Auxiliary variables in Probabilistic Programs

Ekansh Sharma   Daniel M. Roy

University of Toronto

January 9, 2018

Motivation

In existing probabilistic programming languages (PPLs), programmers represent probability models by:

Motivation

In existing probabilistic programming languages (PPLs), programmers represent probability models by:

- Giving an exact sampler for the prior distribution, using probabilistic primitive `sample`

Motivation

In existing probabilistic programming languages (PPLs), programmers represent probability models by:

- Giving an exact sampler for the prior distribution, using probabilistic primitive `sample`
- Weighing the prior by data likelihood, using probabilistic primitive `observe` or `score`

Motivation

In existing probabilistic programming languages (PPLs), programmers represent probability models by:

- Giving an exact sampler for the prior distribution, using probabilistic primitive `sample`
- Weighing the prior by data likelihood, using probabilistic primitive `observe` or `score`

We know, this approach is limiting!

Motivation

In existing probabilistic programming languages (PPLs), programmers represent probability models by:

- Giving an exact sampler for the prior distribution, using probabilistic primitive `sample`
- Weighing the prior by data likelihood, using probabilistic primitive `observe` or `score`

We know, this approach is limiting!

- Certain computable distributions do not have an exact sampling representation
  - Aldous–Hoover representation for an exchangeable array is not necessarily computable (Ackerman et al., 2017)

Motivation

In existing probabilistic programming languages (PPLs), programmers represent probability models by:

- Giving an exact sampler for the prior distribution, using probabilistic primitive `sample`
- Weighing the prior by data likelihood, using probabilistic primitive `observe` or `score`

We know, this approach is limiting!

- Certain computable distributions do not have an exact sampling representation
  - Aldous–Hoover representation for an exchangeable array is not necessarily computable (Ackerman et al., 2017)
- In many cases, it might be difficult to come up with an exact sampling representation.

Motivation

In existing probabilistic programming languages (PPLs), programmers represent probability models by:

- Giving an exact sampler for the prior distribution, using probabilistic primitive `sample`
- Weighing the prior by data likelihood, using probabilistic primitive `observe` or `score`

We know, this approach is limiting!

- Certain computable distributions do not have an exact sampling representation
  - Aldous–Hoover representation for an exchangeable array is not necessarily computable (Ackerman et al., 2017)
- In many cases, it might be difficult to come up with an exact sampling representation.

Contribution

Language construct—`slice-let`—that allows programmer to specify a distribution without giving an exact sampler

Main Idea

Instead of specifying an exact sampler for a random variable x, introduce an auxiliary (random) variable u.
Programmer provides an exact representations for:

- conditional distribution u given x
- conditional distribution x given u

Main Idea

Instead of specifying an exact sampler for a random variable x, introduce an auxiliary (random) variable u.
Programmer provides an exact representations for:

- conditional distribution u given x
- conditional distribution x given u

These two conditional distributions can be used to generate "exact–approximate" inference algorithms, in particular, algorithms called *slice samplers*.

Main Idea

Instead of specifying an exact sampler for a random variable x, introduce an auxiliary (random) variable u.
Programmer provides an exact representations for:

- conditional distribution u given x
- conditional distribution x given u

These two conditional distributions can be used to generate "exact–approximate" inference algorithms, in particular, algorithms called *slice samplers*.

Outline

- Syntax for `slice-let` in the first order language by Staton et al. (2016).
- Measure theoretic semantics for `slice-let`.
- Simple program transformation for an approximate inference algorithm.

First Order Language (Staton et al., 2016)

Syntax

Types:

$$\mathbb{A}, \mathbb{B} ::= \mathbb{R} \mid 1 \mid \mathrm{P}(\mathbb{A}) \mid \mathbb{A} \times \mathbb{B} \mid \sum_{i \in I} \mathbb{A}_i$$

Deterministic Terms:

$$a, b, c ::= x \mid * \mid (a, b) \mid (i, a) \mid \pi_j(a) \mid \texttt{case}\ \ a\ \texttt{of}\ \ \{(i, x) \Rightarrow c_i\}_{i \in I} \mid f(a) \mid \texttt{normalize}(t)$$

Probabilistic Terms:

$$t, u, v ::= \texttt{sample}(a) \mid \texttt{score}(a) \mid \texttt{case}\ \ a\ \texttt{of}\ \ \{(i, x) \Rightarrow t_i\}_{i \in I} \mid \texttt{let}\ \ x = t\ \texttt{in}\ \ u \mid \texttt{return}(a)$$

Program:

$$\mathcal{P} ::= t$$

First Order Language (Staton et al., 2016)

Measure theoretic denotational semantics

1. Types are interpreted as measurable spaces $[\![\mathbb{A}]\!]$
2. Context $\Gamma = (x_1 : \mathbb{A}_1, \ldots, x_n : \mathbb{A}_n)$ interpreted as measurable space, $[\![\Gamma]\!] = \prod_{i=1}^{n} [\![\mathbb{A}_i]\!]$
3. Deterministic terms are interpreted as measurable functions $[\![\Gamma \vdash_d a : \mathbb{A}]\!] : [\![\Gamma]\!] \to [\![\mathbb{A}]\!]$
4. Probabilistic terms are interpreted as sigma finite kernels $[\![\Gamma \vdash_p t : \mathbb{A}]\!] : [\![\Gamma]\!] \times \Sigma_{[\![\mathbb{A}]\!]} \to [0, \infty]$, where $\Sigma_{[\![\mathbb{A}]\!]}$ denotes the σ-algebra on $[\![\mathbb{A}]\!]$

slice-let: Typing Rules

$$\frac{\Gamma \vdash_p t_0 : \mathbb{A}_2 \qquad \Gamma, u : \mathbb{A}_2 \vdash_p t_1 : \mathbb{A}_1 \qquad \Gamma, x : \mathbb{A}_1 \vdash_p t_2 : \mathbb{A}_2 \qquad \Gamma, x : \mathbb{A}_1, u : \mathbb{A}_2 \vdash_p t_3 : \mathbb{B}}{\Gamma \vdash_p \texttt{slice-let}\ u = t_0\ \texttt{with}\ x = t_1\ \texttt{update}\ u = t_2\ \texttt{in}\ t_3 : \mathbb{B}}$$

- $\Gamma \vdash_p t_0 : \mathbb{A}_2$ specifies the initial distribution for auxiliary variable u, which may not be it's marginal distribution
- $\Gamma, u : \mathbb{A}_2 \vdash_p t_1 : \mathbb{A}_1$ specifies the conditional distribution for random variable x given u
- $\Gamma, x : \mathbb{A}_1 \vdash_p t_2 : \mathbb{A}_2$ specifies the conditional distribution for random variable u given x

`slice-let`: Denotational semantics

`slice-let`: Denotational semantics

Semantics for `let` term is given by:

$$[\![\Gamma \vdash_p \texttt{let } x = t_1 \texttt{ in } t_2 : \mathbb{A}]\!](\gamma)(A) = \int [\![t_2]\!](\gamma, x)(A)[\![t_1]\!](\gamma)(dx),$$

where $A \in \Sigma_{[\![\mathbb{A}]\!]}$.

`slice-let`: Denotational semantics

Semantics for `let` term is given by:

$$[\![\Gamma \vdash_p \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \mathbb{A}]\!](\gamma)(A) = \int [\![t_2]\!](\gamma, x)(A)[\![t_1]\!](\gamma)(dx),$$

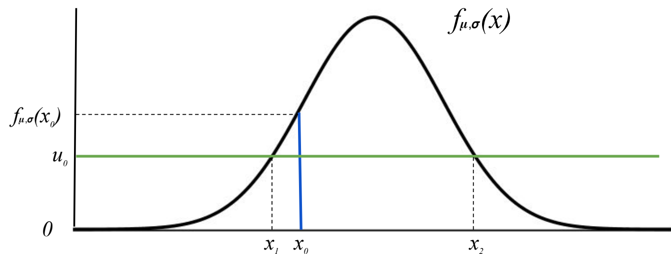
where $A \in \Sigma_{[\![\mathbb{A}]\!]}$.
Semantics for `slice-let` term, similar to `let` term, is given by:

$$\begin{aligned}[\![\texttt{slice-let}\ u = t_0\ \texttt{with}\ x = t_1\ \texttt{update}\ u = t_2\ \texttt{in}\ t_3]\!](\gamma)(A) \\ = \int [\![t_3]\!](\gamma, x, u)(A)\mu(dx, du)\end{aligned}$$

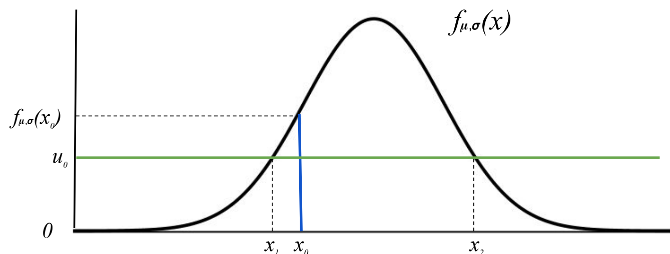
where $\mu : \Sigma_{[\![\mathbb{A}]\!]} \otimes \Sigma_{[\![\mathbb{R}_+]\!]} \to [0, 1]$ is the unique measure such that, for all bounded measurable functions f and for all γ

$$\begin{aligned}\iint f(x', u)[\![t_1]\!](\gamma, u)(dx')\mu(dx, du) &= \iint f(x, u')[\![t_2]\!](\gamma, x)(du')\mu(dx, du) \\ &= \int f(x, u)\mu(dx, du).\end{aligned}$$

Example: Slice sampling Gaussian (Neal, 2003)



Example: Slice sampling Gaussian (Neal, 2003)



For an auxiliary variable representation, programmer needs to specify the following:

- $u|x \sim \mathrm{Uniform}[0, f_{\mu,\sigma}(x)]$, where $f_{\mu,\sigma}$ is the density of a Gaussian with mean μ and standard deviation σ
- $x|u \sim \mathrm{Uniform}\{y \mid f_{\mu,\sigma}(y) > u\}$

Example: Slice sampling Gaussian (Neal, 2003)

Program with slice-let

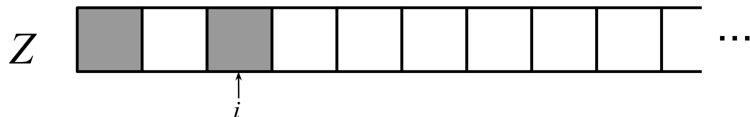
$$\texttt{slice-let}\ u = \texttt{sample}(\mathrm{Uniform}[0, f_{\mu,\sigma}(0)])$$
$$\quad \texttt{with}\ x = \texttt{sample}(\mathrm{Uniform}\{y : u < f_{\mu,\sigma}(y)\})$$
$$\quad \texttt{update}\ u = \texttt{sample}(\mathrm{Uniform}[0, f_{\mu,\sigma}(x)])$$
$$\quad \texttt{in}\ \texttt{return}(x)$$

Example: Bernoulli Process with decaying weights

Let $Z = (z_1, z_2, ...)$ be an infinite dimensional $\{0, 1\}$-valued vector, such that

$$z_i \stackrel{ind}{\sim} \mathrm{Bernoulli}(2^{-i})$$

Interested in representing Z and the last *active* index of Z

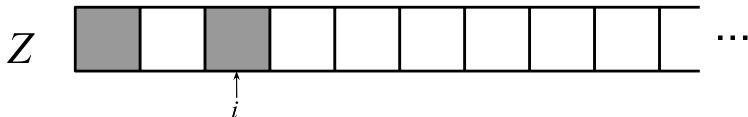


Example: Bernoulli Process with decaying weights

Let $Z = (z_1, z_2, \ldots)$ be an infinite dimensional $\{0, 1\}$-valued vector, such that

$$z_i \overset{ind}{\sim} \mathrm{Bernoulli}(2^{-i})$$

Interested in representing Z and the last *active* index of Z



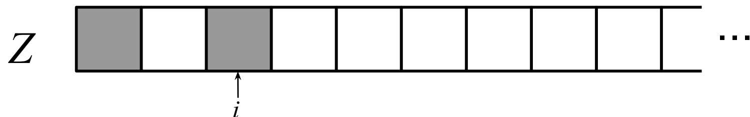
Problem: `last-active-index`: $\{0, 1\}^{\mathbb{N}} \to \mathbb{Z}$ is not continuous.

Example: Bernoulli Process with decaying weights

Let $Z = (z_1, z_2, ...)$ be an infinite dimensional $\{0,1\}$-valued vector, such that

$$z_i \overset{ind}{\sim} \text{Bernoulli}(2^{-i})$$

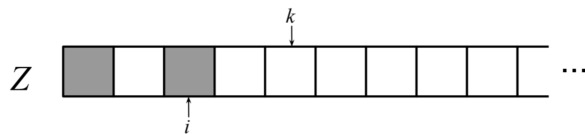
Interested in representing Z and the last *active* index of Z



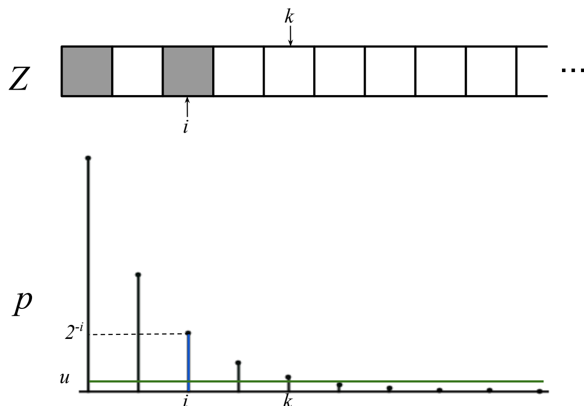
Problem: `last-active-index`: $\{0,1\}^{\mathbb{N}} \to \mathbb{Z}$ is not continuous.

- Want a representation of Z and the last active index that exposes the conditional independence in Z

Example: Bernoulli Process with decaying weights



Example: Bernoulli Process with decaying weights



Introduce an auxiliary variable u such that

- $u|Z, i \sim \mathrm{Uniform}[0, 2^{-i}]$, where i is the last active index
- $p(Z, i|u) \propto p(Z)\frac{1}{2^{-i}}\mathbb{I}(0 \leq u \leq 2^{-i})$

In effect, given u, $\forall j : 2^{-j} < u \implies z_j = 0$. Thus, only need to sample Z up to an index k, given by u.

Program Transformation

Program Transformation for Inference via `unfold`

For the term $\phi := \texttt{slice-let}\ u = t_0\ \texttt{with}\ x = t_1\ \texttt{update}\ u = t_2\ \texttt{in}\ t_3$,
Add syntactic sugar `unfold`, defined inductively as follows:

$$
\begin{aligned}
\texttt{unfold}^1\ \phi :=& \texttt{let}\ u = t_0\ \texttt{in}\ \texttt{let}\ x = t_1\ \texttt{in}\ \texttt{return}(x, u) \\
\texttt{unfold}^{n+1}\ \phi :=& \texttt{let}\ (x, u) = \texttt{unfold}^n\ v\ \texttt{in} \\
& \qquad \texttt{let}\ u = t_2\ \texttt{in} \\
& \qquad\quad \texttt{let}\ x = t_1\ \texttt{in}\ \texttt{return}(x, u)
\end{aligned}
$$

Program Transformation for Inference via unfold

For the term $\phi := \texttt{slice-let}\ u = t_0\ \texttt{with}\ x = t_1\ \texttt{update}\ u = t_2\ \texttt{in}\ t_3$,
Add syntactic sugar unfold, defined inductively as follows:

$$\begin{aligned}
\texttt{unfold}^1\ \phi &:= \texttt{let}\ u = t_0\ \texttt{in}\ \texttt{let}\ x = t_1\ \texttt{in}\ \texttt{return}(x, u) \\
\texttt{unfold}^{n+1}\ \phi &:= \texttt{let}\ (x, u) = \texttt{unfold}^n\ v\ \texttt{in} \\
&\qquad\qquad \texttt{let}\ u = t_2\ \texttt{in} \\
&\qquad\qquad\quad \texttt{let}\ x = t_1\ \texttt{in}\ \texttt{return}(x, u)
\end{aligned}$$

Sequence of terms $\{\phi_n\}_{n\in\mathbb{N}}$ by applying the unfold transformation:

$$\phi_n := \texttt{let}\ (x, u) = \texttt{unfold}^n\ \phi\ \texttt{in}\ t_3$$

Assumption

For the term,
$\phi := \texttt{slice-let}\ u = t_0\ \texttt{with}\ x = t_1\ \texttt{update}\ u = t_2\ \texttt{in}\ t_3$,
we assume that the terms t_0, t_1, and t_2 defined by the programmer satisfy following conditions:

1. $[\![t_1]\!]$ and $[\![t_2]\!]$ are *probability* kernels whose sequential composition define an ergodic Markov chain with a unique stationary distribution μ.

2. $[\![t_0]\!](\gamma)$ is a probability measure, such that $[\![t_0]\!](\gamma) \ll \mu$.

Assumption

For the term,
$\phi :=$ `slice-let` $u = t_0$ `with` $x = t_1$ `update` $u = t_2$ `in` t_3,
we assume that the terms t_0, t_1, and t_2 defined by the programmer satisfy following conditions:

1. $[\![t_1]\!]$ and $[\![t_2]\!]$ are *probability* kernels whose sequential composition define an ergodic Markov chain with a unique stationary distribution μ.
2. $[\![t_0]\!](\gamma)$ is a probability measure, such that $[\![t_0]\!](\gamma) \ll \mu$.

Now, Markov chain theory tells us

$$\forall \gamma, \lim_{n \to \infty} \| [\![\texttt{unfold}^n\ \phi]\!](\gamma) - \mu \|_{TV} = 0.$$

Assumption

For the term,
$\phi := \texttt{slice-let}\ u = t_0\ \texttt{with}\ x = t_1\ \texttt{update}\ u = t_2\ \texttt{in}\ t_3$,
we assume that the terms t_0, t_1, and t_2 defined by the programmer satisfy following conditions:

1. $[\![t_1]\!]$ and $[\![t_2]\!]$ are *probability* kernels whose sequential composition define an ergodic Markov chain with a unique stationary distribution μ.
2. $[\![t_0]\!](\gamma)$ is a probability measure, such that $[\![t_0]\!](\gamma) \ll \mu$.

Now, Markov chain theory tells us

$$\forall \gamma, \lim_{n \to \infty} \| [\![\texttt{unfold}^n\ \phi]\!](\gamma) - \mu \|_{TV} = 0.$$

Question: Does the semantics of the program converge when we apply this transformation?

Convergence Result

For the term

$\phi := \texttt{slice-let}\ u = t_0\ \texttt{with}\ x = t_1\ \texttt{update}\ u = t_2\ \texttt{in}\ t_3$,

Theorem

Let $\mathcal{P}$ be a program consisting of a `slice-let` *term ϕ. Let $\{\phi_n\}$ be a sequence of terms obtained by applying the* `unfold` *transformation, such that replacing ϕ with ϕ_n in $\mathcal{P}$ gives a program $\mathcal{P}_n$. If the following conditions hold:*

1. $[\![\Gamma \vdash_p \mathcal{P} : \mathbb{A}]\!](\gamma)([\![\mathbb{A}]\!]) < \infty$
2. *If ϕ appears inside a term* $\texttt{normalize}(t')$, *then* $[\![\Gamma \vdash_p t' : \mathbb{A}]\!](\gamma)([\![\mathbb{A}]\!]) < \infty$
3. *If ϕ appears inside $f(a)$, then f is a* **continuous** *function*

then, $[\![\mathcal{P}_n]\!](\gamma)$ converges strongly to $[\![\mathcal{P}]\!](\gamma)$.

Summary

- Probabilistic primitives, `sample`, `score`, and `normalize` are not enough for all probability models
- Introduced `slice-let` construct that enables programmer to give exact–approximate representations
- Discussed a program transformation for an inference algorithm.

Appendix: Bernoulli Process with decaying weights

We are interested in representing the Bernoulli sequence, Z, and the last *active* index of Z.
Idea is that we define the term
$\Gamma, u : \mathbb{R}_+ \vdash_p \texttt{truncated-sample} : \{0, 1\}^{\mathbb{N}} \times \mathbb{N}$ such that required program becomes:

$$\begin{array}{l}\texttt{slice-let } u = \texttt{sample}(\mathrm{Uniform}[0, 0.5]) \\ \quad \texttt{with } \ (x, i) = \texttt{return}(\texttt{normalize}(\texttt{truncated-sample})) \\ \quad \texttt{update } \ u = \texttt{sample}(\mathrm{Uniform}[0, 2^{-i}]) \\ \quad \texttt{in return}(x, i)\end{array}$$

Appendix: Bernoulli Process with decaying weights

Define the terms $\Gamma \vdash_d t_n : \{0,1\}^n$ inductively as follows:

$$t_{n+1} := \texttt{let } x = \texttt{flip } (2^{-(n+1)}) \texttt{ in return}(t_n, x)$$
$$t_1 := \texttt{flip } (2^{-1})$$

Define the terms $\Gamma, x : \{0,1\}^n \vdash_d \texttt{last-active}_n : \mathbb{N}$ inductively as follows:

$$\texttt{last-active}_{n+1} := \texttt{if } (\pi_{n+1}(x) == 1) \texttt{ return}(n+1) \texttt{ else let } x = (\pi_1(x), ..., \pi_n(x)) \texttt{ in last-active}_n$$
$$\texttt{last-active}_1 := \texttt{if } (\pi_1(x) == 1) \texttt{ return}(1) \texttt{ else return}(0)$$

Then, we can write the a term $\Gamma, u : \mathbb{R}_+ \vdash_p \texttt{truncated-sample} : \{0,1\}^{\mathbb{N}} \times \mathbb{N}$ as follows:

$$\texttt{truncated-sample} := \texttt{case } (\lfloor -\log_2(u) \rfloor, ()) \texttt{ of } (n, ()) \implies \texttt{let } x = t_n \texttt{ in score}(2^{-\texttt{last-active}_n}); \texttt{return}((x, 0, 0, ...), \texttt{last-active}_n)$$