Neural-Probabilistic Answer Set Programming

Arseny Skryagin[1] , **Wolfgang Stammer**[1] , **Daniel Ochs**[1] ,
Devendra Singh Dhami[1,3] , **Kristian Kersting**[1,2,3]
[1]Computer Science Department, Technical University Darmstadt, Germany
[2]Centre for Cognitive Science, TU Darmstadt
[3]Hessian Center for AI (hessian.AI)
{firstname.surename}@cs.tu-darmstadt.de

Abstract

The goal of combining the robustness of neural networks and the expressivity of symbolic methods has rekindled the interest in Neuro-Symbolic AI. One specifically interesting branch of research is deep probabilistic programming languages (DPPLs) which carry out probabilistic logical programming via the probability estimations of deep neural networks. 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 logical program, united via answer set programming. NPPs are a novel design principle allowing for the unification of 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. We evaluate SLASH on the benchmark task of MNIST addition as well as novel tasks for DPPLs such as missing data prediction, generative learning and set prediction with state-of-the-art performance, thereby showing the effectiveness and generality of our method.[1]

1 Introduction

In recent years, Neuro-Symbolic AI approaches to learning (Hudson and Manning 2019; d'Avila Garcez et al. 2019; Jiang and Ahn 2020; d'Avila Garcez and Lamb 2020) have gained traction. In general, these 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 been shown to have several advantages for various tasks such as visual question answering 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, Schramowski, and Kersting 2021).

Rather than designing specifically tailored neuro-symbolic architectures, where often the neural and symbolic modules are disjoint and trained independently (Yi et al. 2018; Mao et al. 2019; Stammer, Schramowski, and Kersting 2021), deep probabilistic programming languages (DPPLs) provide an exciting alternative (Bingham et al. 2019; Tran et al. 2017; Manhaeve et al. 2018; Yang, Ishay, and Lee 2020). Specifically, DPPLs integrate neural and symbolic modules via a unifying programming framework with probability estimates acting as the *"glue"* between separate modules, thus allowing for reasoning over noisy, uncertain data and, importantly, joint training of the modules. Additionally, prior knowledge and biases in the form of logical rules can easily and explicitly be added into 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 to integrate neural networks easily into downstream logical reasoning tasks.

Recent state of the art DPPLs, such as DeepProbLog (Manhaeve et al. 2018) and NeurASP (Yang, Ishay, and Lee 2020) allow for conditional class probability estimates as both works base their probability estimates on neural predicates. Although certain tasks may only require conditional class probabilities, we argue that for the overall expressive and utilization power of a DPPL, it is necessary to also be able to integrate and process joint probability estimates. The world is uncertain, and often it becomes necessary to reason in settings, e.g., in which variables of an observation might be missing or even manipulated.

In this work, we therefore make two main contributions. First, we propose a novel form of predicates for DPPLs, termed Neural Probabilistic Predicates (NPPs) 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 DeepProbLog and NeurASP, as well as purely probabilistic predicates. Further, we introduce a much more powerful *"flavor"* of NPPs that consist 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 next construct a novel DPPL, SLASH, to efficiently combine 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 and scalable programming language that seamlessly integrates probabilistic logical programming with neural representations and tractable probabilistic estimations. This

[1]Code will be made public upon acceptance.

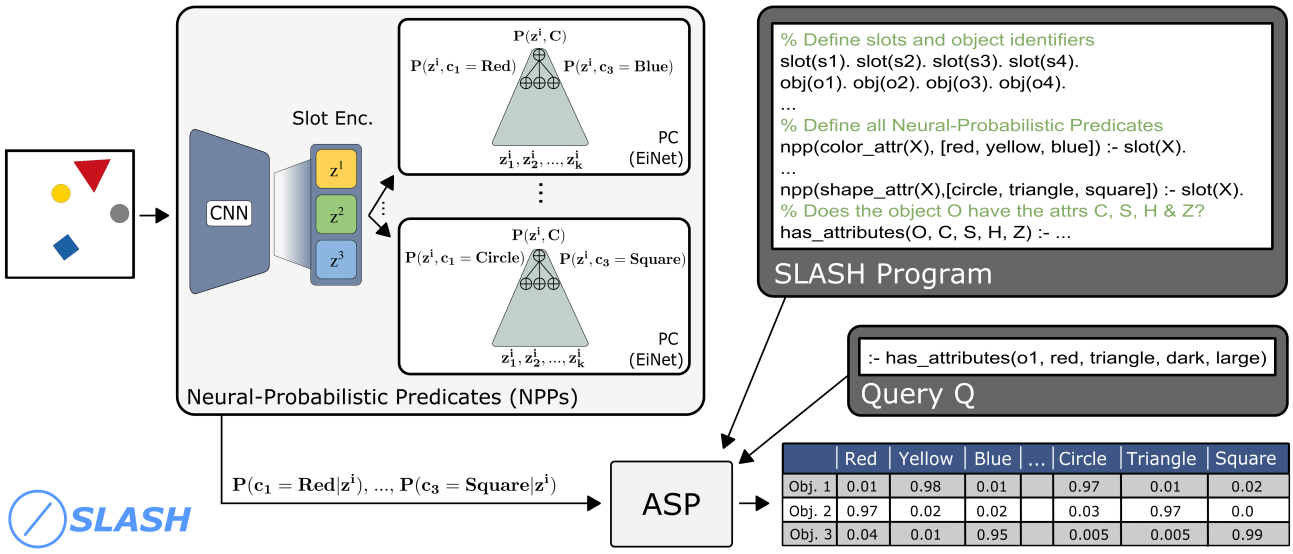

Figure 1: NPPs consist of neural and/or probabilistic circuit modules and can produce task-specific probability estimates. In our novel DPPL, SLASH, NPPs are integrated with a logic program via an ASP module to answer logical queries about data samples. In the depicted instantiation of SLASH (*SLASH Attention*) the Neural-Probabilistic Predicates consist of a slot attention encoder and Probabilistic Circuits (PCs) realized via EiNets. The slot encoder is shared over all NPPs, where each EiNet computes a joint distribution at the root node, thereby, learning the joint distribution over slot encodings, z^i, and object attributes, C, of a specific category, e.g., color attributes. Via targeted queries to the NPPs, one can obtain task-related probabilities, e.g., conditional probabilities for a visual reasoning task.

allows for the integration of all forms of probability estimations, not just class conditionals, thus extending the important works of (Manhaeve et al. 2018) and (Yang, Ishay, and Lee 2020).

Apart from NPPs, SLASH additionally consists of a logical program, containing a set of facts and logical statements that define the state of the world of an underlying task. Lastly, an ASP module is used to combine the NPP(s) with the logic program. Given a logical query about the input data, the logical program and the probability estimates obtained from the NPP(s), the ASP module produces a probability estimate about the truth value of the query. Finally, training in SLASH is performed efficiently in a batch-wise and end-to-end fashion, by integrating the parameters of all modules (neural and probabilistic) into a single loss term.

Fig. 1 exemplifies the building blocks of SLASH and NPPs for a specific instantiation of SLASH, termed SLASH Attention. This is designed for the task of set prediction from images. In this example, the NPPs consist of a slot attention encoder (Locatello et al. 2020) and several EinSum PCs (Peharz et al. 2020). The slot encoder is shared across all NPPs, whereas the PC of each NPP models a separate category of attributes. This way, each NPP models the joint distribution over slot encodings and object attribute values, such as color of an object. By querying the NPP, one can obtain task-related probability estimations, such as the conditional attribute probability. Finally, via the logic program, the user can predefine a set of statements and rules, e.g., of when an object possesses specific set of attributes, and query if an observed image contains a large, dark red triangle.

To show the effectiveness and advantages of SLASH and thus NPPs, we provide extensive experimental evaluations on various data sets and tasks. Specifically, we investigate the advantages of SLASH in comparison to SOTA DPPLs on the benchmark task of MNIST-Addition (Manhaeve et al. 2018). We further expand on this benchmark for a missing data setup as well as generative MNIST Addition task. Our results indicate the advantage of true probabilistic density estimation via appropriate NPPs. Finally, we show that SLASH Attention provides superior results for set prediction in terms of accuracy and generalization abilities compared to a baseline slot attention encoder. Both image generation and set prediction are novel benchmark tasks that no previous DPPL has tackled. With our results, we show that SLASH is a realization of "one system – two approaches" (Bengio 2019), that can successfully be used for performing various tasks and on a variety of data types.

In summary, we make the following contributions:

- introduce neural-probabilistic predicates,
- efficiently integrate answer set programming with probabilistic inference via NPPs within our novel DPPL, SLASH.
- successfully 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 a variety of tasks and data sets compared to state-of-the-art DPPLs and neural models.

We proceed as follows. We start off by introducing NPPs, including its contribution to the loss function when learning. Then we introduce SLASH programs with the corresponding semantics and loss function. Before concluding, we touch upon our experimental evaluation.

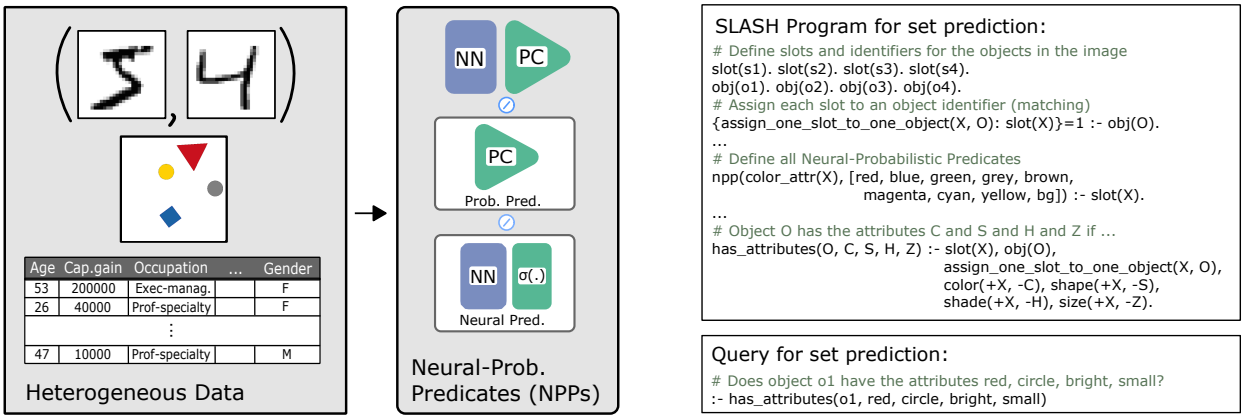

Figure 2: Basic SLASH building blocks and how to use them: (left) NPPs come in various flavors depending on the data and underlying task. Depending on the data set and underlying task, SLASH requires a suitable Neural-Probabilistic Predicate (NPP) that computes query-dependent probability estimates. A NPP can be composed of neural and probabilistic modules, or (depicted via slash symbol) only one of these two. (right) A minimal SLASH program and query for the set prediction task, here only showing the NPP that models the color category per object. For the full program, we refer to the supplements.

2 SLASH by means of NPPs and vice versa

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

2.1 Neural-Probabilistic Predicates

Previous DPPLs, DeepProbLog (Manhaeve et al. 2018) and NeurASP (Yang, Ishay, and Lee 2020), introduced the *Neural Predicate* as an annotated-disjunction or as a propositional atom, respectively, to acquire conditional class probabilities, $P(C|X)$, via the softmax function at the output of an arbitrary DNN. As mentioned in the introduction, this approach has certain limitations concerning inference capabilities. To resolve this issue, we introduce *Neural-Probabilisitic Predicates* (NPPs).

Formally, we denote with

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

a Neural-Probabilistic Predicate h. Thereby, (i) *npp* is a reserved word to label a NPP, (ii) h a symbolic name of either a PC, NN or a joint of a PC and NN (*cf.* Fig. 2)(left), e.g., *color_attr* is the name of a NPP of Fig. 2(right). Additionally, (iii) x denotes a "term" and (iv) $v_1, \ldots, v_n$ are the n possible outcomes of h. For example, the placeholders for *color_attr* are the color attributes of an object (*Red, Blue, Green, etc.*).

A NPP abbreviates a rule of the form $c = v$ with $c \in \{h(x)\}$ and $v \in \{v_1, \ldots, v_n\}$. 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. $r^{color\_attr} = \{c = Red, c = Blue, c = Green, ...\}$ for the example depicted in Fig. 2(right).

Rules of the form $npp\,(h(x), [v_1, \ldots, v_n]) \leftarrow \textit{Body}$ are used as an abbreviation for application to multiple entities, e.g., multiple slots for the task of set prediction (*cf.* Fig. 2(right)). Hereby, *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 largely make use of NPPs that contain probabilistic circuits, which allow for tractable density estimation and modelling of joint probabilities. The term probabilistic circuit (PC) (Choi, Vergari, and Van den Broeck 2020) represents a unifying framework that encompasses all computational graphs which encode probability distributions and guarantee tractable probabilistic modelling. These include Sum-Product Networks (SPNs) (Poon and 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, i.e. $P(X, C)$, $P(X|C)$, $P(C|X)$ 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. Hereby, the neural module learns to map the raw input data into an optimal latent representation, e.g., object-based slot representations. 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.

For making the different probabilistic queries distinguishable in a SLASH program, we introduce the following notation. We denote a given variable with $+$ and the query variable with $-$. E.g., within the running example of set prediction (*cf.* Fig. 1 and 2(right)), with the query $color\_attr(+X, -C)$ one is asking for $P(C|X)$. Similarly, with $color\_attr(-X, +C)$ one is asking for $P(X|C)$ and,

finally, with $color\_attr(+X, +C)$ for $P(X, C)$. And, in the case when no data is available, i.e, $color\_attr(-X, -C)$, we are querying for the prior $P(C)$.

To summarize, a NPP can consist 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, Ishay, and Lee 2020), but also 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 2(left)).

Lastly, for parameter learning NPPs possess a unified loss function of the negative log-likelihood,

$$L_{NPP} := -\log LH(x, \hat{x}) = \sum_{i=1}^{n} LH(x_i, \hat{x}_i) = \\ -\sum_{i=1}^{n} x_i \cdot \log(P_{\xi}^{(X,C)}(x_i)) = -\sum_{i=1}^{n} \log(P_{\xi}^{(X,C)}) \quad (2)$$

whereby LH is an abbreviation for log-likelihood. Further, we assume data to be i.i.d., ground truth x_i to be the all-ones vector, ξ to be the parameters of the NPP and $P_{\xi}^{(X,C)}$ are the predictions $\hat{x}_i$ obtained from the PC encoded in the NPP.

2.2 SLASH: a novel DPPL for integrating NPPs

Let us now introduce SLASH, a novel DPPL that efficiently integrates NPPs with logic programming.

SLASH Language and Semantics We continue in the pipeline on how to use the probability estimates of NPPs for answering logical queries, and begin by formally defining a SLASH program.

Definition 1. *A SLASH program* Π *is the union of* Π^{asp}, Π^{npp}. *Therewith,* 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** to most part. Compared to ProbLog, ASP rarely goes into an infinite loop (*c.f.* chapter 2.9 of (Lifschitz 2019) for a simple code example leading to infinite loops with ProbLog) and is therefore preferable as a backbone. Fig. 2(right) presents a minimal SLASH program for the task of set prediction, exemplifying a set of propositional rules and neural predicates.

We now address how to integrate NPPs into an ASP compatible form to obtain the success probability for the logical query given all possible solutions. Thus, we define **SLASH's semantics**. For SLASH to translate the program Π to the ASP-solver's compatible form, the rules (Eq. 1) will be rewritten to the set of rules:

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

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. We denote a set of ASP constraints in the form $\leftarrow$ *Body*, as *queries* Q (annotation). and each of the solutions with respect to Q as a *potential solution*, I, (referred to as *stable model* in ASP).

With $I|_{r^{npp}}$ we denote the projection of the I onto r^{npp}, $Num(I|_{r^{npp}}, \Pi)$ – the number of the possible solutions of the program Π agreeing with $I|_{r^{npp}}$ on r^{npp}. Because we aim to calculate the success probability of query Q, we formalize the probability of potential solution I beforehand.

Definition 2. *We specify the probability of the potential solution, I, for the program* Π *as the product of the probabilities of all atoms* $c = v$ *in* $I|_{r^{npp}}$ *divided by the number of potential solutions of* Π *agreeing with* $I|_{r^{npp}}$ *on* r^{npp}*:*

$$P_{\Pi}(I) = \begin{cases} \frac{\prod_{c=v \in I|_{r^{npp}}} P_{\Pi}(c=v)}{Num(I|_{r^{npp}}, \Pi)}, & \text{if } I \text{ is pot. sol. of } \Pi, \\ 0, & \text{otherwise.} \end{cases} \quad (4)$$

Thus, we can now define the probability of a query.

Definition 3. *The probability of the query* Q *given the set of possible solutions* I *is defined as*

$$P_{\Pi}(Q) := \sum_{I \models Q} P_{\Pi}(I). \quad (5)$$

Thereby, $I \models Q$ *reads as "*I *satisfies* Q*". The probability of the set of queries* $\mathbf{Q} = \{Q_1, \ldots, Q_l\}$ *is defined as the product of the probability of each. I.e.*

$$P_{\Pi}(\mathbf{Q}) := \prod_{Q_i \in \mathbf{Q}} P_{\Pi}(Q_i) = \prod_{Q_i \in \mathbf{Q}} \sum_{I \models Q} P_{\Pi}(I). \quad (6)$$

Parameter Learning in SLASH We denote with $\Pi(\boldsymbol{\theta})$ the SLASH program under consideration, where $\boldsymbol{\theta}$ is the set of the parameters associated with Π. Further, making the i.i.d. assumption of the query set $\mathbf{Q}$, we follow (Manhaeve et al. 2018) and (Skryagin et al. 2020), and use the *learning from entailment* setting. That is, the training examples are logical queries that are known to be true in the SLASH program $\Pi(\boldsymbol{\theta})$. The goal is to learn the parameters $\boldsymbol{\theta}$ of the program $\Pi(\boldsymbol{\theta})$ so that the observed queries are most likely.

To this end, we employ the negative log-likelihood and the cross-entropy of the observed queries $P_{\Pi(\theta)}(Q_i)$ and their predicted probability value $P^{(X_{\mathbf{Q}},C)}(x_{Q_i})$, assuming the NPPs are fixed:

$$L_{ENT} := \\ -\log LH\left(\log(P_{\Pi(\theta)}(\mathbf{Q})), P^{(X_{\mathbf{Q}},C)}(x_{\mathbf{Q}})\right) = \\ -\sum_{j=1}^{m} \log(P_{\Pi(\theta)}(Q_{ij})) \cdot \log\left(P^{(X_{\mathbf{Q}},C)}(x_{Q_{ij}})\right). \quad (7)$$

This loss function aims at maximizing the estimated success probability. We remark that the defined loss function is true 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. Furthermore, we assume that for the set of queries $\mathbf{Q}$ it holds that,

$$P_{\Pi(\boldsymbol{\theta})}(Q) > 0 \quad \forall\, Q \in \mathbf{Q}.$$

In accordance with the semantics, we seek to reward the right solutions $v = c$ and penalize the wrong ones $v \neq c$. Referring to the probabilities in r^{npp} (i.e., the set of logical rules denoting NPPs; see Def. 2) as $\mathbf{p}$, one can compute their gradients w.r.t. $\boldsymbol{\theta}$ via backpropagation as

$$\sum_{Q \in \mathbf{Q}} \frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \boldsymbol{\theta}} = \sum_{Q \in \mathbf{Q}} \frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \mathbf{p}} \times \frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}} \,. \tag{8}$$

The term $\frac{\partial \mathbf{p}}{\partial \boldsymbol{\theta}}$ can now be computed as usual via backward propagation through the NPPs (see Eq. 13 in the supplementary materials (Suppl.) A for details). By letting p to be the label of the probability of an atom $c = v$ in r^{npp} and denoting $P_{\Pi(\boldsymbol{\theta})}(c = v)$, the term $\frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \mathbf{p}}$ follows from NeurASP (Yang, Ishay, and Lee 2020) as

$$\frac{\partial \log\left(P_{\Pi(\boldsymbol{\theta})}(Q)\right)}{\partial \mathbf{p}} = \frac{\sum\limits_{\substack{I : I \models Q \\ I \models c = v}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v)} - \sum\limits_{\substack{I : I, v' \models Q \\ I \models c = v', v \neq v'}} \frac{P_{\Pi(\boldsymbol{\theta})}(I)}{P_{\Pi(\boldsymbol{\theta})}(c=v')}}{\sum\limits_{I : I \models Q} P_{\Pi(\boldsymbol{\theta})}(I)}. \tag{9}$$

Doing this is a sensible idea. For instance, if a query to be true is not likely to be entailed, the gradient is positive. Putting everything together, the final loss function is

$$L_{SLASH} = L_{NPP} + L_{ENT} \tag{10}$$

and we perform training using coordinate descent, i.e., we train the NPPs, then train the program with fixed NPPs, train the NPPs with the program fixed, and so on.

Complementary, we formulate the proposition in the following, showing exactly the way the parameter learning is performed via the logical entailment loss function L_{ENT}, i.e., we derive its gradients.

Proposition 1. *The average derivative of the logical entailment loss function L_{ENT} defined in Eq. 7 can be estimated as follows*

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

Proof. Applying the i.i.d assumption over the likelihood and the definition of the cross-entropy for two vectors y_i and $\hat{y}_i$ we obtain

$$L_{ENT} = -\log LH(y, \hat{y}) = \sum_{i=1}^{n} H(y_i, \hat{y}_i)$$
$$= -\sum_{i=1}^{n}\sum_{j=1}^{m} \log(P_{\Pi(\theta)}(Q_{ij})) \cdot \log\left(P^{(X_{\mathbf{Q}}, C)}(x_{Q_{ij}})\right).$$

With m being the number of classes defined in the domain of a NPP. Recognizing that the second term of the derivative

$$\frac{\partial}{\partial \mathbf{p}} H\left(y_i, \hat{y}_i\right) =$$
$$-\sum_{j=1}^{m} \left[\frac{\partial \log(P_{\Pi(\theta)}(Q_{ij}))}{\partial \mathbf{p}} \cdot \log\left(P^{(X_{\mathbf{Q}}, C)}(x_{Q_{ij}})\right) + \log(P_{\Pi(\theta)}(Q_{ij})) \cdot \frac{\partial \log\left(P^{(X_{\mathbf{Q}}, C)}(x_{Q_{ij}})\right)}{\partial \mathbf{p}} \right],$$

is a constant and assembling all intermediate steps, we obtain the desired inequality. □

In a similar way one can derive the derivative of the NPP loss function (Eq. 2) as:

$$\frac{\partial}{\partial \xi} L_{NPP} = \frac{\partial x_{Q_i}}{\partial \xi} \left(-\sum_{i=1}^{n} \frac{1}{\left(P_{\xi}^{(X_{\mathbf{Q}}, C)}(x_{Q_i})\right)} \frac{\partial}{\partial x_{Q_i}} \left(P_{\xi}^{(X_{\mathbf{Q}}, C)}(x_{Q_i})\right) \right)$$

For more details of the proof and derivations, we refer to the Suppl. A. It also includes the details on parameter learning for such NPPs, as depicted for SLASH Attention.

Importantly, with the learning schema described above, rather than requiring a novel loss function for each individual task and data set, with SLASH, it is now possible to simply incorporate specific task and data requirements into the logic program. The training loss, however, remains the same.

3 Empirical Illustrations

Let us now turn to experimental evaluations. In particular, a main advantage of SLASH lies in the efficient integration of neural, probabilistic and symbolic computations. To emphasize this, but also the particular advantages of integrating true probabilistic density estimation into logic programming via NPPs, we conduct a variety of experimental evaluations showing the advantages and usability of SLASH on novel tasks for DPPLs.

3.1 Experimental Details.

We use two benchmark data sets, namely MNIST (LeCun, Cortes, and J.C. Burges 1998) for the task of MNIST-Addition and a variant of the ShapeWorld data set (Kuhnle and Copestake 2017) for object-centric set prediction. For all experiments, we present the average and the standard deviation over five runs with different random seeds for parameter initialization.

For ShapeWorld experiments, we generate a data set we refer to as ShapeWorld4. Images of ShapeWorld4 contain between one and four objects, with each object consisting of four attributes: a color (red, blue, green, gray, brown, magenta, cyan or yellow), a shade (bright, or dark), a shape (circle, triangle, or square) and a size (small or big). Thus, each object can be created from 84 different combinations of attributes. Fig. 1 depicts an example image.

We measure performance via classification accuracies in the MNIST-Addition task. In our ShapeWorld4 experiments, we present the average precision. We refer to Suppl. B for

	Test Acc. (%)
DeepProbLog	98.49 ± 0.18
NeurASP	98.21 ± 0.30
SLASH (PC)	95.52 ± 0.32
SLASH (DNN)	$\mathbf{98.99 \pm 0.04}$

	DeepProbLog	NeurASP	**SLASH** (DNN)	**SLASH** (PC)
50%	$\mathbf{97.73 \pm 0.12}$	25.75 ± 31.32	96.0 ± 0.02	97.59 ± 0.01
80%	76.07 ± 18.38	10.29 ± 0.83	95.4 ± 0.01	$\mathbf{96.8 \pm 0.00}$
90%	69.15 ± 29.15	13.28 ± 4.94	75.6 ± 0.18	$\mathbf{95.00 \pm 0.00}$
97%	32.46 ± 22.48	11.94 ± 0.95	67.4 ± 0.09	$\mathbf{84.2 \pm 0.03}$

Table 1: MNIST Addition and missing data MNIST Addition Results. Test accuracy corresponds to the percentage of correctly classified test images. (left) Baseline MNIST Addition task with various DPPLs, including SLASH with a NPP that models the joint probabilities (SLASH (PC)) and one that models only conditional probabilities (SLASH (DNN)). (right) MNIST Addition task with missing pixels. The amount of missing pixels was varied between 50% and 97% of the pixels per image.

the SLASH programs and queries of each experiment, and Suppl. C for a detailed description of hyperparameters and further details. Lastly, we remark to have trained all experiments but Generative MNIST-Addition by discriminative fashion only.

3.2 Experimental Results

Let us now present our experimental findings.

Evaluation 1: SLASH outperforms SOTA DPPLs in MNIST-Addition. The task of MNIST-Addition (Manhaeve et al. 2018) is to predict the sum of two MNIST digits, presented only as raw images. During test time, however, a model should classify the images directly. Thus, although a model does not receive explicit information about the depicted digits, it must learn to identify digits via indirect feedback on the sum prediction.

We compare the test accuracy after convergence between the three DPPLs: DeepProbLog (Manhaeve et al. 2018), NeurASP (Yang, Ishay, and Lee 2020) and SLASH, using a probabilistic circuit (PC) or a deep neural network (DNN) as NPP. Notably, the DNN used in SLASH (DNN) is the LeNet5 model (LeCun et al. 1998) of DeepProbLog and NeurASP. When using the PC as NPP, we have also 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 results can be seen in Tab. 1(left). We observe that training SLASH with a DNN NPP produces SOTA accuracies compared to DeepProbLog and NeurASP, confirming that SLASH's loss computation leads to improved performances. Apart from improved mean accuracies, it is important to note that the training of SLASH (DNN) also led to the massively reduced standard deviation in the test accuracy, suggesting that SLASH allows for much more stable results than the other DPPLs.

We further observe that the test accuracy of SLASH with a PC NPP is slightly below the other DPPLs. However, we argue that this may be since a PC, in comparison to a DNN, is learning a true mixture density rather than just conditional probabilities. The advantages of doing so will be investigated in the next experiments. Note that, optimal architecture search for PCs, e.g., for computer vision, is an open research question. We also refer to the Suppl. E, indicating substantial improvements in terms of inference time (even with a PC-based NPP, thus performing tractable probabilistic inference) due to our efficient implementation of SLASH with to batchwise learning and parallel calls to the ASP solver of SLASH.

These evaluations, in summary, show SLASH's advantages on the benchmark MNIST-Addition task. Additional benefits will be made clear in the following experiments.

Evaluation 2: Handling Missing Data with SLASH. SLASH offers the advantage of a flexibility to use various kinds of NPPs. Thus, in comparison to previous DPPLs, one can easily integrate NPPs into SLASH that perform joint probability estimation. For this evaluation, we again consider the task of MNIST-Addition with missing data. Specifically, in this evaluation, all three DPPLs were trained with the MNIST-Addition task with images in which a percentage of pixels per image has been removed. Importantly, whereas DeepProbLog, NeurASP and SLASH (DNN) handle the missing data simply as background pixels, SLASH (PC) specifically models the missing data as uncertain data by marginalizing the denoted pixels at inference time.

As can be seen in Tab. 1(right), regardless of the NPP's form and the rate of the missing pixels per image, SLASH outperforms other DPPLs, both in terms of higher mean test accuracy and lower standard deviation. We observe that at 50%, DeepProbLog and SLASH produce almost equal accuracies. With 80% percent missing pixels, there is a substantial difference in the ability of the two DPPLs to correctly classify images, with SLASH having quite stable results, and delivering only slightly decreased performance. By further increasing the percentage of missing pixels, this difference becomes even more substantial with SLASH (PC) still reaching a 84% test accuracy even when 97% of the pixels per image are missing, whereas DeepProbLog degrades to an average of 32% test accuracy.

We further note that SLASH, in comparison to DeepProbLog, produces largely reduced standard deviations over runs. Notably, over all seeds and settings, the test accuracy measured by NeurASP diverged after several epochs and did not recover. We presume this to be an effect of exploding gradients. We also refer to Suppl. D, containing results of training on the full MNIST data set, but testing on a set with missing pixels, further indicating benefits of SLASH for out-of-distribution data.

In summary, using the power of true density estimation, SLASH can produce robust results in comparison to other DPPLs in a task with missing data.

Evaluation 3: True density estimation allows for generative learning. As previously mentioned, thanks to the NPP's

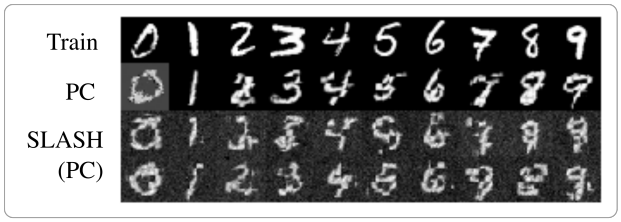

Figure 3: Qualitative comparison of Generative MNIST-Addition: each row is entailing images per class. The first one are ground truths obtained from the dataset, the second one represents the sampled images from a PC trained solely in a generative way. The last two rows are sampled from the NPP trained within SLASH. The third row depicts images after an epoch of the discriminative learning step, and the last row after a one epoch of generative step.

design, SLASH allows us to embrace true density estimation in logic programming. Having such a model for joint probability estimation allows for posing a much broader range of questions, e.g., based on generative learning.

In this evaluation, we investigate the situation where only a part of the overall dataset is available and the task is to discover the missing part through high-level reasoning and generative learning. This evaluation differs from evaluation 2 in that a model receives as input the result of the addition and one of the two images that contributes to the sum. Thus, an entire image is missing, and we wish not only to find the value of the missing digit, but also generate the corresponding image as well. Using the $\pm$ notation, we are thus querying for the class prior of the missing image, a type of query that previous DPPLs cannot handle. Using a NPP in the form of a PC, we show that SLASH can handle this task.

To train SLASH for this challenging task, we require not only discriminative, but also generative learning. For this purpose, we applied coordinate descent with an Adam optimizer (Kingma and Ba 2015) to minimize the SLASH loss (discriminative learning) and expectation maximization (EM) to maximize the log-likelihood (generative learning). In other words, the data set consisted of available and sampled images during the generative phase of the training.

As neither of the previous DPPLs have generative capabilities, we here revert to comparing the generative abilities of SLASH with an PC-based NPP and trained via coordinate descent to a stand-alone PC trained solely via EM. Both PCs were constructed using the same set of settings and entail the same number of learnable parameters. We further revert here to a qualitative analysis of generated samples. Thus, the point of the results is to indicate the quality of generated samples via SLASH in comparison to the current SOTA image generation results via PCs.

As with other tasks, we present the corresponding program in the Suppl. B. Fig. 3 presents the qualitative comparison per class between the ground truths (data), images sampled from a PC trained in the generative fashion and images sampled from the NPP after training. We observe that SLASH generated digits, though not as qualitative as the stand-alone PC digits, still clearly depict characteristic features of the classes. This is a surprising result given that both objectives are perpendicular in their nature towards each other, and thus it is far more challenging for the sampled images to be as distinct and without artifacts as after pure generative training of stand-alone PCs.

Thus, utilizing the generative property of NPPs with the ability of joint probability estimation, we show that via the $\pm$-notation one can easily perform generative learning out-of-the-box via SLASH. Overall, the results are a first step of generative learning via a DPPL. We note that the quality of the generated sample still leaves room for improvements. We hypothesize that employing two different optimizers for each step of coordinate descent surely does not have an optimal effect on the PC NPP for converging towards the optimum of both criteria. A promising approach for future work is to investigate the hybrid generative-discriminative loss proposed in RAT-SPN (eq. 5) (Peharz et al. 2019) that unifies the training and mitigates the need for coordinate descent.

Evaluation 4: Improved Concept Learning via SLASH. Let us now turn to a very different setting of object-centric set prediction, a fascinating, yet challenging task that no DPPL has tackled up to now. Recent advances in object-centric deep learning (OCDL) have shown great architectural improvements for the performances of neural networks in complex visual tasks, such as object set prediction from images (Greff et al. 2019; Lin et al. 2020; Locatello et al. 2020). We propose that these advancements can be improved even further by integrating such neural components into DPPLs, and adding logical constrains about objects and their properties. For the fourth set of evaluations, we thus revert to the ShapeWorld4 dataset.

For set prediction, a model is trained to predict the discrete attributes of a set of objects in an image (*cf.* Fig. 1 for an example ShapeWorld4 image). The difficulty lies therein that the model must match an unordered set of corresponding attributes of a varying number of 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 and, through a competitive softmax-based attention mechanism, can enforce the binding of specific parts of a latent representation into permutation-invariant, task-specific vectors, called slots. In our experiments, we wish to show that by adding logical constraints in the training setting, one can improve the overall performances and generalization properties of such a model. For this, we train SLASH with NPPs as depicted in Fig. 1 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 ShapeWorld4, we thereby have altogether four NPPs. Finally, SLASH is trained via queries of the kind exemplified in Fig. 10 in the supplements. We refer to this configuration as *SLASH Attention*.

We compare SLASH Attention to a baseline slot attention encoder using an MLP and Hungarian loss for predicting the

	Slot Att.	**SLASH Att.**
	ShapeWorld4	
Test Set	90.24 ± 0.93	$\mathbf{95.58 \pm 0.61}$
	CoGenT	
Test Cond. A	90.37 ± 2.19	$\mathbf{96.85 \pm 0.43}$
Test Cond. B	27.15 ± 2.36	$\mathbf{40.58 \pm 1.99}$

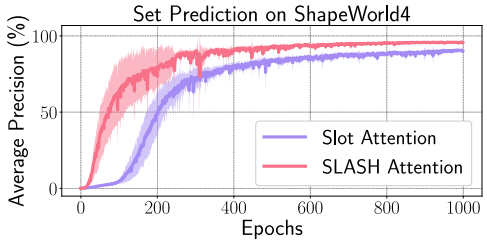

Figure 4: ShapeWorld4 Experiments. (Left) Converged test average precision scores for the set prediction task with ShapeWorld4 (top) and ShapeWorld4 CoGenT (bottom). (Right) Test average precision scores for set prediction with ShapeWorld4 over the training epochs. In these experiments, we compared a baseline slot encoder versus SLASH Attention with slot attention and PC-based NPPs. For the CoGenT experiments, a model is trained on one training set and tested on two separate test conditions. Condition A test set contains attribute compositions which were also seen during training. Condition B test set contains attribute compositions which were not seen during training, e.g., yellow circles were not present in the training set, but present in Condition B test set.

object properties from the slot encodings as in (Locatello et al. 2020). The key difference between these two models lies in the logical constraint we are employing in SLASH Attention. In their original work, the authors of (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 Fig. 2 (right) for a minimal version of the program, and Suppl. C for the full version.

The results of these experiments can be found in Fig. 4 (left). 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. Additionally, in Fig. 4 (right), we observe that SLASH Attention reaches the average precision value of the baseline model in much fewer number of epochs.

The goal of this evaluation was to test if we can observe any overall advantage of SLASH and NPPs for such a challenging visual reasoning task for which we had introduced the specific instantiation of SLASH, SLASH Attention. We can summarize that indeed by adding logical knowledge to the training procedure via SLASH one can greatly improve the capabilities of a neural module exemplified here for set prediction. These results suggest that the question of the independence of object attributes (e.g., an object has one color, one form, etc.) can be solved effectively by logic constraints. Consequentially, the monolithic multicategorical MLP with shared parameters used by (Locatello et al. 2020) is sufficient but not necessary.

Evaluation 5: Improved Compositional Generalization with SLASH. Lastly, to test the hypothesis that SLASH Attention possesses improved generalization properties in comparison to a baseline neural model, we ran experiments on a variant of ShapeWorld4 similar to the CLEVR Compositional Generalization Test (CoGenT) (Johnson et al. 2017). The goal of CoGenT is to investigate a model's ability to handle novel combinations of attributes that were not seen during training.

For this purpose, we established two conditions within a ShapeWorld4 CoGenT data set: **Condition (A)** – the training and test data set contains squares with the colors *gray*, *blue*, *brown*, or *yellow*, triangles with the colors *red*, *green*, *magenta*, or *cyan* and circles of *all* colors. **Condition (B)** – the training set is as in Condition (A). However, the test set contains squares with the colors *red*, *green*, *magenta*, or *cyan*, triangles with the colors *gray*, *blue*, *brown*, or *yellow* and circles of *all* colors. The goal is to investigate how well a model can generalize that, e.g., also squares can have the color red, although never having seen evidence for this during training.

The resulting average precision test scores are presented in Fig. 4a (left). We observe that, even though SLASH Attention with its corresponding program was not explicitly designed to handle composition generalization, it shows greatly improved generalization capabilities. This can be seen in the approx. 13% higher average precision scores on the Condition (B) test set in comparison to the baseline model. Importantly, this trend still holds even when taking the overall higher performance of SLASH Attention observed in Condition (A) into account.

To summarize our findings from the experiments on set prediction: we observe that adding prior knowledge in the form of logical constraints via SLASH can greatly improve a neural module in terms of performance and generalizability. On a side note: training neural networks for novel tasks, often involves defining explicit loss functions, e.g., Hungarian loss for set prediction. This stands in contrast to SLASH: no matter the choice of NPP and underlying task, the training loss remains the same. Task-related requirements simply need to be added as lines of code to the SLASH program. Thus, it highlights SLASH's versatility and flexibility even further.

Summary of Empirical Results. All empirical results together demonstrate that the expressiveness and flexibility of SLASH is highly beneficial and can easily outperform state-of-the-art: one can freely combine what is required to solve the underlying task — (deep) neural networks, PCs, and logic. Importantly, the results indicate the potential of NPPs in general and PC-based NPPs, in particular, via SLASH.

4 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, Lamb, and Gabbay 2009; Mao et al. 2019; Hudson and Manning 2019; d'Avila Garcez et al. 2019; Jiang and Ahn 2020; d'Avila Garcez and Lamb 2020) 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 type of representation by the other, e.g., using the power of symbolic reasoning systems to handle the generalizability issues of neural networks and on the other hand 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, Schramowski, and Kersting 2021). Many of these previous works, however, 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 logical 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 and Domingos 2006). Thereby, the binding link is the Weighted Model Counting (WMC) introduced in LP^{MLN} (Lee and 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), and NeurASP (Yang, Ishay, and Lee 2020).

To resolve the scalability issues of DeepProbLog, which use Sentential Decision Diagrams (SDDs) (Darwiche 2011) as the underlying data structure to evaluate queries, NeurASP (Yang, Ishay, and Lee 2020), offers a solution by utilizing Answer Set Programming (ASP) (Dimopoulos, Nebel, and Koehler 1997; Soininen and Niemelä 1999; Marek and Truszczynski 1999; Calimeri et al. 2020). In contrast to query evaluation in Prolog (Colmerauer and Roussel 1993; Clocksin and 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 similar knowledge representation system, NeurASP generates a set of all possible 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 in an end-to-end fashion, all of these are limited to estimating only conditional class probabilities.

Object-centric deep learning has recently brought forth several exciting avenues of research by introducing inductive biases to neural networks to extract objects from visual scenes in an unsupervised manner (Zhang, Hare, and Prügel-Bennett 2019; Burgess et al. 2019; Engelcke et al. 2020; Greff et al. 2019; Lin et al. 2020; Locatello et al. 2020; Jiang and Ahn 2020). We refer to (Greff, van Steenkiste, and Schmidhuber 2020) for a detailed overview. A motivation for this specific line of investigation, which notably has been around for a longer period of time (Fodor and Pylyshyn 1988; Marcus 2019), is that objects occurring as natural building blocks in human perception and possess advantageous properties for many cognitive tasks, such as scene understanding and reasoning. These works have shown great success in extracting object-based representations via their implicit architectural biases. However, an interesting avenue is to also add explicit logical biases, e.g., via constraints about objects and their properties in the form of logical statements such as color singularity.

5 Conclusion and Future Work

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 are 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 few lines of code. Finally, via Answer Set Programming and Weighted Model Counting, the logical SLASH program and probability estimates from the NPPs are combined to estimate the truth value of a task-specific query. Our experiments show the power and efficiency of SLASH, improving upon previous DPPLs in the benchmark MNIST-Addition task in terms of performance, efficiency, and robustness. Furthermore, the generative MNIST-Addition task demonstrates that via the generation of images which encapsulate logical knowledge bases, we are moving from data-rich to knowledge-rich AI. Importantly, by integrating a SOTA slot attention encoder into NPPs and adding few logical constraints, SLASH demonstrates improved performances and generalizability in comparison to the pure slot encoder for the task of object-centric set prediction; a setting no DPPL has tackled yet. Our results show the effectiveness and improved utility of SLASH over previous DPPLs, as well as the great potential overall of DPPLs to elegantly combine logical reasoning with neural computations and uncertainty estimates.

Interesting avenues for future work include benchmarking SLASH on additional data types and tasks. One should explore unsupervised and weakly supervised learning using logic with SLASH and investigate how far logical constraints can help unsupervised object discovery. An interesting and important avenue for a broader usability of NPPs, in particular, is to investigate the construction of multicategorical NPPs. Furthermore, the ideas of the generative MNIST-Addition task should be investigated further. As mentioned, one interesting approach to this end is the hybrid generative-discriminative loss of (Peharz et al. 2019).

Acknowledgements. 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, 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 "SPAICER" (01MK20015E), the Hessian research priority programme LOEWE within the project WhiteBox, as well as the HMWK cluster projects "The Third Wave of AI" and "The Adaptive Mind".

References

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.; and Goodman, N. D. 2019. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research* 28:1–28:6.

Burgess, C. P.; Matthey, L.; Watters, N.; Kabra, R.; Higgins, I.; Botvinick, M.; and Lerchner, A. 2019. Monet: Unsupervised scene decomposition and representation. *CoRR*.

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

Choi, Y.; Vergari, A.; and Van den Broeck, G. 2020. Probabilistic circuits: A unifying framework for tractable probabilistic models. Technical report, Technical report.

Ciravegna, G.; Giannini, F.; Gori, M.; Maggini, M.; and Melacci, S. 2020. Human-driven FOL explanations of deep learning. In *IJCAI*, 2234–2240.

Clocksin, W. F., and Mellish, C. 1981. *Programming in Prolog*.

Colmerauer, A., and Roussel, P. 1993. The birth of prolog. In *HOPL-II*, 37–52.

Darwiche, A. 2011. SDD: A new canonical representation of propositional knowledge bases. In *IJCAI*, 819–826.

d'Avila Garcez, A., and Lamb, L. C. 2020. Neurosymbolic AI: the 3rd wave. *CoRR*.

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

d'Avila Garcez, A. S.; Lamb, L. C.; and Gabbay, D. M. 2009. *Neural-Symbolic Cognitive Reasoning*. Cognitive Technologies.

Dimopoulos, Y.; Nebel, B.; and Koehler, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *ECP*, Lecture Notes in Computer Science, 169–181.

Engelcke, M.; Kosiorek, A. R.; Jones, O. P.; and Posner, I. 2020. GENESIS: generative scene inference and sampling with object-centric latent representations. In *ICLR*.

Fodor, J. A., and Pylyshyn, Z. W. 1988. Connectionism and cognitive architecture: A critical analysis. *Cognition* 3–71.

Greff, K.; Kaufman, R. L.; Kabra, R.; Watters, N.; Burgess, C.; Zoran, D.; Matthey, L.; Botvinick, M.; and Lerchner, A. 2019. Multi-object representation learning with iterative variational inference. In *ICML*, 2424–2433.

Greff, K.; van Steenkiste, S.; and Schmidhuber, J. 2020. On the binding problem in artificial neural networks. *CoRR*.

Hudson, D. A., and Manning, C. D. 2019. Learning by abstraction: The neural state machine. In *NeurIPS*, 5901–5914.

Jiang, J., and Ahn, S. 2020. Generative neurosymbolic machines. In *NeurIPS*.

Johnson, J.; Hariharan, B.; van der Maaten, L.; Fei-Fei, L.; Zitnick, C. L.; and Girshick, R. B. 2017. CLEVR: A diagnostic dataset for compositional language and elementary visual reasoning. In *CVPR*, 1988–1997.

Kingma, D. P., and Ba, J. 2015. Adam: A method for stochastic optimization. In *ICLR 2015*.

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

LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. 2278–2324.

LeCun, Y.; Cortes, C.; and J.C. Burges, C. 1998. THE MNIST DATABASE of handwritten digits.

Lee, J., and Wang, Y. 2016. Weighted rules under the stable model semantics. In *KR*, 145–154.

Lifschitz, V. 2019. *Answer Set Programming*. Springer International Publishing.

Lin, Z.; Wu, Y.; Peri, S. V.; Sun, W.; Singh, G.; Deng, F.; Jiang, J.; and Ahn, S. 2020. SPACE: unsupervised object-oriented scene representation via spatial attention and decomposition. In *ICLR*.

Locatello, F.; Weissenborn, D.; Unterthiner, T.; Mahendran, A.; Heigold, G.; Uszkoreit, J.; Dosovitskiy, A.; and Kipf, T. 2020. Object-centric learning with slot attention. In *NeurIPS*.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and Raedt, L. D. 2018. Deepproblog: Neural probabilistic logic programming. 3753–3763.

Mao, J.; Gan, C.; Kohli, P.; Tenenbaum, J. B.; and Wu, J. 2019. The neuro-symbolic concept learner: Interpreting scenes, words, and sentences from natural supervision. In *ICLR*.

Marcus, G. F. 2019. *The algebraic mind: Integrating connectionism and cognitive science*.

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

Peharz, R.; Vergari, A.; Stelzner, K.; Molina, A.; Trapp, M.; Shao, X.; Kersting, K.; and Ghahramani, Z. 2019. Random

sum-product networks: A simple and effective approach to probabilistic deep learning. In *UAI*, 334–344.

Peharz, R.; Lang, S.; Vergari, A.; Stelzner, K.; Molina, A.; Trapp, M.; den Broeck, G. V.; Kersting, K.; and Ghahramani, Z. 2020. Einsum networks: Fast and scalable learning of tractable probabilistic circuits. In *ICML*, 7563–7574.

Poon, H., and Domingos, P. M. 2011. Sum-product networks: A new deep architecture. In *UAI*, 337–346.

Richardson, M., and Domingos, P. M. 2006. Markov logic networks. *Machine Learning* 107–136.

Skryagin, A.; Stelzner, K.; Molina, A.; Ventola, F.; and Kersting, K. 2020. Splog: Sum-product logic. In *ProbProg*.

Soininen, T., and Niemelä, I. 1999. Developing a declarative rule language for applications in product configuration. In *PADL*, Lecture Notes in Computer Science, 305–319.

Stammer, W.; Schramowski, P.; and Kersting, K. 2021. Right for the right concept: Revising neuro-symbolic concepts by interacting with their explanations. In *CVPR*, 3619–3629.

Tran, D.; Hoffman, M. D.; Saurous, R. A.; Brevdo, E.; Murphy, K.; and Blei, D. M. 2017. Deep probabilistic programming. In *ICLR*.

Yang, Z.; Ishay, A.; and Lee, J. 2020. NeurASP: Embracing neural networks into answer set programming. In *IJCAI*, 1755–1762.

Yi, K.; Wu, J.; Gan, C.; Torralba, A.; Kohli, P.; and Tenenbaum, J. 2018. Neural-symbolic VQA: disentangling reasoning from vision and language understanding. In *NeurIPS*, 1039–1050.

Zhang, Y.; Hare, J. S.; and Prügel-Bennett, A. 2019. Deep set prediction networks. In *NeurIPS*, 3207–3217.

Supplementary Materials

A – Details on Parameter Learning

In the supplementary materials, we want to discuss details on **parameter learning** in SLASH. Since we use co-ordinate descent for training SLASH we present the derivative of each component of the loss function defined in equation 10 since while optimization, one component has to be kept fixed.

We start with the gradient of the NPP loss function L_{NPP} i.e. the negative log-likelihood, defined in equation 2

$$\begin{aligned}\frac{\partial}{\partial \xi} L_{NPP} &= \frac{\partial}{\partial x_{Q_i}} \cdot \frac{\partial x_{Q_i}}{\partial \xi} L_{NPP} \\ &= \frac{\partial x_{Q_i}}{\partial \xi} \left(-\sum_{i=1}^{n} \frac{\partial}{\partial x_{Q_i}} \log \left(P_{\xi}^{(X_{\mathbf{Q}}, C)}(x_{Q_i}) \right) \right) \\ &= \frac{\partial x_{Q_i}}{\partial \xi} \left(-\sum_{i=1}^{n} \frac{1}{\left(P_{\xi}^{(X_{\mathbf{Q}}, C)}(x_{Q_i}) \right)} \frac{\partial}{\partial x_{Q_i}} \left(P_{\xi}^{(X_{\mathbf{Q}}, C)}(x_{Q_i}) \right) \right)\end{aligned}$$

Here, we remark that $\frac{\partial x_{Q_i}}{\partial \xi}$ will be carried out by back-propagation and the expression after it is the initial gradient.

Next, we derive the gradient of the logical entailment loss function L_{ENT}, as defined in equation 7. One estimates the gradient as follows

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

whereby

- $X_{\mathbf{Q}}$ is the set of random variables associated with the set of the queries $\mathbf{Q}$,
- x_{Q_i} is a training sample, a realization of the set of random variables $X_{\mathbf{Q}}$ associated with the particular query Q_i,
- $P^{(X_{\mathbf{Q}}, C)}(x_{Q_i})$ is the probability of the realization x_{Q_i} estimated by the NPP modelling the joint over the set $X_{\mathbf{Q}}$ and C – the set of classes (the domain of the NPP),
- $\log(P_{\Pi(\theta)}(Q_i))$ – the probability of the query Q_i under the program $\Pi(\theta)$ calculated by SLASH (for the reference see the equation (5)),
- and $\frac{\partial \log(P_{\Pi(\theta)}(Q_i))}{\partial \mathbf{p}}$ is the gradient as defined in Eq.9.

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

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

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 obtain

$$H(y_i, \hat{y}_i) = H \left(\log(P_{\Pi(\theta)}(Q_i)), P^{(X_{\mathbf{Q}}, C)}(x_{Q_i}) \right) = -\sum_{j=1}^{m} \log(P_{\Pi(\theta)}(Q_{ij})) \cdot \log \left(P^{(X_{\mathbf{Q}}, C)}(x_{Q_{ij}}) \right). \tag{11}$$

We remark that m represent the number of classes defined in the domain of an NPP. Now, we differentiate the equation (11) with the respect to p depicted as in Eq. 9 to be the label of the probability of an atom $c = v$ in r^{npp}, denoting $P_{\Pi(\boldsymbol{\theta})}(c = v)$. Since differentiation is linear, the product rule is applicable directly:

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

We do not wish to consider the latter term of $\log(P_{\Pi(\theta)}(Q_i)) \cdot \frac{\partial \log \left(P^{(X_{\mathbf{Q}}, C)}(x_{Q_i}) \right)}{\partial \mathbf{p}}$ because it represents the rescaling and to keep the first since SLASH procure $\frac{\partial \log(P_{\Pi(\theta)}(Q_i))}{\partial \mathbf{p}}$ following Eq. 9. To achieve this, we estimate equation from above downwards as

$$\frac{\partial}{\partial \mathbf{p}} H\left(y_i, \hat{y}_i\right) \geq -\sum_{j=1}^{m} \frac{\partial \log(P_{\Pi(\theta)}(Q_{ij}))}{\partial \mathbf{p}} \cdot \log \left(P^{(X_{\mathbf{Q}}, C)}(x_{Q_{ij}}) \right). \tag{12}$$

Furthermore, let us recall that under i.i.d assumption we obtain from the definition of likelihood

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

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

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

Finally, we obtain the following estimate applying inequality (12)

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

Also, we note that the mathematical transformations listed above hold for any type of NPP and the task dependent queries (NN with Softmax, PC or PC jointly with NN). The only difference will be the second term, i.e., $\log(P^{(C|X_{\mathbf{Q}})}(x_{Q_{ij}}))$ or $\log(P^{(X_{\mathbf{Q}}|C)}(x_{Q_{ij}}))$ depending on the NPP and task. The NPP in a form of a single PC modeling the joint over $X_{\mathbf{Q}}$ and C was depicted to be the example. With that, the derivation of gradients for both loss functions 2 and 7 is complete, and the training is carried out by coordinated descent.

Backpropagation for joint NN and PC NPPs: If within the SLASH program, $\Pi(\boldsymbol{\theta})$, the NPP forwards 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 (8) to

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

Thereby, $\boldsymbol{\gamma}$ is the set of the NN's parameters and $\frac{\partial \boldsymbol{\theta}}{\partial \boldsymbol{\gamma}}$ is computed by the backward propagation through the NN.

B – SLASH Programs

Here, the interested reader will find the SLASH programs which we compiled for our experiments. Figure 5 presents the one for the MNIST Addition task, Figure 7 – for the Generative MNIST Addition task, and Figure 9 – for the set prediction task with slot attention encoder and the subsequent CoGenT test. Note the use of the "+" and "-" notation for indicating whether a random variable is given or being queried for.

```
# 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 5: SLASH Program for MNIST addition. The same program was used for the training with missing data.

```
# Is 7 the sum of the digits in img1 and img2?
:- addition(image_id1, image_id2, 7)
```

Figure 6: Example SLASH Query for MNIST addition. The same type of query was used for the training with missing data

```
# 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 one images and the resulting sum
# Thereby, we add to additional conditions:
## 1. Make the variable N "safe", i.e., one guarantees that there is only
## a countable mount of stable models to be found: N = 0..18
## 2. Insist on the fact that both variables A and B representing images
## are as matter of fact different: A != B
## 3. Since the second image is not available, one has only the prior
## over tha classes P(C) at disposal: digit(-B, -N2)
addition(A, B, N) :- digit(+A, -N1), digit(-B, -N2), N = 0..18, A != B, N - N1 = N2.
```

Figure 7: SLASH Program for the generative MNIST addition.

```
# Is img2 the difference between the sum of the digits and the image encoded in img1?
:- addition(image_id1, _, 2)
```

Figure 8: Example SLASH Query for generative MNIST addition.

```
# 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 9: SLASH Program for ShapeWorld4. The same program was used for the CoGenT experiments.

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

Figure 10: Example SLASH Query for ShapeWorld4 experiments. In other words, this query corresponds to asking SLASH: “Is object 1 a small, bright red circle?”.

C – Experimental Details

ShapeWorld4 Generation

The ShapeWorld4 and ShapeWorld4 CoGenT data sets were generated using the original scripts of (Kuhnle and Copestake 2017) (https://github.com/AlexKuhnle/ShapeWorld). The exact scripts will be added together with the SLASH source code.

Average Precision computation (ShapeWorld4)

For the baseline slot encoder experiments on ShapeWorld4, we measured the average precision score as in (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 an additional binary identifier to the multi-label ground truth vectors, we have added a background (bg) attribute to each category (*cf.* Fig. 9). A slot is thus considered to be empty (i.e., not containing an object) if each NPP returns the maximal conditional probability for the *bg* attribute.
As the ShapeWorld4 prediction task only included discrete object properties both for Slot Attention and for SLASH Attention, the distance threshold for the average precision computation was infinity (thus corresponding to no threshold).

Model Details

For those experiments using NPPs with PC, we have used Einsum Networks (EiNets) for implementing the probabilistic circuits. 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 and Ba 2015) with $\beta 1 = 0.9$ and $\beta 2 = 0.999$, $\epsilon = 1e - 8$ and no weight decay was used.

MNIST-Addition Experiments For the MNIST-Addition experiments, we ran the DeepProbLog and NeurASP programs with their original configurations, as stated in (Manhaeve et al. 2018) and (Yang, Ishay, and Lee 2020), respectively. For the SLASH MNIST-Addition experiments, we have used the same neural module as in DeepProbLog and NeurASP, when training SLASH with the neural NPP (SLASH (DNN)) represented in Tab. 2. When using a PC NPP (SLASH (PC)) we have used an EiNet with the Poon-Domingos (PD) structure (Poon and Domingos 2011) and normal distribution for the leafs. The formal hyperparameters for the EiNet are depicted in Tab. 3.

The learning rate and batch size for the DNN were 0.005 and 100, for DeepProbLog, NeurASP and SLASH (DNN). For the EiNet, these were 0.01 and 100.

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 2: Neural module – LeNet5 for MNIST-Addition experiments.

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

Table 3: Probabilistic Circuit module – EiNet for MNIST-Addition experiments.

ShapeWorld4 Experiments For the baseline slot attention experiments with the ShapeWorld4 data set, we have used the architecture presented in Tab. 4. 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. 4, however, we replaced the final MLPs with 4 individual EiNets with Poon-Domingos structure (Poon and Domingos 2011). Their hyperparameters are represented in Tab. 5.

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

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 4: Baseline slot encoder for ShapeWorld4 experiments.

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 5: Probabilistic Circuit module – EiNet for ShapeWorld4 experiments.

D – Additional Results on Missing Pixel MNIST Addition

	DeepProbLog	**SLASH** (PC)
50%	$\mathbf{79.94 \pm 7.2}$	72.2 ± 12.15
80%	31.6 ± 6.08	$\mathbf{44.2 \pm 8.23}$
90%	16.94 ± 1.76	$\mathbf{29.6 \pm 5.77}$
97%	12.33 ± 0.47	$\mathbf{17.6 \pm 2.97}$

Table 6: Additional MNIST Addition Results. Test accuracy corresponds to the percentage of correctly classified test images. Both models (DeepProbLog and SLASH (PC)) were trained on the full MNIST data, but tested on images with missing pixels. Test accuracies are presented in percent. The amount of missing data was varied between 50% and 97% of the pixels per image.

In addition to the setting considered in Evaluation 2, we adjusted the settings for the MNIST Addition task with missing data into the following way.

Training was performed with the full MNIST data set, however the test data set contained different rates of missing pixels. Whereas using an NPP with a PC allows, among other things, to compute marginalization "out of the box" without requiring an update to the architecture or a retraining, this is not so trivial for purely neural-based predicates as in DeepProbLog. Thus, we allowed SLASH (PC) to marginalize over the missing pixels, where this was not directly possible for DeepProbLog.
The results can be seen in Tab. 6 for 50%, 80%, 90% and 97% of missing pixels per image in the test set. We observe that at 50%, DeepProbLog outperforms SLASH by a small margin. For all other rates, we observe that SLASH (PC) reaches significantly higher test accuracies than DeepProbLog. However, we remark that in this setting, SLASH produces larger standard deviations in comparison to DeepProbLog. These results indicate that the conclusions, drawn in the main part of our work, remain true also in this setting of handling missing data.

E – Computational time Analysis

	RTE (ms)
DeepProbLog	$10m : 9s$
NeurASP	$1m : 24s$
SLASH (PC)	$1m : 15s$
SLASH (DNN)	$\mathbf{0m : 38s}$

Table 7: Run time per epoch analysis for MNIST Addition.

Tab. 7 depicts the average run time training epoch (RTE) in ms per DPPL over the whole training window. Also, here we investigate SLASH both with a PC NPP and a DNN NPP. Through an efficient and batch-wise implementation, SLASH (DNN) greatly surpasses both DeepProbLog and NeurASP in terms of RTE, although all three of them use the same DNN. Notably, SLASH (PC) is also much faster than DeepProbLog and slightly faster than NeurASP, although the PC performs full probabilistic inference. These results were measured on a machine equipped with an AMD Ryzen 7 3800X 8-Core Processor and an Nvidia GeForce RTX 2080 Ti GPU. For SLASH, we were using eight threads for parallel calls upon ASP solver.