Learning Proposals for Probabilistic Programs with Inference Combinators

Sam Stites[1] **Heiko Zimmermann**[1] **Hao Wu**[1] **Eli Sennesh**[1] **Jan-Willem van de Meent**[1]

[1]Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts, USA

Abstract

We develop operators for construction of proposals in probabilistic programs, which we refer to as inference combinators. Inference combinators define a grammar over importance samplers that compose primitive operations such as application of a transition kernel and importance resampling. Proposals in these samplers can be parameterized using neural networks, which in turn can be trained by optimizing variational objectives. The result is a framework for user-programmable variational methods that are correct by construction and can be tailored to specific models. We demonstrate the flexibility of this framework by implementing advanced variational methods based on amortized Gibbs sampling and annealing.

1 INTRODUCTION

One of the major ongoing developments in probabilistic programming is the integration of deep learning with approaches for inference. Libraries such as Edward (Tran et al., 2016), Pyro (Bingham et al., 2018), and Probabilistic Torch (Siddharth et al., 2017), extend deep learning frameworks with functionality for the design and training of deep probabilistic models. Inference in these models is typically performed with amortized variational methods, which learn an approximation to the posterior distribution of the model.

Amortized variational inference is widely used in the training of variational autoencoders (VAEs). While standard VAEs employ an unstructured prior over latent variables, such as a spherical Gaussian, in probabilistic programming, we are often interested in training a neural approximation to a simulation-based model (Baydin et al., 2019), or more generally to perform inference in models that employ programmatic priors as inductive biases. This provides a path to incorporating domain knowledge into methods for learning structured representations in a range of applications, such as identifying objects in an image or a video (Greff et al., 2019), representing users and items in reviews (Esmaeili et al., 2019), and characterizing the properties of small molecules (Kusner et al., 2017).

While amortized variational methods are very general, they often remain difficult to apply to structured domains, where gradient estimates based on a small number of (reparameterized) samples are often not sufficient. A large body of work improves upon standard methods by combining variational inference with more sophisticated sampling schemes based on Markov chain Monte Carlo (MCMC) and importance sampling (e.g. Naesseth et al. (2018); Le et al. (2019); Wu et al. (2020); see Appendix A for a comprehensive discussion). However to date, few of these methods have been implemented in probabilistic programming systems. One reason for this is that many methods rely on a degree of knowledge about the structure of the underlying model, which makes it difficult to develop generic implementations.

In this paper, we address this difficulty by considering a design that goes one step beyond that of traditional probabilistic programming systems. Instead of providing a language for the definition of models in the form of programs, we introduce a language for the definition of sampling strategies that are applicable to programs. This is an instance of a general idea that is sometimes referred to as *inference programming* (Mansinghka et al., 2014). Instantiations of this idea include inference implementations as monad transformers (Ścibior et al., 2018), inference implementations using low-level primitives for integration (Obermeyer et al., 2019), and programming interfaces based on primitive operations for simulation, generation, and updating of program traces (Cusumano-Towner et al., 2019).

Here we develop a new approach to inference programming based on constructs that we refer to as *inference combinators*, which act on probabilistic programs that perform importance sampling during evaluation. A combinator accepts one or more programs as inputs and returns a new

Proceedings of the Thirty-Seventh Conference on Uncertainty in Artificial Intelligence (UAI 2021), PMLR 161:1056–1066.

program. Each combinator denotes an elementary operation in importance sampling, such as associating a proposal with a target, composition of a program and a transition kernel, and importance resampling. The result is a set of constructs that can be composed to define user-programmable importance samplers for probabilistic programs that are valid by construction, in the sense that they generate samples that are properly weighted (Liu, 2008; Naesseth et al., 2015) under the associated target density of the program. These samplers in turn form the basis for nested variational methods (Zimmermann et al., 2021) that minimize a KL divergence to learn neural proposals.

We summarize the contributions of this paper as follows:

- We develop a language for inference programming in probabilistic programs, in which combinators define a grammar over properly-weighted importance samplers that can be tailored to specific models.
- We formalize semantics for inference and prove that samplers in our language are properly weighted for the density of a program.
- We demonstrate how variational methods can be used to learn proposals for these importance samplers, and thereby define a language for user-programmable stochastic variational methods.
- We provide an implementation of inference combinators for the Probabilistic Torch library[1]. We evaluate this implementation by using it to define existing state-of-the-art methods for probabilistic programs that improve over standard methods.

2 PRELIMINARIES

The combinators in this paper define a language for samplers that operate on probabilistic programs. We will refer to this language as the *inference language*. The probabilistic programs that inference operates upon are themselves defined in a *modeling language*. Both languages require semantics. For the inference language, semantics formally define the sampling strategy, whereas the semantics of the modeling language define the target density for the inference problem.

We deliberately opt not to define semantics for a specific modeling language. Our implementation is based on Probabilistic Torch, but combinators are applicable to a broad class of modeling languages that can incorporate control flow, recursion, and higher-order functions. Our main technical requirement is that all sampled and observed variables must have tractable conditional densities. This requirement is satisfied in many existing languages, including Church (Goodman et al., 2008), Anglican (Wood et al., 2014), WebPPL (Goodman, Stuhlmüller, 2014), Turing (Ge et al., 2018), Gen (Cusumano-Towner et al., 2019), Pyro (Bingham et al., 2018), and Edward2 (Tran et al., 2016)[2].

To define semantics for the inference language, we assume the existence of semantics for the modeling language. Concretely, we postulate that there exist *denotational measure semantics* for the density of a program. We will also postulate that programs have corresponding *operational sampler semantics* equivalent to likelihood weighting. In the next section, we will use these axiomatic semantics to define operational semantics for inference combinators such that samplers defined by these combinators are valid by construction, in the sense that evaluation yields *properly weighted* samples for the density denoted by the program. Below, we discuss the preliminaries that we need for this exposition.

2.1 LIKELIHOOD WEIGHTING

Probabilistic programs define a distribution over variables in a programmatic manner. As a simple running example, we consider the following program in Probabilistic Torch

```
η^v, η^x = ... # (initialize generator networks)
def f(s, x):
  # select mixture component
  z = s.sample(Multinomial(1, 0.2*ones(5)),"z")
  # sample image embedding
  v = s.sample(Normal(η^v_μ(z), η^v_σ(z)),"v")
  # condition on input image
  s.observe(Normal(η^x(v), 1), x,"x")
  return s, x, v, z
```

Program 1: A deep generative mixture model

This program corresponds to a density $p(x, v, z)$, in which z and v are unobserved variables and x is an observed variable. We first define a multinomial prior $p(z)$, and then define conditional distributions $p(v \mid z)$ and $p(x \mid v)$ using a set of generator networks η. The goal of inference is to reason about the posterior $p(v, z \mid x)$. We refer to the object `s` as the *inference state*. This data structure stores variables that need to be tracked as side effects of the computation, which we discuss in more detail below.

In this paper, the base case for evaluation performs likelihood weighting. This is a form of importance sampling in which unobserved variables are sampled from the prior, and are assigned an (unnormalized) importance weight according to the likelihood. In the example above, we would typically execute the program in a vectorized manner; we would input a tensor of B samples $x^b \sim \hat{p}(x)$ from an empirical distribution, generate tensors of $B \times L$ samples $v^{b,l}, z^{b,l} \sim p(v, z)$ from the prior, and compute

$$w^{b,l} = \frac{p(x^b, v^{b,l}, z^{b,l})}{p(v^{b,l}, z^{b,l})} = p(x^b \mid v^{b,l}, z^{b,l}).$$

[1] Code is available at `github.com/probtorch/combinators`.

[2] See Appendix A for an extensive discussion of related work.

These weights serve to compute a self-normalized approximation of a posterior expectation of some function $h(x, v, z)$

$$\mathbb{E}_{\hat{p}(x)\, p(v,z|x)}\left[h(x,v,z)\right] \simeq \frac{1}{B}\sum_{b,l}\frac{w^{b,l}}{\sum_{l'} w^{b,l'}} h(x^b, v^{b,l}, z^{b,l}). \tag{1}$$

Self-normalized estimators are consistent, but not unbiased. However, it is the case that each weight $w^{b,l}$ is a unbiased estimate of the marginal likelihood $p(x^b)$.

2.2 TRACED EVALUATION

Probabilistic programs can equivalently denote densities and samplers. In the context of specific languages, these two views can be formalized in terms of denotational *measure semantics* and operational *sampler semantics* (Ścibior et al., 2017). To define operational semantics for inference combinators, we begin by postulating sampler semantics for programs that perform likelihood weighting. We assume these semantics define an evaluation relation

$$c, \tau, \rho, w \leftsquigarrow \mathtt{f}(c').$$

In this relation, $\mathtt{f}$ is a program, c' are its input arguments, and c are its return values. Evaluation additionally outputs a weight w, a density map ρ, and a trace τ.[3]

A trace stores values for all sampled variables in an execution. We mathematically represent a trace as a mapping,

$$\tau = [\alpha_1 \mapsto c_1, \ldots, \alpha_n \mapsto c_n].$$

Here each α_i is an address for a random variable and c_i is its corresponding value. Addresses uniquely identify a random variable. In Probabilistic Torch, as well as in languages like Pyro and Gen, each `sample` or `observe` call accepts an address as the identifier. In the example above, evaluation returns a trace $[\texttt{"v"} \mapsto v, \texttt{"z"} \mapsto z]$, where $z, v \sim p(z, v)$.

The density map stores the value of the conditional density for all variables in the program,

$$\rho = [\alpha_1 \mapsto r_1, \ldots, \alpha_n \mapsto r_n], \qquad r_i \in [0, \infty).$$

Whereas the trace only stores values for unobserved variables, the density map stores probability densities for both observed and unobserved variables. In the example above, evaluation of a program would output a map for the variables with addresses `"x"`, `"v"` and `"z"`,

$$[\texttt{"x"} \mapsto p(x \mid v), \texttt{"v"} \mapsto p(v|z), \texttt{"z"} \mapsto p(z)],$$

in which conditional densities are computed using the traced values $v = \tau(\texttt{"v"})$, $z = \tau(\texttt{"z"})$, and the observed value x, which is an input to the program.

[3] In Probabilistic Torch, we return τ, ρ, and w by storing them in the inference state `s` during evaluation.

We define the weight in the evaluation as the joint probability of all observed variables (i.e. the likelihood). Since the density map ρ contains entries for all variables, and the trace only contains unobserved variables, the likelihood is

$$w = \prod_{\alpha \in \mathrm{dom}(\rho) \setminus \mathrm{dom}(\tau)} \rho(\alpha).$$

To perform importance sampling, we will use a trace from one program as a proposal for another program. For this purpose, we define an evaluation under substitution

$$c, \tau, \rho, w \leftsquigarrow \mathtt{f}(c')[\tau'].$$

In this relation, values in τ' are substituted for values of unobserved random variables in $\mathtt{f}$. This is to say that evaluation *reuses* $\tau(\alpha) = \tau'(\alpha)$ when a value exists at address α, and samples from the program prior when it is not. This is a common operation in probabilistic program inference, which also forms the basis for traced Metropolis-Hastings methods (Wingate et al., 2011).

When performing evaluation under substitution, the set of reused variables is the intersection $\mathrm{dom}(\tau) \cap \mathrm{dom}(\tau')$. Conversely, the newly sampled variables are $\mathrm{dom}(\tau) \setminus \mathrm{dom}(\tau')$. We define the weight of an evaluation under substitution as the joint probability of all observed and reused variables

$$w = \prod_{\alpha \in \mathrm{dom}(\rho) \setminus \left(\mathrm{dom}(\tau) \setminus \mathrm{dom}(\tau')\right)} \rho(\alpha). \tag{2}$$

2.3 DENOTATIONAL SEMANTICS

To reason about the validity of inference approaches, we need to formalize what density a program denotes. For this purpose, we assume the existence of denotational semantics that define a prior and unnormalized density

$$[\![\mathtt{f}(c')]\!]_\gamma(\tau) = \gamma_f(\tau; c'), \qquad [\![\mathtt{f}(c')]\!]_p(\tau) = p_f(\tau; c').$$

Given an unnormalized density, the goal of inference is to approximate the corresponding normalized density

$$\pi_f(\tau; c') = \frac{\gamma_f(\tau; c')}{Z_f(c')}, \qquad Z_f(c') = \int d\tau\, \gamma_f(\tau; c').$$

The reason that we specify a program as a density over traces, rather than as a density over specific variables, is that programs in higher-order languages with control flow and/or recursion may not always instantiate the same set of variables. A program could, for example, perform a random search that instantiates different variables in every evaluation (van de Meent et al., 2016). We refer to van de Meent et al. (2018) for a more pedagogical discussion of this point.

Formal specification of denotational semantics gives rise

to substantial technical questions[4], but in practice the unnormalized density is computable for programs in most probabilistic languages. In the languages that we consider here, in which conditional densities for all random variables are tractable, the unnormalized density is simply the joint probability of all variables in the program. Concretely, we postulate the following relationship between the sampler and the measure semantics of a program

$$\frac{c, \tau, \rho, w \leftsquigarrow \texttt{f}(c')}{[\![\texttt{f}(c')]\!]_\gamma(\tau) = \prod_{\alpha \in \mathrm{dom}(\rho)} \rho(\alpha) \quad [\![\texttt{f}(c')]\!]_p(\tau) = \prod_{\alpha \in \mathrm{dom}(\tau)} \rho(\alpha)}$$

In this notation, the top of the rule lists conditions, and the bottom states their implications. This rule states that, for any trace τ and density map ρ that can be generated by evaluating $\texttt{f}(c')$, the unnormalized density $[\![\texttt{f}(c')]\!]_\gamma(\tau)$ evaluates to the product of the conditional probabilities in ρ, whereas the prior density $[\![\texttt{f}(c')]\!]_p(\tau)$ corresponds to the product for all unobserved variables. This implies that the trace is distributed according to the prior, and that the weight is the ratio between the unnormalized density and the prior

$$w = \gamma_f(\tau; c')/p_f(\tau; c'), \qquad \tau \sim p_f(\tau; c').$$

The above rule implicitly defines the support Ω_f of the density, in that it only defines the density for traces τ that can be generated by evaluating $\texttt{f}(c')$. In the languages that we are interested in here, Ω_f may not be statically determinable through program analysis, but our exposition does not require its explicit characterization.

Finally, we define the density that a program denotes under substitution. We will define the unnormalized density to be invariant under substitution, which is to say that

$$[\![\texttt{f}(c')[\tau']]\!]_\gamma(\tau) = \gamma_{f[\tau']}(\tau; c') = \gamma_f(\tau; c').$$

For the prior under substitution we use the notation

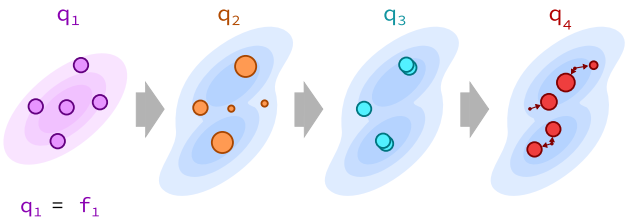
$$[\![\texttt{f}(c')[\tau']]\!]_p(\tau) = p_{f[\tau']}(\tau; c'),$$

to denote a density over newly sampled variables, rather than over all unobserved variables,

$$\frac{c, \tau, \rho, w \leftsquigarrow \texttt{f}(c')[\tau']}{[\![\texttt{f}(c')[\tau']]\!]_p(\tau) = \prod_{\alpha \in \mathrm{dom}(\tau) \setminus \mathrm{dom}(\tau')} \rho(\alpha)}$$

This construction ensures that $w = \gamma_{f[\tau']}(\tau; c')/p_{f[\tau']}(\tau; c')$, as in the case where no substitution is performed.

[4]Traditional measure theory based on Borel sets does not support higher-order functions. Recent work addresses this problem using a synthetic measure theory for quasi-Borel spaces, which in turn serves to formalize denotational semantics for a simply-typed lambda calculus (Ścibior et al., 2017). In practice, these technical issues do not give rise to problems, since existing languages cannot construct objects that give rise to issues with measurability.




```
q1 = f1
q2 = propose(f2, q1)
q3 = resample(q2)
q4 = propose(extend(f2, f4), compose(f3, q3))
```

Figure 1: Inference combinators denote fundamental operations in importance sampling, which can be composed to define a sampling strategy. See main text for details.

As previously, the support $\Omega_{f[\tau']}$ is defined implicitly as the set of traces that can be generated via evaluation under substitution, which is a subset of the original support

$$\Omega_{f[\tau']} = \{\tau \in \Omega_f : \tau(\alpha) = \tau'(\alpha)\ \forall\, \alpha \in \mathrm{dom}(\tau) \cap \mathrm{dom}(\tau')\}\,.$$

3 INFERENCE COMBINATORS

We develop a domain-specific language (DSL) for importance samplers in terms of four combinators, with a grammar that defines their possible compositions

```
f ::= A primitive program
p ::= f | extend(p, f)
q ::= p | resample(q) | compose(q', q) | propose(p, q)
```

We distinguish between three expression types. We use `f` to refer to a *primitive program* in the modeling language, with sampler semantics that perform likelihood weighting as described in the preceding section. We use `p` to refer to a *target program*, which is either a primitive program, or a composition of primitive programs that defines a density on an extended space. Finally, we use `q` to refer to an *inference program* that composes inference combinators.

Figure 1 shows a simple program that illustrates the use of each combinator, for which we will formally specify semantics below. The first step defines a program $\texttt{q}_1$ that generates samples from a primitive program $\texttt{f}_1$, which are equally weighted (i.e. $\texttt{f}_1$ does not have observed variables). The second step defines a program $\texttt{q}_2 = \texttt{propose}(\texttt{f}_2, \texttt{q}_1)$ that uses samples from $\texttt{q}_1$ as proposals for the primitive program $\texttt{f}_2$, which results in weights that are proportional to the ratio of densities. The third step $\texttt{q}_3 = \texttt{resample}(\texttt{q}_2)$ performs importance resampling, which replicates samples with probability proportional to their weights. In the fourth step, we use $\texttt{compose}(\texttt{f}_3, \texttt{q}_3)$ to apply a program $\texttt{f}_3$ that denotes a transition kernel, and use the resulting samples as proposals for a program $\texttt{extend}(\texttt{f}_2, \texttt{f}_4)$, which defines a density on an extended space. To illustrate this extended space construction, we will consider a more representative example of an inference program.

3.1 EXAMPLE: AMORTIZED GIBBS SAMPLING

Figure 2 shows a program that makes use of a combinator DSL that is embedded in Python. This code implements amortized population Gibbs (APG) samplers (Wu et al., 2020), a recently developed method that combines stochastic variational inference with sequential Monte Carlo (SMC) samplers to learn a set of conditional proposals that approximate Gibbs updates. This is an example where combinators are able to concisely express an algorithm that would be non-trivial to implement from scratch, even for experts.

We briefly discuss each operation in this algorithm. The function `pop_gibbs` (Figure 2, left) accepts programs that denote a `target` density, an initial `proposal`, and a set of `kernels`. It returns a program `q` that performs APG sampling. This program is evaluated in the right column to generate samples, compute an objective, and perform gradient descent to train the initial proposal and the kernels.

APG samplers perform a series of *sweeps*, where each sweep iterates over proposals that approximate Gibbs conditionals. To understand the weight computation in this sampler, we consider Program 1 as a (non-representative) example. Here updates could take the form of block proposals $q(v \mid x, z)$ and $q(z \mid x, v)$. At each step, we construct an outgoing sample by either updating v or z given an incoming sample (w, v, z). Here we consider an update of the variable v,

$$w' = \frac{p(x, v', z)\, q(v \mid x, z)}{q(v' \mid x, z)\, p(x, v, z)}\, w, \qquad v' \sim q(\cdot \mid x, z).$$

This weight update makes use of an auxiliary variable construction, in which we compute the ratio between a target density and proposal on an extended space

$$\tilde{p}(x, v', z, v) = p(x, v', z)\, q(v \mid x, z),$$
$$\tilde{q}(v', z, v \mid x) = q(v' \mid x, z)\, p(x, v, z).$$

In the inner loop for each sweep, we use the `extend` combinator to define the extended target $\tilde{p}(x, v', z, v)$. To generate the proposal, we use the `compose` combinator to combine the incoming sampler with the kernel, which defines the extended proposal $\tilde{q}(v', z, v \mid x)$. Since the marginal $\tilde{p}(x, v', z) = p(x, v', z)$, the outgoing sample (w', v', z) is properly weighted for the target density as long as the incoming sample is properly weighted.

3.2 PROGRAMS AS PROPOSALS

When we use a program as a proposal for another program, it may not be the case that the proposal and target instantiate the same set of variables. Here two edge cases can arise: (1) the proposal contains *superfluous* variables that are not referenced in the target, or (2) there are variables in the target that are *missing* from the proposal.

To account for both cases, we define an *implicit* auxiliary variable construction. To illustrate this construction, we consider a proposal that defines a density $q(u, z \mid x)$

```
λᵘ, λᶻ = ... # (initialize encoder networks)
def g(s, x):
  u = s.sample(Normal(λᵘ_μ(x), λᵘ_σ(x)),"u")
  z = s.sample(Multinomial(1, λᶻ(u), "z")
  return s, x, u, z
```

Suppose that we use `propose(f, g)` to associate this proposal with the target in Program 1, which defines a density $p(x, v, z)$. We then have a superfluous variable with address `"u"`, as well as a missing variable with address `"v"`. To deal with this problem, we implicitly extend the target using the conditional density $q(u \mid x)$ from the inference program, and conversely extend the proposal using the conditional density $p(v \mid z)$ from the target program,

$$\tilde{p}(x, v, z, u) = p(x, v, z)\, q(u \mid x),$$
$$\tilde{q}(v, z, u \mid x) = q(u, z \mid x)\, p(v \mid z).$$

Since, by construction, the conditional densities for u and v are identical in the target and proposal, these terms cancel when computing the importance weight

$$w = \frac{\tilde{p}(x, v, z, u)}{\tilde{q}(v, z, u \mid x)} = \frac{p(x, v, z)\, q(u \mid x)}{q(u, z \mid x)\, p(v \mid z)} = \frac{p(x \mid v)\, p(z)}{q(z \mid u)}.$$

This motivates a general importance sampling computation for primitive programs of the following form

$$c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \mathtt{g}(c_0), \qquad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathtt{f}(c_0)[\tau_1].$$

We generate τ_1 from the proposal, and then use substitution to generate a trace τ_2 that reuses as many variables from τ_1 as possible, and samples any remaining variables from the prior. Here the set of missing variables is $\mathrm{dom}(\tau_2) \setminus \mathrm{dom}(\tau_1)$ and the set of superfluous variables is $\mathrm{dom}(\tau_1) \setminus \mathrm{dom}(\tau_2)$. Hence, we can define the importance weight

$$\begin{aligned} w &= \frac{\gamma_f(\tau_2; c_0)\, p_{g[\tau_2]}(\tau_1; c_0)}{\gamma_g(\tau_1; c_0)\, p_{f[\tau_1]}(\tau_2; c_0)}\, w_1, \\ &= \frac{w_2}{\prod_{\alpha \in \mathrm{dom}(\rho_1) \setminus (\mathrm{dom}(\tau_1) \setminus \mathrm{dom}(\tau_2))} \rho_1(\alpha)}\, w_1. \end{aligned} \tag{3}$$

In the numerator, we take the product over all terms in the target density, exclusive of missing variables. Note that this expression is identical to w_2 (Equation 2). In the denominator, we take the product over all terms in the proposal density ρ_1, exclusive of superfluous variables $\mathrm{dom}(\tau_1) \setminus \mathrm{dom}(\tau_2)$.

3.3 PROPERLY WEIGHTED PROGRAMS

The expression in Equation 3 is not just valid for a composition `propose(f, g)` in which `f` and `g` are primitive programs. As we will show, this expression also applies to any composition `propose(p, q)`, in which `p` is a target program, and

```
def pop_gibbs(target, proposal, kernels, sweeps):
  q = propose(partial(target, suffix=0),
              partial(proposal, suffix=0))
  for s in range(sweeps):
    for k in kernels:
      q = propose(
          extend(partial(target, suffix=s+1),
                 partial(k, suffix=s)),
          compose(partial(k, suffix=s+1),
                  resample(q, dim=0)))
    return q
```

```
data, opt = ...
target, proposal, kernels = ...
q = pop_gibbs(target, proposal, kernels)
for _ in range(10000):
  s0 = State(sample_size=[40, 20],
             objective=inc_kl)
  s, *outputs = q(s0, data.next_batch(20))
  s.loss.backward()
  opt.step()
  opt.zero_grad()
```

Figure 2: A combinator-based implementation of amortized population Gibbs sampling (Wu et al., 2020) in python, along with a procedure for training the target model, initial proposal, and each of the kernels that approximate Gibbs conditionals. In this example `partial` is the partial application function from Python 3's `functools` library.

`q` is itself an inference program. While this distinction may seem subtle, it is fundamental; it allows us to use any sampler `q` as a proposal, which yields a new sampler that can once again be used as a proposal.

To demonstrate the validity of combinator composition, we make use of the framework of nested importance samplers and proper weighting (Liu, 2008; Naesseth et al., 2015), which we extend to evaluation of probabilistic programs.

Definition 1 (Properly Weighted Evaluation)**.** *Let* $\mathtt{q}(c')$ *denote an unnormalized density* $[\![\mathtt{q}(c')]\!] = \gamma_q(\cdot\,; c')$*. Let* $\pi_q(\cdot; c')$ *denote the corresponding probability density and let* $Z_q(c')$ *denote the normalizing constant such that*

$$\pi_q(\tau; c') := \frac{\gamma_q(\tau; c')}{Z_q(c')}, \qquad Z_q(c') := \int d\tau\, \gamma_q(\tau; c').$$

We refer to the evaluation of $c, \tau, \rho, w \leftsquigarrow \mathtt{q}(c')$ *as properly weighted for its unnormalized density* $\gamma_q(\cdot\,; c')$ *when, for all measurable functions* h

$$\begin{aligned}\mathbb{E}_{\mathtt{q}(c')}[w\, h(\tau)] &= \mathcal{C}_q(c') \int d\tau\, \gamma_q(\tau; c')\, h(\tau), \\ &= \mathcal{C}_q(c')\, Z_q(c')\, \mathbb{E}_{\pi_q(\cdot; c')}[h(\tau)],\end{aligned}$$

for some constant $\mathcal{C}_q(c') > 0$*. When* $\mathcal{C}_q(c') = 1$*, we refer to the evaluation as strictly properly weighted.*

A properly weighted evaluation can be used to define self-normalized estimators of the form in Equation 1. Such estimators are strongly consistent, which is to say that they converge almost surely in the limit of infinite samples L

$$\frac{\frac{1}{L}\sum_{l=1}^{L} w^l\, h(\tau^l)}{\frac{1}{L}\sum_{l=1}^{L} w^l} \xrightarrow{a.s.} \mathbb{E}_{\pi_q(\cdot; c')}[h(\tau)]\,.$$

Proposition 1 (Proper Weighting of Primitive Programs)**.** *Evaluation of a primitive program* $\mathtt{f}(c')$ *is strictly properly weighted for its unnormalized density* $[\![\mathtt{f}(c')]\!]_\gamma$*.*

Proof. This holds by definition, since in a primitive program w is uniquely determined by τ, which is a sample from the prior density $[\![\mathtt{f}(c')]\!]_p = p_f(\cdot\,; c')$,

$$\begin{aligned}\mathbb{E}_{\mathtt{f}(c')}[w\, h(\tau)] &= \mathbb{E}_{p_f(\cdot; c')}\left[\frac{\gamma_f(\tau; c')}{p_f(\tau; c')}\, h(\tau)\right], \\ &= Z_f(c')\, \mathbb{E}_{p_f(\cdot; c')}\left[\frac{\pi_f(\tau; c')}{p_f(\tau; c')}\, h(\tau)\right], \\ &= Z_f(c')\, \mathbb{E}_{\pi_f(\cdot; c')}[h(\tau)]\,.\end{aligned}$$

□

3.4 OPERATIONAL SEMANTICS

Given a modeling language for primitive programs whose evaluation is strictly properly weighted, our goal is to show that the inference language preserves strict proper weighting. To do so, we begin by formalizing rules for evaluation of each combinator, which together define big-step operational semantics for the inference language.

Compose. We begin with the rule for `compose(q2,q1)`, which performs program composition.

$$\frac{\begin{array}{c}c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \mathtt{q}_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathtt{q}_2(c_1) \\ \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset\end{array}}{c_2, \tau_2 \oplus \tau_1, \rho_2 \oplus \rho_1,\ w_2 \cdot w_1 \leftsquigarrow \mathtt{compose}(\mathtt{q}_2,\ \mathtt{q}_1)(c_0)}$$

This rule states that we can evaluate `compose(q2, q1)(c0)` by first evaluating $\mathtt{q}_1(c_0)$ and using the returned values c_1 as the inputs when evaluating $\mathtt{q}_2(c_1)$. We return the resulting value c_2 with weight $w_2 \cdot w_1$. We combine traces $\tau_2 \oplus \tau_1$ and density maps $\rho_2 \oplus \rho_1$ using the operator $\oplus$, which we define for maps μ_1 and μ_2 with disjoint domains as

$$(\mu_1 \oplus \mu_2)(\alpha) = \begin{cases}\mu_1(\alpha) & \alpha \in \mathrm{dom}(\mu_1), \\ \mu_2(\alpha) & \alpha \in \mathrm{dom}(\mu_2).\end{cases}$$

Extend. The combinator `extend(p, f)` performs a composition between a target `p` and a primitive `f` which defines

a density on an extended space.

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \texttt{p}(c_0) \qquad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \texttt{f}(c_1) \qquad \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset \qquad \mathrm{dom}(\rho_2) = \mathrm{dom}(\tau_2)}{c_2, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2 \leftsquigarrow \texttt{extend(p, f)}(c_0)}$$

The program `f` defines a "kernel", which may not contain observed variables. We enforce this by requiring that $\mathrm{dom}(\rho_2) = \mathrm{dom}(\tau_2)$.

Propose. The extend operator serves to incorporate auxiliary variables into a target density. When evaluating `propose(p, q)`, we discard auxiliary variables to continue the inference computation. For this purpose, we define a transformation `marginal(p)` to recover the original unextended program. We define this transformation recursively

$$\frac{}{\texttt{f} = \texttt{marginal(f)}} \qquad \frac{\texttt{f}' = \texttt{marginal(p)}}{\texttt{f}' = \texttt{marginal(extend(p, f))}}$$

We now define the operational semantics for propose as

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \texttt{q}(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \texttt{p}(c_0)[\tau_1] \quad c_3, \tau_3, \rho_3, w_3 \leftsquigarrow \texttt{marginal(p)}(c_0)[\tau_2] \quad u_1 = \prod_{\alpha \in \mathrm{dom}(\rho_1) \setminus (\mathrm{dom}(\tau_1) \setminus \mathrm{dom}(\tau_2))} \rho_1(\alpha)}{c_3, \tau_3, \rho_3, \; w_2 \cdot w_1 / u_1 \leftsquigarrow \texttt{propose(p, q)}(c_0)}$$

In this rule, the outgoing weight $w_2 \cdot w_1/u_1$ corresponds precisely to Equation 3, since w_2 is equal to the numerator, whereas u_1 is equal to the denominator in this expression.

Evaluation of $\texttt{p}(c_0)[\tau_1]$ applies substitution recursively to sub-expressions (see Appendix C). Note that, by construction, evaluation of $\texttt{marginal(p)}(c_0)[\tau_2]$ is deterministic, and that τ_3 and ρ_3 correspond to the entries in τ_2 and ρ_2 that are associated with the unextended target.

Resample. This combinator performs importance resampling on the return values, the trace, and the density map. Since resampling is an operation that applies to a collection of samples, we use a notational convention in which c^l, τ^l, ρ^l, w^l refer to elements in vectorized objects, and $\vec{c}$, $\vec{\tau}$, $\vec{\rho}$, $\vec{w}$ refer to vectorized objects in their entirety[5]. Resampling as applied to a vectorized program has the semantics

$$\frac{\vec{c}_1, \vec{\tau}_1, \vec{\rho}_1, \vec{w}_1 \leftsquigarrow \texttt{q}(\vec{c}_0) \quad \vec{a}_1 \sim \textsc{resample}(\vec{w}_1) \quad \vec{c}_2, \vec{\tau}_2, \vec{\rho}_2 = \textsc{reindex}(\vec{a}_1, \vec{c}_1, \vec{\tau}_1, \vec{\rho}_1) \quad \vec{w}_2 = \textsc{mean}(\vec{w}_1)}{\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2, \vec{w}_2 \leftsquigarrow \texttt{resample}(\texttt{q})(\vec{c}_0)}$$

In this rule, we make use of three operations. The first samples indices with probability proportional to their weight using a random procedure $\vec{a}_1 \sim \textsc{resample}(\vec{w}_1)$[6]. We then use a function $\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2 = \textsc{reindex}(\vec{a}_1, \vec{c}_1, \vec{\tau}_1, \vec{\rho}_1)$ to replicate objects according to the selected indices,

$$c_2^l = c_1^{a_1^l}, \qquad \tau_2^l = \tau_1^{a_1^l}, \qquad \rho_2^l = \rho_1^{a_1^l}. \tag{4}$$

Finally, we use $\vec{w}_2 = \textsc{mean}(\vec{w}_1)$ to set outgoing weights to the average of the incoming weights, $w_2^l = \sum_{l'} w_1^{l'}/L$.

Denotational Semantics. To demonstrate that program `q` are (strictly) properly weighted, we need to define the density that programs `p` and `q` denote. Owing to space limitations, we relegate discussion of these denotational semantics to Appendix B. Definitions follow in a straightforward manner from the denotational semantics of primitive programs.

Proper Weighting. With this formalism in place, we now state our main claim of correctness for samplers that are defined in the inference language.

Theorem 1 (Strict Proper Weighting of Inference Programs)**.** *Evaluation of an inference program* $\texttt{q}(c)$ *is strictly properly weighted for its unnormalized density* $[\![\texttt{q}(c)]\!]_\gamma$.

We provide a proof in Appendix E, which is by induction from lemmas for each combinator.

4 LEARNING NEURAL PROPOSALS

To learn a proposal `q`, we use properly-weighted samples to compute variational objectives. We consider three objectives for this purpose. The first two minimize a top-level reverse or forward KL divergence, which corresponds to performing stochastic variational inference (Wingate, Weber, 2013) or reweighted wake-sleep style inference (Le et al., 2019). The third implements nested variational inference (Zimmermann et al., 2021) by defining an objective at each level of recursion. We describe the loss and gradient computations at a high level, and provide details in Appendix D and Appendix F respectively.

Objective computation. To optimize the parameters of proposal programs, we slightly modify the operational semantics (for details see Appendix D) such that a user-defined objective function $\ell : (\rho_q, \rho_p, w, v) \to \mathbb{R}$ can be evaluated in the context of the `propose` combinator. Objective functions are defined in terms of the density maps of the proposal and target program ρ_q, ρ_p and the incoming and incremental importance weights w, v at the current level of nesting. The *local* losses computed at the individual levels of nesting are accumulated to a *global* loss in the inference state which consecutively can be used to compute gradients w.r.t. parameters of the target and proposal and programs.

[5] For simplicity we describe resampling with a single indexing dimension. The Probabilistic Torch implementation supports tensorized evaluation. For this reason, we specify a dimension along which resampling is to be performed in Figure 2.

[6] We use systematic resampling in our implementation. For a comparison of methods see (Murray et al., 2016)

Stochastic Variational Inference (SVI). Suppose that we have a program $\mathtt{q}_2 = \mathtt{propose(p,\ q_1)}$ in which the target and proposal have parameters θ and ϕ,

$$[\![\mathtt{p}(\mathtt{c}_0)]\!]_\gamma = \gamma_p(\cdot\,; c_0, \theta), \quad [\![\mathtt{q}_1(\mathtt{c}_0)]\!]_\gamma = \gamma_q(\cdot\,; c_0, \phi). \quad (5)$$

The target program $\mathtt{p}$ and the inference program $\mathtt{q}_2$ denote the same density $[\![\mathtt{p}(\mathtt{c}_0)]\!]_\gamma = [\![\mathtt{q}_2(\mathtt{c}_0)]\!]_\gamma$ and hence, as a result of Theorem 1, the evaluation of $\mathtt{q}_2$ is strictly properly weighted for γ_p. Hence, we can evaluate $c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathtt{q}_2(\mathtt{c}_0)$ to compute a stochastic lower bound (Burda et al., 2016),

$$\begin{aligned}\mathcal{L}(\theta, \phi) &:= \mathbb{E}_{\mathtt{q}_2(c_0)}\left[\log w_2\right] \\ &\leq \log\left(\mathbb{E}_{\mathtt{q}_2(c_0)}\left[w_2\right]\right) = \log Z_p(c_0, \theta),\end{aligned} \quad (6)$$

where the penultimate equality holds by Definition 1. The gradient $\nabla_\theta \mathcal{L}$ of this bound is a biased estimate of $\nabla_\theta \log Z_p$. The gradient $\nabla_\phi \mathcal{L}$ can be approximated using likelihood-ratio estimators (Wingate, Weber, 2013; Ranganath et al., 2014), reparameterized samples (Kingma, Welling, 2013; Rezende et al., 2014), or a combination of the two (Ritchie et al., 2016). In the special case where $\mathtt{q}_1 = \mathtt{f}$ is a primitive program, the gradient of the reverse KL-divergence between $\mathtt{p}$ and $\mathtt{q}_1$ is

$$\begin{aligned}\nabla_\phi \mathcal{L}(\theta, \phi) &= \mathbb{E}_{\mathtt{q}_2(c_0)}\left[\frac{\partial \log w_2}{\partial \tilde{\tau}_2}\frac{\partial \tilde{\tau}_2}{\partial \phi} + \frac{\partial}{\partial \phi}\log w_2\right], \\ &= -\nabla_\phi \mathrm{KL}(\pi_f || \pi_p).\end{aligned}$$

Reweighted Wake-sleep (RWS) Style Inference. To implement variational methods inspired by reweighted wake-sleep, we use samples $c_2^l, \tau_2^l, \rho_2^l, w_2^l \leftsquigarrow \mathtt{q}_2(c_0)$ to compute a self-normalized estimate of the gradient

$$\begin{aligned}\nabla_\theta \log Z_p(c_0, \theta) &= \mathbb{E}_{\pi_p(\tau\,; c_0, \theta)}\left[\nabla_\theta \log \gamma_p(\tau; c_0, \theta)\right], \\ &\simeq \sum_l \frac{w_2^l}{\sum_{l'} w_2^{l'}} \nabla_\theta \log \gamma_p(\tau_2^l; c_0, \theta).\end{aligned} \quad (7)$$

Notice that here we compute the gradient w.r.t. the non-extended density γ_p, which does not include auxiliary variables and hence density terms which would integrate to one. When approximating gradient this allows us to compute lower variance estimates.

Similarly, we approximate the gradient of the forward KL divergence with a self-normalized estimator that is defined in terms of the proposals $c_1^l, \tau_1^l, \rho_1^l, w_1^l \leftsquigarrow \mathtt{q}_1(c_0)$,

$$\begin{aligned}-\nabla_\phi \mathrm{KL}(\pi_p || \pi_q) &= \mathbb{E}_{\pi_p(\tau\,; c_0, \theta)}[\nabla_\phi \log \pi_q(\tau; c_0, \phi)], \\ &\simeq \sum_l \left(\frac{w_2^l}{\sum_{l'} w_2^{l'}} - \frac{w_1^l}{\sum_{l'} w_1^{l'}}\right) \nabla_\phi \log \gamma_q(\tau_1^l; c_0, \phi).\end{aligned} \quad (8)$$

In the special case where the proposal $\mathtt{q}_1 = \mathtt{f}$ is a primitive program without observations (i.e. $w_1^l = 1$), we can drop the second term to recovers the standard RWS estimator.

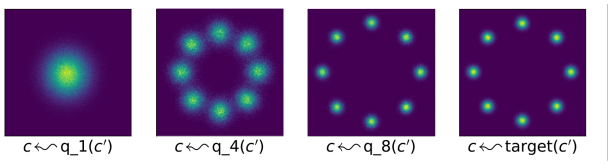


Figure 3: Samples from the initial proposal $\mathtt{q}_1$, learned intermediate proposal $\mathtt{q}_4$, final proposal $\mathtt{q}_8$, and final `target`. Definitions can be found in Figure 2, Appendix G.1.

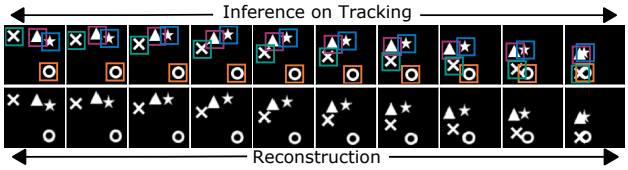


Figure 4: Qualitative results for the tracking task: Top row shows inferred positions of objects, bottom row shows reconstruction of the video frames.

Nested Variational Inference. A limitation of both SVI and RWS is that they are not well-suited to learning parameters in samplers at multiple levels of nesting. To see this, let us consider a program

```
q3 = propose(p3, propose(p2, f1))
```

We denote $[\![\mathtt{q}_3]\!]_\gamma = [\![\mathtt{p}_3]\!]_\gamma = \gamma_3$, $[\![\mathtt{p}_2]\!] = \gamma_2$, and $[\![\mathtt{f}_1]\!] = \gamma_1$. For simplicity, we consider densities with constant support $\Omega_p = \Omega_2 = \Omega_1$. The top-level evaluation $c_3, \tau_3, \rho_3, w_3 \leftsquigarrow \mathtt{q}_3(c_0)$ then yields a trace $\tau_3 = \tau_1$ that contains samples from the prior $[\![\mathtt{f}_1(c_0)]\!] = p_1(\cdot\,; c_0)$ with weight

$$w_3 = \frac{\gamma_3(\tau_1; c_0, \theta)}{\cancel{\gamma_2(\tau_1; c_0, \phi_2)}} \frac{\cancel{\gamma_2(\tau_1; c_0, \phi_2)}}{\cancel{\gamma_1(\tau_1; c_0, \phi_1)}} \frac{\cancel{\gamma_1(\tau_1; c_0, \phi_1)}}{p_1(\tau_1; c_0, \phi_1)}. \quad (9)$$

For this program, the lower bound from Equation 6 does not depend on ϕ_2. Conversely, the RWS-style estimator from Equation 8 does not depend on ϕ_1. Analogous problems arise in non-trivial inference programs that incorporate transition kernels and resampling. One such example are SMC samplers, where we would like to learn a sequence of intermediate densities, which impacts the variance of the final sampler. We consider this scenario Section 5.

Nested Variational Inference (NVI) (Zimmermann et al., 2021) replaces the top-level objectives in SVI and RWS with an objective that contains one term at each level of nesting. In the example above, this objective has the form

$$\mathcal{D} = D_3(\pi_2 \,||\, \pi_3) + D_2(\pi_1 \,||\, \pi_2) + D_1(p_1 \,||\, \pi_1),$$

where each D_i is a forward or a reverse KL divergence or a corresponding stochastic upper or lower bound. The individual terms can be optimized as described above, but additional care has to be taken when optimizing the intermediate target densities, as their normalizing constants might

Table 1: AVO and NVI variants trained for different numbers of annealing steps K and samples per step L for a fixed sampling budget of $K \cdot L = 288$ samples.

	$\log \hat{Z} = \log\left(\frac{1}{L}\sum_l w^l\right)$				ESS			
	K=2	K=4	K=6	K=8	K=2	K=4	K=6	K=8
AVO	1.88	1.99	2.05	2.06	426	291	285	295
NVI	1.88	1.99	2.06	2.07	427	341	308	319
NVIR	1.88	**2.05**	2.07	**2.08**	418	828	934	961
NVI*	1.88	2.03	**2.08**	**2.08**	427	304	414	481
NVIR*	1.88	1.99	**2.08**	**2.08**	418	**981**	**978**	**965**

not be tractable. For details on the computation of the nested variational objective and corresponding gradients we refer to Appendix F. When we apply an NVI objective using an upper bound based on the forward KL divergence to the program in Figure 2, we recover the gradient estimators of APG samplers.

5 EXPERIMENTS

We evaluate combinators in two experiments. First, we learn proposals in an annealed importance sampler that additionally learns intermediate densities. Second, we use an APG sampler to learn proposals and a generative model for an unsupervised multi-object tracking task.

Annealed Variational Inference. We consider the task of generating samples from a 2-dimensional unnormalized density consisting of 8 modes, equidistantly placed along a circle. For purposes of evaluation we treat this density as a blackbox, which we are only able to evaluate pointwise.

We implement an annealed importance sampler (Neal, 2001) (Figure 2 in Appendix G.1) for a sequence of unnormalized densities $\gamma_k(\tau_k; c_0) = \gamma_K(\tau_k; c_0)^{\beta_k}\gamma_1(\tau_k; c_0)^{1-\beta_k}$ that interpolate between an initial proposal γ_1 and the final target γ_K. The sampler employs forward kernels $q_k(\tau_k; c_{k-1}, \phi_k)$ and reverse kernels $r_k(\tau_{k-1}; c_k, \theta_k)$ to define densities on an extended space at each level of nesting. We train the sampler using NVI by optimizing the kernel parameters θ_k and ϕ_k, and parameters for the annealing schedule β_k.

We compare NVI and NVI*, which additionally learns the annealing schedule for the intermediate targets, and corresponding versions, NVIR and NVIR*, which additional employ resampling at every level of nesting, to annealed variational objectives (AVO) (Huang et al., 2018). Learning the annealing schedule (NVI(R)*) results in improved sample quality, in terms of the log expected weight ($log\hat{Z}$) and the effective sample size (ESS). We report results in Table 1 and refer to Appendix G.1 for additional details.

Amortized Population Gibbs. In this task, the data is a corpus of simulated videos that each contain multiple moving objects. Our goal is to learn both the target program (i.e. the generative model) and the inference program using the APG sampler that we introduced in Section 3.1. See Appendix G.2 for implementation details.

Table 2: $\log p_\theta(x, z)$ on test sets that contain D objects and T time steps. L is the number of particles and K is the number of sweeps. We run APG and RWS with computational budget $L \cdot K = 200$, and run HMC-RWS with $L \cdot K = 4000$.

		D=3		D=4	
Model	**Budget**	T=10	T=20	T=10	T=20
RWS	L=200, K=1	-5247	-9396	-8275	-16070
HMC-RWS	L=200, K=20	-5137	-9281	-8124	-15087
APG	L=100, K=2	-2849	-5008	-4411	-8966
APG	L=40, K=5	-2300	-4646	-3529	-6879
APG	L=20, K=10	**-2267**	**-4606**	**-3516**	**-6827**

Figure 4 shows that the APG sampler can (fully unsupervised) identify, track, and reconstruct individual object in each frame. In Table 2 we compare the APG sampler against an RWS baseline and a hand-coded HMC-RWS method, which improves upon RWS proposals using Hamiltonian Monte Carlo. Table 2 shows that APG outperforms both baselines. Moreover, for a fixed computational budget, increasing the number of sweeps improves sample quality.

6 RELATED WORK

This paper builds directly on several lines of work, which we discuss in detail in Appendix A.

Traced evaluation (Section 2) has a long history in systems that extend general-purpose programming languages with functionality for probabilistic modeling and inference (Wingate et al., 2011; Mansinghka et al., 2014; Goodman, Stuhlmüller, 2014; Tolpin et al., 2016), including recent work that combines probabilistic programming and deep learning (Tran et al., 2016; Ritchie et al., 2016; Siddharth et al., 2017; Bingham et al., 2018; Baydin et al., 2018).

The idea of developing abstractions for inference programming has been around for some time (Mansinghka et al., 2014), and several instantiations of such abstractions have been proposed in recent years (Cusumano-Towner et al., 2019; Ścibior et al., 2017; Obermeyer et al., 2019). The combinator-based language that we propose here is inspired directly by the work of Naesseth et al. (2019) on nested importance sampling, as well as on a body of work that connects importance sampling and variational inference.

7 DISCUSSION

We have developed a combinator-based language for importance samplers that are valid by construction, in the sense that samples are properly weighted for the density that a program denotes. We define semantics for these combinators and provide a reference implementation in Probabilistic Torch. Our experiments demonstrate that user-programmable samplers can be used as a basis for sophisticated variational methods that learn neural proposals and/or deep generative models. Inference combinators, which can be implemented as a DSL that extends a range of existing systems, hereby open up opportunities to develop novel nested variational methods for probabilistic programs.

Author Contributions

Sam Stites and Heiko Zimmermann contributed equally to this paper.

Acknowledgements

This work was supported by the Intel Corporation, the 3M Corporation, NSF award 1835309, startup funds from Northeastern University, the Air Force Research Laboratory (AFRL), and DARPA. We would like to thank Adam Ścibior for helpful discussions regarding the combinator semantics.

References

Baydin, A. G., Le, T. A., Heinrich, L., Gram-Hansen, B., Schroeder de Witt, C., Bhimji, W., Cranmer, K., Wood, F. *Pyprob/Ppx*. pyprob. 2018.

Baydin, A. G., Shao, L., Bhimji, W., Heinrich, L., Meadows, L. F., Liu, J., Munk, A., Naderiparizi, S., Gram-Hansen, B., Louppe, G., Ma, M., Zhao, X., Torr, P., Lee, V., Cranmer, K., Prabhat, Wood, F. "Etalumis: Bringing Probabilistic Programming to Scientific Simulators at Scale." *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC19), November 17–22, 2019*. 2019.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., Goodman, N. D. "Pyro: Deep Universal Probabilistic Programming" (Oct. 2018). arXiv: 1810.09538 [cs, stat].

Burda, Y., Grosse, R., Salakhutdinov, R. "Importance Weighted Autoencoders." *International Conference on Representations*. 2016. arXiv: 1509.00519.

Cusumano-Towner, M. F., Saad, F. A., Lew, A. K., Mansinghka, V. K. "Gen: A General-Purpose Probabilistic Programming System with Programmable Inference." *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI 2019. New York, NY, USA: Association for Computing Machinery, June 2019, pp. 221–236. DOI: 10.1145/3314221.3314642.

Esmaeili, B., Huang, H., Wallace, B., van de Meent, J.-W. "Structured neural topic models for reviews." *The 22nd International Conference on Artificial Intelligence and Statistics*. PMLR. 2019, pp. 3429–3439.

Ge, H., Xu, K., Ghahramani, Z. "Turing: A Language for Flexible Probabilistic Inference." *Proceedings of the Twenty-First International Conference on Artificial Intelligence and Statistics (AISTATS)*. Ed. by A. Storkey, F. Perez-Cruz. Vol. 84. 2018, pp. 1682–1690.

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., Tenenbaum, J. B. "Church: A Language for Generative Models." *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*. 2008, pp. 220–229.

Goodman, N. D., Stuhlmüller, A. *The Design and Implementation of Probabilistic Programming Languages*. 2014.

Greff, K., Kaufman, R. L., Kabra, R., Watters, N., Burgess, C., Zoran, D., Matthey, L., Botvinick, M., Lerchner, A. "Multi-object representation learning with iterative variational inference." *International Conference on Machine Learning*. PMLR. 2019, pp. 2424–2433.

Huang, C.-W., Tan, S., Lacoste, A., Courville, A. C. "Improving Explorability in Variational Inference with Annealed Variational Objectives." *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, R. Garnett. Curran Associates, Inc., 2018, pp. 9701–9711.

Kingma, D. P., Welling, M. "Auto-Encoding Variational Bayes." *International Conference on Learning Representations* (2013).

Kusner, M. J., Paige, B., Hernández-Lobato, J. M. "Grammar variational autoencoder." *International Conference on Machine Learning*. PMLR. 2017, pp. 1945–1954.

Le, T. A., Kosiorek, A. R., Siddharth, N., Teh, Y. W., Wood, F. "Revisiting Reweighted Wake-Sleep for Models with Stochastic Control Flow." *Uncertainty in Artificial Intelligence*. 2019.

Liu, J. S. *Monte Carlo strategies in scientific computing*. Springer Science & Business Media, 2008.

Mansinghka, V., Selsam, D., Perov, Y. "Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference." *arXiv* (Mar. 2014), pp. 78–78.

Murray, L. M., Lee, A., Jacob, P. E. "Parallel resampling in the particle filter." *Journal of Computational and Graphical Statistics* 25.3 (2016), pp. 789–805.

Naesseth, C., Linderman, S., Ranganath, R., Blei, D. "Variational Sequential Monte Carlo." en. *International Conference on Artificial Intelligence and Statistics*. PMLR, Mar. 2018, pp. 968–977.

Naesseth, C., Lindsten, F., Schon, T. "Nested Sequential Monte Carlo Methods." *International Conference on Machine Learning*. 2015, pp. 1292–1301.

Naesseth, C. A., Lindsten, F., Schön, T. B. "Elements of Sequential Monte Carlo." *arXiv:1903.04797 [cs, stat]* (Mar. 2019). (Visited on 12/16/2019).

Neal, R. M. "Annealed Importance Sampling." en. *Statistics and Computing* 11.2 (Apr. 2001), pp. 125–139. DOI: 10.1023/A:1008923215028.

Obermeyer, F., Bingham, E., Jankowiak, M., Phan, D., Chen, J. P. "Functional Tensors for Probabilistic Programming." *arXiv preprint arXiv:1910.10775* (2019).

Ranganath, R., Gerrish, S., Blei, D. "Black Box Variational Inference." en. *Artificial Intelligence and Statistics*. Apr. 2014, pp. 814–822.

Rezende, D. J., Mohamed, S., Wierstra, D. "Stochastic Backpropagation and Approximate Inference in Deep Generative Models." *Proceedings of the 31st International Conference on Machine Learning*. Ed. by E. P. Xing, T. Jebara. Vol. 32. Proceedings of Machine Learning Research 2. Bejing, China: PMLR, June 2014, pp. 1278–1286.

Ritchie, D., Horsfall, P., Goodman, N. D. "Deep Amortized Inference for Probabilistic Programs." en (Oct. 2016). arXiv: 1610.05735 [cs, stat].

Ścibior, A., Kammar, O., Ghahramani, Z. "Functional Programming for Modular Bayesian Inference." *Proc. ACM Program. Lang.* 2.ICFP (July 2018), 83:1–83:29. DOI: 10.1145/3236778.

Ścibior, A., Kammar, O., Vákár, M., Staton, S., Yang, H., Cai, Y., Ostermann, K., Moss, S. K., Heunen, C., Ghahramani, Z. "Denotational Validation of Higher-Order Bayesian Inference." *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 60:1–60:29. DOI: 10.1145/3158148.

Siddharth, N., Paige, B., van de Meent, J.-W., Desmaison, A., Goodman, N. D., Kohli, P., Wood, F., Torr, P. "Learning Disentangled Representations with Semi-Supervised Deep Generative Models." *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett. 2017, pp. 5927–5937.

Tolpin, D., van de Meent, J.-W., Yang, H., Wood, F. "Design and Implementation of Probabilistic Programming Language Anglican." *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*. IFL 2016. Leuven, Belgium: ACM, 2016, 6:1–6:12. DOI: 10/ghxhzn.

Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., Blei, D. M. "Edward: A Library for Probabilistic Modeling, Inference, and Criticism" (Oct. 2016). arXiv: 1610.09787 [cs, stat].

van de Meent, J.-W., Paige, B., Tolpin, D., Wood, F. "Black-Box Policy Search with Probabilistic Programs." 2016, 1195–1204.

van de Meent, J.-W., Paige, B., Yang, H., Wood, F. "An Introduction to Probabilistic Programming" (Sept. 2018). arXiv: 1809.10756 [cs, stat].

Wingate, D., Stuhlmueller, A., Goodman, N. D. "Lightweight Implementations of Probabilistic Programming Languages via Transformational Compilation." *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*. 2011, 770_778–770_778.

Wingate, D., Weber, T. "Automated Variational Inference in Probabilistic Programming" (2013), pp. 1–7. arXiv: 1301.1299.

Wood, F., van de Meent, J.-W., Mansinghka, V. "A New Approach to Probabilistic Programming Inference." *Artificial Intelligence and Statistics*. 2014, pp. 1024–1032.

Wu, H., Zimmermann, H., Sennesh, E., Le, T. A., van de Meent, J.-W. "Amortized Population Gibbs Samplers with Neural Sufficient Statistics." *International Conference on Machine Learning*. PMLR. 2020, pp. 10421–10431.

Zimmermann, H., Wu, H., Esmaeili, B., Stites, S., van de Meent, J.-W. "Nested Variational Inference." *3rd Symposium on Advances in Approximate Bayesian Inference* (2021).