Neural-Probabilistic Answer Set Programming

Arseny Skryagin\textsuperscript{1}, Wolfgang Stammer\textsuperscript{1,3}, Daniel Ochs\textsuperscript{1}, Devendra Singh Dhami\textsuperscript{1,3}, Kristian Kersting\textsuperscript{1,2,3}

\textsuperscript{1}Computer Science Department, Technical University Darmstadt, Germany
\textsuperscript{2}Centre for Cognitive Science, TU Darmstadt
\textsuperscript{3}Hessian Center for AI (hessian.AI)

\{arseny.skryagin, wolfgang.stammer, daniel.ochs, devendra.dhami, kersting\}@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 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 symbolic 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 allows for the integration of all forms of probability estima-
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.

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.

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.
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 DPLPL, 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 DPLPLs, 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-Probabilistic Predicates (NPPs).

Formally, we denote with

\[
npp(h(x), [v_1, \ldots, v_n])
\]

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., \( \text{color}_{\text{attr}} \) 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 \( \text{color}_{\text{attr}} \) 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 \( \Pi^{npp} \) 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 \( \Pi^{npp} \), e.g., \( r_{\text{color}_{\text{attr}}} = \{c = \text{Red}, c = \text{Blue}, c = \text{Green}, \ldots\} \) for the example depicted in Fig. 2(right).

Rules of the form \( npp(h(x), [v_1, \ldots, v_n]) \leftarrow Body \) 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, the Body of the rule is identified by \( \top \) (tautology, true) or \( \bot \) (contradiction, false) during grounding. Rules of the form \( \text{Head} \leftarrow Body \) with \( r^{npp} \) appearing in \( \text{Head} \) are prohibited for \( \Pi^{npp} \).

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 \( \text{color}_{\text{attr}}(+X, -C) \) one is asking for \( P(C|X) \). Similarly, with \( \text{color}_{\text{attr}}(-X, +C) \) one is asking for \( P(X|C) \) and,
finally, with $\text{color\_attr}(+X, +C)$ for $P(X, C)$. And, in the case when no data is available, i.e., $\text{color\_attr}(-X, -C)$, 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 can 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,

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

(2)

whereby $LH$ is an abbreviation for likelihood. Further, we assume data to be i.i.d., ground truth $x_i$ to be the all-ones vector, $\xi$ to be the parameters of the NPP and $P_{\xi}^{(X,C)}(x_i)$ 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 $\Pi$ is the union of $\Pi^{\text{asp}}$, $\Pi^{\text{npp}}$. Therewith, $\Pi^{\text{asp}}$ is the set of propositional rules (standard rules from ASP-Core-2 (Calimeri et al. 2020)), and $\Pi^{\text{npp}}$ 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 potential solutions. Thus, we define SLASH’s semantics. For SLASH to translate the program $\Pi$ to the ASP-solver’s compatible form, the rules (Eq. 1) will be rewritten to the set of rules:

$$1 \{ h(x) = v_1; \ldots; h(x) = v_m \} 1$$

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

With $I|_{r^{npp}}$ we denote the projection of the $I$ onto $r^{npp}$, $\text{Num}(I|_{r^{npp}}, \Pi) =$ the number of the potential solutions of the program $\Pi$ agreeing with $I|_{r^{npp}}$ on $r^{npp}$. Because we aim to calculate the success probability of a 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 $\Pi$ as the product of the probabilities of all atoms $c = v$ in $I|_{r^{npp}}$ divided by the number of potential solutions of $\Pi$ agreeing with $I|_{r^{npp}}$ on $r^{npp}$:

$$P_{\Pi}(I) = \left\{ \frac{\prod_{c=v \in I|_{r^{npp}}} P_{\Pi}(c=v)}{\text{Num}(I|_{r^{npp}}, \Pi)}, \quad \text{if } I \text{ is pot. sol. of} \Pi, \quad \text{otherwise.} \right\}$$

(4)

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

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

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

(5)

Thereby, $I \models Q$ reads as “$I$ satisfies $Q$”. The probability of the set of queries $Q = \{Q_1, \ldots, Q_n\}$ is defined as the product of the probability of each. i.e.

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

(6)

**Parameter Learning in SLASH** We denote with $\Pi(\theta)$ the SLASH program under consideration, where $\theta$ is the set of the parameters associated with $\Pi$. Further, making the i.i.d. assumption of the query set $Q$, 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 $\Pi(\theta)$. The goal is to learn the parameters $\theta$ of the program $\Pi(\theta)$ 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 $P^{(XQ,C)}(x_{Q_i})$, assuming the NPPs are fixed:

$$L_{\text{ENT}} := -\log \left( \log (P_{\Pi(\theta)}(Q)), P^{(XQ,C)}(x_{Q_i}) \right) = -\sum_{j=1}^{m} \log (P_{\Pi(\theta)}(Q_{ij})) \cdot \log \left( P^{(XQ,C)}(x_{Q_{ij}}) \right).$$

(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., $P^{(C|XQ)}(x_Q)$ or $P^{(XQ,C)}(x_Q)$ depending
on the NPP and task. Furthermore, we assume that for the
set of queries \( Q \) it holds that,

\[
P_{\Pi(\theta)}(Q) > 0 \quad \forall Q \in Q.
\]

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 \( r_{pp} \) (i.e., the set of logical
rules denoting NPPs; see Def. 2) as \( p \), one can compute their
gradients w.r.t. \( \theta \) via backpropagation as

\[
\sum_{Q \in Q} \frac{\partial \log \left( P_{\Pi(\theta)}(Q) \right)}{\partial \theta} = \sum_{Q \in Q} \frac{\partial \log \left( P_{\Pi(\theta)}(Q) \right)}{\partial p} \times \frac{\partial p}{\partial \theta}.
\]

The term \( \frac{\partial p}{\partial \theta} \) can now be computed as usual via backward
propagation through the NPPs (see Eq.11 in the supplementary
materials (Suppl.) A for details). By letting \( \theta \) to be the
label of the probability of an atom \( c = v \) in \( r_{pp} \) and de-
noting \( P_{\Pi(\theta)}(c = v) \), the term \( \frac{\partial \log \left( P_{\Pi(\theta)}(Q) \right)}{\partial p} \) follows from
NeurASP (Yang, Ishay, and Lee 2020) as

\[
\frac{\partial \log \left( P_{\Pi(\theta)}(Q) \right)}{\partial p} = \frac{\sum_{I : I \in Q} P_{\Pi(\theta)}(I) \cdot \log \left( \frac{\sum_{I : I \in Q} P_{\Pi(\theta)}(I)}{\sum_{I : I \in Q} P_{\Pi(\theta)}(I)} \right)}{\sum_{I : I \in Q} P_{\Pi(\theta)}(I)}.
\]

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

\[
L_{SLASH} = L_{NPP} + L_{ENT}
\]

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 fol-
lowing, 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 entail-
ment loss function \( L_{ENT} \) defined in Eq. 7 can be estimated
as follows

\[
\frac{1}{n} \frac{\partial}{\partial p} L_{ENT} \geq - \frac{1}{n} \sum_{i=1}^{n} \frac{\partial \log \left( P_{\Pi(\theta)}(Q_i) \right)}{\partial p} \cdot \log \left( P^{(X_Q,C)}(x_{Q_i}) \right).
\]

**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

\[
L_{ENT} = - \sum_{i=1}^{n} \log \left( P_{\Pi(\theta)}(Q_i) \right) \cdot H(y_i, \hat{y}_i)
\]

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

\[
\frac{\partial}{\partial p} H(y_i, \hat{y}_i) =
\]

\[
- \sum_{j=1}^{m} \left[ \frac{\partial \log \left( P_{\Pi(\theta)}(Q_{ij}) \right)}{\partial p} \cdot \log \left( P^{(X_Q,C)}(x_{Q_{ij}}) \right) + \log \left( P_{\Pi(\theta)}(Q_{ij}) \right) \cdot \frac{\partial \log \left( P^{(X_Q,C)}(x_{Q_{ij}}) \right)}{\partial p} \right],
\]

is a constant and assembling all intermediate steps, we ob-
tain the desired inequality. \( \Box \)

Similarly one can derive the derivative of the NPP loss
function (Eq. 2) as:

\[
\frac{\partial}{\partial \xi} L_{NPP} =
\]

\[
\frac{\partial x_{Q_i}}{\partial \xi} \left( - \sum_{i=1}^{n} \frac{1}{P^{(X_Q,C)}(x_{Q_i})} \frac{\partial}{\partial x_{Q_i}} \left( P^{(X_Q,C)}(x_{Q_i}) \right) \right).
\]

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 individ-
ual 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 Evaluations

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 (Le-
Cun, 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 de-
VIation over five runs with different random seeds for param-
eter 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 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 the 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 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 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 no previous DPPLs can handle.

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 samples is improvable, and hypothesize that employing two different optimizers for each step of coordinate descent surely does not have an optimal effect on the PC NPP. 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 constraints 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 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. Consequently, 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 \( \text{LP}^{\text{MLN}} \) (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 Truszczyński 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, theoretically, 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 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 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 multiclassical 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


