot entail more entries than the set of all potential solutions, we conclude

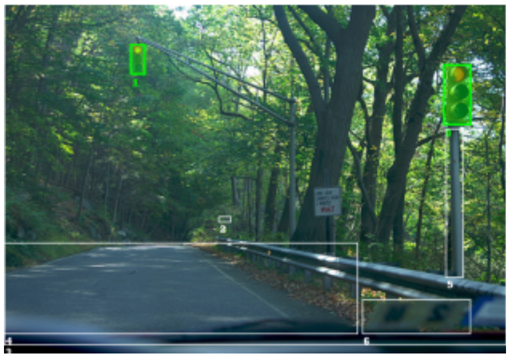
$$\mathcal{I}_{m+s} \to \mathcal{I} = \mathcal{I}_0 \quad \text{for} \quad s \to \infty. \tag{22}$$

We have shown that SAME would add more and more potentials solutions until it reaches the upper bound of all potential solutions which coincide with the query Q. All of the above is true for any arbitrary $m \in \mathbb{N}$, thereby completing the proof. $\square$

Following Thm. 3 (Convergence of SAME), we should choose the threshold t (cf. Eq. 19) to be as high as possible, to guarantee $|\mathcal{I}_m| = 1$ for $m \to \infty$ to hold. Thus, setting t to 99% is a good heuristic, as smaller values for t might be insufficient for having the optimal performance. In the next section, we provide empirical evidence for this phenomenon and for the advantages SAME's utilization brings.

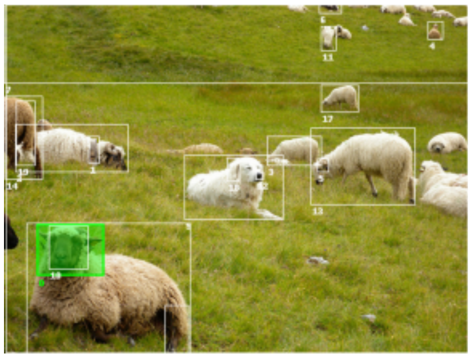
4. Experimental Evaluations

Previously (Skryagin et al., 2022), we showed that the main advantage of SLASH lies in the efficient integration of any combination of neural, probabilistic and symbolic computations. This work extends these findings with new experimental evaluations for SLASH with SAME. In particular, we show how SAME is essential for using SLASH for VQA. Afterward, we conduct an ablation study to evaluate the advantages coming from this combination. For this, we revisit the MNIST addition as conducted by Huang et al. (2021), Sudoku by Yang et al. (2020), and the set prediction task as proposed by Locatello et al. (2020). For different experiments, we had to choose different values for the threshold t. In particularly, for Sudoku, we choose 99.9999% to achieve the best possible performance. For the other experiments, 99% was already sufficient for optimal solving.



target(*O0*) :- *name*(*+O0*, *-N0*), *oa_rel*(*is_used_for*, *N0*, *controlling_flows_of_traffic*).

(a) Traffic lights example from VQAR C_2



target(*O2*) :- *relation*(*+O2*, *+O1*, *-of*), *name*(*+O1*, *-animal*), *name*(*O1*, *object*), *relation*(*+O0*, *+O1*, *-of*).

(b) Flock of sheep example from VQAR C_5

Figure 5: **VQAR example images and programmatic queries:** Bounding boxes are produced by a YOLO network and answer objects are marked with green. On the right, the *name*(*O1*, *object*) predicate is not annotated with the $+/-$ notation and has to be derived via the knowledge graph.

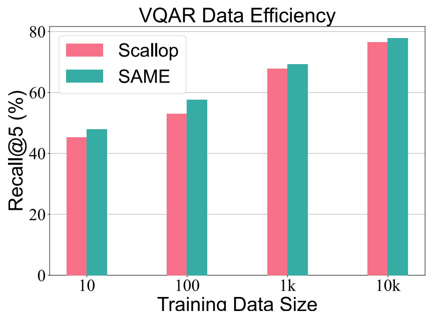
In the ablation study experiments, we present the average over five runs with different random seeds for parameter initialization. For VQA experiments, we used the same single seed to initialize the NPP's parameters following the setting of Huang et al. (2021). We refer to App. A for each experiment's SLASH program, including queries, and App. C for a detailed description of hyperparameters and further experimental details.

4.1 Visual Question Answering

In VQA, a model should produce answers to questions about visual scenes. These questions require a range of capabilities to infer the correct answer. For example, to answer the question "How many red objects are in the scene?" a model has to be able to detect and count red objects. In this experiment, we show how SLASH can be applied to VQA to answer questions that require reasoning.

As of now, few works approach VQA using logic-based DPPLs (Eiter et al., 2022; Huang et al., 2021). Both of these works open up the question of how ASP can be used in an end-to-end trainable setting; for example, questions about scenes from real-world images, such as in the VQAR dataset proposed by Huang et al. (2021). We will now investigate how to apply SLASH to the VQAR dataset.

Task Description – The VQAR dataset consists of 80.178 real-world images. Fig. 1 gives an overview of the task. Each image was fed through a pretrained YOLO Network to obtain bounding boxes and feature maps for recognized objects. Each image has a scene graph (SG), which can have 500 object names, 609 attributes and 229 object relations among the objects. All images share a knowledge graph (KG) encoding 3.387 entries as tuples and triplets, and six rules to traverse. Both graphs are represented in the form of a logic program. There are 4M programmatic queries and answer pairs encoding object



(a) Data efficiency of SLASH with SAME and Scallop on different dataset sizes.

Train →	10k C_2		50k C_{all}	
Test ↓	Scallop	SAME	Scallop	SAME
1k C_2	76.50	77.76	84.34	79.26
1k C_3	76.48	73.75	81.97	72.95
1k C_4	77.00	67.41	80.90	69.50
1k C_5	79.30	68.87	83.76	69.33
1k C_6	76.98	64.86	82.61	63.18
5k C_{all}	77.25	70.52	82.72	70.90

(b) Performance of SLASH with SAME and Scallop on different clauses lengths.

Figure 6: **Performance of SLASH on VQAR:** results on the data efficiency test (a) and generalization test for different clause lengths (b) trained on C_2 (left) and C_{all} (right).

identification questions. The queries' difficulty varies, ranging from two to six occurring clauses (C_2 to C_6), and for each image, ten query answer pairs exist for each clause length. Fig. 5 depicts two examples of VQAR from C_2 and C_5. In Fig. 1, next to the programmatic queries are their corresponding natural language questions to be found. Similarly to Huang et al. (2021), we argue that this work focuses on enabling reasoning for VQA, and as such, we use the programmatic form as input. Some works, such as Yi et al. (2018), translate from natural language to programmatic queries. We leave this for future work.

Approach by SAME – The task is formulated as a multi-label classification task. The feature maps, bounding boxes, the entire knowledge graph and the programmatic query serve as input to predict the objects that answer the programmatic query. Fig. 2 shows the SLASH pipeline for VQA. In our setup, three MLP classifiers are used as NPPs to predict names, attributes, and relations and are trained end-to-end. All three are of the same architecture (cf. App. C.4) as defined by Huang et al. (2021). The NPPs outcomes form the scene graph and build the SLASH program with the KG and the query. The VQA task, in itself, exposes the limits of DPPLs without approximate reasoning. The complexity of the real world is so high that the complete enumeration of all proofs/models is beyond reach. We use a combination of SAME, CLINGO's show statements and iterative solving to deal with the complexity of the task. We refer the interested reader to the App. B, where we look in-depth into our program encoding. In the following, we compare SLASH using SAME with Scallop.

Results – Fig. 6a presents insights on data efficiency: The recall@5 of test queries after training with 10, 100, 1k and 10k training samples on C_2. We see that SAME achieves greater data efficiency than Scallop due to the flexible number of potential solutions.

In Tab. 6b, the recall values are displayed for varying clause lengths to demonstrate our approach's generalizability and overall performance. The left side shows results for training on 10k samples on C_2 and the right side on C_{all}. SAME performs similarly to Scallop (Huang et al., 2021) on C_2 for both settings. As the solution space grows exponentially with the complexity of the questions, we observe that the performance of SAME decreases compared with C_2 on more complex tasks. Comparing Scallop's results with SAME, i.e.,

Neural Model	Task	SLASH	top-k				SAME
			k=1	k=3	k=5	k=10	
DNN	T1	98.80	98.68	98.81	98.60	98.69	98.56
	T2	98.85	98.81	98.76	98.68	98.68	98.82
	T3	98.75	98.77	98.77	98.74	98.77	98.71
PC	T1	95.29	74.89	70.25	79.89	87.59	95.19
	T2	95.26	70.12	64.23	64.42	71.11	94.99
	T3	95.11	30.55	41.03	31.79	32.96	94.94

Table 1: **Regardless of how complex the task is, it is harder to choose the correct k in top-k than for top-k% for PCs:** Accuracy Comparison between top-k and SLASH with and without SAME. We compare the method on three different Tasks, T1-T3: *sum2*(3,7, 10) *sum3*(3,7,5,15) *sum4*(3,7,5,2,17).

top-99%, Huang et al. (2021) use top-10 for each programmatic query; Scallop features directly weighted rules, while SLASH would have to emulate such rules. SLASH uses CLINGO as the underlying solver to produce potential solutions, which are then used to compute a query's success probability. Since we assume the underlying solver to be given, we modify the program to be forwarded to the solver rather than modifying the solver itself. Consequently, we treat the solver as an "off-the-shelf" tool. On the other hand, weighted rules would allow us to navigate the solution spaces more efficiently and solve more complex queries, such as C_6.

In summary, the experimental results show that *SLASH scales with SAME to VQA*. Next, we study the scalability achieved by SAME as an ablation study.

4.2 Scalability of SLASH

Inspired by Huang et al. (2021), we explore how using different subsets of all potential solutions affects the performance and scalability of SLASH on the MNIST addition task.

In the task of MNIST-addition (Manhaeve et al., 2018), the goal is to predict the sum of two images from the MNIST dataset (LeCun et al., 1998b), presented only as raw images. During test time, however, a model should classify the images directly. Thus, the model does not receive explicit information about the depicted digits and must learn to identify digits via indirect feedback on the sum prediction. Using more than two images makes the task significantly harder, as an exponentially growing number of digit combinations has to be considered. Similar to the setup of Scallop (Huang et al., 2021), we test on three different difficulty levels to evaluate the model's scaling capabilities. The difficulty ranges from task T1 with two images *sum2*(3,7,10), to task T3 with four images *sum4*(3,7,5,2,17). We use a PC and a DNN as NPP for the same settings.

The DNN used is the LeNet5 model (LeCun et al., 1998a). When using the PC as NPP, we have extracted conditional class probabilities $P(C|X)$ by marginalizing the class variables C to acquire the normalization constant $P(X)$ from the joint $P(X, C)$, and calculating $P(X|C)$. The models using the NN architecture converge after one or two epochs and only get minor improvements in accuracy thereafter. For the PC architecture, the convergence

	Accuracy after last Epoch			Average Time per Epoch		
Method	T1	T2	T3*	T1	T2	T3*
Scallop top-10	98.95	99.12	97.47	54m:10s	5h:39m:53s	21h:10m:0s
DeepProbLog	98.50	98.75	98.23	8m:3s	15m:36s	34m:54s
DeepStochLog	96.96	97.49	97.54	1m:23s	5m:49s	44m:27s
NeurASP	98.05	98.42	98.03	3m:13s	32m:26s	15h:28m:51s
SLASH-DNN	98.80	98.85	98.75	24s	1m:42s	51m:49s
SLASH-PC	95.29	95.26	95.11	1m:9s	2m:27s	52m:22s
SLASH-DNN top-10	98.69	98.68	98.77	25s	1m:3s	25m:2s
SLASH-PC top-10	87.59	71.11	32.96	1m:8s	1m:47s	26m:52s
SAME-DNN	98.56	98.82	98.71	17s	17s	1m:35s
SAME-PC	95.19	94.99	94.94	1m:3s	1m:23s	16m:40s

Table 2: **SAME scales well with growing task complexity:** Test accuracy in % and runtime comparison. The runtime is averaged over ten epochs for all methods. Light green indicates high accuracy or low time, while blue stands for the opposite. (*) Please note, that since training Scallop and NeurASP would have taken too long, they were stopped after one epoch and therefore did not converge as the other DPPLs.

takes more epochs and increases with the task difficulty. We report test accuracies after 10 and 20 epochs for the DNN and PC architecture, respectively. Tab. 3 in App. C.1 shows the convergence of SLASH with PC as NPP on different tasks.

Performance using subsets of all potential solutions – First, let us look at what happens if we prune away some potential solutions given our NPP probabilities. We compute the potential solutions in three ways: SLASH with all potential solutions, SLASH with a top-k variant (SLASH-top-k) and SLASH with SAME. For top-k, we use CLINGO's minimization constraints to put the NPP output probabilities in the logic program, cf. App. A. The solver then gives us the potential solutions sorted by their probability $P_{\Pi(\boldsymbol{\theta})}(I)$, from which we keep the k most probable solutions. For an example program for SLASH top-k, see App. A.

Tab. 1 lists the results for the test on partial solutions. SLASH and SAME achieve almost identical or slightly worse performance on all tasks and different NPPs. With neural networks as our NPP, SLASH-top-k achieves similar performance for all k's compared to SLASH. Using PCs as NPP, we get a worse performance. With increasing task difficulty, we lose most of the predictive performance of our model. With a high k on T1, most potential solutions are still covered, resulting in only a small drop in accuracy. For example, on T1 there are nine ways to add two digits to ten, which is the query with the most potential solutions. With increasing task difficulty, though, many more potential solutions are not covered when selecting k=10 as in Scallop Huang et al. (2021), since there are 73 for T2 and 633 for T3. At the beginning of training, our model gives us uniform predictions over all digits, as it has not learned anything yet. Therefore, the randomness of model initialization influences which solution falls into the top-k range. If we prune the true solution, our model cannot learn to detect the correct class with that query, and it has to rely on other queries that might have the true solution in the top-k range. Empirically, we see that with

Model	Epochs	T1	T2	T3
SAME-DNN	1	21s	39s	13m:37s
	1-10	17s	17s	1m:35s
	2-10	17s	15s	14s
SAME-PC	1	1m:14s	2m:32s	53m:26s
	1-20	1m:3s	1m:23s	16m:40s
	2-20	1m:2s	1m:15s	12m:35s

Table 3: **Due to pruning, SAME gets faster in later iterations:** The average time per epoch is shown for different epochs.

DNNs, we can still learn to detect digits, while with PCs, we cannot. We argue that the DNN architecture is more robust to these incorrect inputs and, over time, accumulates an increasing proportion of the correct digits in the top-k selection because it is better suited for object detection equipped with the visual inductive biases of convolutional layers. PCs, on the other hand, learn false classes at the beginning and reinforce the false prediction by repeatedly predicting them as most likely.

In contrast, SAME works on both PCs and DNNs as it only prunes certainly unlikely options. At first, we do not prune anything, and over time, after learning, we can safely regard the unlikely solutions, which explains why SAME is the better choice for both NNs and PCs.

SAME reduces training time by pruning unlikely outcomes – After seeing that SLASH with SAME achieves on-par performance, we now want to look at the time savings we get by using it. Tab. 2 shows the average training time per epoch and the test accuracy. We provide results for other state-of-the-art DPPLs: Scallop (Huang et al., 2021), DeepProbLog (Manhaeve et al., 2018), its cousin DeepStochLog (Winters et al., 2022), and NeurASP (Yang et al., 2020). These DPPLs again use the LeNet5 architecture (LeCun et al., 1998a). For Scallop and NeurASP, we report the accuracy after one epoch on T3, as the training time for ten epochs would take almost a week. NeurASP and SLASH both use CLINGO as their ASP solver. On T1, the difference in speed can mainly be explained by batch-wise computations employed in SLASH, while NeurASP processes one query at a time. On the harder tasks where the solution space grows exponentially, we see that SAME helps to accelerate the solving process, while NeurASP still has to evaluate the whole solution space by enumerating all stable models.

SLASH with and without SAME achieves state-of-the-art accuracy similar to the other models on all task difficulties using the same DNN architecture. We further observe that the test accuracy of SLASH with a PC NPP is slightly below the other DPPLs. However, this may be because a PC, compared to a DNN, is learning a true mixture density rather than just conditional probabilities. Moreover, it is a question of engineering, and optimal architecture search for PCs, e.g., for computer vision, is an open research question.

Regarding training time, we see that top-k yields small improvements. With SAME, we improve the training time by a huge fraction when considering numerous potential solutions. For example, on T3 with NNs, we only need 3% of SLASH's original training time over 10

	(i) $Acc_{identify}$ of $M_{identify}$		(ii) $Acc_{identify}$ of $M_{identify} + \Pi_{Sudoku} \backslash r$		(iii) $Acc_{identify}$ of $M_{identify} + \Pi_{Sudoku}$		
train samples	NeurASP	SAME	NeurASP	SAME	NeurASP	SAME	avg. outcomes
15	15	38	49	59	71	68	99.43
17	31	84	62	93	80	96	87.05
19	72	96	90	99	95	99	85.36
21	85	97	95	99	98	100	84.57
23	93	100	99	100	100	100	83.05

Table 4: **End-to-end training with SLASH improves data efficiency on Sudoku**: Shown are accuracies in % of (i) the perception module without any corrections, (ii) corrected by the three Sudoku constraints (unique values per column, row, and block) and (iii) fulfilled Sudoku constraints plus the corrected grid has a solution. Yang et al. (2020) showed that ASP improves the pretrained perception module with constraints. Additionally, end-to-end training of the module with SLASH yields bigger improvements in data efficiency. Furthermore, with SAME, we consider exclusively the solutions aligning with perception and the average number of outcomes grows even smaller the more data is accessible for training.

epochs (see Fig. 3). Tab. 3 gives a more detailed overview of SLASH training times with SAME. Interestingly, after one epoch of training, the average runtime per epoch for epochs 2-10 is the same for all three difficulties for the DNN, as the model converges for the most part after the first epoch. It is even a bit faster on T3 because the number of queries is less on the T3 dataset (60k samples/number of images per query).

These evaluations, in summary, show that SAME is an efficient extension of SLASH which saves a lot of computing resources at the cost of tiny to no differences in performance.

4.3 Correcting Sudoku boards with SAME

In this section, we consider solving a Sudoku puzzle, where the board configuration must first be extracted from an input image as proposed by Yang et al. (2020). Within the pipeline, a neural network first predicts the initial configuration of a 9×9 Sudoku grid, which is then corrected and solved by ASP.

Using SAME, we use the program for Sudoku (Π_{Sudoku}) originally proposed by Yang et al. (2020) (see Fig. 11). First, we train the perception module, which we call $M_{identify}$, end-to-end within SLASH using SAME. During test time, we use SAME as well to reduce the considered outcomes. Please note that using SAME during training is not necessary, since we supervise each cell's outcome. To this end, we employ constraints encoding the expected value for each grid cell and set the training time window for 3k epochs. Second, the proposed three constraints (unique numbers per row, column, and block) are used to correct the outputs of the perception module for testing upon training completion. We measure the accuracy of the perception module (denoted with $Acc_{identify}$ of $M_{identify}$) for the whole Sudoku board represented within the image at once, i.e., a prediction is counted as one if and only if every cell's prediction is correct. Next, we are interested in the accuracy derived by correcting the perception with ASP through the three constraints (unique numbers per row, column, and block), but not checking if the predicted board offers a correct solution ($Acc_{identify}$ of $M_{identify} + \Pi_{Sudoku} \backslash r$). If, for example, in the first row, digit 2 appears twice, the solver will check which is the second most likely solution, and if it fulfills all constraints, choose this as the correct number. The rule r corresponds to line 6 of the program (Π_{Sudoku}) enlisted in Fig. 11. The line ensures a number of fills to each empty cell. Lastly, we include r to test additionally if the predicted board fulfills all three constraints and offers a unique solution ($Acc_{identify}$ of $M_{identify} + \Pi_{Sudoku}$).

Tab. 4 shows the results of our approach compared with NeurASP. In the experimental setup, Yang et al. (2020) pretrain the perception module using different amounts of images representing the initial Sudoku configuration. They employ minimization constraints to correct the perception module to find the most probable stable model that aligns with the Sudoku constraints.

The results indicate that training end-to-end in SLASH with SAME greatly improves data efficiency. Using 17 images for training allows us to achieve 53% improvement in the predictive performance of the perception module. Further, correcting the perception model improves the predictive performance, as shown by Yang et al. (2020). The corrections made by NeurASP and SAME are similar. Nonetheless, SAME needs to consider only a fraction of the possible outcomes. In NeurASP, every possible outcome (810 in total) is annotated with a minimization weight from the probability provided by the perception module. Given all outcomes and their weights, the ASP solver must decide which is the likeliest model. In contrast, SAME simplifies this process by considering only the most likely outcomes given the perception module. For example, when using the model training with 17 samples, the ASP solver predicts a high probability for most of the 81 cells. In the last column, the value of 87.05 indicates SAME considers on average around ($\frac{87.05}{81} \approx$)1.07 outcomes instead of 10 per cell as possible corrections by ASP. Furthermore, having more training samples, SAME improves the predictive performance while reducing the average number of outcomes for consideration. Thus, SAME works as intended, as it only considers outcomes that align with the perception from DNN.

4.4 Object-centric learning

Now, we turn to a very different task of object-centric set prediction. We presume that recent advancements in object-centric learning (Greff et al., 2019; Lin et al., 2020; Locatello et al., 2020) can be further improved by integrating such neural components into DPPLs and adding logical constraints about objects and their properties. Similarly, we want to find out how much SAME speeds up SLASH possibly without loss of performance.

For set prediction, a model is trained to predict the discrete attributes of a set of objects in an image (cf. Fig. 2 in the top-left corner for an example CLEVR image). The difficulty therein is that the model must match an unordered set of corresponding attributes of various objects with its internal representations of the image.

The slot attention module introduced by Locatello et al. (2020) allows for an attractive object-centric approach to this task. Specifically, this module represents a pluggable, differentiable module that can be easily added to any architecture. Through a competitive softmax-based attention mechanism, the model can enforce the binding of specific parts of a latent representation into permutation-invariant, task-specific vectors called slots.

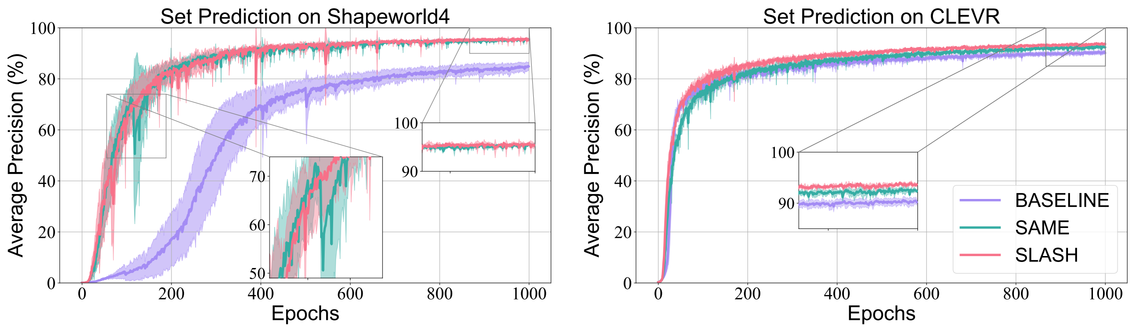


Figure 7: **SLASH can converge faster:** Average Precision on ShapeWorld4 (left) and CLEVR (right). SLASH converges faster on ShapeWorld4 compared to the Baseline. During training, we observe temporary crashes in AP, which get smaller over time (see zoomed windows). Furthermore, the standard deviation is much smaller for SLASH than for the baseline. On CLEVR, all three models converge similarly after roughly 200 epochs. SLASH performs slightly better than SAME, which, in turn, performs a little better than the Baseline.

We train SLASH with and without SAME based on NPPs consisting of a shared slot encoder and separate PCs, each modelling the mixture of latent slot variables and the attributes of one category, e.g., color. For each dataset, ShapeWorld4 and CLEVR, we have four NPPs in total. Finally, the model is trained via queries exemplified in Fig. 14 in App A. We refer to this configuration as *SLASH Attention*.

We compare SLASH Attention to a baseline of slot attention encoder using single multicategorical MLP and Hungarian loss to predict object properties from the slot encodings as in (Locatello et al., 2020). The key difference between these two models lies in the employed logical constraints in SLASH Attention. In their work, Locatello et al. (2020) utilize a single MLP trained via Hungarian loss, i.e., they assume shared parameters for all attributes. In comparison, in SLASH attention, we make an independence assumption about the parameters for the object attributes and encode this via logical constraints. We refer to App. A for the program.

One limitation is that matching objects to slots has $n!$ possible assignments. To overcome this, we adopt a similar strategy to external functions in CLINGO. We use Hungarian matching (Kuhn, 1955) and make the resulting assignments a part of the logic program. The Hungarian matching algorithm scales polynomial with time complexity of $\mathcal{O}(n^3)$. This enables SLASH for training on CLEVR, which can contain up to ten objects per image. For images containing an order of magnitude more objects, the matching might become a bottleneck again, and we leave it for future work. The results of these experiments can be found in Tab. 5.

On ShapeWorld4, we observe that the average precision after convergence on the held-out test set with SLASH Attention is greatly improved to that of the baseline model. More interesting, SAME provides the best results in this setting while having the smallest deviation, cf. Fig. 7 (left). ShapeWorld4 has significantly fewer data entries than CLEVR,

	Accuracy		Time	
Method	ShapeWorld4	CLEVR	ShapeWorld4	CLEVR
Slot Att.	85.25	90.21	2h:36m	1d:2h:26m
SLASH	94.98	93.47	1d:3h:53m	6d:16h:49m
SAME	95.21	92.50	15h:29m	5d:6h:24m

Table 5: **SLASH improves upon Slot Attention:** Test average precision and training times for the Slot Attention baseline and SLASH with and without SAME.

which may account for the huge improvement in performance with SLASH Attention. This is evidence that we are moving closer to knowledge-rich AI. Additionally, we observe that SLASH Attention reaches the average precision value of the baseline model in much fewer epochs. On CLEVR, this tendency also holds, but the difference in performance is smaller, but we still get around 2-3% more average precision with SLASH and SAME.

Regarding the training times, we observed that in the case of ShapeWorld4, using SAME allows truncating the training window by **44.47%**, compared to the results of SLASH without SAME. For CLEVR, we obtained a solution and are getting it **21.4%** faster thanks to SAME.

These observations suggest that SAME applies to any form of NPPs and is a good step towards unraveling the solving bottleneck to lift the symbolic overhead. Nonetheless, there is a difference in the number of learnable parameters between the neural baseline and SLASH attention. Namely, SLASH attention consists of four PCs, for which the time spent on forward- and backward-pass is higher compared to the single multicategorical DNN used in the slot attention module. We refer to App. C.5 for in-depth discussion. Finally, we draw attention to the fact that the symbolic overhead is a direct result of all DPPLs under consideration using the CPU for high-level reasoning, while low-level perception is based on the GPU. To further reduce the symbolic overhead, tight integration of solving with neural processing may be a promising research direction.

Summary of Empirical Results. All empirical results together demonstrate that the expressiveness and flexibility of SLASH are highly beneficial and improve upon the state-of-the-art: One can freely combine what is required to solve the underlying task — (deep) neural networks, PCs, and logic. The experiments demonstrate SAME to be the natural extension of SLASH. Further, the results indicate that utilization of SAME comes with a tiny, if any, performance loss in comparison to the analytical weighted model counting.

5. Related Work

Neuro-Symbolic AI can be divided into two lines of research, depending on the starting point, though both have the same final goal: To combine low-level perception with logical constraints and reasoning. A key motivation of Neuro-Symbolic AI (d'Avila Garcez et al., 2009; Mao et al., 2019; Hudson & Manning, 2019; d'Avila Garcez et al., 2019; Jiang & Ahn, 2020; d'Avila Garcez & Lamb, 2023) is to combine the advantages of symbolic and neural representations into a joint system. This is often done in a hybrid approach where a neural network acts as a perception module that interfaces with a symbolic reasoning system, e.g., Mao et al. (2019), Yi et al. (2018). The goal of such an approach is to mitigate the issues of one by the other, e.g., using the power of symbolic reasoning systems to handle the generalizability issues of neural networks and handle the difficulty of noisy data for symbolic systems via neural networks. Recent work has also shown the advantage of approaches for explaining and revising incorrect decisions (Ciravegna et al., 2020; Stammer et al., 2021). However, many of these previous works train the sub-symbolic and symbolic modules separately.

Deep Probabilistic Programming Languages (DPPLs) are programming languages that combine deep neural networks with probabilistic models and allow a user to express a probabilistic model via a logic program. Similar to neuro-symbolic architectures, DPPLs thereby unite the advantages of different paradigms. DPPLs are related to earlier works such as Markov Logic Networks (MLNs) (Richardson & Domingos, 2006). Thereby, the binding link is the Weighted Model Counting (WMC) introduced in LP$^{\mathrm{MLN}}$ (Lee & Wang, 2016). Several DPPLs have been proposed by now, among which are Pyro (Bingham et al., 2019), Edward (Tran et al., 2017), DeepProbLog (Manhaeve et al., 2018), DeepStochLog (Winters et al., 2022), NeurASP (Yang et al., 2020), and Scallop (Huang et al., 2021).

To resolve the scalability issues of DeepProbLog, which uses Sentential Decision Diagrams (SDDs) (Darwiche, 2011) as the underlying data structure to evaluate queries, NeurASP (Yang et al., 2020), offers a solution by utilizing ASP (Dimopoulos et al., 1997; Soininen & Niemelä, 1999; Marek & Truszczynski, 1999; Calimeri et al., 2020). In contrast to query evaluation in Prolog (Colmerauer & Roussel, 1993; Clocksin & Mellish, 1981), which may lead to an infinite loop, many modern answer set solvers use Conflict-Driven-Clause-Learning (CDPL), which, in principle, always terminates. In this way, NeurASP changes the paradigm from query evaluation to model generation, i.e., instead of constructing an SDD or a similar knowledge representation system, NeurASP generates a set of all potential solutions (one model per solution) and estimates the probability for the truth value of each of these solutions. Of those DPPLs that handle learning in a relational, probabilistic setting and end-to-end fashion are limited to estimating only conditional class probabilities. Particularly, the inference is limited to $P(C|X)$ obtained from a neural network using Softmax.

Another research branch focuses on approximate inference for DPPLs to allow scaling to harder problems. The goal is to incorporate probabilities into the solving process to obtain only a subset of all proofs. Manhaeve et al. (2021) propose an A*-like search for proofs, and Huang et al. (2021) introduce a top-k mechanism based on Datalog to only keep likely proofs. In ASP, a program is first grounded and then solved, sometimes making the grounding itself a bottleneck. Existing work, therefore, aims at grounding on demand. The two main candidates are Lazy Grounding (Palù et al., 2009) and Magic Sets for ASP (Alviano & Faber, 2011). To the best of our knowledge, both techniques have not been applied in a probabilistic setting with ASP yet.

Visual Question Answering has seen a lot of attention from the computer vision and natural language processing community. We refer to Manmadhan and Kovoor (2020) and Kodali and Berleant (2022) for a detailed review. Recently, more neuro-symbolic approaches to VQA have been proposed. Yi et al. (2018) proposed a model which creates a structural scene representation of the image, parses a natural language question into a program, and then executes the program to obtain an answer. A few works utilize logic programming: Scallop's (Huang et al., 2021) top-k approach allows for answering complex reasoning questions on real-world images. Eiter et al. (2022) showed how ASP could be used on top of the outputs of a pretrained YOLO network to answer CLEVR questions (Johnson et al., 2017).

6. Conclusions

We introduce SLASH, a novel DPPL that integrates neural computations with tractable probability estimates and logical statements. The key ingredient of SLASH to achieve this is Neural-Probabilistic Predicates (NPPs) that can be flexibly constructed out of neural and/or probabilistic circuit modules based on the data and underlying task. With these NPPs, one can produce task-specific probability estimates. The details and additional prior knowledge of a task are neatly encompassed within a SLASH program with only a few lines of code. Finally, via ASP and Weighted Model Counting, the logic program and probability estimates from the NPPs are combined within SLASH to estimate the truth value of a task-specific query. Additionally, the SAME technique addresses the question of scalability. Proven to converge to only one solution, SAME is the natural extension of SLASH and generally applicable to any problem.

Our experiments on the VQAR dataset show the power, efficiency, and scalability of SLASH, paving the way to handle extremely difficult real-world applications. As one of many consequences, we found the following shortcomings, which we leave to be resolved in future work. First, VQAR shows bigger parts of a program are optional to answer the programmatic query and, thus, should be ignored during grounding. SAME can be seen as a form of stochastic lazy grounding, and thus helps to reduce the computation costs for NPPs. It remains to be seen if and how similar technique(s) can be used for grounding the rest of the program after applying SAME.

Second, should there be an exponential number of potential solutions, as in some VQAR queries, we would no longer able to answer the query. Weighted rules and facts might be insightful in finding ways to navigate solution spaces more efficiently. Finally, for WMC to be computed most efficiently regardless of the number of potential solutions and the queries, it must take place simultaneously with solving, i.e., becoming an inseparable part of it.

Apart from that, our ablation study provided a detailed evaluation of the computation speed of SAME, improving upon previous DPPLs in the benchmark MNIST-Addition tasks yet retaining the performance. On Sudoku, we showed that SAME works as designed, reducing the number of outcomes per grid's cell to the smallest possible. Additionally, invoking Python routines allowed for the seamless invocation of the Hungarian matching algorithm into SLASH Attention. Together with SAME, we solved the task of object-centric set prediction for the CLEVR dataset, which none of the previous DPPLs has tried to solve yet, and reduced the training time of SLASH.

With SLASH on the set prediction task, we effectively use elements of functional programming within SLASH. Similarly, the used ASP-solver CLINGO can invoke Python routines at the grounding time via external functions. These pave the way for merging functional programming with SLASH. Neural Logic Machines (Dong et al., 2019) serve as an example of a similar combination. Going in this direction will allow us to treat logically constrained regression problems, which would benefit fundamental sciences such as particle physics. Yu et al. (2021) show how PCs can be used for multi-output regression tasks, and it appears to be the natural next step to integrate them in SLASH.

Acknowledgments

This work was partly supported by the Federal Minister of Education and Research (BMBF) and the Hessian Ministry of Science and the Arts (HMWK) within the National Research Center for Applied Cybersecurity ATHENE, as well as via the DEPTH group CAUSE of the Hessian Center for AI (hessian.ai), the ICT-48 Network of AI Research Excellence Center "TAILOR" (EU Horizon 2020, GA No 952215, and the Collaboration Lab with Nexplore "AI in Construction" (AICO). It also benefited from the BMBF AI lighthouse project, the Hessian research priority programme LOEWE within the project WhiteBox, the HMWK cluster projects "The Third Wave of AI" and "The Adaptive Mind", the German Center for Artificial Intelligence (DFKI) project "SAINT".

Appendix A. SLASH Programs

Here, the interested reader will find the SLASH programs which we compiled for our ablation studies on MNIST addition and Object-centric learning. Where, Fig. 13 and Fig. 14 are for the set prediction task with slot attention encoder.

MNIST addition Fig. 10 shows the weak constraints for SLASH top-k. In the brackets,

```
# Define images
img(i1). img(i2).
# Define Neural-Probabilistic Predicate
npp(digit(X), [0,1,2,3,4,5,6,7,8,9]) :- img(X).
# Define the addition of digits given two images and the resulting sum
addition(A, B, N) :- digit(+A, -N1), digit(+B, -N2), N = N1 + N2.
```

Figure 8: SLASH Program for MNIST addition with two images.

```
# Is 7 the sum of the digits in img1 and img2?
:- addition(i1, i2, 7)
```

Figure 9: Example SLASH Query for MNIST addition.

the first value is the probability of the corresponding ground atom in log space. The second and third values together make up a unique identifier for the belonging atom, which is used by CLINGO.

```
# weak constraints for image 1
:~ digit(1,i1,0). [1866, 0, 0]
:~ digit(1,i1,1). [2901, 0, 1]
...
:~ digit(1,i1,9). [2468, 0, 9]
# weak constraints for image 2
:~ digit(1,i2,0). [1761, 1, 0]
:~ digit(1,i2,1). [2922, 1, 1]
...
:~ digit(1,i2,9). [2517, 1, 9]
```

Figure 10: Weak constraints for SLASH top-k

Sudoku

```
# NPP for identifying empty cells and filled in cells (1-9)
npp(identify(81, img), [empty,1,2,3,4,5,6,7,8,9]).
# we assign one number at each position (R,C)
a(R,C,N) :- identify(Pos, +img, -N), R=Pos/9, C=Pos\9, N!=empty.
{a(R,C,N): N=1..9}=1 :- identify(Pos, +img, -empty), R=Pos/9, C=Pos\9.
# it's a mistake if the same number shows 2 times in a row
:- a(R,C1,N), a(R,C2,N), C1!=C2.
# it's a mistake if the same number shows 2 times in a column
:- a(R1,C,N), a(R2,C,N), R1!=R2.
# it's a mistake if the same number shows 2 times in a 3*3 grid
:- a(R,C,N), a(R1,C1,N), R!=R1, C!=C1, ((R/3)*3 + C/3) = ((R1/3)*3 + C1/3).
```

Figure 11: SLASH Program for Sudoku

```
# Assign cell 42 the number 7
:- identify(41, +img, -7).
...
# Assign cell 81 the value empty
:- identify(80, +img, empty).
```

Figure 12: Example SLASH Query for Sudoku experiments.

Object-centric learning

```
# Define slots
slot(s1). slot(s2). slot(s3). slot(s4).
# Define identifiers for the objects in the image
# (there are up to four objects in one image).
obj(o1). obj(o2). obj(o3). obj(o4).
# Assign each slot to an object identifier
{assign_one_slot_to_one_object(X, O): slot(X)}=1 :- obj(O).
# Make sure the matching is one-to-one between slots
# and objects identifiers.
:- assign_one_slot_to_one_object(X1, O1),
   assign_one_slot_to_one_object(X2, O2),
   X1==X2, O1!=O2.
# Define all Neural-Probabilistic Predicates
npp(color_attr(X), [red, blue, green, grey, brown,
                    magenta, cyan, yellow, bg]) :- slot(X).
npp(shape_attr(X), [circle, triangle, square, bg]) :- slot(X).
npp(shade_attr(X), [bright, dark, bg]) :- slot(X).
npp(size_attr(X), [big,small,bg]) :- slot(X).
# Object O has the attributes C and S and H and Z if ...
has_attributes(O, C, S, H, Z) :- slot(X), obj(O),
                    assign_one_slot_to_one_object(X, O), color(+X, -C),
                    shape(+X, -S), shade(+X, -H), size(+X, -Z).
```

Figure 13: SLASH Program for ShapeWorld4.

```
# Does object o1 have the attributes red, circle, bright, small?
:- has_attributes(o1, red, circle, bright, small)
```

Figure 14: Example SLASH Query for ShapeWorld4 experiments. In other words, this query corresponds to asking SLASH: “Is object 1 a small, bright red circle?”.

Appendix B. VQA Program Encoding and dealing with complexity

In this section, we will explain how the SLASH program for the VQA task is constructed and how we deal with the complexity of the task and thus avoid producing an infeasible number of potential solutions for difficult questions. As depicted in Fig. 1, the VQA task comprises multiple parts in the SLASH program. One thing to highlight here is the length of the program, which usually has more than 3k lines.

The KG makes 1424 "is-a" tuples and 1963 "object-attribute-relation" triplets, as well as six rules for the fixed part of every program. For n objects, the SG includes n attributes, n names, and $n * (n - 1)$ obj-to-obj relations, excluding relations of objects to themselves. Each object can have multiple attributes at once, so each attribute is modelled as a NPP with two outcomes: Having or not having the attribute.

Fig. 5 shows two images, object bounding boxes, and a target rule specifying what targets should entail. E.g., the provided target query from C_2 in Fig. 5a is depicted.

```
:- not target(0); not target(1).
:- target(2).:- target(3).:- target(4).:- target(5).:- target(6).
```

It restricts objects 0 and 1 to be targets, while others are not. Combined with the stated target rule, the name NPP outcomes of objects 0 and 1 are restricted to inferring a name that can be substituted for variable N0 in the oa_rel(is_used_for, N0, controlling_flows_of_traffic) predicate. From the knowledge graph, we can infer for N0 to be replaced by "traffic lights". In this case, all other names for the non-target objects are restricted to not being traffic lights. They can take on all other 499 outcomes of the name NPP. Attributes and relations are not restricted as well by the query. CLINGO's show statements are used to show exclusively the predicates of the programmatic query in any potential solution. For the example under consideration, the show rule is depicted below.

```
#show.
#show name(O0, X) : target(O0), name(O0,X), name(O0,N0), oa_rel(is_used_for,
    N0,controlling_flows_of_traffic).
```

These lines tell CLINGO to itemize name predicates of objects which satisfy all target predicates. Upon solving, we obtain the following potential solutions.

$$\{\text{target(0), name(0,traffic\_lights)}\}$$
$$\{\text{target(1), name(1,traffic\_lights)}\}$$

Moving on to the more complex example of C_5 in Fig. 5b. Here, our target rule consists of five predicates. Two name predicates restrict the target to be animals and objects. Additionally, two relation predicates specify the relation of the target object O2 to other objects. For such a query, computing all potential solutions can be infeasible. Particularly, we have 16 objects which form 16*(16-1)=240 relation NPPs. Substituting them into the target relations at once can quickly lead to millions of potential solutions, should a programmatic query contains multiple relation predicates. Instead, we employ iterative solving. At each iteration, the next five relations with the highest probability are added until we have 100 potential solutions or a specified timeout of 30 seconds is reached.

In the last step, SAME helps out by pruning unlikely NPP outcomes. The target rule of this example stipulates only one name predicate should be an object. From the KG, we see the ontological concept telling us about what falls under the category of objects, such as furniture, vehicles, or animals. Asking for these broader categories restricts the NPP outcomes only partially. In the case of objects, most Name NPP's outcomes are part of this category. Here again, for some queries, this can make computing all potential solutions infeasible. Our solution – combining top-k pruning with SAME: To keep the k most probable outcomes for each Name NPP and to prune more with SAME. Precisely, SAME will point to the Name outcome, that is, the actual one belonging to the object category and will prune any remaining ones.

Appendix C. Experimental Details

C.1 PC Convergence

Fig. 15 shows the convergence of SAME with PCs on the T1, T2 and T3. The accuracy converges to the almost same value, and we see that the harder the task is, the more epochs it takes to converge.

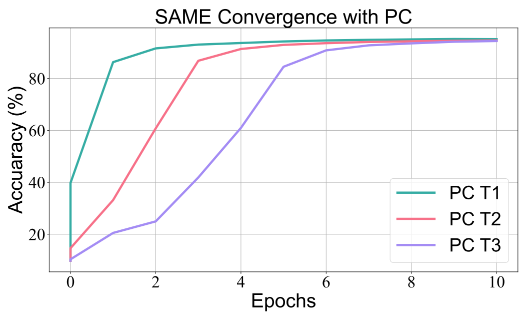


Figure 15: Convergence with SAME-PC on MNIST Addition.

C.2 ShapeWorld4 Generation

The ShapeWorld4 dataset was generated using the original scripts of Kuhnle and Copestake (2017)[2]. The exact scripts will be added together with the SLASH source code.

C.3 Average Precision computation

For the baseline slot encoder experiments on the Object-centric learning tasks, we measured the average precision score as by Locatello et al. (2020). In comparison to the baseline slot encoder, when applying SLASH Attention, however, we handled the case of a slot not containing an object, e.g., only background variables, differently. Whereas Locatello et al. (2020) add another binary identifier to the multi-label ground truth vectors, we have added a background (bg) attribute to each category (cf. Fig. 13). A slot is thus considered empty (i.e., not containing an object) if each NPP returns a high conditional probability for the bg attribute.

C.4 Model Details

For those experiments using NPPs with PC, we have used Einsum Networks (EiNets) for implementing the PC. EiNets are a novel implementation design for SPNs introduced by Peharz et al. (2020) that minimize the issue of computational costs that initial SPNs had suffered. This is accomplished by combining several arithmetic operations via a single monolithic einsum-operation.

For all experiments, the ADAM optimizer (Kingma & Ba, 2015) with $\beta_1 = 0.9$ and $\beta_2 = 0.999$, $\epsilon = 1e - 8$ and no weight decay was used.

2. https://github.com/AlexKuhnle/ShapeWorld

VQA Experiments The architecture for the VQA experiments is the same as in Huang et al. (2021) and is shown in Tab. 6. The name, relation, and attribute classifier share the same architecture. A YOLO network produces object features of size 2048 which are fed into the classifiers. The relation classifier takes as input the features and bounding boxes of two objects, resulting in an input dimension of $4104 = (2048 + 4) * 2$. For the name and relation classifier, a Softmax is used. The attribute classifier has a Sigmoid activation, encoding multiple attributes over each output neuron.

Type	Size/Channels	Activation	Comment
MLP	input_dim, 1024	ReLU	-
BatchNorm + Dropout	1024	-	dropout-rate 0.3/0.5
MLP*	1024, 1024	ReLU	
BatchNorm + Dropout*	1024	-	dropout-rate 0.3
MLP	1014, num_classes	Softmax/Sigmoid	-

Table 6: VQA Neural Model. Layers marked with * are only used in the attribute and name classifier.

MNIST-Addition Experiments For the MNIST-Addition experiments, we ran all baseline programs with their original configurations, as stated in Huang et al. (2021), Manhaeve et al. (2018), Winters et al. (2022), Yang et al. (2020), respectively. For the MNIST Addition experiments, we have used the same neural module as in the baselines when training SLASH and SAME with the neural NPP represented in Tab. 8. When using a PC NPP, we have used an EiNet with the Poon-Domingos (PD) structure (Poon & Domingos, 2011) and normal distribution for the leaves. The formal hyperparameters for the EiNet are depicted in Tab. 9. The learning rate and batch size for SLASH and the baselines are shown in Tab. 7.

Model	learning rate	batch size
Scallop	0.001	64
DeepProbLog	0.0001	2
DeepStochLog	0.001	100
NeurASP	0.001	-
SLASH-DNN	0.005	100
SLASH-PC	0.01	100

Table 7: Learning rate and batch size for the baselines and SLASH.

ShapeWorld4 Experiments For the baseline slot attention experiments with the ShapeWorld4 data set, we have used the architecture presented in Tab. 10. For further details on this, we refer to the original work of Locatello et al. (2020). The slot encoder had a number of 4 slots and 3 attention iterations over all experiments.

For the SLASH Attention experiments with ShapeWorld4, we have used the same slot encoder as in Tab. 10, however, we replaced the final MLPs with 4 individual EiNets with

Type	Size/Channels	Activation	Comment
Encoder	-	-	-
Conv 5 x 5	1x28x28	-	stride 1
MaxPool2d	6x24x24	ReLU	kernel size 2, stride 2
Conv 5 x 5	6x12x12	-	stride 1
MaxPool2d	16x8x8	ReLU	kernel size 2, stride 2
Classifier	-	-	-
MLP	16x4x4,120	ReLU	-
MLP	120,84	ReLU	-
MLP	84,10	-	Softmax

Table 8: Neural module – LeNet5 for MNIST-Addition experiments.

Variables	Width	Height	Number of Pieces	Class count
784	28	28	[4,7,28]	10

Table 9: Probabilistic Circuit module – EiNet for MNIST-Addition experiments.

Type	Size/Channels	Activation	Comment
Conv 5 x 5	32	ReLU	stride 1
Conv 5 x 5	32	ReLU	stride 1
Conv 5 x 5	32	ReLU	stride 1
Conv 5 x 5	32	ReLU	stride 1
Position Embedding	-	-	-
Flatten	axis: [0, 1, 2 x 3]	-	flatten x, y pos.
Layer Norm	-	-	-
MLP (per location)	32	ReLU	-
MLP (per location)	32	-	-
Slot Attention Module	32	ReLU	-
MLP	32	ReLU	-
MLP	16	Sigmoid	-

Table 10: Baseline slot encoder for ShapeWorld4 experiments.

Poon-Domingos structure (Poon & Domingos, 2011). Their hyperparameters are represented in Tab. 11.

On CLEVR, we also used the “bigger” slot encoder architecture for the CLEVR images as in Locatello et al. (2020) which have higher resolution than the Shapeworld4 images. The PC architecture used is the same for CLEVR, but the number of slots is increased to 10.

The learning rate for the baseline slot encoder was 0.0004 and 512. The learning rate and batch size for SLASH Attention were 0.01 and 512 for ShapeWorld4 and CLEVR for the PCs, and 0.0004 for the slot encoder.

EiNet	Variables	Width	Height	Number of Pieces	Class count
Color	32	8	4	[4]	9
Shape	32	8	4	[4]	4
Shade	32	8	4	[4]	3
Size	32	8	4	[4]	3

Table 11: Probabilistic Circuit module – EiNet for ShapeWorld4 experiments.

C.5 Training times for SLASH Attention

	ShapeWorld4			CLEVR		
	Baseline	SLASH	SAME	Baseline	SLASH	SAME
Forward pass	-	1.7	1.7	-	85.7	78.1
Potential Solutions	-	50.1	8.8	-	140.5	61.7
Gradients	-	12.2	11.9	-	71.3	64.2
Backward pass	-	32.9	31.4	-	273.1	243.2
Σ for training	9.4	96.9	53.8	95.2	570.6	447.3

Table 12: Average training times per epoch in seconds. The four training stages as well as the total training time per epoch are listed.

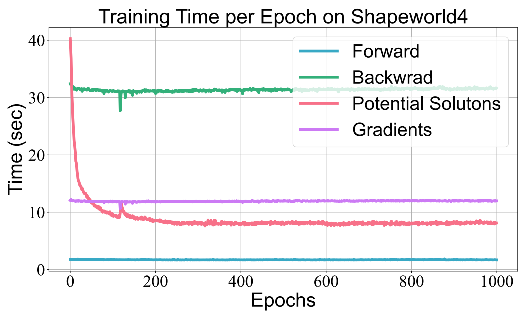


Figure 16: Training time of the four training steps of the SLASH pipeline with SAME. Over time, SAME reduces the time spent on computing Potential Solutions, while all other steps stay constant in time.

In Sec. 4.4 we saw that there is still some gap between SLASH and its baseline. Here we want to have a closer look at where the overhead is coming from. The training of SLASH can be seen as four steps: The forward pass, computing potential solutions with ASP, computing gradients and lastly the backward pass. Tab. 12 gives an overview of the average time spent on each of these steps per epoch. The first observation we make is that the forward and backward pass in sum takes longer than the total training of the baseline. This is because we are using Einsum Network's as the NPPs and that we are using a NPP for each object concept instead of using a single MLP for all concepts and objects at once. As a result, a lot more parameters are used in total, which increases the time spent on neural computations. The biggest bottleneck though is computing the potential solutions, which makes up more than 50% of the training time. Fig. 16 shows how SAME helps to mitigate this overhead and reduces the average time to compute Potential Solutions from 49 seconds to 8.8 seconds, making it not longer the training bottleneck. Computing the gradients stays constant over time and is responsible for 20% of the total training time for SAME. In general, DPPLs as of now utilize a GPU for neural computations, while solving and computing gradients happens on the CPU. As argued before, this suggests that an interesting research direction would be to find a closer integration of the neural and symbolic components of the pipelines for parallel and faster training.

References

Alviano, M., & Faber, W. (2011). Dynamic magic sets and super-coherent answer set programs. *AI Communications*, *24*, 125–145.

Bengio, Y. (2019). From System 1 Deep Learning to System 2 Deep Learning. Invited talk NeurIPS.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P. A., Horsfall, P., & Goodman, N. D. (2019). Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, *20*, 28:1–28:6.

Calimeri, F., Faber, W., Gebser, M., Ianni, G., Kaminski, R., Krennwallner, T., Leone, N., Maratea, M., Ricca, F., & Schaub, T. (2020). Asp-core-2 input language format. *Theory and Practice of Logic Programming*, *20*, 294–309.

Choi, Y., Vergari, A., & Van den Broeck, G. (2020). Probabilistic circuits: A unifying framework for tractable probabilistic models. Tech. rep., UCLA.

Ciravegna, G., Giannini, F., Gori, M., Maggini, M., & Melacci, S. (2020). Human-driven FOL explanations of deep learning. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 2234–2240.

Clocksin, W. F., & Mellish, C. (1981). *Programming in Prolog*. Springer.

Colmerauer, A., & Roussel, P. (1993). The birth of prolog. In *Proceedings of History of Programming Languages Conference (HOPL-II)*, pp. 37–52.

Cropper, A., Dumancic, S., Evans, R., & Muggleton, S. H. (2022). Inductive logic programming at 30. *Machine Learning*, *111*, 147–172.

Darwiche, A. (2011). SDD: A new canonical representation of propositional knowledge bases. In *Proceedings of the 22nd International Joint Conference on Artificial Intelligence*, pp. 819–826.

d'Avila Garcez, A., & Lamb, L. C. (2023). Neurosymbolic ai: The 3rd wave. *Artificial Intelligence Review*.

d'Avila Garcez, A. S., Gori, M., Lamb, L. C., Serafini, L., Spranger, M., & Tran, S. N. (2019). Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *Journal of Applied Logics*, 611–632.

d'Avila Garcez, A. S., Lamb, L. C., & Gabbay, D. M. (2009). *Neural-Symbolic Cognitive Reasoning*. Springer.

Dimopoulos, Y., Nebel, B., & Koehler, J. (1997). Encoding planning problems in nonmonotonic logic programs. In *Proceedings of Recent Advances in AI Planning, 4th European Conference on Planning*, Vol. 1348, pp. 169–181.

Dong, H., Mao, J., Lin, T., Wang, C., Li, L., & Zhou, D. (2019). Neural logic machines. In *Proceedings of the 7th International Conference on Learning Representations*.

Eiter, T., Higuera, N., Oetsch, J., & Pritz, M. (2022). A neuro-symbolic ASP pipeline for visual question answering. *Theory and Practice of Logic Programming*, *22*, 739–754.

Greff, K., Kaufman, R. L., Kabra, R., Watters, N., Burgess, C., Zoran, D., Matthey, L., Botvinick, M. M., & Lerchner, A. (2019). Multi-object representation learning with iterative variational inference. In *Proceedings of the 36th International Conference on Machine Learning*, Vol. 97, pp. 2424–2433.

Huang, J., Li, Z., Chen, B., Samel, K., Naik, M., Song, L., & Si, X. (2021). Scallop: From probabilistic deductive databases to scalable differentiable reasoning. In *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems*, pp. 25134–25145.

Hudson, D. A., & Manning, C. D. (2019). Learning by abstraction: The neural state machine. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems*, pp. 5901–5914.

Jiang, J., & Ahn, S. (2020). Generative neurosymbolic machines. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*.

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 Conference on Computer Vision and Pattern Recognition*, pp. 1988–1997.

Kingma, D. P., & Ba, J. (2015). Adam: A method for stochastic optimization. In *Proceedings of the 3rd International Conference on Learning Representations*.

Kodali, V., & Berleant, D. (2022). Recent, rapid advancement in visual question answering: a review. In *Proceedings of the International Conference on Electro Information Technology*, pp. 139–146.

Kuhn, H. W. (1955). The hungarian method for the assignment problem. *Naval research logistics quarterly*, *2*, 83–97.

Kuhnle, A., & Copestake, A. A. (2017). Shapeworld - A new test methodology for multimodal language understanding. *CoRR*.

LeCun, Y., Bottou, L., Bengio, Y., & Haffner, P. (1998a). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, *86*, 2278–2324.

LeCun, Y., Cortes, C., & J.C. Burges, C. (1998b). MNIST handwritten digit database. *http://yann.lecun.com/exdb/mnist/*.

Lee, J., & Wang, Y. (2016). Weighted rules under the stable model semantics. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 145–154.

Lin, Z., Wu, Y., Peri, S. V., Sun, W., Singh, G., Deng, F., Jiang, J., & Ahn, S. (2020). SPACE: unsupervised object-oriented scene representation via spatial attention and decomposition. In *Proceedings of the 8th International Conference on Learning Representations*.

Locatello, F., Weissenborn, D., Unterthiner, T., Mahendran, A., Heigold, G., Uszkoreit, J., Dosovitskiy, A., & Kipf, T. (2020). Object-centric learning with slot attention. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems*.

Manhaeve, R., Dumancic, S., Kimmig, A., Demeester, T., & Raedt, L. D. (2018). DeepProbLog: Neural probabilistic logic programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems*, pp. 3753–3763.

Manhaeve, R., Marra, G., & Raedt, L. D. (2021). Approximate inference for neural probabilistic logic programming. In *Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning*, pp. 475–486.

Manmadhan, S., & Kovoor, B. C. (2020). Visual question answering: a state-of-the-art review. *Artificial Intelligence Review*, *53*, 5705–5745.

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*.

Marek, V. W., & Truszczynski, M. (1999). Stable models and an alternative logic programming paradigm. In *The Logic Programming Paradigm - A 25-Year Perspective*, pp. 375–398. Springer.

Palù, A. D., Dovier, A., Pontelli, E., & Rossi, G. (2009). GASP: Answer Set Programming with Lazy Grounding. *Fundamenta Informaticae*, *96*, 297–322.

Peharz, R., Lang, S., Vergari, A., Stelzner, K., Molina, A., Trapp, M., den Broeck, G. V., Kersting, K., & Ghahramani, Z. (2020). Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *Proceedings of the 37th International Conference on Machine Learning*, Vol. 119, pp. 7563–7574.

Poon, H., & Domingos, P. M. (2011). Sum-product networks: A new deep architecture. In *Proceedings of the 27th Conference on Uncertainty in Artificial Intelligence*, pp. 337–346.

Redmon, J., Divvala, S. K., Girshick, R. B., & Farhadi, A. (2016). You only look once: Unified, real-time object detection. In *Proceedings of the Conference on Computer Vision and Pattern Recognition*, pp. 779–788.

Richardson, M., & Domingos, P. M. (2006). Markov logic networks. *Machine Learning*, *62*, 107–136.

Seide, F., Fu, H., Droppo, J., Li, G., & Yu, D. (2014). 1-bit stochastic gradient descent and its application to data-parallel distributed training of speech dnns. In *Proceedings of the 15th Annual Conference of the International Speech Communication Association*, pp. 1058–1062.

Skryagin, A., Stammer, W., Ochs, D., Dhami, D. S., & Kersting, K. (2022). Neural-probabilistic answer set programming. In *Proceedings of the 19th International Conference on Principles of Knowledge Representation and Reasoning.*

Soininen, T., & Niemelä, I. (1999). Developing a declarative rule language for applications in product configuration. In *Proceedings of First International Workshop on Practical Aspects of Declarative Languages*, Vol. 1551, pp. 305–319.

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 Conference on Computer Vision and Pattern Recognition*, pp. 3619–3629.

Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., & Blei, D. M. (2017). Deep probabilistic programming. In *Proceedings of the 5th International Conference on Learning Representations.*

Winters, T., Marra, G., Manhaeve, R., & De Raedt, L. (2022). Deepstochlog: Neural stochastic logic programming. In *Proceedings of the Thirty-Sixth AAAI Conference on Artificial Intelligence*, pp. 10090–10100.

Yang, Z., Ishay, A., & Lee, J. (2020). NeurASP: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence*, pp. 1755–1762.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., & Tenenbaum, J. (2018). Neural-symbolic VQA: disentangling reasoning from vision and language understanding. In *Advances in Neural Information Processing Systems: Annual Conference on Neural Information Processing Systems*, pp. 1039–1050.

Yu, Z., Zhu, M., Trapp, M., Skryagin, A., & Kersting, K. (2021). Leveraging probabilistic circuits for nonparametric multi-output regression. In *Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence*, Vol. 161, pp. 2008–2018.