Implementing Inference Algorithms for Probabilistic Programs

Hongseok Yang

KAIST

1 Introduction

This is a lecture note from the probabilistic-programing course given at KAIST in the fall of 2017. Its goal formally describes one popular approach of implementing an inference algorithm for probabilistic programs. The approach is sometimes called evaluation-based inference, and it works for programs that might use unboundedly many random variables. The approach views an inference algorithm as a non-standard interpreter for probabilistic programs, and implements this interpreter using standard techniques from the programming-language research. In the note, we explain this approach using so called operational semantics, a formalism for specifying an interpreter for programs precisely.

2 Syntax of a Probabilistic Programming Language

We consider an expressive probabilistic programming language that supports higher-order functions, such as Anglican. The syntax of expressions (or programs) in the language is given by the following grammar:

Expressions	$e ::= c$	Constants
	$\mid x$	Variables
	$\mid (\texttt{fn}\ [x_1 \ldots x_n]\ e)$	Function Definitions
	$\mid (e_0\ e_1 \ldots e_n)$	Function Applications
	$\mid (\texttt{if}\ e_0\ e_1\ e_2)$	Conditional Expression

The first constant case includes constant values (real numbers and booleans), standard primitive operations, and non-standard operations for constructing distribution objects, sampling from such objects and conditioning with observed values. Here is the syntax for constants:

$c ::= 1.2 \mid \ldots$	Real Numbers
$\mid \texttt{true} \mid \texttt{false}$	Booleans
$\mid + \mid \ldots$	Arithmetic Operations
$\mid \texttt{and} \mid \texttt{or} \mid \ldots$	Boolean Operations
$\mid \texttt{flip} \mid \texttt{normal} \mid \texttt{poisson} \mid \ldots$	Distribution Constructor
$\mid \texttt{sample}_\alpha$	Sampling Operator
$\mid \texttt{observe}_\alpha$	Conditioning Operator

The subscript α of $\texttt{sample}_\alpha$ or $\texttt{observe}_\alpha$ is a label unique to a particular occurrence of sample and observe in a given probabilistic program. The labels identify sample and observe expressions in the program syntactically, and they are used in some of the inference algorithms, which need to distinguish different sample and observe expressions.

We use the letter d for the constants for constructing distribution objects:

$$d ::= \texttt{flip} \mid \texttt{normal} \mid \texttt{poisson} \mid \ldots$$

3 Likelihood-Weighted Importance Sampler

The likelihood-weighted importance sampler is perhaps the simplest inference algorithm. When $p(x)$ is a prior probability distribution on the latent variable $x \in X$ and $p(y|x)$ is the likelihood for the observed variable y, the likelihood-weighted importance sampler draws (independent) samples from $p(x)$ and returns them with their weights $p(y|x)$:

$$x_i \sim p(x) \text{ and } w_i = p(y|x_i) \quad \text{for all } i = 1, \ldots, N.$$

We can then use these weighted samples for estimating the expectation of a (measurable) function $f : X \to \mathbb{R}$ with respect to the posterior distribution as follows:

$$\mathbb{E}_{p(x|y)}[f(x)] \approx \sum_{i=1}^{N} \frac{w_i}{\sum_{j=1}^{N} w_j} \cdot f(x_i).$$

Intuitively implementing this likelihood-weighted importance sampler for probabilistic program is easy. We just execute a given probabilistic program multiple times. The execution is mostly standard except for three things. It starts with a 1-initialised global variable w. When we encounter the invocation of sample, we draw a sample from the distribution parameter of this invocation. When we encounter the call of observe, we compute the likelihood w' of the observed value, multiply w' with the value stored in the variable w, and update the variable w with the result of this multiplcation.

We specify this intuitive implementation formally using operational semantics from the programming-language research. The (small-step) operational semantics in this case is the binary relation $\rightsquigarrow$ on pairs of expression e and non-negative real numbers w:

$$e,\, w \rightsquigarrow e',\, w'$$

This notation says that (e, w) and (e', w') are related by the relation $\rightsquigarrow$. Intuitively it means that evaluating the expression e with the current importance weight w in one step may lead to (e', w').

We need two further concepts in order to define the $\rightsquigarrow$ relation. The first is a subclass of expressions, called *values*:

Values	$v ::= c$	Constants
	$\mid x$	Variables
	$\mid$ (`fn` $[x_1 \ldots x_n]$ e)	Function Definitions
	$\mid (d\ v_1 \ldots v_n)$	Distribution Objects

A value v is a particular kind of expression. It is constant, variable, function definition or the application of distribution contructor d to parameters that themselves are values. The last case represents a distribution object such as (`normal` 0 1) and (`flip` 0.7).

The second is the notion of *evaluation context* C that is defined by the following grammar:

Evaluation Context	$C ::= [-]$
	$\mid (v_0\ v_1 \ldots v_{m-1}\ C\ e_{m+1} \ldots e_{m+n})$
	$\mid$ (`if` $C\ e_1\ e_2$)

A evaluation context C is a certain kind of expression with one hole. We write $C[e]$ for the expression obtained by filling in the hole of C with e. For every expression e', we can always decompose e' uniquely to $C[e]$ for some context C and expression e, and this decomposition identifies which sub-expression of e' should be evaluated next: it is the inner expression e of $C[e]$. The name "evaluation context" comes from this property.

We specify the relation

$$e,\, w \rightsquigarrow e',\, w'$$

using the inference-rule notation:

$$\frac{\text{Premises}}{\text{Conclusion}}$$

This notation says that the conclusion below the bar holds if all the premises above the bar hold. Here are inference rules that define the binary relation $\rightsquigarrow$

for the likelihood-weighted importance sampler:

$$\frac{}{C[(\texttt{if true}\ e_1\ e_2)],\ w \rightsquigarrow C[e_1],\ w}$$

$$\frac{}{C[(\texttt{if false}\ e_1\ e_2)],\ w \rightsquigarrow C[e_2],\ w}$$

$$\frac{}{C[((\texttt{fn}\ [x_1\ \ldots\ x_n]\ e)\ v_1\ \ldots\ v_n)],\ w \rightsquigarrow C[e[x_1 := v_1, \ldots, x_n := v_n]],\ w}$$

$$\frac{c \notin \{\texttt{sample}, \texttt{observe}\} \qquad c' \leftarrow \text{compute\_op}(c, [v_1, \ldots, v_n])}{C[(c\ v_1\ \ldots\ v_n)],\ w \rightsquigarrow C[c'],\ w}$$

$$\frac{c' \leftarrow \text{sample\_dist}(d, [v_1, \ldots, v_n])}{C[(\texttt{sample}_\alpha\ (d\ v_1\ \ldots\ v_n))],\ w \rightsquigarrow C[c'],\ w}$$

$$\frac{w' \leftarrow \text{score\_dist}(d, [v_1, \ldots, v_n], c)}{C[(\texttt{observe}_\alpha\ (d\ v_1\ \ldots\ v_n)\ c)],\ w \rightsquigarrow C[c'],\ w \cdot w'}$$

The first two describes how to evaluate a conditional expression. The third rule says that the application of a function definition ($\texttt{fn}\ [x_1\ \ldots\ x_n]\ e$) to value arguments $v_1, \ldots v_n$ leads to the function body e with all parameter variables $x_1, \ldots, x_n$ substituted by the actual parameters $v_1, \ldots, v_n$. In the rule, we use the notation $e[x_1 := v_1, \ldots, x_n := v_n]$ to mean the result of this substitution. The fouth rule uses the operator compute_op that evaluates a primitive operator c on parameters $v_1, \ldots, v_n$. There we write $[v_1, \ldots, v_n]$ for the sequence consisting of $v_1, \ldots, v_n$. The operator may be undefined as in the case that compute_op$(3, [1, 2])$. The rule implicitly requires that this undefined case should not happen.

The last two rules are the most important, and describe the likelihood-weighted importance sampler. The rule for sample just samples a constant c' from a distribution object $(d\ v_1\ \ldots\ v_n)$; the operator sample_dist in the rule performs this sampling. The rule for observe computes the probability w' of the observed value c according to the distribution $(d\ v_1\ \ldots\ v_n)$ using th operator score_dist, and updates w by $w \cdot w'$.

We now use the $\rightsquigarrow$ relation to describe the top-level routine of the likelihood-weighted importance sampling algorithm. Assume that we are given an expression e in our probabilistic programming language, and that we would like to generate N weighted samples.

1. Run e with the initial weight 1 using $\rightsquigarrow$ until we reach a value. Repeat this for N times.
$$e,\ 1 \rightsquigarrow^*\ v_i,\ w_i \quad \text{for } i = 1, \ldots, N.$$
2. Return the sequence
$$[(v_1, w_1),\ (v_2, w_2),\ \ldots,\ (v_N, w_N)].$$

4 General Importance Sampler

Operational semantics (i.e. the $\rightsquigarrow$ relation) in the previous section is a general technique for specifying an interpreter or a runtime of a programming language in a high level. Since inference algorithms for probabilistic programs can often be understood as interpreters, they can be specified by means of operational semantics. In fact, this benefit is the reason that we formulated the likelihood-weighted importance sampler in the previous section. In the rest of this note, we describe other inference algorithms starting from a general importance sampler.

A general importance sampler uses a proposal distribution $q(x)$ that may be different from a prior $p(x)$. It draws (independent) samples $x_1, \ldots, x_N$ from q, computes their weights

$$w_i = \frac{p(x, y)}{q(x)} = \frac{p(x)}{q(x)} \cdot p(y|x) \tag{1}$$

and returns a sequence of weighted samples:

$$[(w_1, x_1), \ldots, (w_N, w_N)],$$

which can be used to estimate the expectation as in the case of the likelihood-weighted importance sampler.

The first step of implementing this general importance sampler for probabilistic programs is to pick a proposal distribution for each sample statement in a given program. Let *Dists* be the set of expressions of the form

$$(d\ v_1\ \ldots\ v_n),$$

which denotes a distribution object created by calling the constructor d on parameters $v_1, \ldots, v_n$. A proposal map M has the type

$$M : \mathit{Dists} \to \mathit{Dists},$$

and $M(d\ v_1\ \ldots\ v_n)$ specifies a proposal distribution to use for the prior $(d\ v_1\ \ldots\ v_n)$.

The second step is to change the rule for sampling as follows:

$$\frac{\begin{array}{ll}(d'\ v'_1\ \ldots\ v'_m) = M(d\ v_1\ \ldots\ v_n) & c' \leftarrow \text{sample\_dist}(d', [v'_1, \ldots, v'_m]) \\ p_0 \leftarrow \text{score\_dist}(d, [v_1, \ldots, v_n], c') & q_0 \leftarrow \text{score\_dist}(d', [v'_1, \ldots, v'_m], c')\end{array}}{C[(\texttt{sample}_\alpha\ (d\ v_1\ \ldots\ v_n))],\ w \rightsquigarrow C[c'],\ w \cdot (p_0/q_0)}$$

This rule says that a sample is drawn from a proposal distribution $(d'\ v'_1\ \ldots\ v'_m)$, which may be different from the prior distribution $(d\ v_1\ \ldots\ v_n)$. The use of a proposal, not a prior, is accounted for by the multiplication of the total weight by p_0/q_0 in the rule. The ratio p_0/q_0 corresponds to $p(x)/q(x)$ in (1).

5 Metropolis-Hastings Algorithm with Independent Proposals

Next we consider the Metropolis-Hastings algorithm (in short, MH algorithm) with so called independent proposals. This is an instatiation of the algorithm that uses the prior distribution as a proposal distribution. Thus, in this case, the acceptance ratio becomes:

$$\alpha(x, x') = \min\left\{1, \frac{p(x', y) \cdot q(x|x')}{p(x, y) \cdot q(x'|x)}\right\} = \min\left\{1, \frac{(p(x')p(y|x')) \cdot p(x)}{(p(x)p(y|x)) \cdot p(x')}\right\}$$
$$= \min\left\{1, \frac{p(y|x')}{p(y|x)}\right\}$$

That is, we just need to track the ratio of the likelihood.

We can easily implement this instantiation of the MH algorithm by reusing the $\rightsquigarrow$ relation in Section 3 but employing the top-level routine of the MH algorithm instead of the one of the importance sampler. Although there are no changes, we repeat the rules for sampling and conditioning again here:

$$\frac{c' \leftarrow \text{sample\_dist}(d, [v_1, \ldots, v_n])}{C[(\texttt{sample}_\alpha\ (d\ v_1\ \ldots\ v_n))],\ w \rightsquigarrow C[c'],\ w}$$

$$\frac{w' \leftarrow \text{score\_dist}(d, [v_1, \ldots, v_n], c)}{C[(\texttt{observe}_\alpha\ (d\ v_1\ \ldots\ v_n)\ c)],\ w \rightsquigarrow C[c'],\ w \cdot w'}$$

The top-level routine is just the one of the MH algorithm adapted for probabilistic programs. Assume that we would like to generate posterior samples for a given expression e.

1. Run $e, 1 \rightsquigarrow^* v', w'$.
2. $(v_1, w_1) \leftarrow (v', w')$.
3. Repeat the following for $n = 2, \ldots, N$:
   (a) Run $e, 1 \rightsquigarrow^* v'', w''$.
   (b) $\alpha \leftarrow \min(1, w''/w_{n-1})$.
   (c) Sample u from the uniform distribution on $[0, 1]$.
   (d) If $u \leq \alpha$, then $(v_n, w_n) \leftarrow (v'', w'')$; otherwise, $(v_n, w_n) \leftarrow (v_{n-1}, w_{n-1})$.
4. Return the sequence $[v_1, \ldots, v_N]$.

6 Lightweight Metropolis-Hastings Algorithm

We now consider a well-known instantiation of the MH algorithm for probabilistic programs that proposes a candidate new sample based on single-site update and re-execution. Although simple, this instantiation works very well for quite a few probabilistic programs and is implemented in Anglican and other popular higher-order probabilistic programming languages.

To describe this algorithm, we need a few notations. Let *Labels* be the set of labels $\alpha, \beta, \ldots$ used as annotations for the sample and observe expressions (for instance, ($\texttt{sample}_\alpha$ ($\texttt{normal}$ 0 1))). Let *Values* be the set of values v. For two sequences S_1 and S_2, we write $S_1 @ S_2$ for the concatenation of S_1 with S_2. Also, we write [] for the empty sequence, and $[a_1, \ldots, a_n]$ for the sequence consisting of $a_1, \ldots, a_n$.

The interpreter corresponding to this algorithm is more complex than all the others seen before, and keeps tracks of more information than those. It is thus specified in terms of the relation on tuples (e, w, D, S), which is again denoted by $\rightsquigarrow$:

$$e, w, D, S \rightsquigarrow e', w', D', S'$$

where D, D' are finite partial maps and S, S' sequences of the following types:

$$D, D' : \mathit{Labels} \times \mathit{Dists} \to_{\text{fin}} \mathit{Values}, \qquad S, S' : (\mathit{Labels} \times \mathit{Dists} \times \mathit{Values})^*.$$

That is, D and D' map pairs of label and distribution object to values, and S and S' are sequences of triples of label, distribution object and value. S' is always an extension of S, and this extended part records all the samples drawn during the execution of e. The maps D and D' are always the same, and they store information about values sampled in the previous iteration of the MH algorithm's loop body.

As before, we define the $\rightsquigarrow$ relation using inference rules. All the rules except those for sampling and conditioning are essentially the same as before; we just need to say that the D and S parts do not change.

$$\frac{}{C[(\texttt{if true}\ e_1\ e_2)],\ w,\ D,\ S \rightsquigarrow C[e_1],\ w,\ D,\ S}$$

$$\frac{}{C[(\texttt{if false}\ e_1\ e_2)],\ w,\ D,\ S \rightsquigarrow C[e_2],\ w,\ D,\ S}$$

$$\frac{}{C[((\texttt{fn}\ [x_1\ \ldots\ x_n]\ e)\ v_1\ \ldots\ v_n)],\ w,\ D,\ S \rightsquigarrow C[e[x_1 := v_1, \ldots, x_n := v_n]],\ w,\ D,\ S}$$

$$\frac{c \notin \{\texttt{sample}, \texttt{observe}\} \qquad c' \leftarrow \text{compute\_op}(c, [v_1, \ldots, v_n])}{C[(c\ v_1\ \ldots\ v_n)],\ w,\ D,\ S \rightsquigarrow C[c'],\ w,\ D,\ S}$$

The tricky rules are the ones for sampling and conditioning. For sampling, we have two rules, one for the case that D provides a sample and the other for the case that D does not provide a sample.

$$\frac{c' = D(\alpha, (d'\ v_1\ \ldots\ v_n)) \qquad p' \leftarrow \text{score\_dist}(d, [v_1, \ldots, v_n], c')}{C[(\texttt{sample}_\alpha\ (d\ v_1\ \ldots\ v_n))],\ w,\ D,\ S \rightsquigarrow C[c'],\ c \cdot p',\ D,\ S @ [(\alpha, d', c')]}$$

$$\frac{\begin{array}{l}\mathit{undefined} = D(\alpha, (d\ v_1\ \ldots\ v_n)) \\ c' \leftarrow \text{sample\_dist}(d, [v_1, \ldots, v_n]) \qquad p' \leftarrow \text{score\_dist}(d, [v_1, \ldots, v_n], c')\end{array}}{C[(\texttt{sample}_\alpha\ (d\ v_1\ \ldots\ v_n))],\ w,\ D,\ S \rightsquigarrow C[c'],\ w \cdot p',\ D,\ S @ [(\alpha, d', v)]}$$

$$\frac{w' \leftarrow \text{score\_dist}(d, [v_1, \ldots, v_n], c)}{C[(\texttt{observe}_\alpha\ (d\ v_1\ \ldots\ v_n)\ c)],\ w,\ D,\ S \rightsquigarrow C[c'],\ w \cdot w', D,\ S}$$

For each sequence $S \in (\mathit{Labels} \times \mathit{Dists} \times \mathit{Values})^*$, we can take the length-n prefix S_n of S, and convert S_n to a map $D_n : \mathit{Labels} \times \mathit{Dists} \to_{\mathrm{fin}} \mathit{values}$ using the most recent value in case of duplications. We denote this map by $\mathrm{map}(S, n)$.

The top-level algorithm is essentially the same as before, except for two things: we use the new $\rightsquigarrow$ relation, and the acceptance ratio is changed. Assume that we would like to generate posterior samples for a given expression e. Let $\emptyset$ be the empty finite partial map.

1. Run $e, 1, \emptyset, [] \rightsquigarrow^* v', w', D', S'$.
2. $(v_1, w_1, S_1) \leftarrow (v', w', S')$.
3. Repeat the following for $n = 2, \ldots, N$:
   (a) Sample l uniformly from $\{1, \ldots, |S_{n-1}|\}$.
   (b) $D \leftarrow \mathrm{map}(S_{n-1}, l)$.
   (c) Run $(e, 1, D, []) \rightsquigarrow^* (v', w', D', S')$.
   (d) $\alpha \leftarrow \min(1, (w' \cdot |S_{n-1}|)/(w_{n-1} \cdot |S'|))$.
   (e) Sample u from the uniform distribution on $[0, 1]$.
   (f) If $u \leq \alpha$, then $(v_n, w_n, S_n) \leftarrow (v', w', S')$; otherwise, $(v_n, w_n, S_n) \leftarrow (v_{n-1}, w_{n-1}, S_{n-1})$.
4. Return the sequence $[v_1, \ldots, v_N]$.