Sum-Product Loop Programming: From Probabilistic Circuits to Loop Programming

Viktor Pfanschilling[1] , **Hikaru Shindo**[1] , **Devendra Singh Dhami**[1,3] , **Kristian Kersting**[1,2,3]
[1]Computer Science Department, Technical University of Darmstadt, Germany
[2]Center for Cognitive Science, Technical University of Darmstadt, Germany
[3]Hessian Center for AI (hessian.AI), Darmstadt, Germany
{viktor.pfanschilling, hikaru.shindo, devendra.dhami, kersting}@cs.tu-darmstadt.de

Abstract

Recently, Probabilistic Circuits such as Sum-Product Networks have received growing attention, as they can represent complex features but still provide tractable inference. Although quite successful, unfortunately, they lack the capability of handling control structures, such as for and while loops. In this work, we introduce Sum-Product Loop Language (SPLL), a novel programming language that is capable of tractable inference on complex probabilistic code that includes loops. SPLL has dual semantics: every program has generative semantics familiar to most programmers and probabilistic semantics that assign a probability to each possible result. This way, the programmer can describe how to generate samples almost like in any standard programming language. The language takes care of computing the probability values of all results for free at run time. We demonstrate that SPLL inherits the beneficial properties of PCs, namely tractability and differentiability, while generalizing to other distributions and programs, and retains substantial computational similarities.

1 Introduction

Sum-Product Networks (SPNs) can be viewed as a deep learning architecture as well as a graphical model with tractable inference (Poon and Domingos 2011). SPNs are based on the concept of modeling network polynomials and are closely related to arithmetic circuits (Darwiche 2003). They exploit the efficiency of deep learning while abstracting the representation of the underlying model by implementing compositions of functions. As a result of such compositions, SPNs can model different distributions efficiently when compared to pure deep learning models and can compute any marginalization and conditioning query in time linear to the size of the network. That is, we can answer questions such as "what is the probability of X being 0.3 and Y being any value" or "what is the probability of X being greater than 0.3 and Y being greater than 0.5" efficiently. The inherent probabilistic formulation of SPNs enables effective reasoning about the uncertainty in the underlying domain.

Deep probabilistic programming languages (DPPLs) such as DeepProblog (Manhaeve et al. 2018), Pyro (Bingham et al. 2019) and Edward (Tran et al. 2017), to name a few, leverage the expressive power of deep neural networks within probabilistic programming systems, especially at inference time. These deep neural architectures allow sampling in the DPPLs but the computation of marginal probabilities remains an issue. Such a computation can quickly become intractable, especially for continuous variables, thus limiting the types of queries that can be handled efficiently by DPPLs.

```
A = coinflip(theta1)
if(A) {
  B = coinflip(theta2)
} else {
  B = A
}
```

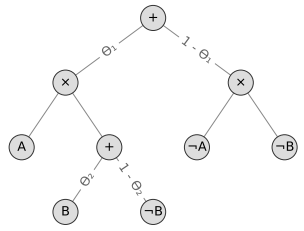


Figure 1: A simple program with randomness (left) and an SPN encoding the same distribution(right).

Looking at programs with random outcomes as ways to encode distributions, for some programs we can find SPNs or other probabilistic representations that encode the same distribution as that program (Holtzen, Millstein, and Broeck 2019). For example, consider figure 1 that shows an SPN representation for a simple program. This probabilistic representation is useful because it enables us to learn, as the distribution is differentiable in its parameters. It also encodes the joint distribution in a way that enables more complicated queries like maximum a posteriori (MAP) or most probable explanations (MPE).

Hand-crafting the equivalent SPN structure for a given program is tedious, and quite complicated once we introduce recursion or other complex control flow, such as for and while loops, into the program. In order for an SPN to be a valid distribution, one of the constraints is that of *completeness* which states that each child of a sum node must cover the same set of variables, that is, have the same scope. If a loop that generates more samples is introduced, then the children of the sum node that decides on whether to keep iterating or to stop will have different scopes, thereby violating the completeness constraint and rendering the SPN invalid.

Consider for example the following program where crafting the SPN structure is impossible, due to the number of involved variables:

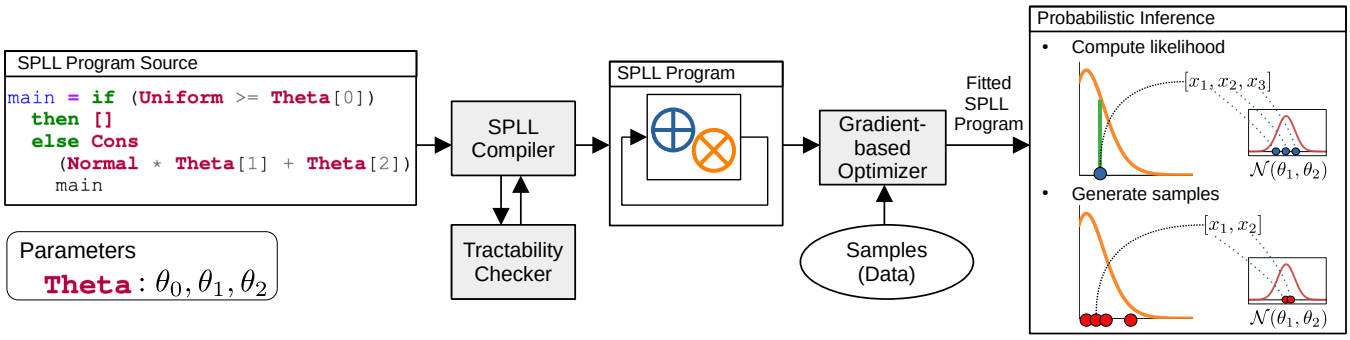


Figure 2: An overview of SPLL, for the task of parameter estimation. The source program of SPLL is compiled into executable SPLL programs by the compiler. During the compilation, SPLL checks the tractability of the program. The parameters in the program can be trained by gradient descent using samples. Using the fitted SPLL program, users can perform different types of probabilistic inference, e.g. compute the likelihood given samples or generate samples. We note that each data point in SPLL can be structured, e.g. as a list of values. In the example code, each data point is a list of values, and each value is following a Gaussian distribution which is parameterized by θ_1 and θ_2.

```
main = if coinflip(theta1) {
   (coinflip(theta2) : main)
} else {
   []
}
```

To handle these control structures, we introduce Sum-Product Loop Language (SPLL), a novel programming language capable of tractable inference on complex code that includes loops and produces a probabilistic representation from the program code. By allowing recursive programs, we can e.g. model distributions that describe a heterogeneous number of variables. In SPLL, every program has generative semantics that most programmers are familiar with and probabilistic semantics that assign a probability to each possible result. Thus, SPLL enables programmers to describe a generative process similarly to any standard programming language. The language produces code for computing the probability of values for free. At the core of SPLL lie two interpreters: one generates a sample from a program and its parameterization while the other differentiably computes the probability of a sample given a program and its parameterization. Overall, we make the following contributions:

1. We propose sum-product loop language (SPLL), a novel probabilistic programming language capable of handling loops in complex probabilistic code.
2. We describe the syntax as well as the generative and probabilistic semantics of SPLL (Section 2).
3. We demonstrate that SPLL is at least as capable as Sum Product Networks and inherits the beneficial properties of PCs, namely tractability and differentiability(Section 5).
4. We illustrate that SPLL can neatly solve some problems that are fundamentally unsolvable using Sum Product Networks (Section 6).

We proceed as follows. We start off by introducing SPLL, including its corresponding semantics. Then we introduce the notion of tractable programs within SPLL and then show the training procedure. Before concluding we show SPLL is at least as powerful as sum-product networks and also how SPLL extends beyond them.

2 Sum-Product Loop Language (SPLL)

Figure 2 shows the overall architecture of SPLL as presented for the task of parameter estimation. The SPLL source code, written in the manner of a standard programming language, is compiled to an SPLL program after going through a tractability checker. The tractability checker determines if the source SPLL program is efficiently solvable. The SPLL program parameters are then trained using gradient descent. The fitted program can then be used for inference and sample generation.

2.1 SPLL by Example

For an intuitive introduction, we construct a few compact SPLL programs and give a brief explanation from the view of generative semantics. We will cover the path to probabilistic semantics in section 2.3. Our first program will simply draw from a $\mathcal{N}(0, 1)$ distribution.

```
main = Normal
```

Our next program will linearly transform the $\mathcal{N}(0, 1)$ samples using parameters *Theta* to produce a $\mathcal{N}(\mu, \sigma)$-distribution.

```
main = Normal * Theta[0] + Theta[1]
```

We can also produce boolean variables, for example by comparing a uniform random sample against a parameter. This results in a Bernoulli distribution.

```
main = Uniform >= Theta[0]
```

Our final example will introduce list construction and conditional statements. It returns either [True] or [].

```
main = if Uniform >= Theta[0]
  then Cons True Null
  else Null
```

2.2 SPLL Syntax

We now introduce the core syntax of SPLL. An SPLL program can consist of multiple functions, each assigning an expression to a name. These functions can call each other, also recursively, thereby extending existing first-order functional

$$
\begin{aligned}
&\mathcal{E} \rightarrow \textit{if } \mathcal{E} \textit{ then } \mathcal{E} \textit{ else } \mathcal{E}\\
&\mathcal{E} \rightarrow \mathcal{E} >= \mathcal{E}\\
&\mathcal{E} \rightarrow \textit{theta}[\mathbb{Z}]\\
&\mathcal{E} \rightarrow \textit{uniform}\\
&\mathcal{E} \rightarrow \textit{normal}\\
&\mathcal{E} \rightarrow \textit{constant } \mathcal{V}\\
&\mathcal{E} \rightarrow \mathcal{E} * \mathcal{E}\\
&\mathcal{E} \rightarrow \mathcal{E} + \mathcal{E}\\
&\mathcal{E} \rightarrow \textit{null}\\
&\mathcal{E} \rightarrow \textit{cons } \mathcal{E}\ \mathcal{E}\\
&\mathcal{E} \rightarrow \textit{call } \mathcal{F}
\end{aligned}
$$

Figure 3: The syntax for SPLL.

languages such as DICE (Holtzen, Millstein, and Broeck 2019). SPLL programs can be queried by an interpreter.

Formally, the syntax for SPLL expressions is defined as in Fig. 3. $\mathcal{E}$ refers to the expressions in SPLL and $\mathbb{Z}$ refers to the set of whole numbers and is used to index the θ vector. The $\mathcal{V}$ nonterminal refers to the set of all values supported by the language and $\mathcal{F}$ refers to any function in the SPLL program. Overall, SPLL's syntax follows the syntax of standard functional programming languages, e.g. Haskell, closely.

2.3 SPLL Semantics

Now that we have covered the syntax, let us turn to the semantics. We first show how to draw samples and then derive the probability distribution of the generative semantics.

(1) Generative Semantics The generative semantics of SPLL are accessed by calling the *generate* function on an SPLL program. The semantics again draw from functional languages such as Haskell, albeit with randomness as a first-class part of the language. The generative semantics of the syntax elements are aligned with the syntax, i.e. the programs can be read as a generative description of its distribution. We use *cons* and *null* for list construction. Types in the generative semantics are restricted to *Bool*, *Float*, and *Lists*. SPLL prevents certain ill-typed expressions at compile time such as using a non-boolean for the condition of an if-expression.

Loops are implemented in SPLL using recursion. This as well as the functional language simplify control flow and thus simplify reasoning about the resulting distribution. θ denotes the parameters; each index refers to a real-valued variable marked as subject to parameter optimization. We adopt the definition of most syntax elements from Haskell.

Theorem 1 (Generative Semantics)**.** *Any properly typed SPLL program with a complete parameterization θ that is not recursing indefinitely can be used to draw samples from its distribution.*

Proof. The syntax of SPLL programs encompasses a set of simple functional programs with well-known semantics. We can execute these programs to generate samples. The only syntax not usually found in functional languages is our θ-parameteriziation, which will yield a result as long as the provided θ-vector is long enough with respect to the indices used. □

(2) Probabilistic Semantics The probabilistic semantics of SPLL, given a sample, follow a similar recursive formulation. The resulting probabilities will add to 1 if integrated or summed over all possible outcomes of the program. They express the chance of that sample being emitted by the *generate* function. Some expressions introduce requirements and assumptions beyond what is readily apparent from their generative semantics. For example, *cons*-statements assume statistical independence of the head and tail elements, which is trivially true, since SPLL does not allow variable bindings, thus preventing information flow between the subexpressions.

We will next prove that the generative and probabilistic semantics describe the same distribution:

Theorem 2. *For any properly typed SPLL program with a complete parameterization θ, the analytical probability function accurately represents the empirical sample probability resulting from the associated generative process.*

Proof. In order to demonstrate that the *probability* function is aligned with the sample probability, we look at how it computes probabilities for each sample. In Table 1, we provide the probability $p(x|\texttt{expr})$ of a sample x arising from a program `expr`. We use these probabilities directly in interpreting the *probability* function. Intuitively, conditionals weigh the outcomes of each branch by the probability of the condition selecting that branch. Meanwhile, comparisons (which have one integrable and one deterministic argument), such as $p(\top|\texttt{a} >= \texttt{b})$, integrate the probability of the random variable up to the deterministic boundary. Please mind the distinction of which side is deterministic and which side of c is integrated. As for arithmetic expressions, here we compute the deterministic operand and use it to invert the operation to deduce the value of the random subexpression. For multiplication, we additionally need to apply a correction factor: The inversion widens the graph of the distribution. To ensure it integrates to 1, we divide the probability by y. Finally, keep in mind that we can safely assume independence of the subexpressions of *cons*. These probabilities were derived from the generative semantics. □

These semantics preserve and expect data type consistency. For example, queries such as $p(\top|\texttt{Normal})$ will not arise and the condition in if-statements must return a boolean. This, as well as the typing rules of the generative process, guarantee type safety. The type system of the probabilistic semantics ensures that (sub-)expressions that are integrated or deterministically evaluated in the above computations are indeed integrable or deterministic. Please refer back to the semantics in Table 1 for when these conditions apply and see Section 3 for how these conditions are enforced.

$$p(x|\texttt{if c then a else b}) = p(\top|\texttt{c})p(x|\texttt{a}) + p(\bot|\texttt{c})p(x|\texttt{b})$$

$$p(x|\texttt{uniform}) = \varphi_{\mathcal{U}(0,1)}(x)$$

$$p(x|\texttt{normal}) = \varphi_{\mathcal{N}(0,1)}(x)$$

$$p(x|\texttt{constant y}) = \begin{cases} 1, \textit{if } x = y \\ 0, \textit{if } x \neq y \end{cases}$$

$$p(x|\texttt{theta[y]}) = \begin{cases} 1, \textit{if } x = \theta_y \\ 0, \textit{if } x \neq \theta_y \end{cases}$$

$$p(x|\texttt{a} * \texttt{b}) = \begin{cases} p(x/y|\texttt{a})y \textit{ if det(b)} \to y \\ p(x/y|\texttt{b})y \textit{ if det(a)} \to y \end{cases}$$

$$p(x|\texttt{a} + \texttt{b}) = \begin{cases} p(x-y|\texttt{a}) \textit{ if det(b)} \to y \\ p(x-y|\texttt{b}) \textit{ if det(a)} \to y \end{cases}$$

$$p(x|\texttt{null}) = \begin{cases} 1 \textit{ if x = []} \\ 0 \textit{ otherwise} \end{cases}$$

$$p(x|\texttt{cons a b}) = \begin{cases} 0 \textit{ if x = []} \\ p(y|\texttt{a})p(ys|\texttt{b}) \\ \quad \textit{if x = (y:ys)} \end{cases}$$

$$p(x|\texttt{call "f"}) = p(x|\texttt{f})$$

$$p(\top|\texttt{a >= b}) = \int_c^\infty p(x|\texttt{a})\,dx \quad \textit{where det(b)} \to c$$

$$p(\bot|\texttt{a >= b}) = 1 - \int_c^\infty p(x|\texttt{a})\,dx \quad \textit{where det(b)} \to c$$

$$p(\top|\texttt{a >= b}) = \int_{-\infty}^c p(x|\texttt{b})\,dx \quad \textit{where det(a)} \to c$$

$$p(\bot|\texttt{a >= b}) = 1 - \int_{-\infty}^c p(x|\texttt{b})\,dx \quad \textit{where det(a)} \to c$$

Table 1: Semantics for SPLL expressions. We denote 'expression a is deterministic with result x' with $det(a) \to x$; the density function of a distribution d with φ_d, and the name of a function and its implementation with "f" and f. The difference of probability densities and probabilities is left implicit.

3 Tractable Programs via Typing

In order to help the user avoid intractable programs, we provide rudimentary type inference that returns a compile-time error if no efficient implementation of the probability function is available. These probability types describe the operations we can efficiently do on a given expression. Generally, the type of an expression depends on the types of its subexpressions, but can have unique requirements such as the need to integrate over values of one subexpression or the need for a subexpression to be deterministic. Therefore, the type system keeps track of whether an expression is deterministic or allows efficient integration. If SPLL does not have a tractable solution to the probability function of a program, SPLL will raise a type error.

We developed rules that derive the type of an expression. For example, uniform and normal distributions can be integrated. Parameters θ and constants behave deterministically.

Conditional statements can be integrated over if both branches can be integrated over. The condition, being of generative type *bool*, is either tractably solvable or not; playing no further role in typing beyond failing if it is intractable.

The $a >= b$ operator requires that one argument be deterministic and the other be integrable, as the semantics integrate the random variable up to the deterministic boundary. Arithmetics are considered tractable if either argument is deterministic.

Different algorithms and along with them different typing rules can coexist within the SPLL compiler. For example, the solution of addition has two symmetrical variants. It is possible to add more variants that cover additional cases, thus expanding the set of valid programs. We will elaborate on such future expansions in section 8.

Tractable Programs as SPNs Now that we have a way to guarantee that the program is tractable, we can try to find a way back towards more familiar tractable structures. First, let us constrain the set of considered programs further, in order to characterize programs with useful gradients:

Definition 1. *An SPLL program is consistent if it has the following properties: (A) For any sample, every execution path resulting in this sample must have finite length. (B) The probability functions of* constant *and* theta *are not called or integrated.*

Note that (A) is true in particular for programs that recursively construct lists, as this way the recursion depth is connected to the number of list elements. Meanwhile (B) implies that deterministic expressions are only used where the probability rules expect them.

Theorem 3. *For any sample and consistent program, the computation of the probability function is equivalent to a finite SPN. That is, a program induces a set of SPNs, potentially one per sample.*

Proof. Looking at the previous formulas for the probability of a sample in Table 1, large parts collapse at compile-time into structures familiar from SPNs: leaf nodes containing

primitive distributions, weighted sums, and products. The indicator functions associated with deterministic behavior in `constant` and `theta` are not actually called upon in the probability computation of consistent programs. Some expressions result in behaviors that are not trivially mapped to an SPN. We now illustrate how these map to SPNs:

(A) Arithmetics query its subexpressions normally, analogous to an SPN. Since y is deterministic, the transformations upon x it imparts do not change the distribution fundamentally. In order to construct an equivalent SPN, these transformations can be represented as changes to the leaf nodes.

(B) Function calls attach another function's computational graph in place, potentially recursively. Due to consistency of the SPLL program, the resulting graph has a finite depth.

(C) The indicator functions in the probabilities of *cons* and *null* develop interesting behavior though: They can be seen as rejecting samples that are ill-structured. In particular, *cons* rejects empty lists or shortens a list for further evaluation, thus guaranteeing termination for finite lists when applied recursively. We can also view this behavior as pruning an infinite, recursive SPN down to a finite SPN, subject to information from the sample.

This way, we can represent the computation of the probability function as an SPN, where the structure and parameters of that SPN depend on compile-time information and runtime information from the sample. □

Tractability Inference in general is known to be an intractable problem (Darwiche 2009). We have demonstrated that consistent SPLL programs are computationally equivalent to a set of SPNs. With SPNs known to be tractable, this makes SPLLs tractable, at least with respect to the size of the equivalent SPN model. The size of this model is dependent on recursion behavior and the SPLL code length. For any program that can be constrained in its recursion behavior, for example by recursively constructing lists, we can thus show tractability. We discuss the impact of tractability on the design decisions of SPLL and on future expansions in Section 8.

4 Training SPLL Programs

Now that we have formalized the syntax and semantics, we will look at the task of parameter estimation, using the following learning setup:

> **Given** a set of samples X and a consistent SPLL program, **find** the parameterization θ of the program that best maximizes log-likelihood $\mathcal{L}$ of all samples given a program and parameters: $\mathcal{L} = \sum_{x \in X} -log(p(x|\theta))$.

For this learning setting, it is natural to follow a standard gradient approach for maximizing log-likelihood, and we will describe that next. However, we have to be a bit more careful and we will explain that afterward.

4.1 Gradients via Automatic Differentiation

The gradient of the likelihood function with respect to a program's `theta[i]` can be computed using automatic differentiation. This works well for most expressions except for the following: Uniform, deterministic expressions and list construction.

Specifically, *Uniform* distributions do not exhibit useful gradients. However, when integrated they are very useful. We thus use them in the argument of comparisons, so as to produce more understandable code. Furthermore, the indicator functions for *deterministic expressions* provide no usable gradients, but for consistent SPLL programs, this is no problem, as their probability function will not be evaluated. *List construction* has, in spite of appearances, proper gradients, as the discrete jump between empty and full list does not happen via variation of `theta` but only by deconstructing x.

While it may seem like we can not train thetas, as the derivative of $p(x|\texttt{theta[y]})$ is 0 everywhere, this is not true. Note how other probabilities, namely arithmetics and comparisons, depend on the result of deterministic expressions. The derivatives of their probabilities will contain the derivatives of said result, as opposed to the derivative of the probability of the result. The derivative of the result of a `theta[i]`-expression is a vector with 1 at index i, thus providing a non-zero base case for SPLL gradients.

Given the gradients and a learning rate λ, we can train the parameters θ of an SPLL program in the following way: We initialize θ randomly and iteratively update it using $\theta := \theta + \lambda \cdot \partial\mathcal{L}/\partial\theta$. Assuming that the given program is adequate for the dataset, we can reconstruct the exact distribution of the data.

Although the above approach seems natural, consider the following SPLL program:

```
main = if 0.5 >= Theta[0]
  then True
  else False
```

This program is interesting as it has gradients that do not allow the training of θ at all, even though θ affects the distribution. While it would be easy to brush this program aside as inconsistent, let us instead investigate remedial actions for it and programs like it.

4.2 Gradients via Continuous Relaxations

We can use continuous relaxations to produce a gradient that resembles that of the original distribution, while also being much better behaved. The consequence of this is giving up exactness as the relaxations will result in approximate solutions.

In the case where both $det(l) \rightarrow l_d$ and $det(r) \rightarrow r_d$, by smoothing the discrete comparison result with a sigmoid, the probabilities of comparisons can be written as

$$p(\top|\texttt{l >= r}) = \sigma(l_d - r_d)$$
$$p(\bot|\texttt{l >= r}) = 1 - \sigma(l_d - r_d)$$

and their corresponding gradients can be computed as:

$$\nabla p(\top|\texttt{l >= r}) = \sigma'(l_d - r_d)\nabla l_d - \sigma'(l_d - r_d)\nabla r_d$$
$$\nabla p(\bot|\texttt{l >= r}) = \sigma'(l_d - r_d)\nabla r_d - \sigma'(l_d - r_d)\nabla l_d \ .$$

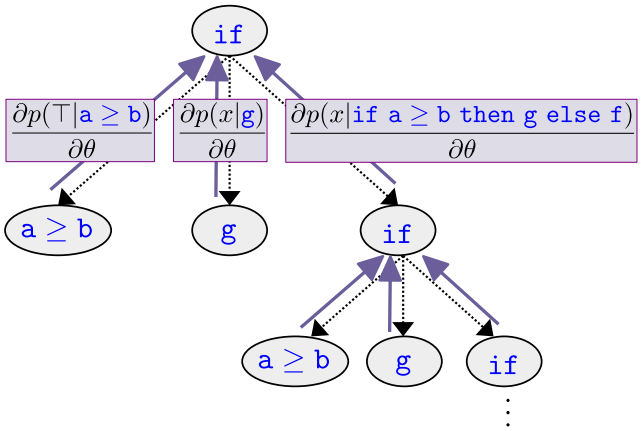


Figure 4: Gradient flow in the SPLL program. Each node represents a component of the program. Dotted arrows represent the forward path, and colored arrows represent the backward path, respectively. The gradient of an entire SPLL program can be computed by automatic differentiation even if the program contains complex control flow such as recursions and conditionals.

The changes affect conditional statements and thus propagate even to recursive calls. Consider how this affects conditionals: With the probability of the condition returned to smoothness using the above relaxations, we can again compute reasonable gradients for conditional statements.

Let us now return to the above code example. We will assume a sample of $x = \texttt{True}$. We will omit a few terms due to redundancy arising from symmetry. To disambiguate the factor 0.5 from the program constant 0.5, we will write out the usually implicit `constant` expression. The gradient of the program, ignoring the symmetrical $\perp$ case would be the following:

$$
\begin{aligned}
&\frac{\partial}{\partial\theta} p(\top|\texttt{if c then a else b}) \\
&= p(\top|\texttt{a}) \cdot \frac{\partial}{\partial\theta} p(\top|\texttt{c}) + p(\top|\texttt{c}) \cdot \frac{\partial}{\partial\theta} p(\top|\texttt{a}) + \ldots \\
&= 1 \cdot \frac{\partial}{\partial\theta} p(\top|\texttt{constant 0.5 >= theta[0]}) + p(\top|\texttt{c}) \cdot 0 \\
&= \sigma'(0.5 - \theta_0) \frac{\partial}{\partial\theta} (\texttt{constant 0.5}) - \\
&\quad \sigma'(0.5 - \theta_0) \frac{\partial}{\partial\theta} (\texttt{theta[0]}) \\
&= -\sigma'(0.5 - \theta_0)\ ,
\end{aligned}
$$

where the last step is because θ consists of `theta[0]` only.

As we can see, we restored the gradient of θ. We can convince ourselves that the gradient is viable by observing that the gradient is always negative, most strongly so around $\theta = 0.5$. From the source code, we can learn that a lower theta will tend to produce more `True` samples, as it will move the program towards the respective side of the decision boundary of the condition. That is exactly the property we want to maximize.

We can restore gradients even if a branch of the conditional recurses: Note the original semantics for recursion are quite benign; the heavy lifting of computing likelihoods and gradients happens in the conditional and comparison expressions. The question of whether to recurse or not is decided by the plausibility of the desired outcome under the recursive and the non-recursive branch, and that decision is made in the conditional. Thus, restoring gradients to conditionals is sufficient to restore gradients even to more complex, recursive programs.

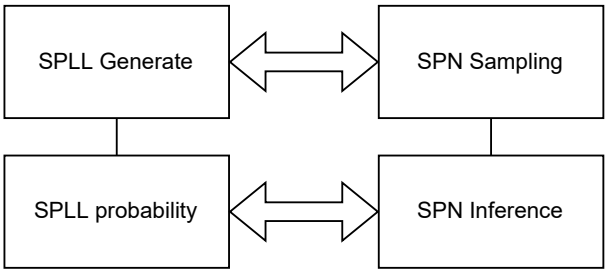


Figure 5: Mapping between SPN and SPLL functions for simple SPLL programs.

By using Automatic Differentiation, the gradient for an entire SPLL program, including any recursions can be computed by computing the gradient of each component and applying the chain rule. Consider the following SPLL program:

```
f = if a >= b
  then g
  else f
```

where `f` is a recursive function that calls itself or the function `g`. Figure 4 shows the gradient flow of the program. The gradient of each component can be computed as presented above. SPLL allows computing the gradients even if the program contains complex control flows such as if-then-else and recursions.

5 SPLL Covers Sum-Product Networks

Because SPNs offer good guarantees of e.g. tractability, it is desirable to demonstrate that SPLL can fall back on those guarantees if the problem at hand is sufficiently simple. These guarantees are also valuable if the SPLL program can divide the problem into SPN-solvable sub-problems. Therefore, we will now show how any SPN can be expressed in SPLL, and which subset of SPLL programs maps cleanly to SPNs.

Let us start off by showing that we can express any SPN in our language simply by writing out its sampling procedure. That is to say, we need an SPLL code's generative process to match an SPN's generative process, i.e. its sampling procedure. Under that condition, their distributions also align (See Figure 5). Naturally, this symmetry can not hold if the SPLL is sufficiently complex, such that an SPN with an identical sampling procedure can not be found.

Let us look now in more detail at how this sampling procedure can be implemented in SPLL and why the distributions line up: Concatenation of two disjointed sets of variables and weighted random sampling from

two branches are exactly equivalent to if-then-else and list concatenation, with the respective probabilities being weighted sums and products respectively. More explicitly, `if Bernoulli(p) then A else B` is equivalent to a sum node that associates node A with weight p and node B with weight $(1-p)$. Similarly, the concatenation of two independent lists is equivalent to a product node, in that the probability of the concatenation is the product of the probabilities of the sublists. Consequently, we interpret the list [x1, x2, x3] as a sample of a trivariate distribution. For completeness, we should add that proper SPN-equivalence requires the use of "chaotic" concatenation, where we can e.g. concatenate [x1, x3] and [x2, x4, x5] into [x1, x2, x3, x4, x5]. This is necessary because product nodes in SPNs can split the domain of variables arbitrarily. This can be implemented in our language as an operation that rearranges a single list. SPN nodes with multiple children in SPNs can be represented as nested statements in our language.

Thus, a minimal grammar for SPLL programs that cover SPNs would be:

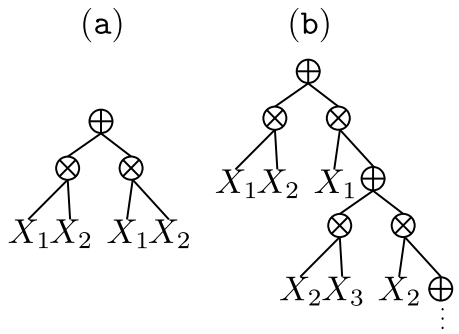
$$\begin{aligned}
&\textit{Expr} \rightarrow \textit{if (bernoulli } p\textit{) then Expr else Expr}\\
&\textit{Expr} \rightarrow \textit{normal } \mu\ \sigma\\
&\textit{Expr} \rightarrow \textit{null}\\
&\textit{Expr} \rightarrow \textit{cons Expr Expr}
\end{aligned}$$

Every other element of the grammar of SPLL, like function calls or arithmetic, is there to support more complex programs or to aid in flexibility of the language.

Experimentally, we confirm the above theoretical considerations by modelling a simple mixture of Gaussians using an SPN and using our language. To this end, we used two Gaussians centered at (0.3, 0.3) and (0.7, 0.7), with $\sigma = 0.1$ and a mixture ratio of 0.4 to 0.6. We modeled this distribution using the SPLL code shown in Listing 1. We will compare this to an SPN using SPFlow (Molina et al. 2019).

We find that in both cases, the parameters of the respective structures match those found in the distribution: SPN and SPLL both get the mixture ratios and the underlying parameters of the normal distribution right to within a small approximation error. Please refer also to Figure 7: We display the probability density of the list $[X, Y]$ for each model used. That is, for each X and Y, we plot the probability density of that pair, as modeled by the SPN and the SPLL program. Both models exhibit a stronger peak at (0.7, 0.7) and a weaker peak at (0.3, 0.3), which reflects the method of data synthesis, including the mixture ratio. The resulting figures line up to the point of being indistinguishable.

Figure 6: SPNs are SPLL programs. (a) A valid SPN. (b) An invalid SPN caused by recursions. Note that the topmost sum node in (b) covers different sets of variables in its left and right branch.



Listing 1 Multivariate mixture of Gaussians

```
main = if Uniform >= Theta[0]
  then [Normal * Theta[1] + Theta[2],
        Normal * Theta[3] + Theta[4]]
  else [Normal * Theta[5] + Theta[6],
        Normal * Theta[7] + Theta[8]]
```

Listing 2 Recursive lists of mixture of Gaussians

```
main = if Uniform >= Theta[0]
  then Null
  else Cons
    if Uniform >= Theta[1]
      then Normal * Theta[2] + Theta[3]
      else Normal * Theta[4] + Theta[5]
    main
```

6 SPLL Moves Beyond SPNs

To illustrate why recursive programs can generate more varied distributions than SPNs, consider for example a process that generates one sample from a Gaussian mixture distribution per iteration, with some probability of stopping each time. The resulting distribution contains lists of variable length with no upper limit, where longer lists are exponentially less likely. Distributions such as this can not be modelled by SPNs. Since the resulting number of variables varies, an SPN would need to cover a heterogeneous number of variables, which would violate completeness. For example, consider the SPN in Fig. 6(a) which is defined over two random variables X_1 and X_2. Note that the scope of the childern of the only sum node is $\{X_1, X_2\}$ as both the random variables are present on both the left and right child product nodes. Now consider the SPN in Fig. 6(b) obtained after recursion on the underlying SPN in Fig. 6(a) where one of the leaf node is replaced by the underlying SPN but can contain a new random variable X_3. The children of the topmost sum node then have the scope $\{X_1, X_2\}$ for the left child product node and $\{X_1, X_2, X_3\}$ for the right child product node thus specifying different scopes and violating the completeness property of the SPN.

With SPLL, we can solve this by modelling the generative process (see Listing 2). The likelihood function correctly determines the amount of times the recursion condition fired before failing and consequently provides a gradient for optimizing θ_0. The gradient is also passed through to the parameters of the Gaussian distributions. While estimating θ_0 is of course possible from just the list lengths, SPLL can do this without much modelling work and for more complex programs where connections are not as readily apparent. For

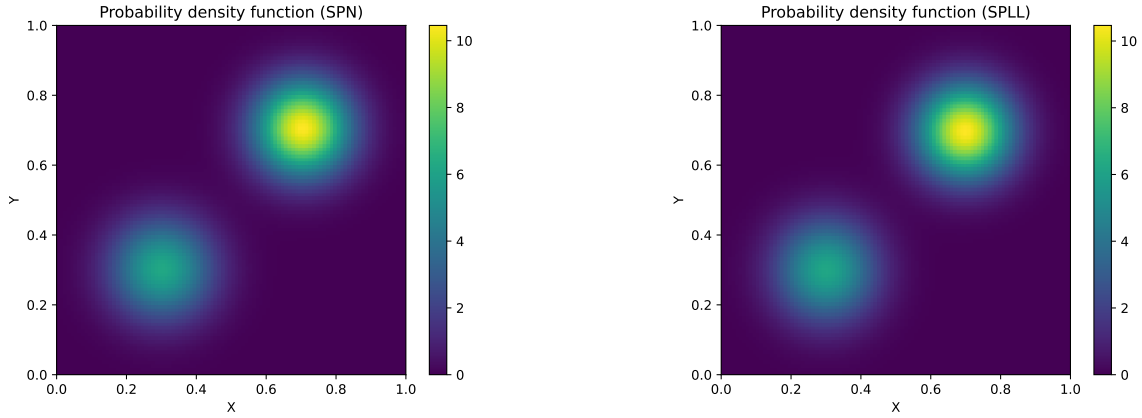


Figure 7: Comparison of the learned density functions of an SPN (left) and SPLL (right). We can see that the density functions learned by both are identical thus showing empirically that SPLLs are at least as powerful as SPNs.

such a recursive mixture model as well as a recursive simple gaussian model, we conducted experiments by sampling from the ground truth parameterization 1000 times and training a random parameterization via maximum likelihood.

The convergence behavior of these programs can be seen in Figure 8. We trace the path parameters take while the program is training, in relation to the ground truth configuration. Note that in the mixture model, most of the parameters except θ_0 go through a phase of non-monotonic behavior while the program can not provide a clear assignment of samples to mixture components. Note that due to model symmetry it is possible for θ to converge on an inverted but equivalent parameterization. This would not affect sample quality or log-likelihood.

7 Related Work

Deep probabilistic programming languages (DPPLs) such as DeepProblog (Manhaeve et al. 2018), Pyro (Bingham et al. 2019) and Edward (Tran et al. 2017) have been developed to enable users to write probabilistic programs as SPLL. However, they do not ensure tractable inference. DICE (Holtzen, Van den Broeck, and Millstein 2020) is a probabilistic programming language that can perform exact discrete probabilistic program inference. DICE supports tractable inference but unlike SPLL only on discrete, non-recursive programs. Church (Goodman et al. 2008) is a functional language for generative models. It has a similar motivation as SPLL to introduce a language to describe stochastic processes in a functional programming style. However, church has less of a focus on tractability and thus can not provide the guarantees we do around tractable programs. For both languages, continuous relaxations for training have not been proposed yet.

Another important line of research is that of logic-based probabilistic programming (Raedt, Kimmig, and Toivonen 2007; Riguzzi and Swift 2011; Fierens et al. 2014; Vlasselaer et al. 2015). In contrast to these frameworks, SPLL defines its own syntax and semantics supporting traditional programming language constructs.

Probabilistic models to perform tractable inference have been proposed such as Sum Product Network (SPNs) (Poon and Domingos 2011) and Probabilistic Sentential Decision Diagrams (PSDDs) (Kisa et al. 2014). These models are called *Probabilistic Circuits (PCs)* (Choi, Vergari, and Van den Broeck 2020). However, these models are not capable of handling structured programs that contain loops or recursion. SPLL offers the capability of tractable inference and the handling of structured programs. Sum-Product Probabilistic Language (SPPL) has been proposed as a generalization of SPNs to a programming language with an extension of symbolic expressions (Saad, Rinard, and Mansinghka 2021). SPPL achieves tractable exact inference. However, it does so by restricting the user to bounded loops.

Since we pass a gradient through a program, we can also compare our work to Continuous Relaxations (Berthet et al. 2020; Petersen et al. 2021), which smoothes over discrete control flow decisions by interpolating the outcomes. In contrast, SPLL offers exact computations, where the distribution of the generator is guaranteed to align with that of the likelihood function. Meanwhile Continuous Relaxations can lead to unintended program behavior, where the act of relaxing control flow conditions can lead to e.g. impossible paths in the program being possible. The cost of this exactness is that some programs can not be expressed in SPLL and some others scale badly in their computational cost. (Petersen et al. 2021) has proposed to use continuous relaxations to produce a gradient that resembles the original distribution, while also being much better behaved. SPLL adopts the approach to compute the gradients for parameters in places where it not otherwise possible.

Previous work has addressed the integration of symbolic programs and neural networks (Manhaeve et al. 2018; Yang, Ishay, and Lee 2020; Evans and Grefenstette 2018; Rocktäschel and Riedel 2017; Mao et al. 2019). SPLL can be placed here as a new approach to use a differentiable probabilistic programming language for neuro-symbolic learning, where the available set of computations

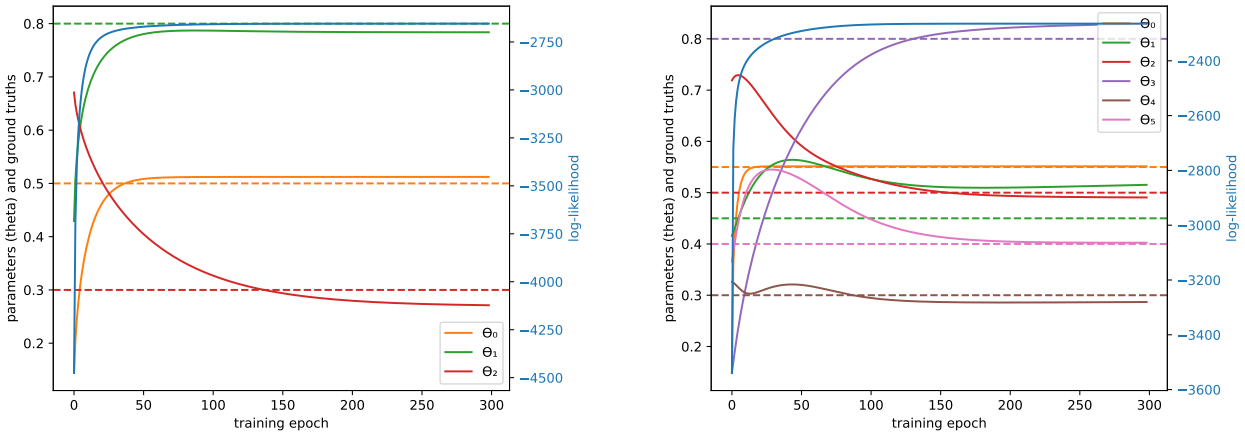


Figure 8: Training SPLL programs: Parameter convergence and likelihood fit(blue) for variable-length Gaussian sequences (left) as well as for variable length mixture-of-Gaussian sequences(right). Dashed lines indicate the ground truth of the respective parameter. Slightly off convergence points can arise from the sampling error of the training set.

on the symbolic representation receives a new degree of freedom.

8 Conclusions

Triggered by the success of probabilistic circuits such as sum-product networks, we have moved beyond circuits and introduced Sum-Product Loop Language, a novel programming language that is capable of tractable inference on complex probabilistic code that includes recursion. SPLL has dual semantics: every program has generative semantics familiar to most programmers and probabilistic semantics that assign a probability to each possible result. This way, the programmer can describe how to generate samples almost like in any standard programming language. The language takes care of computing the probability values of all results for free at run time. Overall, we showed that SPLL inherits the beneficial properties of PCs, namely tractability and differentiability while generalizing to other distributions and programs.

Our work provides several interesting avenues for future work on tractability for more complicated programs, the introduction of memory, more supported queries, and incorporating deep learning by means of neural function calls.

Tractability. Limitations of the language arise from the tractability of the probabilistic semantics: It is quite simple to produce code for which the analytical computation of the probability is impossible. For example, we might encounter problems in a program that adds two random variables: Computing the probability of the result generally involves integrating over all possible values of one variable, making even a relatively simple program intractable. Consider, however, the case of the sum of two Gaussian variables. The sum of two Gaussian-distributed variables is again Gaussian. Such an algorithm and many like it could be added to SPLL. Thus, it is possible to expand tractability beyond what our language currently offers. In such cases, identifying and pushing the boundary of intractability will be important future work.

Memory. Another interesting avenue is the introduction of memory: If we could tractably compute $p(x|\texttt{let y = e1 in e2})$, we would introduce many different interesting programs into the language. These programs could for example be made efficient if the result "witnesses" y as would be the case in $[\texttt{y}, \texttt{f(y)}]$, if we restrict the permitted subexpressions, or if the type of y is small enough to allow enumeration. However, care must be taken to not accidentally allow intractable programs. For example, a program like

```
main = let x1 = expr
  in let x2 = expr2 x1
  in expr3 x2
```

might result in evaluating a number of execution paths exponential in the number of variables x_i.

Queries. SPNs support many more queries than SPLL. Implementing these queries for the SPN-equivalent subset of the language should straightforwardly follow their SPN implementation, while for the wider language, this might offer interesting insights into degrees of tractability while making the language more usable. Instead of implementing these as different interpretation methods of the same program, a promising approach could be to implement them within the language. For example, if we refer back to the recursive lists of Gaussians in Listing 2, we might query the length of the resulting list like this:

```
query = length main >= 2
```

to find the probability that a list is at least 2 elements long. Using this approach, enabling more queries would entail expanding the expressiveness of the language fundamentally.

Neural Function Calls. In order to integrate SPLL with Neural Networks, which offer high performance in many fields of machine learning, we could introduce a new syntax element. First, we impose that compatible neural networks must output their predictions "distribution-like", e.g. using a softmax layer. The *probability* function is the output of the neural network, while the *generate* function draws a sample

from this distribution. Implementing it this way also guarantees that we can train the neural network end to end, as the gradients can be propagated to the softmax layer. We could then solve tasks such as MNIST-Addition (Manhaeve et al. 2018) as simply as:

```
main img1 img2 =
  (read_NN img1) + (read_NN img2)
```

With these additions to SPLL, it becomes feasible to tackle problems that integrate neural and symbolic representations and move into the domain of neuro-symbolic AI.

Acknowledgements

This work was supported by the Federal Ministry for Economic Affairs and Climate Action (BMWK) AI lighthouse project "SPAICER" (01MK20015E), the EU ICT-48 Network of AI Research Excellence Center "TAILOR" (EU Horizon 2020, GA No 952215), and the Collaboration Lab "AI in Construction" (AICO) with Nexplore/HochTief. The work has also benefited from the Hessian Ministry of Higher Education, Research, Science and the Arts (HMWK) cluster projects "The Third Wave of AI" and "The Adaptive Mind".

References

Berthet, Q.; Blondel, M.; Teboul, O.; Cuturi, M.; Vert, J.-P.; and Bach, F. 2020. Learning with differentiable pertubed optimizers. In *NeurIPS*.

Bingham, E.; Chen, J. P.; Jankowiak, M.; Obermeyer, F.; Pradhan, N.; Karaletsos, T.; Singh, R.; Szerlip, P.; Horsfall, P.; and Goodman, N. D. 2019. Pyro: Deep universal probabilistic programming. *The Journal of Machine Learning Research* 20(1):973–978.

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

Darwiche, A. 2003. A differential approach to inference in Bayesian networks. *JACM*.

Darwiche, A. 2009. *Modeling and reasoning with Bayesian networks*. Cambridge university press.

Evans, R., and Grefenstette, E. 2018. Learning explanatory rules from noisy data. *JAIR*.

Fierens, D.; den Broeck, G. V.; Renkens, J.; Shterionov, D.; Gutmann, B.; Thon, I.; Janssens, G.; and Raedt, L. D. 2014. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming* 15(3).

Goodman, N. D.; Mansinghka, V. K.; Roy, D. M.; Bonawitz, K. A.; and Tenenbaum, J. B. 2008. Church: a language for generative models. In *UAI*.

Holtzen, S.; Millstein, T.; and Broeck, G. V. d. 2019. Symbolic exact inference for discrete probabilistic programs. *arXiv preprint arXiv:1904.02079*.

Holtzen, S.; Van den Broeck, G.; and Millstein, T. 2020. Scaling exact inference for discrete probabilistic programs. *Proc. ACM Program. Lang.*

Kisa, D.; Van den Broeck, G.; Choi, A.; and Darwiche, A. 2014. Probabilistic sentential decision diagrams. In *KR*.

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.

Mao, J.; Gan, C.; Kohli, P.; Tenenbaum, J. B.; and Wu, J. 2019. The Neuro-Symbolic Concept Learner: Interpreting Scenes, Words, and Sentences From Natural Supervision. In *ICLR*.

Molina, A.; Vergari, A.; Stelzner, K.; Peharz, R.; Subramani, P.; Mauro, N. D.; Poupart, P.; and Kersting, K. 2019. Spflow: An easy and extensible library for deep probabilistic learning using sum-product networks.

Petersen, F.; Borgelt, C.; Kuehne, H.; and Deussen, O. 2021. Learning with algorithmic supervision via continuous relaxations. In *NeurIPS*.

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

Raedt, L. D.; Kimmig, A.; and Toivonen, H. 2007. Problog: A probabilistic prolog and its application in link discovery. In Veloso, M. M., ed., *IJCAI*.

Riguzzi, F., and Swift, T. 2011. The pita system: Tabling and answer subsumption for reasoning under uncertainty. *Theory and Practice of Logic Programming* 11(4-5):433–449.

Rocktäschel, T., and Riedel, S. 2017. End-to-end Differentiable Proving. In *NeurIPS*.

Saad, F. A.; Rinard, M. C.; and Mansinghka, V. K. 2021. Sppl: Probabilistic programming with fast exact symbolic inference. In *ICPLI*.

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

Vlasselaer, J.; Van den Broeck, G.; Kimmig, A.; Meert, W.; and De Raedt, L. 2015. Anytime inference in probabilistic logic programs with Tp-compilation. In *IJCAI*.

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