Implementing Inference Algorithms for Probabilistic Programs

Hongseok Yang

KAIST

1 Introduction

This is one of lecture notes from the probabilistic-programming course given at KAIST in the spring of 2019. Its goal is to describe formally 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
$\mid \texttt{nil}$	Nil
$\mid \texttt{and} \mid \texttt{or} \mid \ldots$	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 occurrences of sample and observe.

We use the letter d for the operators 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 x_i from $p(x_i)$ and returns them with their weights $w_i = p(y|x_i)$:

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

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:

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

Implementing this likelihood-weighted importance sampler for probabilistic programs is easy. We just execute a given probabilistic program multiple times. The execution is mostly standard except for three things. First, it starts with a 1-initialised global variable w. Second, when we encounter the invocation of sample, we draw a sample from the distribution parameter of this invocation. Third, 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 multiplication.

We formalise this description using small-step operational semantics from the programming-language research. The operational semantics in this case refers to 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 for one step may transform e to e' and update w with 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 a constant, a variable, a function definition, or the application of a distribution constructor d to value parameters. The last case represents a distribution object such as (`normal` 0 1) and (`flip` 0.7).

The second is the notions of *redex* r and *reduction context* C that are defined by the following grammar:

Redex	$r ::= (v_0\, v_1 \ \ldots \ v_m)$	
	$\mid$ (`if` $v\ e_1\ e_2$)	
Reduction 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 redex r is an expression that represents unfinished computation and can be reduced or transformed at least for one step by the runtime. The word "redex" is an abbreviation of "reducible expression". A reduction 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' without any free variables, there is at most one way to split $e' = C[r]$ into a reduction context C and a redex r. This decomposition identifies which sub-expression of e' should be evaluated or reduced next: it is the inner expression e of $C[e]$. The name "reduction context" comes from this property. When the splitting fails, e' is a value representing the final outcome of computation.

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. We have one inference rule for the binary relation $\rightsquigarrow$:

$$\frac{r,\, w \Longrightarrow e',\, w'}{C[r],\, w \rightsquigarrow C[e'],\, w'}$$

which is defined in terms of the relation $\Longrightarrow$ between redexes and expressions:

$$\frac{}{(\texttt{if true}\ e_1\ e_2),\, w \Longrightarrow e_1,\, w}$$

$$\frac{v \not\equiv \texttt{true}}{(\texttt{if}\ v\ e_1\ e_2),\, w \Longrightarrow e_2,\, w}$$

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

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

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

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

$$\frac{(v_0\ v_1\ \ \ldots\ \ v_n) \text{ is not one of the cases from above}}{(v_0\ v_1\ \ \ldots\ \ v_n),\, w \Longrightarrow \texttt{nil},\, w}$$

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 fourth 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 next 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 the operator score_dist, and updates w by $w \cdot w'$.

The last rule covers errorneous computation. Our rule says that when faced an error, the expression return `nil` instead of terminating the computation and signaling an error. We adopted this slightly ugly choice in order to simplify our formalisation of inference algorithms.

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 using operational semantics in the previous section.

In the rest of this note, we describe other inference algorithms using operational semantics. We start with 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

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

and returns a sequence of weighted samples:

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

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 *Labels* be the set of labels α and β, which are used to annotate sample and observe expressions, as in $\texttt{sample}_\alpha$ and $\texttt{observe}_\beta$. Let *Dists* be the set of expressions of the form

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

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

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

and $M(\alpha, (d\ v_1\ \ldots\ v_n))$ specifies a proposal distribution to use for the prior $(d\ v_1\ \ldots\ v_n)$ sampled by the α-labelled sample expression.

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

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

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 multiplying the total weight w with 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 instantiation of the algorithm that uses the prior distribution as a proposal distribution. Thus, in this case, the acceptance ratio becomes:

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

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

We can easily implement this instantiation of the MH algorithm by reusing the $\rightsquigarrow$ and $\Longrightarrow$ relations 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 here:

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

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

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 *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 $a_1, \ldots, a_n$. Finally, for a sequence S and a number $m \leq |S|$, we write $\text{prefix}(S, m)$ for the length-m prefix of S.

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, R, S), which is again denoted by $\rightsquigarrow$:

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

where R, R', S, S' are sequences of values:

$$R, R', S, S' : \mathit{Values}^*.$$

The sequence R records some of random choices that have been made during the previous iteration of the MH algorithm. Thus, it is fixed before the execution of the expression e. Some or none of

these choices is used during the one-step execution of e, and the unused part of R is kept in R'. Note that R' is, therefore, a suffix of R. The sequence S' is always an extension of S, and this extended part keeps all the samples drawn during the one-step execution of e.

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 R and S parts do not change.

$$\frac{}{(\texttt{if true}\ e_1\ e_2),\ w,\ R,\ S \Longrightarrow e_1,\ w,\ R,\ S}$$

$$\frac{v \not\equiv \texttt{true}}{(\texttt{if}\ v\ e_1\ e_2),\ w,\ R,\ S \Longrightarrow e_2,\ w,\ R,\ S}$$

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

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

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

$$\frac{}{(\texttt{sample}_\alpha\ (d\ v_1\ \ldots\ v_n)),\ w,\ [c']@R',\ S \Longrightarrow c',\ w,\ R',\ S@[c']}$$

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

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

The last rule for covering the error case is the same as before.

$$\frac{(v_0\ v_1\ \ldots\ v_n) \text{ is not one of the cases from above}}{(v_0\ v_1\ \ldots\ v_n),\ w \Longrightarrow \texttt{nil},\ w}$$

The top-level algorithm is essentially the same as before, except for two changes. First, we use the new $\rightsquigarrow$ relation. Second, the acceptance ratio is changed. Assume that we would like to generate posterior samples for a given expression e.

1. Run $e, 1, [], [] \rightsquigarrow^* v', w', R', S'$.
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 $\{0, \ldots, |S_{n-1}| - 1\}$.
   (b) $R \leftarrow \text{prefix}(S_{n-1}, l)$.
   (c) Run $e, 1, R, [] \rightsquigarrow^* v', w', R', S'$.
   (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 $[v_1, \ldots, v_N]$.

Question 1. Why is the acceptance ratio correct? Can you argue semi-formally that the ratio is in fact an instance of the acceptance ratio of the MH algorithm?

7 Sequential Monte Carlo

Our next example is the Sequential Monte Carlo (SMC) algorithm. This algorithm and its recent variants (such as particle cascade and interactive particle MCMC) have been implemented in Anglican and other probabilistic programming languages. These variants of SMC are usually considered the state-of-the-art techniques for performing posterior inference for probabilistic programs.

In this section, we pick one of the most basic versions of the SMC algorithm, which even omits one of the key aspects of SMC, namely, the estimation of marginal likelihood. We make this choice so as to clarify the algorithmic aspect of SMC, such as the use of concurrency.

Just like what we have done so far, we specify SMC by means of one-step execution relations $\rightsquigarrow$ and $\Longrightarrow$, and a top-level routine. Let us start with the $\Longrightarrow$ relation. The definition of the other $\rightsquigarrow$ remains the same. The $\Longrightarrow$ relation is defined almost the same way as that for general importance sampling. The only difference is that in SMC, we do not include any inference rule for observe expression.

Omitting the case for observe means that when the next thing to evaluate is an observe expression, the $\rightsquigarrow$ relation does not suggest what to do. As a result, our notion of values becomes more permissive and includes this case involving observe:

Values	$v ::= c$	Constants
	$\mid x$	Variables
	$\mid (\texttt{fn}\ [x_1\ \ldots\ x_n]\ e)$	Function Definitions
	$\mid (d\ v_1\ \ldots\ v_n)$	Distribution Objects
	$\mid C[(\texttt{observe}_\alpha\ (d\, v_1, \ldots, v_n)\ c)]$	Conditioning

The last case in this grammar is new. It describes expressions where the next thing to evaluate is an observe expression.

For completeness, we repeat the inference rules for the $\Longrightarrow$ relation for SMC:

$$\frac{}{(\texttt{if true}\ e_1\ e_2),\ w \Longrightarrow e_1,\ w}$$

$$\frac{v \not\equiv \texttt{true}}{(\texttt{if}\ v\ e_1\ e_2),\ w \Longrightarrow e_2,\ w}$$

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

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

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

$$\frac{(v_0\ v_1\ \ldots\ v_n) \text{ is not one of the cases from above}}{(v_0\ v_1\ \ldots\ v_n),\ w \Longrightarrow \texttt{nil},\ w}$$

We now move on to the top-level routine. This routine is the most creative part of the SMC algorithm. Suppose that we are told to generate samples of the posterior distribution of an expression e. Intuitively, the SMC algorithm works by running multiple importance samplers on e simultaneously while coordinating them from time to time. Let N be a positive integer that specifies the number of importance samplers, and `discrete` be a constructor for discrete distributions—$(\texttt{discrete}\ n_1\ \ldots\ n_m)$ represents a distribution on $\{1, \ldots, n\}$ that assigns the probability $n_i / \sum_{j=1}^{m} n_m$ to each i. Assume that every execution of the expression e encounters the same number of observe expressions.

1. $E[i] \leftarrow e$ for $i = 1, \ldots, N$.
2. Repeat the following.
   (a) For $i = 1, \ldots, N$,
      i. run $(E[i], 1) \rightsquigarrow^* (v', w')$;
      ii. $(E[i], W[i]) \leftarrow (v', w')$.
   (b) If none of $E[i]$ has the form $C_i[(\texttt{observe}_\alpha \ \ldots)]$, then return $[(E[1], W[1]), \ldots, (E[N], W[N])]$.
   (c) If some $E[i]$ has the form $C_i[(\texttt{observe}_\alpha \ \ldots)]$ but some doesn't, then raise the incorrect program exception.
   (d) If every $E[i]$ has the form $C_i[(\texttt{observe}_\alpha \ (d_i \ v_{i,1} \ \ldots \ v_{i,n_i}) \ c'_i)]$, then
      i. $W[i] \leftarrow W[i] \times \text{score\_dist}(d_i, [v_{i,1}, \ldots, v_{i,n_i}], c'_i)$ for all $i = 1, \ldots, N$;
      ii. $A[i] \leftarrow \text{sample\_dist}(\texttt{discrete}, [W[1], \ldots, W[N]])$ for all $i = 1, \ldots, N$;
      iii. $(E[1], \ldots, E[N]) \leftarrow (C_{A[1]}[c'_{A[1]}], \ldots, C_{A[N]}[c'_{A[N]}])$.

This algorithm runs the given expression e with N threads according to the rules of general importance sampler, until these threads reach an observe expression or complete their execution. In the latter case, it returns weighted samples of these N threads. In the former case, it updates the weights of the threads with the observe expression, resamples these threads according to their weights, replaces the observe expression of each resampled thread by the observed value, and then continues the executions of these threads. Intuitively, this resampling kills executions whose random choices so far do not match past observations well, and clones executions that have the opposite property.

8 Baby Black-box Variational Inference

Let us start with a simple version of black-box variational inference (BBVI). Because of its naivety, we call it baby BBVI. But we want to emphasise that this baby BBVI is very close to the implementation of BBVI in Anglican.

Recall our notational convention. We use d for distribution constructors, and α for labels attached to sample and observe operators appearing in a given program. Of course, this means that our presentation assumes a fixed probabilistic program as in the previous sections.

One key data structure of baby BBVI is a map P from (α, d) pairs to $(d', [\theta_1, \ldots, \theta_m])$ pairs, where $\theta_1, \ldots, \theta_m$ are specific parameters to the distribution constructor d', that is, $(d' \ \theta_1 \ \ldots \ \theta_m)$ is a distribution object. Intuitively,

$$P(\alpha, d) = (d', [\theta_1, \ldots, \theta_m])$$

instructs us to use the distribution object $(d' \ \theta_1 \ \ldots \ \theta_m)$ whenever the program execution encounters the sampling operator labelled with α and applied to a distribution object of the form $(d \ \ldots)$. In most cases, d' is morally the same as d; it just uses a different parameterisation from d's so as to ensure that each parameter of d' may range over the unconstrained $\mathbb{R}^k$ for some k. For instance, when $d = \texttt{normal}$, we commonly use d' that describes the normal distribution but has the mean parameter θ_1 and the log-standard-deviation parameter θ_2. Thus,

$$\text{score\_dist}(d', [\theta_1, \theta_2], c) = \text{score\_dist}(d, [\theta_1, \exp(\theta_2)], c).$$

Also note that θ_2 is unconstrained unlike the usual standard deviation parameter and may have any real number. This d' part is fixed during inference. But the remaining $[\theta_1, \ldots, \theta_m]$ part changes. In fact, the purpose of inference is to find the most appropriate $[\theta_1, \ldots, \theta_m]$.

The operational semantics of baby BBVI is given by the following transition relation:

$$e, w, G \rightsquigarrow_P e', w', G'$$

where w is a non-negative real and G is a function mapping (α, d) to a parameter for the distribution constructor d' in $(d', [\theta_1, \ldots, \theta_n]) = P(\alpha, d)$. This parameter should be understood as

the gradient of the sum of the log densities of all sampled values c' related to α and d. The top-level algorithm works by gathering such gradient information by running e repeatedly with $\rightsquigarrow_P$, updating P with that information, and repeating this process multiple times.

The rule for the $\rightsquigarrow_P$ relation is the same as before. But it uses $\Longrightarrow_P$ that has a different definition. Although most rules for $\Longrightarrow_P$ are standard, the ones for sample and observe are not and they implement the core of the BBVI algorithm:

$$\frac{\begin{array}{ll}(d',\,[\theta_1,\ldots,\theta_m]) = P(\alpha,\,d) & c' \leftarrow \text{sample\_dist}(d',\,[\theta_1,\ldots,\theta_m]) \\ q_0 \leftarrow \text{score\_dist}(d',\,[\theta_1,\ldots,\theta_m],\,c') & g_0 \leftarrow \text{grad\_log\_score\_dist}(d',\,[\theta_1,\ldots,\theta_m],\,c') \\ p_0 \leftarrow \text{score\_dist}(d,\,[v_1,\ldots,v_n],\,c') & G' \leftarrow G[(\alpha,d) \mapsto G[(\alpha,d)] + g_0]\end{array}}{(\texttt{sample}_\alpha\ (d\ v_1\ \ldots\ v_n)),\,w,\,G \Longrightarrow_P c',\,w \cdot p_0/q_0,\,G'}$$

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

In the rule for sample, we use the grad_log_score_dist operator. It takes a tuple $(d',\,[\theta_1,\ldots,\theta_m],\,c')$, and returns the gradient of the log density of the distribution object $(d'\ \theta_1\ \ldots\ \theta_m)$ at the sampled value c'.

The top-level algorithm works as follows:

1. Initialize P using appropriate distribution constructors but choosing their initial parameters reasonably randomly.
2. Repeat the following for N times.
   (a) $G_0 \leftarrow \lambda(\alpha, d).\,0$.
   (b) $(e,\,1,\,G_0) \rightsquigarrow_P^* (v_k,\,w_k,\,G_k)$ for $k = 1,\ldots,K$.
   (c) $P \leftarrow \lambda(\alpha, d).\left(d',\,g' + \eta \cdot \sum_{k=1}^{K} \log(w_k) \cdot G_k(\alpha, d)/K\right)$ where $(d',\,g') = P(\alpha,\,d)$.
3. Return P.