Approximate Inference for Neural Probabilistic Logic Programming

Robin Manhaeve[1] , **Giuseppe Marra**[1] , **Luc De Raedt**[1,2]
[1]KU Leuven, Dept. of Computer Science; Leuven.AI
[2]Örebro University
{robin.manhaeve, giuseppe.marra,luc.deraedt}@cs.kuleuven.be

Abstract

DeepProbLog is a neural-symbolic framework that integrates probabilistic logic programming and neural networks. It is realized by providing an interface between the probabilistic logic and the neural networks. Inference in probabilistic neural symbolic methods is hard, since it combines logical theorem proving with probabilistic inference and neural network evaluation. In this work, we make the inference more efficient by extending an approximate inference algorithm from the field of statistical-relational AI. Instead of considering all possible proofs for a certain query, the system searches for the best proof. However, training a DeepProbLog model using approximate inference introduces additional challenges, as the best proof is unknown at the start of training which can lead to convergence towards a local optimum. To be able to apply DeepProbLog on larger tasks, we propose: 1) a method for approximate inference using an A*-like search, called DPLA* 2) an exploration strategy for proving in a neural-symbolic setting, and 3) a parametric heuristic to guide the proof search. We empirically evaluate the performance and scalability of the new approach, and also compare the resulting approach to other neural-symbolic systems. The experiments show that DPLA* achieves a speed up of up to 2-3 orders of magnitude in some cases.

1 Introduction

There has been a recent surge of interest in neural symbolic computation, the integration of neural networks with symbolic reasoning, cf. the many special tracks and debates on this topic at major conferences. Many approaches to neural symbolic computation now exist, cf. recent surveys (Besold et al., 2017; Garcez et al., 2019; De Raedt et al., 2020). While most approaches focus on pushing the symbols and the knowledge inside the neural network, others have argued for an interface layer between the neural and the symbolic side (Manhaeve et al., 2018). One such interface is built between the probabilistic logic programming language ProbLog and neural networks in DeepProbLog. ProbLog (Fierens et al., 2015) belongs to the statistical relational artificial intelligence paradigm and combines theorem proving with knowledge compilation to perform inference and learning. It is well known that probabilistic logic inference is computationally hard. While the statistical relational AI community has contributed many approximate inference techniques, so far only exact inference has been used for DeepProbLog.

We contribute the first approximate inference technique for DeepProbLog, called *DPLA**. It applies ideas from A* to the SLD theorem proving of DeepProbLog to find a small set of best proofs, which are then compiled into a circuit for learning and inference. Which proofs are best depends on the likelihood of these proofs, which in turn depends on the parameters of the program. An important challenge is to determine the best proofs while learning the parameters themselves, as any change in parameters will have an effect on the likelihood of the proofs. This challenge is particularly important when neural networks come into the picture as neural networks can perform better when their parameters are suitably initialized (cf. the success of curriculum learning). By using an A*-like approach, we are able to simultaneously improve upon the heuristic that is used in theorem proving and the parameters of the DeepProbLog model.

Our contributions are threefold: 1) we introduce an approximate inference technique for DeepProbLog called DPLA* that achives a speed-up over several magnitudes over exact inference, 2) the application of curriculum learning and exploration to overcome the issues faced when learning with approximate inference, and 3) a parametric heuristic for guiding the proving process of the approximate inference.

2 ProbLog and DeepProbLog

Prolog Prolog is a logic programming language based on definite clauses. These are expressions of the form $h \leftarrow b_1, ..., b_n$ where h and the b_i are logical atoms[1]. This clause states that h is true whenever all b_i are true. When $n = 0$, the clause is a fact. A substitution θ is an expression of the form $\{V_1 = t_1, ..., V_n = t_n\}$ where the V_i are different variables and the t_i terms. Applying a substitution θ to an expression e (term or clause) yields the instantiated expression $e\theta$ where all variables V_i in e have been replaced by their corresponding terms t_i in e. For example, applying the substitution $\theta = \{X = an, Y = bob\}$ to the term $parent(X, Y)$ results in the term $parent(an, bob)$. Prolog's SLD-resolution theorem prover can be used to decide whether a ground goal, i.e., a conjunction of atoms $g_1 \wedge ... \wedge g_n$ is logically entailed by the

[1]A logical atom $a(t_1, ..., t_n)$ consists of a predicate a of arity n followed by n terms t_i. Terms then are either constants, logical variables or structured terms of the form $f(t_1, ..., t_k)$ with f a functor and the t_j terms. A ground term is a term without variables.

program. Briefly put, SLD resolution works by the repeated application of clauses to a goal. If we have a goal $g_1 \wedge ... \wedge g_n$ and a clause $h \leftarrow b_1 \wedge ... \wedge b_m$ such that h unifies with g_1 with substitution θ (i.e. the two terms can be made identical by substituting certain variables in the terms), SLD-resolution derives the new goal $(b_1 \wedge ... \wedge b_m \wedge g_2 \wedge ... \wedge g_n)\theta$. This is repeated until an empty goal is achieved, or no more rules can be applied. For more detail on this, we refer to standard works on logic programming (Flach, 1994).

ProbLog extends Prolog by introducing probabilistic facts of the form $p :: atom$, where p is a probability and $atom$ is a logical atom. A probabilistic fact is true with probability p. If the atom contains variables, then all its ground instances $atom\theta$ are true with probability p.[2] Furthermore, all the probabilistic facts are assumed to be independent from one another. ProbLog's inference mechanism computes the probability of a goal g as

$$P(g) = \sum_{M \models g} \prod_{f \in M} p_f \prod_{f \notin M} (1 - p_f)$$

For ease of modeling, ProbLog also allows the use of annotated disjunctions (ADs)

$$p_1 :: h_1; ...; p_n :: h_n : -b_1, ...b_k$$

with with $\sum_i p_i \leq 1$, which states that whenever the condition part of the rule is true, one of the h_i will be true according to the probabilities p_i. If $\sum_i p_i < 1$, then the probability that none of the h_i becomes true is equal to $1 - \sum_i p_i$.

DeepProbLog extends ProbLog by the addition of the neural predicate. The neural predicate is defined by means of a neural AD, i.e., an expression of the form $nn(mr, I, O, D) :: r(I, O)$, where mr is a neural network taking I as input and computing O as output. Furthermore, the range of possible outputs is specified in the (discrete) domain $D = \{d_1, ..., d_k\}$. Given specific inputs i and o, the probability of the neural predicate $r(i, o)$ is determined as

$$nn(mr, i, d_1) :: r(i, d_1); ...; nn(mr, i, d_k) :: r(i, d_k)$$

where $nn(mr, i, d_j)$ represents the probability that the neural network outputs d_j on input i. The semantics of DeepProbLog reduces to that of ProbLog (but see (Manhaeve et al., 2021)).

Example 1 (MNIST Addition). Consider the predicate `addition(X,Y,Z)`, where `X` and `Y` are images of handwritten digits from the MNIST dataset (LeCun et al., 1998), and `Z` is the natural number corresponding to the sum of these digits. DeepProbLog can make a probabilistic estimate on the validity of, for example, `addition(`  `,`  `,1)` using the program:

```
nn(m,[I],N,[0,...,9]) :: digit(I,N).
addition(I1,I2,R) :-
   digit(I1,N1), digit(I2,N2),
   R is N1 + N2.
```

The first line is a neural predicate that defines the classification of MNIST images. The subsequent lines define the addition. The line `R is N1 + N2` in essence defines a constraint on the outputs of the neural network.

Inference in ProbLog and DeepProbLog Inference in ProbLog is the process of calculating the probability of a query $P(q)$. To calculate this, we have to find the proofs for the query q, and calculate the probability of the disjunction of these proofs. Previously, DeepProbLog only calculated the *success probability* of a query that is, the probability of all the disjunction of all proofs. In this work, we will consider calculating the *explanation probability* of a query, namely the highest probability of any proof for the query. Although ProbLog inference is usually described with respect to grounding, it can also be described with respect to SLD-resolution as done in the original paper (De Raedt, Kimmig, and Toivonen, 2007).

Inference for computing the probability $P(q)$ for both ProbLog and DeepProbLog happens in several steps.

1. **Proving step:** During the proving procedure, the set of all proofs $S(q)$ for the query q is computed. The proofs are computed using SLD resolution. Without negation, each proof can be considered a conjunction of probabilistic and neural facts.
2. **Rewrite step:** The proofs are turned into a logical expression $\bigvee_{s \in S(g)} \bigwedge_{p_f \in s} p_f$ where p_f is a probabilistic fact or neural fact used in proof s.
3. **Knowledge compilation:** Computing $P(q)$ directly on this formula is not efficient. That's why techniques from knowledge compilation (Darwiche and Marquis, 2002) are used, which turns the formula into a structure that allows for efficient evaluation.
4. **Arithmetic circuit evaluation:** The compiled structure can be trivially turned into an arithmetic circuit where the leafs (probabilistic and neural facts) are connected using additions and multiplications. $P(q)$ is then computed by evaluating the neural networks to calculate the probability of the neural facts, and a subsequent bottom-up evaluation of the arithmetic circuit.

Example 2 (MNIST Addition). We evaluate $P($`addition(``,``,1)`$)$ from Example 1. The SLD-tree for this query is shown in Figure 1. The ground logical expression produced in the second step is (`digit(``,0)` $\wedge$ `digit(``,1)`$)$ $\vee$ (`digit(``,1)` $\wedge$ `digit(``,0)`$)$. The arithmetic circuit produced after knowledge compilation is shown in Figure 2.

[2] A substitution $\theta = \{V_1 = t_1, ..., V_n = t_n\}$ is an expression where the V_i are logical variables and the t_i are terms. Applying a substitution θ to an expression e yields the expression $e\theta$ where all V_i have been simultaneously replaced by the t_i.

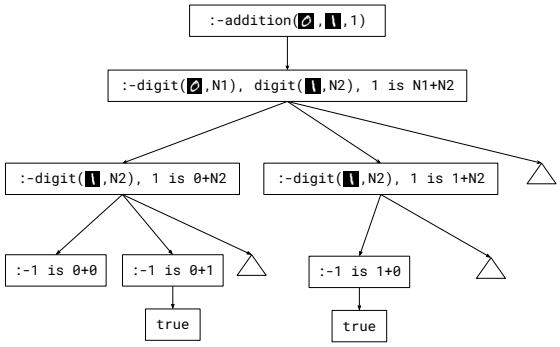


Figure 1: The SLD-tree for query addition(0, 1, 1). Triangles represent branches omitted for brevity.

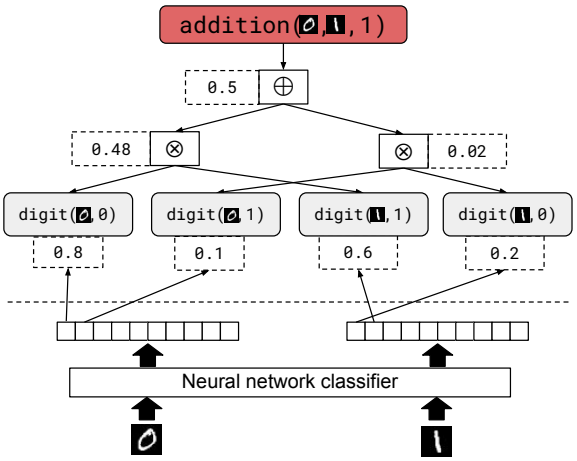


Figure 2: The arithmetic circuit for query addition(0, 1, 1). Figure adapted from Manhaeve et al. (2021).

Learning in ProbLog and DeepProbLog Learning in DeepProblog follows the learning from entailment setting(Frazier and Pitt, 1993). Given a DeepProbLog program with parameters Θ, a set $\mathcal{Q}$ of tuples $(q, \mathcal{X}, p)$ with q a query, $\mathcal{X}$ the neural input for this query and p its desired success probability, and a loss function $\mathcal{L}$, learning from entailment computes

$$\text{argmin}_\Theta \frac{1}{|\mathcal{Q}|} \sum_{(q,\mathcal{X},p)\in\mathcal{Q}} \mathcal{L}(P(q|\mathcal{X},\Theta), p)$$

ProbLog and DeepProbLog can estimate the parameters of the probabilistic facts, the neural predicates and the underlying neural networks using gradient descent. This happens in two steps. In the first step, the gradient is propagated to the leaves of the logical circuit, and in the second step these gradients are propagated to the weights in the neural network[3]. In this work, DeepProbLog uses cross-entropy as the loss function.

Note that, for most tasks, the supervision is not on the level of the neural predicate, so the neural networks cannot be directly trained. In the case of the MNIST addition, there are only supervised examples for the addition, and not for the individual digits. These neural predicates however are often classifiers themselves, and it will be useful for the remainder of this work to consider the *neural predicate accuracy*. This is the accuracy of the neural predicate alone, measured on a relevant dataset (e.g., for the MNIST addition task, neural predicate accuracy is defined on the MNIST test dataset).

[3]Technically this is done using semi-rings and algebraic ProbLog, see (Kimmig, Van den Broeck, and De Raedt, 2011).

3 Approximate Inference

There are essentially two computational problems in (Deep)ProbLog inference. First, one needs to compute the set of all ground proofs, and secondly, one needs to compile the resulting ground logical formula into an arithmetic circuit. It is well-known that for some domains the number of possible ground proofs can explode, and also that knowledge compilation is computationally hard (Darwiche and Marquis, 2002). This has inspired various approximate inference techniques for probabilistic logic programming. Especially relevant are the techniques that do not compute the full set of proofs but focus on a subset of the proofs (Renkens, Van den Broeck, and Nijssen, 2012; De Raedt, Kimmig, and Toivonen, 2007), e.g., the k-best proofs. If fewer proofs are used then compilation also becomes simplified and faster. In this work, we will optimize the parameters in the model with only the highest probability proof (cf. Viterbi training (Jelinek, 1976)), which is equivalent to optimizing the explanation probability instead of the success probability for our query. The method discussed here will only be used to find the best proof, but it also capable of finding the k best proofs, or even all proofs. In this work, we will only consider programs without negation. Below, we introduce our approximate inference method for DeepProbLog, called **DPLA***.

3.1 Heuristically Searching the SLD-Tree

The core component of DPLA* is an A*-based search in the SLD tree for the best proof. The standard SLD-resolution procedure used in Prolog and ProbLog searches the tree depth-first, left-to-right. An SLD-tree is illustrated in Figure 1 for the program from Example 1. Each node in this SLD tree represents a *partial* proof. The path from the root to the node captures what has already been proven, and the node itself contains the remaining part of the goal that still needs to be proven. As we are only interested in the probabilistic and neural facts in the proof, we define partial proofs as follows:

Definition 3.1. A partial proof is a pair $\mathcal{E} = (\mathcal{E}^f, \mathcal{E}^g)$. $\mathcal{E}^f$ is the conjunction of ground probabilistic or neural facts that have already been used in the proof, and $\mathcal{E}^g$ denotes the goal that still needs to be proven.

If $\mathcal{E}^g$ contains no literals, then $\mathcal{E}$ represents a complete proof and contains the set of ground probabilistic or neural facts used in the proof. A query q corresponds to a partial proof $\mathcal{E} = (\emptyset, q)$. The probability of a partial proof is defined as

$$P(\mathcal{E}) = P(\bigwedge_{f\in\mathcal{E}^f} f \bigwedge_{g\in\mathcal{E}^g} g)$$

For example, a partial proof encountered while proving the query addition(0, 1, 1) in Example 1 could

be $\mathcal{E} = (\{\texttt{digit}(\texttt{}, 0)\}, \{\texttt{digit}(\texttt{  }, \texttt{N2}), \texttt{1 is 0} + \texttt{N2}\})$. This means that we have already derived that the first image represents the number 0 and we still need to find which number N2 is represented by the second image so that the two numbers sum to 1.

A* Search The heuristic that we will use is an estimate of of the probability of a partial proof $\mathcal{E}$. This can be factorized as

$$P(\mathcal{E}) = \left(\prod_{f \in \mathcal{E}^f} P(f)\right) P(\bigwedge_{g \in \mathcal{E}^g} g \mid \bigwedge_{f \in \mathcal{E}^f} f)$$

By taking the negative logarithm of this we get the formulation for the value function as used in A*, namely

$$\text{f-value}(\mathcal{E}) = \text{cost}(\mathcal{E}) + h(\mathcal{E})$$

with:

$$\text{cost}(\mathcal{E}) = -\log\left(\prod_{f \in \mathcal{E}^f} P(f)\right)$$

$$h(\mathcal{E}) = -\log H(\mathcal{E}) \approx -\log P(\bigwedge_{g \in \mathcal{E}^g} g | \bigwedge_{f \in \mathcal{E}^f} f)$$

It will be convenient to define $h(\mathcal{E})$ as $-\log H(\mathcal{E})$. To have an admissible heuristic, $h(\mathcal{E})$ should never overestimate $-\log P(\bigwedge_{g \in \mathcal{E}^g} g | \bigwedge_{f \in \mathcal{E}^f} f)$, or equivalently $H(\mathcal{E})$ should never underestimate $P(\bigwedge_{g \in \mathcal{E}^g} g | \bigwedge_{f \in \mathcal{E}^f} f)$. If A* uses an admissible heuristic, it is guaranteed to find the best result, which is the proof with the highest probability. It's easy to see that $H(\mathcal{E}) = 1$ is an admissible heuristic. If the heuristic, is not admissible, than we do not have this guarantee, and in general will only be able to approximate the explanation probability. (Pearl, 1984) shows that if the overestimation of the heuristic is bounded $h(\mathcal{E}) - \left[-\log P(\bigwedge_{g \in \mathcal{E}^g} g | \bigwedge_{f \in \mathcal{E}^f} f)\right] \leq \epsilon$, then the difference between the result and the optimal solution is also bounded: $[-\log P(\mathcal{E})] - [-\log P(\mathcal{E}^*)] \leq \epsilon'$, with $P(\mathcal{E}^*)$ the explanation probability and $P(\mathcal{E})$ the probability of the proof found by the search. Rewriting this from negative log probabilities to probabilities, for $\epsilon = e^{-\epsilon'}$, we have: $P(\mathcal{E}) \geq \epsilon P(\mathcal{E}^*)$.

3.2 Constant Heuristic

We define the *constant heuristic* as $H_c(\mathcal{E}) = c$. If $c = 1$, then the search becomes uniform-cost search (UCS), and it will always find the optimal solution, but it might traverse more nodes than a better informed heuristic. Uniform-cost search is equivalent to the k-best method as explored in previous work for approximate inference in ProbLog (Renkens, Van den Broeck, and Nijssen, 2012; De Raedt, Kimmig, and Toivonen, 2007).

3.3 Geometric Mean Heuristic

We consider an improvement over the constant heuristic for a specific case that occurs often in the neural-symbolic setting, namely that when the neural networks is completely correct and confident, the success probability is equal to the explanation probability. This is the case when there is only one correct proof, and all other proofs have a 0 probability. In many neural symbolic tasks, there is only one correct proof for each query. This is clear in the example of the MNIST addition. There are 9 proof for the query $\texttt{addition}(\texttt{}, \texttt{}, 9)$, but only the proof $\texttt{digit}(\texttt{}, 4) \wedge \texttt{digit}(\texttt{}, 5)$ is correct and should carry all of the probability mass, while all other proofs have a probability of zero. The same reasoning holds for many other neural-symbolic tasks. To define this heuristic, we re-write $P(\mathcal{E}) = \prod_{f \in \mathcal{E}^{f*}} P(f) = {\mu^*}^N$, where $\mathcal{E}^{f*}$ is the set facts in the complete proof, μ^* the geometric mean of their probabilities and $N = |\mathcal{E}^{f*}|$. When $n = |\mathcal{E}^f|$, we can write $P(\mathcal{E}) = {\mu^*}^n {\mu^*}^{N-n}$. Let us assume that the geometric mean μ^* of the probabilities of the facts in $\mathcal{E}^{f*}$ is close to the geometric mean μ in $\mathcal{E}^f$. This assumption holds when $P(f) \approx \mu$ for all $f \in \mathcal{E}^{f*}$, or, equivalently, when the variance of the log probabilities of all the facts in the proof $\mathcal{E}^f$ is small. The geometric mean is chosen over the arithmetic mean, as the probabilities of individual facts are not summed, but multiplied. Under this assumption, we define the *geometric mean heuristic* as $H_g(\mathcal{E}) = \mu^{N-n}$. Note that the f-value for each node is $\mu^n \mu^{N-n} = \mu^N$. If we assume N is the same for all proofs we do not have to know N. We can replace the f-value by μ as x^N is monotonically increasing in $[0, 1]$.

4 Approximate Learning

Learning with approximate inference in the neural symbolic setting holds additional challenges. The main challenge is that DPLA* only uses the best proof, but the selection of the best proof is based on the probabilities of facts and neural predicates in the program. At the start of learning, these probabilities are not correct, so the danger is that the selected proof is not correct. This prevents DPLA* from correctly training the neural networks. There are two ways to overcome this issue. The first is to use curriculum learning to make sure that the accuracy of the neural predicates is better than random. This implies that the correct proofs will be selected more often than the wrong proofs, which allows the neural predicates to be trained. The second way to do this is to incorporate a form of exploration. If diversity in the proofs is not enforced, depending on the initialization, DPLA* can get stuck in a local optimum as no correct proofs are selected, preventing the neural predicates from being trained. Exploration can make sure that, regardless of the initialization, the proofs are sufficiently diverse so that he neural networks are able to identify the patterns only present in the correct proofs and are able to learn from these.

4.1 Curriculum Learning

Due to the flexibility of the DeepProbLog framework, curriculum learning is a very natural setting. Consider the multi-digit MNIST addition experiment. Training on long sequences of digits is hard, as the number of proofs grows rapidly. This means that exact inference quickly becomes infeasible. Approximate inference without any pre-initialization would also struggle as it is hard to determine the correct proof for the

given sum. One solution is to incorporate a few examples of shorter sequences. Alternatively, one can provide a few labeled examples for the neural predicate. Because of the flexibility of DeepProbLog as a programming language, no changes to the model have to be made to allow for this form of curriculum learning. For example, in the addition program, one could directly maximize the probability of a few `digit` facts, like `digit(`  `, 1)` or `digit(`  `, 6)`. While in general it is not hard to collect a few labeled examples, they are valuable to initialize the neural predicates to guide the approximate inference.

4.2 Exploration

To include exploration in the algorithm, we borrow an idea from reinforcement learning, namely the upper confidence bound (UCB) (Lai and Robbins, 1985), which has been used in techniques such as Monte Carlo tree search. UCB assumes rewards in the interval $[0, 1]$, so it is appropriate for probabilities. In UCB, actions are selected based on their observed mean reward and an additional exploration term $\sqrt{\frac{\log(p)}{2N_a}}$ where p is an additional parameter controlling the amount of exploration (often taken to be t^{-4}), t is the total number of actions taken and N_a is the number of times action a was selected. The problem with learning using approximate inference is that when neural networks are initialized with a slight preference for one output, the proof containing that output will be selected more often. Due to the optimization, the neural network will be trained towards this output even more, which may lead to converging to a local optimum. To encourage exploration in the proof selection, we augment $cost(\mathcal{E})$ with an additional UCB term for each neural predicate:

$$\text{cost}(\mathcal{E}) = -\log\left(\prod_{f\in\mathcal{E}^f} P(f) + P_{\text{UCB}}(f) - P(f)P_{\text{UCB}}(f)\right)$$

where $P_{\text{UCB}}(f) = min(1, \sqrt{\frac{2\log(N_r)}{N_r^j}})$ if $f = r(i_1, ..., i_n)$ is a neural predicate, and 0 otherwise. Here, N_r^j is the number of times output j for neural predicate r was selected during the proving throughout the training process and N_r is the sum of all such N_r^j. As we treat these quantities as probabilities, we use the general disjunction rule instead of addition.

5 Parametric Heuristics

The goal of the parametric heuristic is to predict the probability of certain goals before they are proven. With the right predictions, the proving will be guided towards the correct proofs. This will not only speed up inference, but will also make training more efficient as fewer wrong proofs will be considered. This idea is similar to that of neurally-guided theorem proving (Wang et al., 2017; Rawson and Reger, 2019).

To implement a parametric heuristic, we assume that the probability of a goal is independent of other goals and the probabilistic facts in the partial proof. This allows us to factorize the probability of the partial proof as

$$P(\mathcal{E}) = \prod_{f\in\mathcal{E}^f} P(f) \prod_{g\in\mathcal{E}^g} P(g)$$

. Under this assumption, we can also factorize the heuristic $H(\mathcal{E}) = \prod_{g\in\mathcal{E}^g} H(g)$. This factorization allows us to estimate $P(g)$ for every goal separately. If this assumption does not hold, then $\prod_{g\in\mathcal{E}^g} P(g) < P(\bigwedge_{g\in\mathcal{E}^g} g \mid \bigwedge_{f\in\mathcal{E}^f} f)$ and the heuristic will be a worse underestimation. The goals for which we should calculate such a heuristic is problem dependent, which is why we define a subset $\mathcal{R}$ of all the predicates in the program for which a heuristic should be estimated. We define the heuristic as $H_n(r(i_1, ..., i_n)) = f_r(i_1, ..., i_n)$ if $r \in \mathcal{R}$, and 1 otherwise. f_r is an external implementation provided to the framework (e.g. a neural network).

Training the Parametric Heuristic The advantage of using a parametric heuristic is that its parameters can be trained to fit the properties of a specific task. Training parametric heuristics is a common technique in neurally-guided theorem proving, where the heuristic is pre-trained on data of known proofs. However, in this setting, the proofs found in the training procedure cannot be used as supervision, as these can be wrong or may not yield the correct probability due to the inaccuracy of neural predicates. To overcome this issue we use a different approach. In order to train the parametric heuristic in parallel with the neural predicates, we optimize in parallel an equivalent DeepProbLog program where all definitions of predicate $r(i_1, ..., i_n)$ are replaced by a neural predicate representing the heuristic. The idea here is that we train the heuristic as shortcuts in the proving procedure. While the heuristic needs to solve a harder, higher-level task, it does not have to be perfectly trained as it will be used only as a heuristic during the actual search and doesn't affect the probability of the selected proof.

Example 3 (Neural Heuristics). Consider the following program. It represents a multi-instance setting of the MNIST addition example.

```
nn(m,[I],N,[0,...,9]) :: digit(I,N).
addition(I1,I2,R) :-
   digit(I1,N1), digit(I2,N2),
   R is N1+N2.
mil_addition(Bag,R) :-
   member([I1,I2],Bag),
   addition(I1,I2,R).
```

`mil_addition([[`  `, `  `], [`  `, `  `]], 7)` is true if any of the pairs in the outer list sums to 7. We could use a heuristic on the `addition` predicate that can estimate if the sum holds for a given input pair. This will be able to guide the heuristic towards the pair in the bag that are most likely to have the correct sum. To train the neural heuristic, we train the following DeepProbLog program instead:

```
nn(h,[I1,I2],R,[0,...,18]) :: h_addition
    (I1,I2,R).
mil_addition(Bag,R) :-
   member([I1,I2],Bag),
   h_addition(I1,I2,R).
```

6 Experiments

The goal of our experiments is to investigate whether DPLA* can be applied to tasks that are intractable for exact inference in DeepProbLog. We will compare the DPLA* with different heuristics to DeepProbLog with exact inference and other neural-symbolic frameworks. Our questions are grouped in three topics: approximate inference, approximate learning and parametric heuristics.

First, we will look at how approximate inference, the heuristics and neural predicate accuracy interact. We hypothesize that approximate inference will outperform exact inference, but that its performance depends on the heuristic and the neural predicate accuracy. We will verify that by answering the following questions.

Q1.1 How does approximate inference compare to exact inference and related frameworks in terms of speed?

Q1.2 How do the different heuristics compare to each other?

Q1.3 What is the impact of neural predicate accuracy on the inference speed of the approximate inference?

Performing learning with approximate inference has potential complications. The proving process is guided by the probabilities which are the output of the neural networks, which are themselves being trained. This can lead to issues where the correct proofs are not selected and the system converges to a local optimum. We answer the following questions.

Q2.1 What is the influence of approximate inference on the accuracy and convergence of the learning?

Q2.2 How does curriculum learning impact the performance of the different heuristics?

Q2.3 What is the impact of exploration on the learning process?

Finally, we will also investigate how a parametric heuristic influences inference and learning. We answer the following questions.

Q3.1 Can parametric heuristics be used to guide the proving process?

Q3.2 Can parametric heuristics be learned in conjunction with the neural predicates?

6.1 Tasks

In order to answer these questions, we will use four datasets, each of them coupled with a different neural symbolic task. A common feature of all the tasks is that they require to symbolically reason on sub-symbolic representations (either images or natural language sentences).

MNIST Addition The MNIST addition example has been introduced in the original DeepProbLog paper (Manhaeve et al., 2018). The task is to predict the sum of two integer numbers represented as two sequences of MNIST images, e.g. $[\boxed{3}, \boxed{4}] + [\boxed{1}, \boxed{5}] = 49$. During training, we are not provided with any direct supervision for the images but, only the sum of the numbers they represent. The dataset is composed of 60000 images and $\frac{60000}{2N}$ pairs. Each pair contains two equally long sequences of length N. For this task, N varies from 1 to 5.

Multi-Instance Learning MNIST Addition Here, we introduce a new task that is an extension of the *MNIST Addition* task. The difference is that it is extended to a multi-instance learning task. Several pairs are combined in a bag, and the bag is labeled with a sum randomly selected out of all the pairs in the bag. Example: $(\boxed{3} + \boxed{4} = 7) \vee (\boxed{1} + \boxed{5} = 7)$. The goal of this task is being able to train the neural predicates to recognize MNIST digits using the weak training signal provided by the multi-instance learning setting. This task is interesting as it investigates the multi-instance learning setting, but also because large parts of the proof space are irrelevant as only a few items from each bag are interesting. A well-informed proving algorithm should be able to leverage this sparsity and avoid unnecessary work on proving and neural network evaluation.

Hand-written Formulas (HWF) The Handwritten Formula (HWF) dataset was introduced in Li et al. (2020). The task is to predict the result of an expression represented as a sequence of images. Each image represents either a digit or an operator($+, -, \times, \div$). An example of an expression is:  . As for the addition, no supervision is provided on the single images, but only the result of the expression is provided. We do not consider the curriculum learning setting for this task (i.e., we train and test only on expressions of exactly length N)

CLUTRR The CLUTRR dataset was proposed in (Sinha et al., 2019). Each example is a natural language sentence describing a kinship graph of variable size. The goal is to deduce a family relation between two nodes of the graph. The query relation is not directly mentioned in the text but can be deduced by chaining relations mentioned in the text. The setting we consider here is slightly different than the original. Instead, we consider the setting as used in Manhaeve et al. (2021), where the rules describing family relations are specified as background knowledge, and the task is to train neural networks to extract the relations from the natural language sentences.

6.2 Baselines

We compare the proposed approach with two neural-symbolic systems.

NeurASP: NeurASP (Yang, Ishay, and Lee, 2020) inherits neural predicates from DeepProbLog but they are used to extend Answer Set Programming instead of ProbLog. While being an approach to exact inference in probabilistic logic programming, NeurASP was showed to be a more efficient implementation than the original DeepProbLog paper in some tasks.

NGS: Neural-Grammar-Symbolic (NGS) (Li et al., 2020) combines grammar parsing with neural networks at the level of terminal symbols, which resembles the interface layer implemented by DeepProbLog using neural predicates. However, the solution to the neural symbolic integration is quite different. Instead of relying on a probabilistic semantics of the logic, NGS provides a backward (i.e. correction) module for propagating the error through the symbolic grammar. By being tailored to the specific task, the backward module can be very efficient. This method is less general than DeepProbLog, where the correction is automatically computed

given any possible logic program.

6.3 Results

To concisely describe the results, we will use the following abbreviated terms. *Query time*: The time it takes for a method to perform the proving for a single query. For exact methods, this involves all proofs, and for DPLA*, this involves a single proof (unless otherwise specified). *Neural predicate accuracy*: The accuracy of the neural predicate measured on the relevant dataset. For tasks based on MNIST images, this is the test set, unless specified otherwise. For all graphs, a line represents the median, and the shaded area spans between the first and third quartiles. For all tables, we report the mean and standard deviation. All experiments were run 5 times.[4]

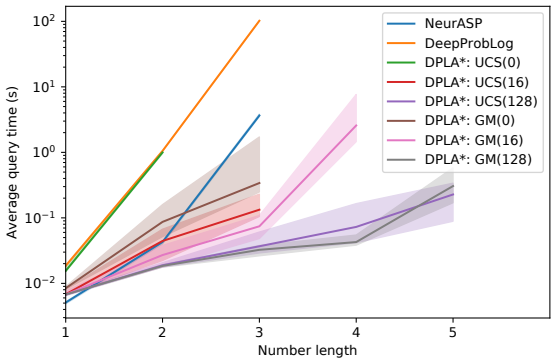
Q1: Scaling Inference For this first question, we are only interested in the scalability of inference, and we do not consider the learning aspect. The aim of this experiment is to show how approximate inference is capable of answering queries faster than exact inference, and thus can scale to larger examples. To answer questions **Q1.1** and **Q1.2**, we evaluate DPLA* for both the uniform cost search and geometric mean heuristics, DeepProbLog with exact inference and NeurASP. To investigate the effect of the accuracy of the neural predicates, we use varying numbers of examples to pre-train the neural predicates. As the pre-training trains the neural predicates on an easier task first,it is a form of curriculum learning.

The results are shown in Figure 3a. When we compare the methods that use exact inference: NeurASP and DeepProbLog, we see that the average time to answer a query grows rapidly and quickly becomes intractable. This is expected as these methods consider all proofs for a query, which grows rapidly with the number length. We can also see that NeurASP is slightly faster than DeepProbLog.

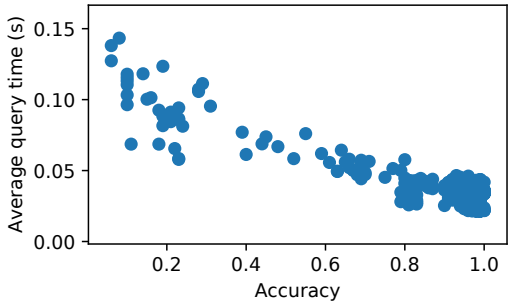
However, DPLA* with both Uniform-cost Search (UCS) and the Geometric Mean (GM) approximate inference with DeepProbLog can answer the queries several order of magnitude faster than the exact methods. The geometric mean heuristic is also considerably faster than the uniform-cost search when the neural predicate accuracy is low, but the difference grows smaller as this accuracy increases. For this reason, we will use the geometric mean heuristic in the remainder of the DPLA* experiments. To answer question **Q1.3**, we plot the query time with respect to the accuracy during training. In Figure 3b, we see that the speed of inference for DPLA* depends on the accuracy of the neural predicates. As the accuracy goes up, the times it takes to find a proof in DPLA* decreases.

To conclude, DPLA* is able to prove queries orders of magnitude faster than exact methods. The geometric mean heuristic outperforms uniform cost search if the neural predicate accuracy is low, but the difference decreases as the accuracy improves. The speed of DPLA* improves as the neural predicate accuracy increases.

[4]The code is available here: https://github.com/ML-KULeuven/deepproblog



(a) **Q1.1, Q1.2:** The average query time for different methods on the MNIST addition set. The number between brackets is the number of examples used to pre-train the neural predicates.



(b) **Q1.3:** The average query time on the MNIST addition task (N=2) for DPLA* with the geometric mean heuristic, with respect to the neural predicate accuracy.

Figure 3: **Q1:** Time to answer queries on the MNIST addition task.

Q2: Scaling Learning To answer this question, we compare DPLA*, NeurASP and NGS on the MNIST addition, HWF and CLUTRR datasets. **Q2.1** investigates how approximate inference impacts the learning in terms of accuracy, speed and convergence. We first compare DeepProbLog, NeurASP and DPLA* on MNIST addition for $N = 1..3$ for uninitialized neural networks and neural-networks pre-trained with limited amounts of direct supervision. As can be seen in Figure 4, DPLA* performs only slightly worse than DeepProbLog with exact inference. Without pre-training, DPLA* converges slower than exact inference. Due to the wider spread and lower mean, we can also see that the training is less stable and does not converge reliably to the correct solution. In Table 1, we compare the accuracy of the different methods on the test set of the MNIST addition. We can see that the exact methods reach the highest accuracy, but can only scale to $N = 2$. DPLA* with exploration attains a slightly lower performance than the exact methods. Note, however, that DPLA* with exploration does not scale to $N = 3$, since it considers too many proofs at the start of training. DPLA* achieves the lowest accuracy of all considered methods, but it is the only one that scales to $N = 3$. DPLA* without exploration, but pre-trained using 16 MNIST digits, performs similar to DPLA* with exploration, but it

can also scale to $N = 3$, where it performs significantly better.

We further explore the same question using the HWF task. We split off 10% of the training dataset as the validation set, which we use to select the best model during training. It is important to note that this is different from the testing methodology from the original paper where the test set was used to perform the selection. We use a validation set as this gives a less biased result. We report the accuracy on the test set, averaged over five runs. In Table 2, we see the performance of DeepProbLog, DPLA* and NGS on handwritten formulas of varying length. Exact inference in DeepProbLog can handle only expressions of length up to 3. Using approximate inference with the geometric mean heuristic allows DeepProbLog to scale to all the expressions in the dataset. We also note that DPLA* is able to learn without curriculum learning. The performance of both DeepProbLog and DPLA* for expressions of lengths up to 5 is close to the state-of-the-art NGS, which is quite interesting since DPLA* uses a task-agnostic approach. However, we have found a sharp decrease in mean accuracy for expressions of length 7 for NGS, as for some initializations NGS fails to converge. For the CLUTRR dataset we used the same setting as in Manhaeve et al. (2021) but with forward inference (implemented on a meta-level). Due to the cyclical nature of the rules, forward inference is more efficient than backward reasoning. The results in Figure 5 show that DPLA* is able to achieve good performance on an NLP setting as well, and is comparable to that of exact inference as shown in Manhaeve et al. (2021).

In question **Q2.2**, we analyse the convergence on the MNIST addition task. The results are shown in Figure 4. Without pre-training, DPLA* converges noticeably slower than exact inference. However, by pre-training the neural predicates with a small amount of images (only 16), approximate proving converges considerably faster. When pre-training the neural predicates, DPLA* shows the same convergence trend and the same performances as the exact algorithm with the same pre-training, with the latter taking a lot more time to answer each query, cf. Figure 3a.

For question **Q2.3**, we investigate whether exploration mitigates the drop in performance in absence of pre-training. We run the experiment with exploration for the neural predicates. The result can be seen in Figure 4. Here, we can see that, when using exploration, the spread in the accuracy between the pre-training and non pre-training settings is small, indicating that the convergence is reliable. This is not the case without exploration, indicating that for some runs, the learning does not converge on the correct solution. We can also see that exploration slows convergence.

To conclude, learning with approximate inference has a negative impact on learning in absence of pre-training or exploration. Convergence is slower and the final accuracy is also lower, as some runs do not converge onto a good solution. Using curriculum learning to pre-train the neural predicate makes the method converge as fast as exact inference and with the same accuracy. Moreover, the use of exploration on the neural predicates allows all the runs to converge on a good solution, but it converges slower than without exploration.

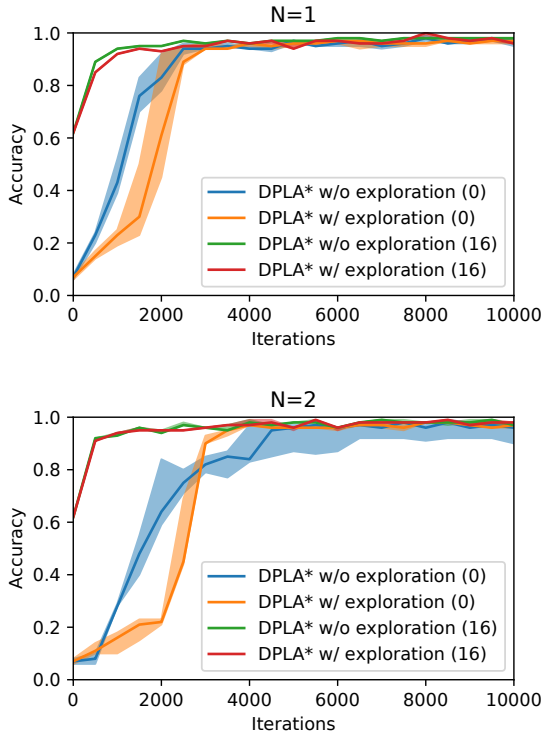


Figure 4: **Q2:**The neural predicate accuracy during training curves for DeepProbLog and DPLA* on the MNIST addition task, with and without pre-training and exploration. The number between brackets is the number of examples used to pre-train the neural predicates.

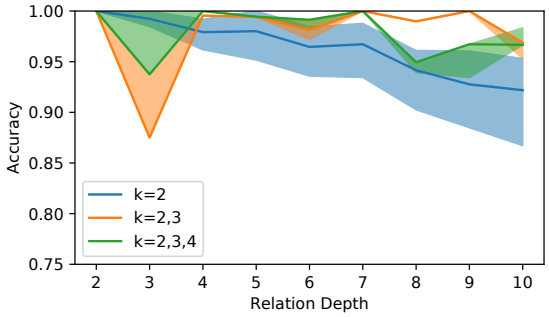


Figure 5: **Q2** The results on the systematic generalization of the CLUTRR dataset. k indicates the relation lengths present in the training data.

Q3: Parametric Heuristic To answer this question, we consider the task of multi-instance MNIST addition. In this task, we investigate the use of defining a heuristic over the `addition/3` goal. The idea is that by being able to estimate the probability of this goal, the correct bags are more likely to be selected and the training is more efficient. We pre-train the heuristic with varying number of examples of additions. We also continue training of the heuristic throughout the training of the neural predicates.

Figure 6 shows the results for the multi-instance MNIST

Methods	N 1	2	3
DeepProbLog	97.2 ± 0.5	95.2 ± 1.7	–
NeurASP	97.3 ± 0.3	93.9 ± 0.7	–
DPLA*	88.9 ± 14.8	83.6 ± 23.7	77.4 ± 33.3
DPLA* w/ pre-training (16)	95.8 ± 0.4	93.7 ± 0.2	88.9 ± 1.6
DPLA* w/ exploration	96.7 ± 0.2	93.4 ± 0.7	–

Table 1: **Q2** The accuracy on the MNIST addition test set for N=1..3 when trained on datasets of the same length.

addition problem on bag size 16. We see that training of the neural predicate is only effective when the proving is guided by the heuristic. A more informed heuristic leads to more efficient training, and a higher final accuracy. We also see that a more informed and accurate heuristic leads to more stable convergence and a higher final accuracy. We can also see that we are able to further fine-tune the heuristic during training, and that exploration has a positive effect on learning the heuristic as well. Interestingly, although exploration usually makes a method slower, here, it makes the method faster instead as the heuristic accuracy is higher, which results in faster inference.

To conclude, we show that parametric heuristic is effective at guiding the proving process when it is sufficiently accurate, reaching a higher accuracy and making the proving process faster. We are also able to fine-tune the heuristic throughout the training process.

7 Related Work

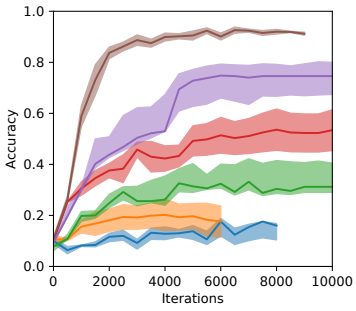
7.1 Approximate Inference

Exact inference in ProbLog is hard as it involves weighted model counting (WMC), which is #P-complete. This explains why there has already been a lot of research into approximate inference for probabilistic logic programming. However, these techniques have not yet been used or evaluated in the context of neural-symbolic computation. Furthermore, this context brings additional complications that were not considered in this previous work. The majority of the parameters in neural-symbolic methods reside in the neural networks, and cannot be directly optimized. This can lead to convergence towards local optima if the methods do not explore sufficiently. On the other hand, we leverage certain properties of the neural-symbolic AI to further improve the approximate inference. The assumption that in many tasks the success probability is equal to the explanation probability is used to make a more informed non-parametric heuristic. In addition, similarities between different queries (e.g. $3 + 4 = 7$ vs. $3 + 1 = 7$) can be used by a parametric heuristic to further guide the search. Approximate inference for probabilistic logic programming can be divided into two categories: approximate proving and approximate WMC.

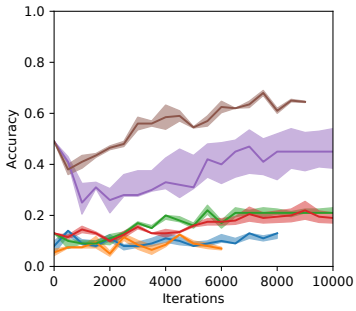
Approximate Proving These methods use only a subset of the proofs in the SLD-tree, which is also what DPLA* does. The k-best approach (Gutmann et al., 2008) is a branch-and-bound algorithm that only includes the top-k most likely proofs. This makes it very similar to DPLA* with uniform-cost search without exploration. However, k-best does neither consider exploration nor curriculum learning to avoid getting stuck in local optima when using approximate inference. The idea of k-best was further refined with k-optimal (Renkens, Van den Broeck, and Nijssen, 2012). K-best considers the probability of each proof separately. K-optimal, on the other hand, looks at the total (disjoint) probability of the k proofs that it uses, and selects as next proof the one that yields the maximum increase in total probability. A related approach is that of bounded approximation. Here, partial proofs are pruned when they reach a certain depths. The disjunction of the proofs that have been found already are used to compute the lower bound of the query probability, as finding more proofs can only increase the probability. The partial proofs that were not fully resolved are used to compute an the upper bound. If the difference between these two bounds is too large, the procedure is restarted with a deeper depth bound.

Approximate Weighted Model Counting Some methods rely on only approximating the weighted model count. So everything up to this step is the same as before (i.e. the grounding / proving phase, and turning the resulting program / proofs in to a logical formula). These methods try to avoid having to deal with the disjoint sum problem. For example, in DNF sampling (Shterionov et al., 2010), a sampling approach is used to approximate the real WMC from a DNF formulation of the logical formula without having to explicitly solve the disjoint sum problem. Similarly, MCMC estimation Moldovan et al. (2013) estimates conditional probabilities by implementing a Monte Carlo Markov chain on the and-or tree produced by the grounding phase.

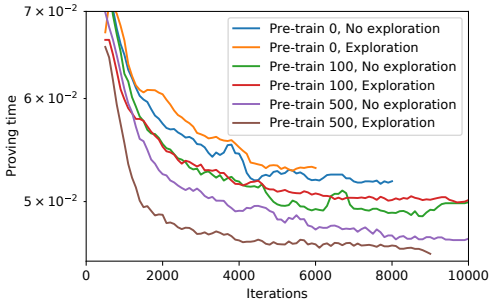
Other Approaches Some approaches deviate from the standard inference procedure. One such approach is program sampling (Kimmig et al., 2008). In this approach, one repeatedly samples a logic program by determining the truth value for all probabilistic facts by sampling according to their specified probability. For each sampled program we check if the query is entailed. The fraction of samples that entail the query is the probability of the query. T_p compilation Vlasselaer et al. (2016) combines forward reasoning with formula construction. At each step of the forward inference, if a clause can be applied, it both adds the head to the logic program, and adds a logical formula representing that clause to the logical formula representing the query. By also performing the compilation at this point, we get an efficient algorithm that can be stopped at any time to calculate the approximate probability of the query. It is also interesting to note that incrementally compiling the formula is efficient.



(a) Neural predicate accuracy on the first 100 images of the MNIST dataset



(b) Heuristic accuracy measured on the MNIST addition test dataset



(c) Moving average (window size 5) of the proving time for different methods.

Figure 6: **Q3:** The MIL Addition task with bag size 16.

	Expression length			
Method	1	3	5	7
NGS	90.2 ± 1.6	85.7 ± 1.0	91.7 ± 1.3	20.4 ± 37.2
DeepProbLog	90.8 ± 1.3	85.6 ± 1.1	–	–
DPLA*	90.9 ± 0.2	81.1 ± 2.4	89.0 ± 0.8	94.8 ± 0.5

Table 2: **Q2.1** Accuracy on the HWF dataset for expressions of length 1 to 7. Comparison of the proposed approach (DPLA*) with exact DeepProbLog inference and NGS.

7.2 Neural-Symbolic Methods

As argued above and in (De Raedt et al., 2020), inference in neural symbolic systems shares many properties with inference in (probabilistic) logic programs. As a consequence, approximate inference is an important, though not yet widely explored problem in neural symbolic computation. Many systems in the neural-symbolic scenario focus on approximating WMC using MCMC or variational approaches (Marra and Kuželka, 2019; Zhang et al., 2020). However, there are few systems that retain the focus on approximating the proving step like Neural Theorem Provers (NTP) (Rocktäschel and Riedel, 2017). NTPs are, like DeepProbLog, inspired by Prolog. The NTP uses a mechanism called soft-unification, which is a relaxation of the unification procedure used in Prolog. While unification applies only if two terms can be made identical, soft unification applies to any pair of terms that are similar in the latent space. However, because every term can be compared to every other term with the same arity, proving becomes easily intractable. Two approximate inference schemes for NTP have been proposed. In the first approach (Minervini et al., 2020a), nearest neighbourhood search is used to focus only on those branches where soft-unification provides the higher scores. This resembles how, in our approach, approximate inference uses only the most promising proofs. In the second approach (Minervini et al., 2020b), a neural module provides the next most promising rule(s) to expand. This is similar to how the parametric heuristic is used to guide the proving process in this work. NTPs, however, have been exclusively applied in structure learning tasks in the setting of knowledge-base completion, where the program is not known in advance, but it is learned as a by-product of the main task. Structure learning has not yet been investigated for DeepProbLog, and conversely, the NTP has not yet been applied to the type of task considered in this work.

Another related neural-symbolic approach is NeuroLog (Tsamoura and Michael, 2020). NeuroLog proposes to approximate the inference by *neurally guiding* the abduction. The idea is that the output of the neural network is used to instantiate only a subset of the possible neural predicates. Then, the framework considers only the proofs that contain the facts in this subset or small similar *perturbations*. The concepts of perturbation and similarity are, however, domain dependent and they are not provided as part of the framework. This makes this last system more similar to the NGS baseline than the problem-agnostic approach we propose in this paper.

8 Conclusion

We introduced an approximate inference technique for DeepProbLog called DPLA*. It extends an approximate inference algorithm from the field of statistical-relational AI to the neural-symbolic setting. Instead of considering all possible proofs for a certain query, the system searches for the best proof using an A*-like search. For this A*-like search, we considered several heuristics, including a parametric heuristic that can be trained on additional data to guide the proving process better. We showed on 4 datasets that DPLA* is more scalable than exact inference methods such as DeepProbLog and NeurASP, and can be applied on larger tasks. We also address the convergence issue that arises when the system learns using approximate inference using curriculum learning and UCB-based exploration.

Acknowledgements

This research has been funded by the Research Foundation - Flanders and the KU Leuven Research Fund (C14/18/062)". It has also been supported by the European Research Council Advanced Grant project SYNTH (ERC AdG-694980), the Flemish Government under the "Onderzoeksprogramma Artificiële Intelligentie (AI) Vlaanderen" programme, and the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation.

References

Besold, T. R.; Garcez, A. d.; Bader, S.; Bowman, H.; Domingos, P.; Hitzler, P.; Kühnberger, K.-U.; Lamb, L. C.; Lowd, D.; Lima, P. M. V.; et al. 2017. Neural-symbolic learning and reasoning: A survey and interpretation. *arXiv preprint arXiv:1711.03902*.

Darwiche, A., and Marquis, P. 2002. A knowledge compilation map. *Journal of Artificial Intelligence Research* 17:229–264.

De Raedt, L.; Dumančić, S.; Manhaeve, R.; and Marra, G. 2020. From statistical relational to neuro-symbolic artificial intelligence. In Bessiere, C., ed., *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI-20*, 4943–4950.

De Raedt, L.; Kimmig, A.; and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI*, volume 7, 2462–2467. Hyderabad.

Fierens, D.; Van den Broeck, G.; Renkens, J.; Shterionov, D.; Gutmann, B.; Thon, I.; Janssens, G.; and De Raedt, L. 2015. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming* 15(3):358–401.

Flach, P. 1994. *Simply logical: intelligent reasoning by example*. John Wiley & Sons, Inc.

Frazier, M., and Pitt, L. 1993. Learning from entailment: An application to propositional horn sentences. In *Machine Learning, Proceedings of the Tenth International Conference, University of Massachusetts, Amherst, MA, USA, June 27-29, 1993*, 120–127.

Garcez, A.; Gori, M.; Lamb, L.; Serafini, L.; Spranger, M.; and Tran, S. 2019. Neural-symbolic computing: An effective methodology for principled integration of machine learning and reasoning. *Journal of Applied Logics* 6(4):611–632.

Gutmann, B.; Kimmig, A.; Kersting, K.; and De Raedt, L. 2008. Parameter learning in probabilistic databases: A least squares approach. In Daelemans, W.; Goethals, B.; and Morik, K., eds., *Machine Learning and Knowledge Discovery in Databases*, 473–488. Berlin, Heidelberg: Springer Berlin Heidelberg.

Jelinek, F. 1976. Continuous speech recognition by statistical methods. *Proceedings of the IEEE* 64(4):532–556.

Kimmig, A.; Costa, V. S.; Rocha, R.; Demoen, B.; and De Raedt, L. 2008. On the efficient execution of problog programs. In *International Conference on Logic Programming*, 175–189. Springer.

Kimmig, A.; Van den Broeck, G.; and De Raedt, L. 2011. An algebraic prolog for reasoning about possible worlds. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 25.

Lai, T. L., and Robbins, H. 1985. Asymptotically efficient adaptive allocation rules. *Advances in applied mathematics* 6(1):4–22.

LeCun, Y.; Bottou, L.; Bengio, Y.; and Haffner, P. 1998. Gradient-based learning applied to document recognition. *Proceedings of the IEEE* 86(11):2278–2324.

Li, Q.; Huang, S.; Hong, Y.; Chen, Y.; Wu, Y. N.; and Zhu, S.-C. 2020. Closed loop neural-symbolic learning via integrating neural perception, grammar parsing, and symbolic reasoning. In *International Conference on Machine Learning (ICML)*.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2018. Deepproblog: Neural probabilistic logic programming. *Advances in Neural Information Processing Systems* 31:3749–3759.

Manhaeve, R.; Dumancic, S.; Kimmig, A.; Demeester, T.; and De Raedt, L. 2021. Neural probabilistic logic programming in deepproblog. *Artificial Intelligence* 298.

Marra, G., and Kuželka, O. 2019. Neural markov logic networks. *arXiv preprint arXiv:1905.13462*.

Minervini, P.; Bošnjak, M.; Rocktäschel, T.; Riedel, S.; and Grefenstette, E. 2020a. Differentiable reasoning on large knowledge bases and natural language. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, 5182–5190.

Minervini, P.; Riedel, S.; Stenetorp, P.; Grefenstette, E.; and Rocktäschel, T. 2020b. Learning reasoning strategies in end-to-end differentiable proving. In *International Conference on Machine Learning*, 6938–6949. PMLR.

Moldovan, B.; Thon, I.; Davis, J.; and Raedt, L. D. 2013. MCMC estimation of conditional probabilities in probabilistic programming languages. In van der Gaag, L. C., ed., *Symbolic and Quantitative Approaches to Reasoning with Uncertainty - 12th European Conference, ECSQARU 2013, Utrecht, The Netherlands, July 8-10, 2013. Proceedings*, volume 7958 of *Lecture Notes in Computer Science*, 436–448. Springer.

Pearl, J. 1984. Intelligent search strategies for computer problem solving. *Addision Wesley*.

Rawson, M., and Reger, G. 2019. A neurally-guided, parallel theorem prover. In *International Symposium on Frontiers of Combining Systems*, 40–56. Springer.

Renkens, J.; Van den Broeck, G.; and Nijssen, S. 2012. k-optimal: A novel approximate inference algorithm for problog. *Machine learning* 89(3):215–231.

Rocktäschel, T., and Riedel, S. 2017. End-to-end differentiable proving. *arXiv preprint arXiv:1705.11040*.

Shterionov, D.; Kimmig, A.; Mantadelis, T.; and Janssens, G. 2010. Dnf sampling for problog inference. In *Proceedings International Colloquium on Implementation of Constraint and LOgic Programming Systems (CICLOPS)*, 15.

Sinha, K.; Sodhani, S.; Dong, J.; Pineau, J.; and Hamilton, W. L. 2019. Clutrr: A diagnostic benchmark for inductive reasoning from text. *Empirical Methods of Natural Language Processing (EMNLP)*.

Tsamoura, E., and Michael, L. 2020. Neural-symbolic integration: A compositional perspective. *arXiv preprint arXiv:2010.11926*.

Vlasselaer, J.; den Broeck, G. V.; Kimmig, A.; Meert, W.; and Raedt, L. D. 2016. T_p-compilation for inference in probabilistic logic programs. *Int. J. Approx. Reason.* 78:15–32.

Wang, M.; Tang, Y.; Wang, J.; and Deng, J. 2017. Premise selection for theorem proving by deep graph embedding. *Advances in neural information processing systems*.

Yang, Z.; Ishay, A.; and Lee, J. 2020. Neurasp: Embracing neural networks into answer set programming. In *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI*, 1755–1762.

Zhang, Y.; Chen, X.; Yang, Y.; Ramamurthy, A.; Li, B.; Qi, Y.; and Song, L. 2020. Efficient probabilistic logic reasoning with graph neural networks. *arXiv preprint arXiv:2001.11850*.