
Learning Proposals for Probabilistic Programs with Inference Combinators

Sam Stites¹ Heiko Zimmermann¹ Hao Wu¹ Eli Sennesh¹ Jan-Willem van de Meent¹

¹Khoury 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 learn-

ing 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

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 library¹. 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, Stuhmüller, 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

```

 $\eta^v, \eta^x = \dots$  # (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( $\eta^v_\mu(z), \eta^v_\sigma(z)$ ), "v")
    # condition on input image
    s.observe(Normal( $\eta^x(v), 1$ ), 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 $p(v | z)$ and $p(x | v)$ using a set of generator networks η . The goal of inference is to reason about the posterior $p(v, z | x)$. 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

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

¹Code is available at github.com/probtorch/combinators.

²See 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 $h(x, v, z)$

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

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

$$c, \tau, \rho, w \leftarrow f(c').$$

In this relation, 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,

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

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 $["v" \mapsto v, "z" \mapsto z]$, where $z, v \sim p(z, v)$.

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

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

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"`,

$$["x" \mapsto p(x | v), "v" \mapsto p(v|z), "z" \mapsto p(z)],$$

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

³In 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

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

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

$$c, \tau, \rho, w \leftarrow f(c')[\tau'].$$

In this relation, values in τ' are substituted for values of unobserved random variables in 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 $\text{dom}(\tau) \cap \text{dom}(\tau')$. Conversely, the newly sampled variables are $\text{dom}(\tau) \setminus \text{dom}(\tau')$. We define the weight of an evaluation under substitution as the joint probability of all observed and reused variables

$$w = \prod_{\alpha \in \text{dom}(\rho) \setminus (\text{dom}(\tau) \cup \text{dom}(\tau'))} \rho(\alpha). \quad (2)$$

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

$$\llbracket f(c') \rrbracket_{\gamma}(\tau) = \gamma_f(\tau; c'), \quad \llbracket f(c') \rrbracket_p(\tau) = p_f(\tau; c').$$

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

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

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 questions⁴, 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

$$\frac{c, \tau, \rho, w \leftarrow f(c')}{\llbracket f(c') \rrbracket_\gamma(\tau) = \prod_{\alpha \in \text{dom}(\rho)} \rho(\alpha) \quad \llbracket f(c') \rrbracket_p(\tau) = \prod_{\alpha \in \text{dom}(\tau)} \rho(\alpha)}$$

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 $f(c')$, the unnormalized density $\llbracket f(c') \rrbracket_\gamma(\tau)$ evaluates to the product of the conditional probabilities in ρ , whereas the prior density $\llbracket f(c') \rrbracket_p(\tau)$ 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

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

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 $f(c')$. 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

$$\llbracket f(c')[\tau'] \rrbracket_\gamma(\tau) = \gamma_{f[\tau']}(\tau; c') = \gamma_f(\tau; c').$$

For the prior under substitution we use the notation

$$\llbracket f(c')[\tau'] \rrbracket_p(\tau) = p_{f[\tau']}(\tau; c'),$$

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

$$\frac{c, \tau, \rho, w \leftarrow f(c')[\tau']}{\llbracket f(c')[\tau'] \rrbracket_p(\tau) = \prod_{\alpha \in \text{dom}(\tau) \setminus \text{dom}(\tau')} \rho(\alpha)}$$

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

⁴Traditional 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.

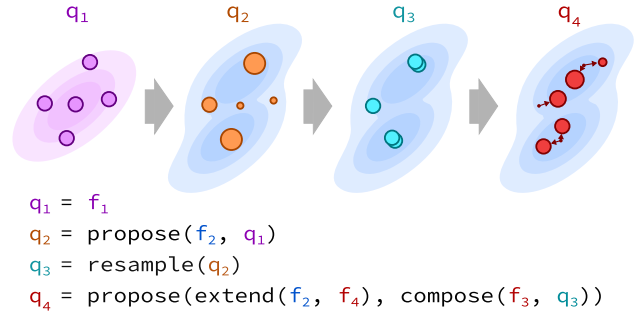


Figure 1: Inference combinators denote fundamental operations in importance sampling, which can be composed to define a sampling strategy. See main text for details.

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

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

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
 $p ::= f \mid \text{extend}(p, f)$
 $q ::= p \mid \text{resample}(q) \mid \text{compose}(q', q) \mid \text{propose}(p, q)$

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 q_1 that generates samples from a primitive program f_1 , which are equally weighted (i.e. f_1 does not have observed variables). The second step defines a program $q_2 = \text{propose}(f_2, q_1)$ that uses samples from q_1 as proposals for the primitive program f_2 , which results in weights that are proportional to the ratio of densities. The third step $q_3 = \text{resample}(q_2)$ performs importance resampling, which replicates samples with probability proportional to their weights. In the fourth step, we use $\text{compose}(f_3, q_3)$ to apply a program f_3 that denotes a transition kernel, and use the resulting samples as proposals for a program $\text{extend}(f_2, f_4)$, 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 $q(v | x, z)$ and $q(z | x, v)$. 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 ,

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

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

$$\begin{aligned} \tilde{p}(x, v', z, v) &= p(x, v', z) q(v | x, z), \\ \tilde{q}(v', z, v | x) &= q(v' | x, z) p(x, v, z). \end{aligned}$$

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 $\tilde{q}(v', z, v | x)$. 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 $q(u, z | x)$

```
λu, λz = ... # (initialize encoder networks)
def g(s, x):
    u = s.sample(Normal(λμu(x), λσu(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 $q(u | x)$ from the inference program, and conversely extend the proposal using the conditional density $p(v | z)$ from the target program,

$$\begin{aligned} \tilde{p}(x, v, z, u) &= p(x, v, z) q(u | x), \\ \tilde{q}(v, z, u | x) &= q(u, z | x) p(v | z). \end{aligned}$$

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

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

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

$$c_1, \tau_1, \rho_1, w_1 \leftarrow g(c_0), \quad c_2, \tau_2, \rho_2, w_2 \leftarrow f(c_0)[\tau_1].$$

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 $\text{dom}(\tau_2) \setminus \text{dom}(\tau_1)$ and the set of superfluous variables is $\text{dom}(\tau_1) \setminus \text{dom}(\tau_2)$. Hence, we can define the importance weight

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

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 $\text{dom}(\tau_1) \setminus \text{dom}(\tau_2)$.

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 $q(c')$ denote an unnormalized density $\llbracket q(c') \rrbracket = \gamma_q(\cdot; c')$. Let $\pi_q(\cdot; c')$ denote the corresponding probability density and let $Z_q(c')$ denote the normalizing constant such that*

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

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

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

for some constant $C_q(c') > 0$. When $C_q(c') = 1$, 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 $f(c')$ is strictly properly weighted for its unnormalized density $\llbracket f(c') \rrbracket_\gamma$.*

Proof. This holds by definition, since in a primitive program w is uniquely determined by τ , which is a sample from the prior density $\llbracket f(c') \rrbracket_p = p_f(\cdot; c')$,

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

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`(q_2, q_1), which performs program composition.

$$\frac{c_1, \tau_1, \rho_1, w_1 \rightsquigarrow q_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \rightsquigarrow q_2(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset}{c_2, \tau_2 \oplus \tau_1, \rho_2 \oplus \rho_1, w_2 \cdot w_1 \rightsquigarrow \text{compose}(q_2, q_1)(c_0)}$$

This rule states that we can evaluate `compose`(q_2, q_1)(c_0) by first evaluating $q_1(c_0)$ and using the returned values c_1 as the inputs when evaluating $q_2(c_1)$. 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

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

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.

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftarrow p(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftarrow f(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \text{dom}(\rho_2) = \text{dom}(\tau_2)}{c_2, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2 \leftarrow \text{extend}(p, f)(c_0)}$$

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

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

$$\frac{f' = \text{marginal}(p)}{f = \text{marginal}(f)} \quad \frac{f' = \text{marginal}(p)}{f' = \text{marginal}(\text{extend}(p, f))}$$

We now define the operational semantics for `propose` as

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

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 $p(c_0)[\tau_1]$ applies substitution recursively to sub-expressions (see Appendix C). Note that, by construction, evaluation of $\text{marginal}(p)(c_0)[\tau_2]$ 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 entirety⁵. Resampling as applied to a vectorized program has the semantics

$$\frac{\vec{c}_1, \vec{\tau}_1, \vec{\rho}_1, \vec{w}_1 \leftarrow q(\vec{c}_0) \quad \vec{a}_1 \sim \text{RESAMPLE}(\vec{w}_1) \quad \vec{c}_2, \vec{\tau}_2, \vec{\rho}_2 = \text{REINDEX}(\vec{a}_1, \vec{c}_1, \vec{\tau}_1, \vec{\rho}_1) \quad \vec{w}_2 = \text{MEAN}(\vec{w}_1)}{\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2, \vec{w}_2 \leftarrow \text{resample}(q)(\vec{c}_0)}$$

In this rule, we make use of three operations. The first samples indices with probability proportional to their weight

⁵For 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.

using a random procedure $\vec{a}_1 \sim \text{RESAMPLE}(\vec{w}_1)$ ⁶. We then use a function $\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2 = \text{REINDEX}(\vec{a}_1, \vec{c}_1, \vec{\tau}_1, \vec{\rho}_1)$ to replicate objects according to the selected indices,

$$c_2^l = c_1^{a_1^l}, \quad \tau_2^l = \tau_1^{a_1^l}, \quad \rho_2^l = \rho_1^{a_1^l}. \quad (4)$$

Finally, we use $\vec{w}_2 = \text{MEAN}(\vec{w}_1)$ 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 $q(c)$ is strictly properly weighted for its unnormalized density $\llbracket q(c) \rrbracket_\gamma$.*

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 $\ell : (\rho_q, \rho_p, w, v) \rightarrow \mathbb{R}$ 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.

⁶We 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 $q_2 = \text{propose}(p, q_1)$ in which the target and proposal have parameters θ and ϕ ,

$$\llbracket p(c_0) \rrbracket_\gamma = \gamma_p(\cdot; c_0, \theta), \quad \llbracket q_1(c_0) \rrbracket_\gamma = \gamma_q(\cdot; c_0, \phi). \quad (5)$$

The target program p and the inference program q_2 denote the same density $\llbracket p(c_0) \rrbracket_\gamma = \llbracket q_2(c_0) \rrbracket_\gamma$ and hence, as a result of Theorem 1, the evaluation of q_2 is strictly properly weighted for γ_p . Hence, we can evaluate $c_2, \tau_2, \rho_2, w_2 \leftarrow q_2(c_0)$ to compute a stochastic lower bound (Burda et al., 2016),

$$\begin{aligned} \mathcal{L}(\theta, \phi) &:= \mathbb{E}_{q_2(c_0)} [\log w_2] \\ &\leq \log \left(\mathbb{E}_{q_2(c_0)} [w_2] \right) = \log Z_p(c_0, \theta), \end{aligned} \quad (6)$$

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 $q_1 = f$ is a primitive program, the gradient of the reverse KL-divergence between p and q_1 is

$$\begin{aligned} \nabla_\phi \mathcal{L}(\theta, \phi) &= \mathbb{E}_{q_2(c_0)} \left[\frac{\partial \log w_2}{\partial \tau_2} \frac{\partial \tau_2}{\partial \phi} + \frac{\partial}{\partial \phi} \log w_2 \right], \\ &= -\nabla_\phi \text{KL}(\pi_f || \pi_p). \end{aligned}$$

Rewighted Wake-sleep (RWS) Style Inference. To implement variational methods inspired by reweighted wake-sleep, we use samples $c_2^l, \tau_2^l, \rho_2^l, w_2^l \leftarrow q_2(c_0)$ to compute a self-normalized estimate of the gradient

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

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 $c_1^l, \tau_1^l, \rho_1^l, w_1^l \leftarrow q_1(c_0)$,

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

In the special case where the proposal $q_1 = f$ is a primitive program without observations (i.e. $w_1^l = 1$), we can drop the second term to recovers the standard RWS estimator.

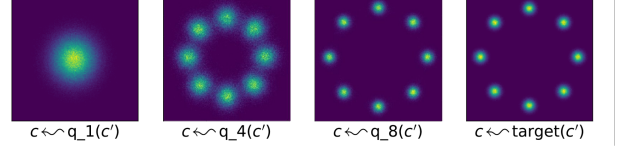


Figure 3: Samples from the initial proposal q_1 , learned intermediate proposal q_4 , final proposal q_8 , and final target. Definitions can be found in Figure 2, Appendix G.1.

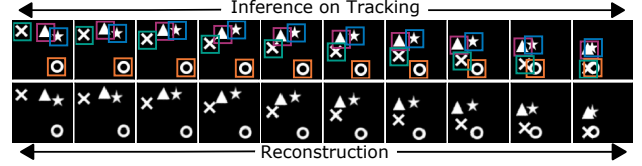


Figure 4: Qualitative results for the tracking task: Top row shows inferred positions of objects, bottom row shows reconstruction of the video frames.

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

$$q_3 = \text{propose}(p_3, \text{propose}(p_2, f_1))$$

We denote $\llbracket q_3 \rrbracket_\gamma = \llbracket p_3 \rrbracket_\gamma = \gamma_3$, $\llbracket p_2 \rrbracket = \gamma_2$, and $\llbracket f_1 \rrbracket = \gamma_1$. For simplicity, we consider densities with constant support $\Omega_p = \Omega_2 = \Omega_1$. The top-level evaluation $c_3, \tau_3, \rho_3, w_3 \leftarrow q_3(c_0)$ then yields a trace $\tau_3 = \tau_1$ that contains samples from the prior $\llbracket f_1(c_0) \rrbracket = p_1(\cdot; c_0)$ with weight

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

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

$$D = D_2(\pi_2 || \pi_3) + D_2(\pi_1 || \pi_2) + D_1(p_1 || \pi_1),$$

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.

	$\log \hat{Z} = \log(\frac{1}{L} \sum_l w^l)$				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 combinatorics 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 additionally 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

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 $L \cdot K = 4000$.

Model	Budget	D=3		D=4	
		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

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.

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](https://arxiv.org/abs/1810.09538) [cs, stat].
- Burda, Y., Grosse, R., Salakhutdinov, R. “Importance Weighted Autoencoders.” *International Conference on Representations*. 2016. arXiv: [1509.00519](https://arxiv.org/abs/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](https://doi.org/10.1145/3314221.3314642).
- Esmaili, 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](https://doi.org/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. Beijing, 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](https://doi.org/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](https://doi.org/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](https://doi.org/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., Stuhlmüller, 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).

Supplementary Material: Learning Proposals for Probabilistic Programs with Inference Combinators

Sam Stites^{*1}

Heiko Zimmermann^{*1}

Hao Wu¹

Eli Sennesh¹

Jan-Willem van de Meent¹

¹Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts, USA
¹{stites.s, zimmermann.h, wu.hao10, sennesh.e, j.vandemeent}@northeastern.edu

A RELATED WORK

A.1 IMPORTANCE SAMPLING AND MCMC IN VARIATIONAL INFERENCE

This work fits into a line of recent methods for deep generative modeling that seek to improve inference quality in stochastic variational methods. We briefly review related literature on standard methods before discussing advanced methods that combine variational inference with importance sampling and MCMC, which inspire this work.

Variants of SVI maximize a lower bound, which equates to minimizing a reverse KL divergence, whereas methods that derive from RWS minimize an upper bound, which equates to minimizing a forward KL divergence. In the case of SVI, gradients can be approximated using reparameterization, which is widely used in variational autoencoders (Kingma, Welling, 2013; Rezende et al., 2014) or by using likelihood-ratio estimators (Wingate, Weber, 2013; Ranganath et al., 2014). In the more general case, where the model contains a combination of reparameterized and non-reparameterized variables, the gradient computation can be formalized in terms of stochastic computation graphs (Schulman et al., 2015). Work by Ritchie et al. (2016) explains how to operationalize this computation in probabilistic programming systems. In the case of RWS, gradients can be computed using simple self-normalized estimators, which do not require reparameterization (Bornschein, Bengio, 2015). This idea was recently revisited in the context of probabilistic programming systems by (Le et al., 2019), who demonstrate that RWS-based methods are often competitive with SVI.

Importance-weighted Variational Inference. There is a large body of work that improves upon standard SVI by defining tighter bounds, which results in better gradient estimates for the generative model. Many of these approaches derive from importance-weighted autoencoders (IWAEs) (Burda et al., 2016), which use importance sampling to define a stochastic lower bound $\mathbb{E}[\log \hat{Z}] \leq \log Z$ based on an unbiased estimate $\mathbb{E}[\hat{Z}] = Z$ (see Section G.1). Since any strictly properly weighted importance sampler can be used to define an unbiased estimator $\hat{Z} = \frac{1}{L} \sum_l w^l$, this gives rise to many possible extensions, including methods based on SMC (Le et al., 2018; Naesseth et al., 2018; Maddison et al., 2017) and thermodynamic integration (Masrani et al., 2019). Salimans et al. (2015) derive a stochastic lower bound for variational inference which uses an importance weight defined in terms of forward and reverse kernels in MCMC steps. Caterini et al. (2018) extend this work by optimally selecting reverse kernels (rather than learning them) using inhomogeneous Hamiltonian dynamics. The work on AVO (Huang et al., 2018) learn a sequence of transition kernels that performs annealing from the initial encoder to the posterior, which we use as a baseline in our annealing experiments.

A somewhat counter-intuitive property of these estimators is tightening the bound typically improves the quality of gradient estimates for the generative model, but can adversely affect the signal-to-noise ratio of gradient estimates for the inference model (Rainforth et al., 2018). These issues can be circumvented by using RWS-style estimators¹, or by implementing doubly-reparameterized estimators (Tucker et al., 2019).

¹Note that the gradient estimate for the generative model in Equation 7 is identical to the gradient estimate of the corresponding IWAE bound, so these two approaches only differ in the gradient estimate that they compute for the inference model.

SMC Samplers and MCMC. In addition to enabling implementation of variational methods based on SMC, this work enables the interleaving of resampling and move operations to define so-called SMC samplers (Chopin, 2002). One recent example of are the APG samplers that we consider in our experiments (Wu et al., 2020). It is also possible to interleave importance sampling with any MCMC operator, which preserves proper weighting as long as the stationary distribution of this operator is the target density. In this space, there exists a large body of relevant work.

Hoffman (2017) apply Hamiltonian Monte Carlo to samples that are generated from the encoder, which serves to improve the gradient estimate w.r.t. the generative model, while learning the inference network using a standard reparameterized lower bound objective. Li et al. (2017) also use MCMC to improve the quality of samples from an encoder, but additionally use these samples to train the encoder by minimizing the forward KL divergence relative to the filtering distribution of the Markov chain. Since the filtering distribution after multiple MCMC steps is intractable, Li et al. (2017) use an adversarial objective to minimize the forward KL. Wang et al. (2018) develop a meta-learning approach to learn Gibbs block conditionals. This work assumes a setup in which it is possible to sample data and latent variables from the true generative model. This approach minimizes the forward KL, but uses the learned conditionals to define an (approximate) MCMC sampler, rather than using them as proposals in a SMC sampler.

Proper weighting and nested variational inference. This work directly builds on seminal work by Liu (2008); Naesseth et al. (2015); Naesseth et al. (2019) that formalizes proper weighting. This formalism makes it possible to reason compositionally about validity of importance samplers and corresponding variational objectives (Zimmermann et al., 2021).

A.2 PROBABILISTIC PROGRAMMING

Probabilistic programming systems implement methods for inference in programmatically specified models. A wide variety of systems exist, which differ in the base languages they employ, the types of inference methods they provide, and their intended use cases. One widely used approach to the design of probabilistic programming systems is to define a language in which all programs are amenable to a particular style of inference. Exemplars of this approach include Stan (Carpenter et al., 2017), which emphasizes inference using Hamiltonian Monte Carlo methods in differentiable models with statically-typed support, Infer.NET (Minka et al., 2010) which emphasizes message passing in programs that denote factor graphs, Problog (De Raedt et al., 2007) and Dice (Holtzen et al., 2020), in which programs denote binary decision diagrams, and LibBi (Murray, 2013), which emphasizes particle-based methods for state space models. More generally, systems that emphasize MCMC methods in programs with statically typed support fit this mold, including early systems like BUGS (Spiegelhalter et al., 1995) and JAGS (Plummer, 2003), as well as more recent systems like PyMC3 (Salvatier et al., 2016).

A second widely used approach to probabilistic programming is to extend a general-purpose language with functionality for probabilistic modeling, and implement inference methods that are generally applicable to programs in this language. The advantage of this design is that it becomes more straightforward to develop simulation-based models that incorporate complex deterministic functions, or to design programs that incorporate recursion and control flow. A well-known early exemplar of this style of probabilistic programming is Church (Goodman et al., 2008), whose modeling language is based on Scheme. Since then, many existing languages have been adapted to probabilistic programming, including Lisp variants (Mansinghka et al., 2014; Wood et al., 2014), Javascript (Goodman, Stuhlmüller, 2014), C (Paige, Wood, 2014), Scala (Pfeffer, 2009), Go (Tolpin, 2018), and Julia (Ge et al., 2018; Cusumano-Towner et al., 2019).

This second class of probabilistic programming systems is the most directly relevant to the work that we present here. A key technical consideration in the design of probabilistic programming systems is whether the modeling language is first-order or higher-order (sometimes also referred to as “universal”) (van de Meent et al., 2018). In first-order languages, a program denotes a density in which the support is statically determinable at compile time. Since most general-purpose languages support higher-order functions and recursion, the support of probabilistic programs in these languages is generally not statically determinable. However, the traced evaluation model that we describe in Section 2.2 can be implemented in almost any language, and inference combinators can therefore be implemented as a DSL for inference in most of these systems.

In the context of our work, we are particularly interested in systems that extend or interoperate with deep learning systems, including Pyro (Bingham et al., 2018), Edward2 (Tran et al., 2018), Probabilistic Torch (Siddharth et al., 2017), Scruff (Pfeffer, Lynn, 2018), Gen (Cusumano-Towner et al., 2019) and PyProb (Baydin et al., 2019). These systems provide first-class support for inference with stochastic gradient methods. While support for stochastic variational inference in probabilistic programming has been around for some time (Wingate, Weber, 2013; van de Meent et al., 2016b; Ritchie et al., 2016), this style of inference has become much more viable in systems where variational distributions can be parameterized using neural networks. However, to date, the methods that are implemented in these systems are typically limited to standard

SVI, RWS, and IWAE objectives. We are not aware of systems that currently support stochastic variational methods that incorporate importance resampling, such as autoencoding SMC or APG samplers.

A.3 INFERENCE PROGRAMMING

The combinator-based language for inference programming that we develop in this paper builds on ideas that have been under development in the probabilistic programming community for some time.

The Venture paper described “inference programming” as one of the desiderata for functionality of future systems (Mansinghka et al., 2014). Venture also implemented a stochastic procedure interface for manipulating traces and trace fragments, which can be understood as a low-level programming interface for inference. WebPPL (Goodman, Stuhlmüller, 2014) and Anglican (Tolpin et al., 2016) define an interface for inference implementations that is based on continuation-passing-style (CPS) transformations, in which a black-box deterministic computation returns continuations to an inference backend, which implements inference computations and continues execution (van de Meent et al., 2016a). A number of more recent systems have implemented similar interfaces, albeit by different mechanisms than CPS transformations. Turing (Ge et al., 2018) uses co-routines in Julia as a mechanism for interruptible computations. Pyro (Bingham et al., 2018) and Edward2 (Tran et al., 2018) implement an interface in which requests to the inference backend are dispatched using composable functions that are known as “messengers” or “tracers”. The arguably most general instantiation of this idea is found in PyProb (Baydin et al., 2019), which employs a cross-platform Probabilistic Programming eXecution (PPX) protocol based on flatbuffers to enable computation between a program and inference backend that can be implemented in different languages (Baydin et al., 2018). For a pedagogical discussion, see the introduction by van de Meent et al. (2018).

All of the above programming interfaces for inference are low-level, which is to say that it is the responsibility of the developer to ensure that quantities like importance weights and acceptance probabilities are computed in a manner that results in a correct inference algorithm. This means that users of probabilistic programming systems can in principle create their own inference algorithms, but that doing so may require considerable expertise and debugging. For this reason, recent systems have sought to develop higher-level interfaces for inference programming. Edward (Tran et al., 2016) provides a degree of support for interleaving inference operations targeting different conditionals. Birch (Murray, Schön, 2018) provides constructs for structural motifs, such as state space models, which are amenable to inference optimizations.

Exemplars of systems that more explicitly seek to enable inference programming include Gen (Cusumano-Towner et al., 2019) and Functional Tensors (Obermeyer et al., 2019). Gen provides support for writing user-level code to interleave operations such as MCMC updates or MAP estimation, which can be applied to subsets of variables. For this purpose, Gen provides a generative function interface consisting of primitives `generate`, `propose`, `assess`, `update`, and `choice_gradients`. Recursive calls to these primitives at generative function call sites serve to compositionally implement Gen’s built-in modeling language, along with hierarchical traces. Lew et al. (2020) considered a type system for execution traces in probabilistic programs, which allowed them to verify the correctness of certain inference algorithms at compile time.

Functional Tensors (funsors) provide an abstraction for integration that aims to unify exact and approximate inference. Funsors providing a grammar and type system which take inspiration from modern autodifferentiation libraries. Expressions in funsors denote discrete factors, Gaussian factors, point mass, delayed values, function application, substitution, marginalization, and plated products. Types encapsulate tensor dimensions, which permit funsors to support broadcasting. This defines an intermediate representation that can be used for a variety of probabilistic programs and inference methods.

The inference language that we develop here differs from these above approaches in that is designed to ensure that any composition of combinators yields an importance sampler that is valid, in the sense that evaluation is properly-weighted for the unnormalized density that a program denotes. In doing so, our work takes inspiration from Haku (Narayanan et al., 2016), which frames inference as program transformations that be composed so as to preserve a measure-theoretic denotation (Zinkov, Shan, 2017), as well as the work on validity of inference in higher-order probabilistic programs by Ścibior et al. (2017), which we discuss in the next Section.

A.4 SAMPLING AND MEASURE SEMANTICS

Programming languages have been studied in terms of two kinds of semantics: what the programs do, operational semantics, and what they mean, denotational semantics. For probabilistic programming languages, this has usually led to a denotational *measure semantics* that interpret generative model programs as measures, and an operational *sampler semantics* that interpret programs as procedures for using randomness to sample from a specified distribution. In the terms we use for our combinators library, model composition changes the target density of a program, and thus its measure semantics, while

inference programming should alter the sampling semantics in a way that preserves the measure semantics.

Early work by [Borgström et al. \(2011\)](#); [Toronto et al. \(2015\)](#) characterized measure semantics for first-order probabilistic programs, with the first enabling inference by compilation to a factor graph, and the second via semi-computable preimage functions. This demonstrates one of the simplest, but most analytically difficult, ways to perform inference in a probabilistic program: get rid of the sampling semantics and use the measure semantics to directly evaluate the relevant posterior expectation. When made possible by program analysis, this can even take the form of symbolic disintegration of the joint distribution entailed by a generative program into observations and a posterior distribution, as in [Shan, Ramsey \(2017\)](#).

[Fong \(2013\)](#) gave categorical semantics to causal Bayesian networks, and via the usual compilation to a graphical model construction, to first-order probabilistic programs as well. Further work by [Clerc et al. \(2017\)](#); [Dahlqvist et al. \(2018\)](#) has extended the consideration of categorical semantics for first-order PPLs.

Quasi-Borel spaces were discovered by [Heunen et al. \(2017\)](#) and quickly found to provide a good model for higher-order probabilistic programming. [Ścibior et al. \(2017\)](#); [Ścibior et al. \(2018\)](#) applied these categorical semantics to prove that inference algorithms could be specified by monad transformers on sampling strategies that would preserve the categorical measure semantics over the underlying generative model. The work of [Ścibior et al. \(2018\)](#) included an explicit consideration of importance weighting, which we have extended to cover the proper weighting of importance samples from arbitrarily nested and extended inference programs.

B DENOTATIONAL SEMANTICS OF TARGET AND INFERENCE PROGRAMS

To reason about validity of inference, we need to specify what density a target program p or inference program q denotes. We begin with target programs, which have the grammar $p ::= f \mid \text{extend}(p, f)$. For a primitive program f , the denotational semantics inherit trivially from the axiomatic denotational semantics of the modeling language. For a program $\text{extend}(p, f)$ we define the density as the composition of densities of the inputs

$$\frac{c_1, \tau_1, \rho_1, w_1 \sim p(c_0) \quad c_2, \tau_2, \rho_2, w_2 \sim f(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \text{dom}(\rho_2) = \text{dom}(\tau_2) \quad \tau_3 = \tau_2 \oplus \tau_1}{\llbracket \text{extend}(p, f)(c_0) \rrbracket(\tau_3) = \llbracket f(c_1) \rrbracket(\tau_2) \llbracket p(c_0) \rrbracket(\tau_1)}$$

In this rule, we omit subscripts $\llbracket \cdot \rrbracket_\gamma$ and $\llbracket \cdot \rrbracket_p$, since this rule applies to both prior and the unnormalized density. As in the case of primitive programs, we here adopt a convention in which the support is implicitly defined as the set of traces $\tau_3 = \tau_2 \oplus \tau_1$ by combining disjoint traces τ_1 and τ_2 that can be generated by evaluating the composition of p and f .

Inference programs have a grammar $q ::= p \mid \text{compose}(q', q) \mid \text{resample}(q) \mid \text{propose}(p, q)$. For each expression form, we define the density that a program denotes in terms of the density of its corresponding target program. To do so, we define a program transformation $\text{target}(q)$. This transformation replaces all sub-expressions of the form $\text{propose}(p, q)$ with their targets p and all sub-expressions $\text{resample}(q)$ with q . We define the transformation recursively

$$\frac{}{p = \text{target}(p)} \quad \frac{}{p = \text{target}(\text{propose}(p, q))} \quad \frac{q' = \text{target}(q)}{q' = \text{target}(\text{resample}(q))} \quad \frac{q'_1 = \text{target}(q_1) \quad q'_2 = \text{target}(q_2)}{\text{compose}(q'_2, q'_1) = \text{target}(\text{compose}(q_2, q_1))}$$

Since this transformation removes all instances of propose and resample forms, transformed programs $q' = \text{target}(q)$ define a simplified grammar $q' ::= p \mid \text{compose}(q'_1, q'_2)$. We now define the denotational semantics for inference programs as

$$\frac{q' = \text{target}(q)}{\llbracket q(c) \rrbracket = \llbracket q'(c) \rrbracket}$$

The denotational semantics for a transformed program are trivially inherited from those for target programs when $q' = p$, whereas the denotational semantics for a composition $\text{compose}(q'_2, q'_1)$ are analogous to those of an extension $\text{extend}(p, f)$

$$\frac{c_1, \tau_1, \rho_1, w_1 \sim q'_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \sim q'_2(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \tau_3 = \tau_1 \oplus \tau_2}{\llbracket \text{compose}(q'_2, q'_1)(c_0) \rrbracket(\tau_3) = \llbracket q'_2(c_1) \rrbracket(\tau_2) \llbracket q'_1(c_0) \rrbracket(\tau_1)}$$

C EVALUATION UNDER SUBSTITUTION

To use an extended program as a target for a proposal, we need to define its evaluation under substitution. We define the operational semantics of this evaluation in a manner that is analogous to the unconditioned case

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftarrow p(c_0)[\tau_0] \quad c_2, \tau_2, \rho_2, w_2 \leftarrow f(c_1)[\tau_0] \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \text{dom}(\rho_2) = \text{dom}(\tau_2)}{c_1, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2 \leftarrow \text{extend}(p, f)(c_0)[\tau_0]}$$

This rule, like other rules, defines a recursion. In this case, the recursion ensures that we can perform conditioned evaluation for any target program p .

We define evaluation under substitution of an inference program q by performing an evaluation under substitution for the corresponding target program

$$\frac{c, \tau, \rho, w \leftarrow \text{target}(q)(c_0)[\tau_0]}{c, \tau, \rho, w \leftarrow q(c_0)[\tau_0]}$$

As in the previous section, the transformed programs define a grammar $q' ::= p \mid \text{compose}(q'_1, q'_2)$. In the base case $q' = p$, evaluation under substitution is defined as above. Evaluation under substitution of a program $\text{compose}(q'_1, q'_2)$ is once again defined by recursively evaluating inputs under substitution, as with the extend combinator

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftarrow q'_1(c_0)[\tau_0] \quad c_2, \tau_2, \rho_2, w_2 \leftarrow q'_2(c_1)[\tau_0] \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset}{c_1, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2 \leftarrow \text{compose}(q'_2, q'_1)(c_0)[\tau_0]}$$

Note that the operational semantics do not rely on evaluation under substitution of inference programs q , since conditional evaluation is only every performed for target programs. However, the proofs in Section E do make use of the definition of the prior under substitution, which is identical to the definition for primitive programs

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftarrow q(c_0)[\tau_0]}{[[p(c_0)[\tau_0]]_p(\tau_1) = p_{q[\tau_0]}(\tau_1; c_0) = \prod_{\alpha \in \text{dom}(\tau_1) \setminus \text{dom}(\tau_0)} \rho_1(\alpha)}$$

D EVALUATION IN CONTEXT

To perform variational inference, we need to update a variational objective \mathcal{L} each time we evaluate a **propose** combinator. For this purpose we define an evaluation in the context of a user-defined objective function $\ell : (\rho_q, \rho_p, w, v) \rightarrow \mathbb{R}$. For this purpose we introduce the notation

$$\langle \mathcal{L}, \ell, (c, \tau, \rho, w) \rangle \leftarrow \langle \mathcal{L}', \ell, q(c') \rangle.$$

In this notation, evaluation of the program q in the context of an objective \mathcal{L}' and an objective function ℓ returns the tuple (c, τ, ρ, w) according to the operational semantics for traced evaluation, along with an updated objective \mathcal{L} and the original loss function ℓ .

The base case for primitive programs is trivial, we simply perform a traced evaluation and leave the loss $\mathcal{L} = \mathcal{L}'$ invariant

$$\frac{c, \tau, \rho, w \leftarrow f(c')}{\langle \mathcal{L}, \ell, (c, \tau, \rho, w) \rangle \leftarrow \langle \mathcal{L}, \ell, f(c') \rangle}$$

The semantics of evaluation in context for **compose**, **extend**, and **resample** are similarly trivial, in the sense that they only serve thread their input \mathcal{L} through the evaluation. We show inference rules for these combinators in Figure 1. The only evaluation in which loss terms are computed is that of the **propose** combinator, for which we define the semantics

$$\frac{\mathcal{L}_2 = \ell(\rho_1, \rho_2, w_1, w_2/u_1) + \mathcal{L}_1 \quad \langle \mathcal{L}_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle \leftarrow \langle \mathcal{L}_0, \ell, q(c_0) \rangle \quad c_2, \tau_2, \rho_2, w_2 \leftarrow p(c_0)[\tau_1] \quad c_3, \tau_3, \rho_3, w_3 \leftarrow \text{marginal}(p)(c_0)[\tau_2] \quad u_1 = \prod_{\alpha \in \text{dom}(\rho_1) \setminus (\text{dom}(\tau_1) \cup \text{dom}(\tau_2))} \rho_1(\alpha)}{\langle \mathcal{L}_2, \ell, (c_3, \tau_3, \rho_3, w_2 \cdot w_1/u_1) \rangle \leftarrow \langle \mathcal{L}_0, \ell, \text{propose}(p, q)(c_0) \rangle}$$

$$\begin{array}{c}
\langle \mathcal{L}_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle \leftarrow \langle \mathcal{L}_0, \ell, q_1(c_0) \rangle \quad \langle \mathcal{L}_2, \ell, (c_2, \tau_2, \rho_2, w_2) \rangle \leftarrow \langle \mathcal{L}_1, \ell, q_2(c_1) \rangle \\
\text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \\
\hline
\langle \mathcal{L}_2, \ell, (c_2, \tau_2 \oplus \tau_1, \rho_2 \oplus \rho_1, w_2 \cdot w_1) \rangle \leftarrow \langle \mathcal{L}_0, \ell, \text{compose}(q_2, q_1)(c_0) \rangle \\
\langle \mathcal{L}_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle \leftarrow \langle \mathcal{L}_0, \ell, p(c_0) \rangle \quad \langle \mathcal{L}_2, \ell, (c_2, \tau_2, \rho_2, w_2) \rangle \leftarrow \langle \mathcal{L}_1, \ell, f(c_1) \rangle \\
\text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \text{dom}(\rho_2) = \text{dom}(\tau_2) \\
\hline
\langle \mathcal{L}_2, \ell, (c_2, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2) \rangle \leftarrow \langle \mathcal{L}_0, \ell, \text{extend}(p, f)(c_0) \rangle \\
\langle \mathcal{L}_1, \ell, (\vec{c}_1, \vec{\tau}_1, \vec{\rho}_1, \vec{w}_1) \rangle \leftarrow \langle \mathcal{L}_0, \ell, q(\vec{c}_0) \rangle \quad \vec{a}_1 \sim \text{RESAMPLE}(\vec{w}_1) \\
\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2 = \text{REINDEX}(\vec{a}_1, \vec{c}_1, \vec{\tau}_1, \vec{\rho}_1) \quad \vec{w}_2 = \text{MEAN}(\vec{w}_1) \\
\hline
\langle \mathcal{L}_1, \ell, (\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2, \vec{w}_2) \rangle \leftarrow \langle \mathcal{L}_0, \ell, \text{resample}(q)(\vec{c}_0) \rangle
\end{array}$$

Figure 1: Operational semantics for evaluating `compose`, `extend`, and `resample` combinators in the context of a loss function ℓ and accumulated loss \mathcal{L} .

E PROPER WEIGHTING OF PROGRAMS

Lemma 1 (Strict proper weighting of the extend combinator). *Evaluation of a target program $p_2 = \text{extend}(p_1, f)$ is strictly properly weighted for its unnormalized density $\llbracket p_2 \rrbracket_\gamma$ when evaluation of p_1 is strictly properly weighted for $\llbracket p_1 \rrbracket_\gamma$.*

Proof. Recall from Section 3.4, that the program p_2 denotes a composition

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftarrow p_1(c_0) \quad c_f, \tau_f, \rho_f, w_f \leftarrow f(c_1) \quad \tau_2 = \tau_1 \oplus \tau_f \quad \rho_2 = \rho_1 \oplus \rho_f}{c_1, \tau_2, \rho_2, w_1 \leftarrow p_2(c_0) \quad \llbracket p_2 \rrbracket_\gamma(\tau_2; c_0) = \llbracket p_1 \rrbracket_\gamma(\tau_1; c_0) \cdot \llbracket f \rrbracket_\gamma(\tau_f; c_1)} \quad (1)$$

Our induction hypothesis is that p_1 is strictly properly weighted for its density $\llbracket p_1 \rrbracket_\gamma := \gamma_1$. Since the primitive program f may only include unobserved variables, $w_f = 1$ and its evaluation is properly weighted relative to the prior density $\llbracket f \rrbracket_\gamma = \llbracket f \rrbracket_p := p_f$. Strict proper weighting with respect to $\llbracket p_2 \rrbracket_\gamma = \gamma_2$ follows directly from definitions

$$\begin{aligned}
\mathbb{E}_{p_2(c_0)} [w_1 h(\tau_2)] &= \mathbb{E}_{p_1(c_0)} [w_1 \mathbb{E}_{f(c_1)} [h(\tau_1 \oplus \tau_f)]], \\
&= \int d\tau_1 \gamma_1(\tau_1; c_0) \int d\tau_f p_f(\tau_f; c_1) h(\tau_1 \oplus \tau_f), \\
&= \int d\tau_1 d\tau_f \gamma_2(\tau_1 \oplus \tau_f; c_0) h(\tau_1 \oplus \tau_f) \\
&= \int d\tau_2 \gamma_2(\tau_2; c_0) h(\tau_2).
\end{aligned}$$

Note in particular that $Z_1(c_0) = Z_2(c_0)$, since the normalizing constant $Z_f(c_1) = 1$ for the program f . □

Theorem 1 (Strict proper weighting of target programs). *Evaluation of a target program p is (strictly) properly weighted for the unnormalized density $\llbracket p \rrbracket_\gamma$ that it denotes.*

Proof. By induction on the grammar $p ::= f \mid \text{extend}(p, f)$.

- *Base case:* $p = f$. This follows from Proposition 1.
- *Inductive case:* $p_2 = \text{extend}(p_1, f)$. This follows from Lemma 1. □

Lemma 2 (Strict proper weighting of the resample combinator). *Evaluation of a program $q_2 = \text{resample}(q_1)$ is strictly properly weighted for its unnormalized density $\llbracket q_2 \rrbracket_\gamma$ when evaluation of q_1 is strictly properly weighted for $\llbracket q_1 \rrbracket_\gamma$.*

Proof. In Section 3.4, we defined the operational semantics for the `resample` combinator as

$$\frac{\vec{c}_1, \vec{\tau}_1, \vec{\rho}_1, \vec{w}_1 \leftarrow \mathbf{q}_1(\vec{c}_0) \quad \vec{a}_1 \sim \text{RESAMPLE}(\vec{w}_1) \quad \vec{c}_2, \vec{\tau}_2, \vec{\rho}_2 = \text{REINDEX}(\vec{a}_1, \vec{c}_1, \vec{\tau}_1, \vec{\rho}_1) \quad \vec{w}_2 = \text{MEAN}(\vec{w}_1)}{\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2, \vec{w}_2 \leftarrow \text{resample}(\mathbf{q}_1)(\vec{c}_0)}$$

Whereas other combinators act on samples individually, the `resample` combinator accepts and returns a collection of samples. Bold notation signifies tensorized objects. For notational simplicity, we assume in this proof that all objects contain a single dimension, e.g. $\vec{\tau} = [\tau^1, \dots, \tau^L]$, which is also the dimension along which resampling is performed, but this is not a requirement in the underlying implementation.

Let $\llbracket \mathbf{q}_2(c_0) \rrbracket_\gamma = \llbracket \mathbf{q}_1(c_0) \rrbracket_\gamma = \gamma(\cdot; c_0)$ denote the unnormalized density of the program. Our goal is to demonstrate that outgoing samples are individually strictly properly weighted for the unnormalized density $\gamma(\cdot; c_0)$,

$$\mathbb{E}_{\mathbf{q}_2(c_0)} [w_2^l h(\tau_2^l)] = \int d\tau' \gamma(\tau'; c_0) h(\tau'),$$

under the inductive hypothesis that incoming samples are strictly properly weighted,

$$\mathbb{E}_{\mathbf{q}_1(c_0)} [w_1^l h(\tau_1^l)] = \int d\tau' \gamma(\tau'; c_0) h(\tau').$$

The `resample` combinator randomly selects ancestor indices $\vec{a}_1 \sim \text{RESAMPLE}(\vec{w}_1)$. Informally, this procedure selects $a_1^l = k$ with probability proportional to w_1^k . More formally, this procedure must satisfy

$$\mathbb{E}_{\text{RESAMPLE}(\vec{w}_1)} [\mathbb{I}[a^l = k]] = \frac{w_1^k}{\sum_l w_1^l}.$$

The outgoing return value, trace, and weight, are then reindexed according to \vec{a}_1 , whereas the outgoing weights are set to the average of the incoming weights

$$c_2^l = c_1^{a_1^l}, \quad \tau_2^l = \tau_1^{a_1^l}, \quad \rho_2^l = \rho_1^{a_1^l}, \quad w_2^l = \frac{1}{L} \sum_{l'=1}^L w_1^{l'}.$$

Strict proper weighting now follows directly from definitions

$$\begin{aligned} \mathbb{E}_{\mathbf{q}_2(c_0)} [w_2^l h(\tau_2^l)] &= \mathbb{E}_{\mathbf{q}_2(c_0)} \left[\left(\frac{1}{L} \sum_{l'} w_1^{l'} \right) h(\tau_1^{a_1^l}) \right] \\ &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\left(\frac{1}{L} \sum_{l'} w_1^{l'} \right) \mathbb{E}_{\text{RESAMPLE}(\vec{w}_1)} [h(\tau_1^{a_1^l})] \right] \\ &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\left(\frac{1}{L} \sum_{l'} w_1^{l'} \right) \mathbb{E}_{\text{RESAMPLE}(\vec{w}_1)} \left[\sum_k \mathbb{I}[a_1^l = k] h(\tau_1^k) \right] \right] \\ &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\left(\frac{1}{L} \sum_{l'} w_1^{l'} \right) \sum_k \mathbb{E}_{\text{RESAMPLE}(\vec{w}_1)} [\mathbb{I}[a_1^l = k]] h(\tau_1^k) \right] \\ &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\left(\frac{1}{L} \sum_{l'} w_1^{l'} \right) \sum_k \frac{w_1^k}{\sum_{l''} w_1^{l''}} h(\tau_1^k) \right] \\ &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\frac{1}{L} \sum_k w_1^k h(\tau_1^k) \right] = \frac{1}{L} \sum_k \mathbb{E}_{\mathbf{q}_1(c_0)} [w_1^k h(\tau_1^k)]. \end{aligned}$$

□

Lemma 3 (Strict proper weighting of the `compose` combinator). *Evaluation of the program $\mathbf{q}_3 = \text{compose}(\mathbf{q}_1, \mathbf{q}_2)$ is strictly properly weighted for its unnormalized density $\llbracket \text{compose}(\mathbf{q}_1, \mathbf{q}_2) \rrbracket_\gamma$ when evaluation of \mathbf{q}_1 and \mathbf{q}_2 is strictly properly weighted for the unnormalized densities $\llbracket \mathbf{q}_1 \rrbracket_\gamma$ and $\llbracket \mathbf{q}_2 \rrbracket_\gamma$.*

Proof. In Section 3.4, we defined the operational semantics for the compose combinator as

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftarrow q_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftarrow q_2(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \tau_3 = \tau_2 \oplus \tau_1 \quad \rho_3 = \rho_2 \oplus \rho_1 \quad w_3 = w_2 \cdot w_1}{c_2, \tau_3, \rho_3, w_3 \leftarrow \text{compose}(q_2, q_1)(c_0)},$$

and its denotation as the product of conditional densities

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftarrow q_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftarrow q_2(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset}{\llbracket \text{compose}(q_2, q_1)(c_0) \rrbracket(\tau_1 \oplus \tau_2) = \llbracket q_2(c_1) \rrbracket(\tau_2) \llbracket q_1(c_0) \rrbracket(\tau_1)}.$$

Let $\llbracket q_3 \rrbracket_\gamma = \gamma_3$ denote the unnormalized density of the composition. We will show that, for any measurable function $h(\tau_3)$,

$$\mathbb{E}_{q_3(c_0)} [w_3 h(\tau_3)] = Z_3(c_0) \int d\tau_3 \gamma_3(\tau_3; c_0) h(\tau_3).$$

We can express the expectation with respect to the program q_3 as a nested expectation with respect to q_1 and q_2

$$\mathbb{E}_{q_3(c_0)} [w_3 h(\tau_3)] = \mathbb{E}_{q_1(c_0)} \left[w_1 \mathbb{E}_{q_2(c_1)} [w_2 h(\tau_2 \oplus \tau_1)] \right],$$

Let $\llbracket q_1 \rrbracket_\gamma = \gamma_1$, $\llbracket q_2 \rrbracket_\gamma = \gamma_2$ of the inputs and their composition. By the induction hypothesis, we can express rewrite both expectations as integrals with respect to γ_1 and γ_2 . Strict proper weighting follows from definitions

$$\begin{aligned} \mathbb{E}_{q_3(c_0)} [w_3 h(\tau_3)] &= \int d\tau_1 \gamma_1(\tau_1; c_0) \int d\tau_2 \gamma_2(\tau_2; c_1) h(\tau_1 \oplus \tau_2; c_0) \\ &= \int d\tau_1 \int d\tau_2 \gamma_1(\tau_1; c_0) \gamma_2(\tau_2; c_1) h(\tau_1 \oplus \tau_2; c_0) \\ &= \int d\tau_3 \gamma_3(\tau_3; c_0) h(\tau_3; c_0). \end{aligned}$$

□

Lemma 4 (Strict proper weighting of the propose combinator). *Evaluation of a program $\text{propose}(p, q)$ is strictly properly weighted for the unnormalized density $\llbracket \text{propose}(p, q) \rrbracket_\gamma$ when evaluation of q is strictly properly weighted for the unnormalized density $\llbracket q \rrbracket_\gamma$.*

Proof. Recall from Section 3.4 that that the operational semantics for propose are

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

Our aim is to demonstrate that evaluation of $\text{propose}(p, q)$ is strictly properly weighted for

$$\llbracket \text{propose}(p, q)(c_0) \rrbracket_\gamma = \llbracket \text{marginal}(p)(c_0) \rrbracket_\gamma = \gamma_p(\cdot; c_0).$$

This is to say that, for any measurable $h(\tau_3)$

$$\mathbb{E}_{q(c_0)} \left[\frac{w_2 w_1}{u_1} h(\tau_3) \right] = \int d\tau_3 \gamma_p(\cdot; c_0) h(\tau_3).$$

We start by expressing the expectation with respect to $q_2(c_0)$ as an expectation with respect to $q_1(c_0)$ and $p(c_0)[\tau_1]$, and use the inductive hypothesis to express the first expectation as an integral with respect to $\llbracket q(c_0) \rrbracket_\gamma = \gamma_q(\cdot; c_0)$,

$$\begin{aligned} \mathbb{E}_{q_2(c_0)} \left[\frac{w_2 w_1}{u_1} h(\tau_3) \right] &= \mathbb{E}_{q_1(c_0)} \left[w_1 \mathbb{E}_{p(c_0)[\tau_1]} \left[\frac{w_2}{u_1} h(\tau_3) \right] \right] \\ &= \int d\tau_1 \gamma_q(\tau_1; c_0) \mathbb{E}_{p(c_0)[\tau_1]} \left[\frac{w_2}{u_1} h(\tau_3) \right]. \end{aligned}$$

We use $\llbracket p(c_0) \rrbracket = \tilde{\gamma}_p(\cdot; c_0)$ to refer to the density that p denotes, which possibly extends the density $\gamma_p(\cdot; c_0)$ using one or more primitive programs. We then use Equation 3 to replace w_2/u_1 with the relevant extended-space densities, and simplify the resulting expressions

$$\begin{aligned}
\int d\tau_1 \gamma_1(\cdot; c_0) \mathbb{E}_{p(c_0)[\tau_1]} \left[\frac{w_2}{u_1} h(\tau_3) \right] &= \int d\tau_1 \gamma_q(\tau_1; c_0) \mathbb{E}_{p(c_0)[\tau_1]} \left[\frac{\tilde{\gamma}_p(\tau_2; c_0) p_{q[\tau_2]}(\tau_1; c_0)}{\gamma_q(\tau_1; c_0) p_{p[\tau_1]}(\tau_2; c_0)} h(\tau_3) \right] \\
&= \int d\tau_1 \frac{\gamma_q(\tau_1; c_0)}{\gamma_q(\tau_1; c_0)} \mathbb{E}_{p(c_0)[\tau_1]} \left[\frac{\tilde{\gamma}_p(\tau_2; c_0) p_{q[\tau_2]}(\tau_1; c_0)}{p_{p[\tau_1]}(\tau_2; c_0)} h(\tau_3) \right] \\
&= \int d\tau_1 \int d\tau_2 p_{p[\tau_1]}(\tau_2; c_0) \frac{\tilde{\gamma}_p(\tau_2; c_0) p_{q[\tau_2]}(\tau_1; c_0)}{p_{p[\tau_1]}(\tau_2; c_0)} h(\tau_3) \\
&= \int d\tau_1 \int d\tau_2 \frac{p_{p[\tau_1]}(\tau_2; c_0)}{p_{p[\tau_1]}(\tau_2; c_0)} \tilde{\gamma}_p(\tau_2; c_0) p_{q[\tau_2]}(\tau_1; c_0) h(\tau_3) \\
&= \int d\tau_2 \tilde{\gamma}_p(\tau_2; c_0) h(\tau_3) \int d\tau_1 p_{q[\tau_2]}(\tau_1; c_0), \\
&= \int d\tau_3 \gamma_p(\tau_3; c_0) h(\tau_3).
\end{aligned}$$

In the final equality, we rely on the fact that $\gamma_p(\tau_3; c_0)$ is the marginal of $\tilde{\gamma}_p(\tau_2; c_0)$ with respect to the set of auxiliary variables $\text{dom}(\tau_2) \setminus \text{dom}(\tau_3)$. □

Theorem 2 (Strict proper weighting of compound inference programs). *Compound inference programs q are strictly properly weighted for their unnormalized densities γ_q .*

Proof. By induction on the grammar for q .

- *Base case:* $q = p$. Theorem 1 above provides a proof.
- *Inductive case:* $q_2 = \text{resample}(q_1)$. This follows from Lemma 2.
- *Inductive case:* $q_3 = \text{compose}(q_1, q_2)$. This follows from Lemma 3.
- *Inductive case:* $q_2 = \text{propose}(p, q_1)$. This follows from Lemma 4. □

F GRADIENT COMPUTATIONS

F.1 STOCHASTIC VARIATIONAL INFERENCE (SVI).

Let $q_2 = \text{propose}(p, q_1)$ be a program in which the initial inference program q_1 , target program p , and inference program q_2 denote the densities

$$\llbracket q_1(c_0) \rrbracket_\gamma = \gamma_q(\cdot; c_0, \phi) \quad \llbracket p(c_0) \rrbracket_\gamma = \tilde{\gamma}_p(\cdot; c_0, \theta), \quad \llbracket q_2(c_0) \rrbracket_\gamma = \gamma_p(\cdot; c_0, \theta),$$

with parameters θ and ϕ respectively. Notice that the target program p and the inference program q_2 denote the same density $\llbracket p(c_0) \rrbracket_\gamma = \llbracket q_2(c_0) \rrbracket_\gamma$ and hence, as a result of Theorem 1, the evaluation of q_2 is strictly properly weighted for γ_p . Applying definition 1 we can now write

$$Z_p(c_0; \theta) \mathbb{E}_{\pi_p(\cdot; c_0)}[h(\tau)] = \mathbb{E}_{q_2(c_0)}[w_2 h(\tau_2)], \quad c_2, \tau_2, \rho_2, w_2 \leftarrow q_2(c_0),$$

for any measurable function h . Given evaluations $c_1, \tau_1, \rho_1, w_1 \sim q_1(c_0)$ and $c'_2, \tau'_2, \rho'_2, w'_2 \sim p[\tau_1](c_0)$ we can similarly compute a stochastic lower bound (Burda et al., 2016),

$$\mathcal{L} = \mathbb{E}_{q_2(c_0)}[\log w_2] \leq \log \left(\mathbb{E}_{q_2(c_0)}[w_2] \right) = \log \left(Z_p(c_0, \theta) \mathbb{E}_{\pi(\cdot; c_0)}[1] \right) = \log Z_p(c_0, \theta),$$

using the constant function $h(\tau) = 1$. The gradient of this bound

$$\begin{aligned}\nabla_{\theta} \mathbb{E}_{\mathbf{q}_2(c_0)} [\log w_2] &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\nabla_{\theta} \log w_1 + \nabla_{\theta} \mathbb{E}_{\mathbf{p}(c_0)[\tau_1]} \left[\log \frac{\tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta) p_{q[\tilde{\tau}_2]}(\tau_1; c_0, \phi)}{\gamma_q(\tau_1; c_0, \phi) p_{p[\tau_1]}(\tilde{\tau}_2; c_0, \theta)} \right] \right] \\ &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\nabla_{\theta} \mathbb{E}_{\mathbf{p}(c_0)[\tau_1]} \left[\log \frac{\tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta) p_{q[\tilde{\tau}_2]}(\tau_1; c_0, \phi)}{p_{p[\tau_1]}(\tilde{\tau}_2; c_0, \theta)} \right] \right]\end{aligned}$$

is a biased estimate of $\nabla_{\theta} \log Z_p$, where we use Equation 3 to replace w_2 with the incoming importance weight w_1 and the relevant extended-space densities.

If the target program does not introduce additional random variables, i.e. $\text{dom}(\tilde{\tau}_2) \setminus \text{dom}(\tau_1) = \emptyset$, the traces produced by the conditioned evaluation $\tilde{c}_2, \tilde{\tau}_2, \tilde{\rho}_2, \tilde{w}_2 \sim \mathbf{p}[\tau_1](c_0)$ do not depend on θ , as all variables are generated from the inference program, and the prior term $p_{p[\tau_1]}(\tilde{\tau}_2; c_0) = 1$. As a result we can move the gradient operator inside the inner expectation,

$$\begin{aligned}\mathbb{E}_{\mathbf{q}_1(c_0)} \left[\nabla_{\theta} \mathbb{E}_{\mathbf{p}(c_0)[\tau_1]} \left[\log \tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta) p_{q[\tilde{\tau}_2]}(\tau_1; c_0) \right] \right] &= \mathbb{E}_{\mathbf{q}_1(c_0)} \left[\mathbb{E}_{\mathbf{p}(c_0)[\tau_1]} \left[\nabla_{\theta} \log \tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta) p_{q[\tilde{\tau}_2]}(\tau_1; c_0) \right] \right] \\ &= \mathbb{E}_{\mathbf{q}_2(c_0)} \left[\nabla_{\theta} \log \tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta) p_{q[\tilde{\tau}_2]}(\tau_1; c_0) \right].\end{aligned}$$

If, additionally, all random variables in the inference program are reused in the target program, i.e. $\text{dom}(\tau_1) \setminus \text{dom}(\tilde{\tau}_2) = \emptyset$, the prior term $p_{q[\tilde{\tau}_2]}(\tau_1; c_0) = 1$. Hence, in the case where the set of random variables in the proposal and target program is the same, i.e. $\text{dom}(\tilde{\tau}_2) = \text{dom}(\tau_1)$, we recover the standard variational inference gradient w.r.t. the model parameters θ ,

$$\mathbb{E}_{\mathbf{q}_2(c_0)} \left[\nabla_{\theta} \log \gamma_p(\tilde{\tau}_2; c_0, \theta) \right].$$

The gradient with respect to the proposal parameters $\nabla_{\phi} \mathcal{L}$ 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). Here we only consider the fully reparameterized case, which allows us to move the gradient operator inside the expectation

$$\begin{aligned}\nabla_{\phi} \mathbb{E}_{\mathbf{q}_2(c_0)} [\log w_2] &= \mathbb{E}_{\mathbf{q}_2(c_0)} [\nabla_{\phi} \log w_2] \\ &= \mathbb{E}_{\mathbf{q}_2(c_0)} \left[\frac{\partial \log w_2}{\partial \tilde{\tau}_2} \frac{\partial \tilde{\tau}_2}{\partial \phi} + \frac{\partial \log w_2}{\partial \phi} \right] \\ &= \mathbb{E}_{\mathbf{q}_2(c_0)} \left[\frac{\partial \log w_2}{\partial \tilde{\tau}_2} \frac{\partial \tilde{\tau}_2}{\partial \phi} + \frac{\partial}{\partial \phi} \log \frac{p_{q[\tilde{\tau}_2]}(\tau_1; c_0, \phi)}{\gamma_q(\tau_1; c_0, \phi)} \right]\end{aligned}$$

In the case where the set of random variables in the proposal and target program is the same, i.e. $\text{dom}(\tilde{\tau}_2) = \text{dom}(\tau_1)$ and $\mathbf{q}_1 = \mathbf{f}$ is a primitive program, we can write $w_1 = \gamma_f(\tau_1; c_0, \phi) / p_f(\tau_1; c_0, \phi)$, and we recover the standard variational inference gradient w.r.t. ϕ ,

$$\begin{aligned}\mathbb{E}_{\mathbf{q}_2(c_0)} \left[\frac{\partial}{\partial \tilde{\tau}_2} \log \left(\frac{\tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta)}{p_f(\tau_1; c_0, \phi)} \right) \frac{\partial \tilde{\tau}_2}{\partial \phi} - \frac{\partial}{\partial \phi} \log p_f(\tau_1; c_0, \phi) \right] \\ = \mathbb{E}_{\mathbf{q}_2(c_0)} \left[\frac{\partial}{\partial \tilde{\tau}_2} \log \left(\frac{\tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta)}{p_f(\tau_1; c_0, \phi)} \right) \frac{\partial \tilde{\tau}_2}{\partial \phi} - \frac{\partial}{\partial \phi} \log p_f(\tau_1; c_0, \phi) \right] \\ = -\nabla_{\phi} \text{KL}(p_f || \pi_p),\end{aligned}$$

effectively minimizing a reverse KL-divergence. Noticing that the second term is zero in expectation, due to the reinforce trick, we can derive the lower variance gradient

$$\mathbb{E}_{\mathbf{q}_2(c_0)} \left[\frac{\partial}{\partial \tilde{\tau}_2} \log \frac{\gamma_p(\tilde{\tau}_2; c_0, \theta)}{p_f(\tau_1; c_0, \phi)} \frac{\partial \tilde{\tau}_2}{\partial \phi} - \frac{\partial}{\partial \phi} \log p_f(\tau_1; c_0, \phi) \right] = \mathbb{E}_{\mathbf{q}_2(c_0)} \left[\frac{\partial}{\partial \tilde{\tau}_2} \log \frac{\gamma_p(\tilde{\tau}_2; c_0, \theta)}{p_f(\tau_1; c_0, \phi)} \frac{\partial \tilde{\tau}_2}{\partial \phi} \right],$$

which can be approximated using reparameterized weights obtained by evaluating \mathbf{q}_2 .

F.2 REWEIGHTED WAKE-SLEEP (RWS) STYLE INFERENCE.

To implement variational methods inspired by reweighted wake-sleep (Hinton et al., 1995; Bornschein, Bengio, 2015; Le et al., 2019), we compute a self-normalized estimate of the gradient

$$\begin{aligned}
 \nabla_{\theta} \log Z_p(c_0, \theta) &= \frac{1}{Z_p(c_0, \theta)} \nabla_{\theta} \int d\tau \gamma_p(\tau; c_0, \theta) \\
 &= \frac{1}{Z_p(c_0, \theta)} \int d\tau \gamma_p(\tau; c_0, \theta) \nabla_{\theta} \log \gamma_p(\tau; c_0, \theta) \\
 &= \int d\tau \pi_p(\tau; c_0, \theta) \nabla_{\theta} \log \gamma_p(\tau; c_0, \theta) \\
 &= \mathbb{E}_{\pi_p(\cdot; c_0, \theta)} [\nabla_{\theta} \log \gamma_p(\tau; c_0, \theta)].
 \end{aligned}$$

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. In practice this allows us to compute lower variance approximation. Using definition 1 and traces obtained from the evaluation of our inference program $c_2^l, \tau_2^l, \rho_2^l, w_2^l \leftarrow q_2(c_0)$ we can derive a consistent self-normalized estimator

$$\mathbb{E}_{\pi_p(\cdot; c_0, \theta)} [\nabla_{\theta} \log \gamma_p(\tau; c_0, \theta)] = Z_p^{-1}(c_0, \theta) \mathbb{E}_{q_2} [w_2 \nabla_{\theta} \log \gamma_p(\tau_2; c_0, \theta)] \simeq \sum_l \frac{w_2^l}{\sum_{l'} w_2^{l'}} \nabla_{\theta} \log \gamma_p(\tau_2^l; c_0, \theta).$$

We can similarly approximate the gradient of the forward KL divergence with a self-normalized estimator,

$$\begin{aligned}
 -\nabla_{\phi} \text{KL}(\pi_p || \pi_q) &= \mathbb{E}_{\pi_p(\cdot; c_0, \theta)} [\nabla_{\phi} \log \pi_q(\tau; c_0, \phi)] \\
 &= \mathbb{E}_{\pi_p(\cdot; c_0, \theta)} [\nabla_{\phi} \log \gamma_q(\tau; c_0, \phi)] - \nabla_{\phi} Z_q(c_0, \phi) \\
 &= \mathbb{E}_{\pi_p(\cdot; c_0, \theta)} [\nabla_{\phi} \log \gamma_q(\tau; c_0, \phi)] - \mathbb{E}_{\pi_q(\cdot; c_0, \theta)} [\nabla_{\phi} \log \gamma_q(\tau; c_0, \phi)] \\
 &= Z_p^{-1}(c_0, \theta) \mathbb{E}_{q_2} [w_2 \nabla_{\phi} \log \gamma_q(\tau_2; c_0, \phi)] - Z_q^{-1}(c_0, \phi) \mathbb{E}_{q_1} [w_1 \nabla_{\phi} \log \gamma_q(\tau_1; c_0, \phi)] \\
 &\simeq \sum_l \left(\frac{w_2^l}{\sum_{l'} w_2^{l'}} - \frac{w_1^l}{\sum_{l'} w_1^{l'}} \right) \nabla_{\phi} \log \gamma_q(\tau_1^l; c_0, \phi).
 \end{aligned}$$

In the special case where the proposal $q_1 = f$ is a primitive program without observations (i.e. $w_1^l = 1$) we have that

$$\mathbb{E}_{\pi_q(\cdot; c_0, \theta)} [\nabla_{\phi} \log \gamma_q(\tau; c_0, \phi)] = \mathbb{E}_{\pi_q(\cdot; c_0, \theta)} [\nabla_{\phi} \log \pi_q(\tau; c_0, \phi)] = 0,$$

by application of the reinforce trick. In this case we drop the second term, which is introducing additional bias, through self-normalization, and variance to the estimator, and recover the standard RWS estimator

$$\sum_l \frac{w_2^l}{\sum_{l'} w_2^{l'}} \nabla_{\phi} \log \gamma_q(\tau_1^l; c_0, \phi).$$

G IMPLEMENTATION DETAILS FOR EXPERIMENTS

G.1 ANNEALED VARIATIONAL INFERENCE

In the annealing task we implement Annealed Variational Inference (Zimmermann et al., 2021) and learn to sample from a multimodal Gaussian distribution γ_K , composed of eight equidistantly spaced modes with covariance matrix of $I \cdot 0.5$ on a circle of radius 10. We define our initial inference program and annealing path to be:

$$\llbracket q_0(c) \rrbracket = \text{Normal}(\mu = 0, \sigma = 5) \quad \llbracket q_k(c) \rrbracket_{\gamma} = q_1(c)^{1-\beta_k} \gamma(c)_{K}^{\beta_k}, \quad \beta_k = \frac{k-1}{K-1}, \quad \text{for } k = 1 \dots K$$

Implementations of these programs can be found in Figure 2. Additionally, each forward- and reverse- kernel program $q_k(c)$ is defined by a neural network:

```

def target(s, x):
    zs, xs = s.sample(Categorical(K), "zs"), []
    for k in K:
        count = sum(zs==k)
        dist = Normal(mu(k), sigma(k))
        xs.append(s.sample(dist, str(k), count))
    return s, shuffle(cat(xs))

def q_0(s):
    c = s.sample(Normal(mu(k), sigma(k)), "q_0")
    return s, c

eta = ... # initialize neural network for kernel k
def q_k(s, c):
    c' = s.sample(Normal(eta_mu(k), eta_sigma(k)), "q_k")
    return s, c'

def gamma_k(s, log_gamma, q_0, beta=1.0):
    # sample from the initial proposal
    x = s.sample(q_0, "x")
    # add a heuristic factor
    s.factor(beta * (log_gamma(x) - q_0.log_prob(x)))
    return x

```

Figure 2: models defined for Annealed Variational Inference

$$x = \text{Linear } 50. \text{ReLU}(c) \quad \mu_k = \text{Linear } 2(x) + c \quad \text{cov}_k = \text{DiagEmbed} . \text{Softplus } 2(x)$$

Learned intermediate densities of β_k are embedded by a logit function and is extracted by sigmoid function.

A combinators implementation of nested variational inference is defined:

```

def step(q, intermediate, do_resample):
    (fwd, rev), p = intermediate
    q' = resample(q) if do_resample else q
    return propose(extend(p, rev), compose(q', fwd))

path, kernels = ...
ixs = list(range(len(path)))
do_resamples = map(lambda i -> i == ixs[-1], ixs)
nvir = reduce(step, zip(kernels, path[1:], do_resamples[1:], path[0]))

```

We implement nested variational inference (NVI), nested variational inference with resampling (NVIR), as well as nested variational inference with learned intermediate densities (NVI*), and nested variational inference with resampling and learned intermediate densities (NVI*). When implementing any NVI algorithm with resampling, we additionally implement nested variational inference with resampling (NVIR*) by applying the `resample` combinator after all but the final proposal combinator.

We evaluate our model by training each model for 20,000 iterations with a sampling budget of 288 samples, distributed across K intermediate densities. Metrics of $\log \hat{Z}$ and effective sample size average over 100 batches of 1,000 samples and results are calculated using the mean of 10 training runs using unique, fixed seeds. In the evaluation of NVIR* we do not resample at test time.

G.2 AMORTIZED POPULATION GIBBS SAMPLERS

Many inference task requires learning a deep generative model. For this purpose, we we evaluate combinators in an unsupervised tracking task. In this task, the data is a corpus of simulated videos that each contain multiple moving objects. Out goal is to learn both the target program (i.e. the generative model) and the inference program using the APG sampler.

Consider a sequence of video frames $x_{1:T}$, which contains T time steps and D different objects. We assume that the k th object in the t th frame x_t can be represented by some object feature z_d^{what} and a time-dependent position variable $z_{d,t}^{\text{where}}$. The deep generative model takes the form

$$z_d^{\text{what}} \sim \text{Normal}(0, I), \quad z_{d,1}^{\text{where}} \sim \text{Normal}(0, I), \quad z_{d,t}^{\text{where}} \sim \text{Normal}(z_{d,t-1}^{\text{where}}, \sigma_0^2 I),$$

$$x_t \sim \text{Bernoulli}\left(\sigma\left(\sum_d \text{ST}(g_\theta(z_d^{\text{what}}), z_{d,t}^{\text{where}})\right)\right), \quad d = 1, 2, \dots, D, \quad t = 1, 2, \dots, T.$$

where $\sigma_0 = 0.1$ and $z_d^{\text{what}} \in \mathbb{R}^{10}$, $z_{d,t}^{\text{where}} \in \mathbb{R}^2$. To perform inference for this model, APG sampler learns neural proposals to iterate conditional updates to blocks of variables, which consists of one block of object features and T blocks of each time-dependent object position as

$$\{z_{1:D}^{\text{what}}\}, \quad \{z_{1:D,1}^{\text{where}}\}, \quad \{z_{1:D,2}^{\text{where}}\}, \quad \dots, \quad \{z_{1:D,T}^{\text{where}}\}$$

We train the model on 10000 video instances, each containing 10 timesteps and 3 different objects. We train with batch size 5, sample size 20, Adam optimizer with $\beta_1 = 0.9, \beta_2 = 0.99$ and $\text{lr} = 2e - 4$.

Architecture for Generative Model. We learn a deep generative model of the form

$$p_\theta(x_{1:T} | z_{1:D}^{\text{what}}, z_{1:T}^{\text{where}}) = \prod_{t=1}^T \text{Bernoulli}\left(x_t \mid \sigma\left(\sum_d \text{ST}(g_\theta(z_d^{\text{what}}), z_{d,t}^{\text{where}})\right)\right)$$

Given each object feature z_d^{what} , the APG sampler reconstruct a 28×28 object image using a MLP decoder, the architecture of which is

Decoder $g_\theta(\cdot)$
Input $z_d^{\text{what}} \in \mathbb{R}^{10}$
FC 200. ReLU. FC 400. ReLU. FC 784. Sigmoid.

Then we put each reconstructed image $g_\theta(z_d^{\text{what}})$ onto a 96×96 canvas using a spatial transformer ST which takes position variable $z_{d,t}^{\text{where}}$ as input. To ensure a pixel-wise Bernoulli likelihood, we clip on the composition as

$$\text{For each pixel } p_i \in \left(\sum_d \text{ST}(g_\theta(z_d^{\text{what}}), z_{d,t}^{\text{where}})\right), \quad \sigma(p_i) = \begin{cases} p_i = 0 & \text{if } p_i < 0 \\ p_i = p_i & \text{if } 0 \leq p_i \leq 1 \\ p_i = 1 & \text{if } p_i > 1 \end{cases}$$

Architecture for Gibbs Neural Proposals. The APG sampler in the bouncing object employs neural proposals of the form

$$\begin{aligned} q_\phi(z_{1:D,t}^{\text{where}} | x_t) &= \prod_{d=1}^D \text{Normal}\left(z_{d,t}^{\text{where}} \mid \tilde{\mu}_{d,t}^{\text{where}}, \tilde{\sigma}_{d,t}^{\text{where}^2} I\right), & \text{for } t = 1, 2, \dots, T, \\ q_\phi(z_{1:D,t}^{\text{where}} | x_t, z_{1:D}^{\text{what}}) &= \prod_{d=1}^D \text{Normal}\left(z_{d,t}^{\text{where}} \mid \tilde{\mu}_{d,t}^{\text{where}}, \tilde{\sigma}_{d,t}^{\text{where}^2} I\right), & \text{for } t = 1, 2, \dots, T, \\ q_\phi(z_{1:D}^{\text{what}} | x_{1:T}, z_{1:T}^{\text{where}}) &= \prod_{d=1}^D \text{Normal}\left(z_d^{\text{what}} \mid \tilde{\mu}_d^{\text{what}}, \tilde{\sigma}_d^{\text{what}^2} I\right). \end{aligned}$$

We train the proposals with instances containing $D = 3$ objects and $T = 10$ time steps and test them with instances containing up to $D = 5$ objects and $T = 100$ time steps. We use the tilde symbol $\tilde{\cdot}$ to denote the parameters of the conditional neural proposals (i.e. approximate Gibbs proposals).

The APG sampler uses these proposals to iterate over the $T + 1$ blocks

$$\{z_{1:D}^{\text{what}}\}, \quad \{z_{1:D,1}^{\text{where}}\}, \quad \{z_{1:D,2}^{\text{where}}\}, \quad \dots, \quad \{z_{1:D,T}^{\text{where}}\}.$$

For the position features, the proposal $q_\phi(z_{1:D,t}^{\text{where}} | x_t)$ and proposal $q_\phi(z_{1:D,t}^{\text{where}} | x_t, z_{1:D}^{\text{what}})$ share the same network, but contain different pre-steps where we compute the input of that network. The initial proposal $q_\phi(z_{1:D,t}^{\text{where}} | x_t)$ will convolve the frame x_t with the mean image of the object dataset; The conditional proposal $q_\phi(z_{1:D,t}^{\text{where}} | x_t, z_{1:D}^{\text{what}})$ will convolve the frame x_t with each reconstructed object image $g_\theta(z_d^{\text{what}})$. We perform convolution sequentially by looping over all objects $d = 1, 2, \dots, D$. Here is pseudocode of both pre-steps:

We employ a MLP encoder $f_\phi^L(\cdot)$ that takes the convolved features as input and predict the variational parameters for positions $\{z_{d,t}^{\text{where}}\}_{d=1}^D$ at step t , i.e. vector-valued mean $\tilde{\mu}_{d,t}^{\text{where}}$ and logarithm of the diagonal covariance $\log \tilde{\sigma}_{d,t}^{\text{where}^2}$ as

$$\tilde{\mu}_{d,t}^{\text{where}}, \log \tilde{\sigma}_{d,t}^{\text{where}^2} \leftarrow f_\phi^L(x_{d,t}^{\text{conv}}), \quad d = 1, 2, \dots, D.$$

Algorithm 1 Convolution Processing for $q_\phi(z_{1:D,t}^{\text{where}} | x_t)$

- 1: **Input** frame $x_t \in \mathbb{R}^{9216}$, mean image of object dataset $mm \in \mathbb{R}^{784}$
 - 2: **for** $d = 1$ **to** D **do**
 - 3: $x_{d,t}^{\text{conv}} \leftarrow \text{Conv2d}(x_t)$ with kernel mm , stride = 1, no padding.
 - 4: **end for**
 - 5: **Output** Convolved features $\{x_{d,t}^{\text{conv}} \in \mathbb{R}^{4761}\}_{d=1}^D$
-

Algorithm 2 Convolution Processing for $q_\phi(z_{1:D,t}^{\text{where}} | x_t, z_{1:D}^{\text{what}})$

- 1: **Input** frame $x_t \in \mathbb{R}^{9216}$, reconstructed object objects $\{g_\theta(z_d^{\text{what}}) \in \mathbb{R}^{784}\}_{d=1}^D$
 - 2: **for** $d = 1$ **to** D **do**
 - 3: $x_{d,t}^{\text{conv}} \leftarrow \text{Conv2d}(x_t)$ with kernel $g_\theta(z_d^{\text{what}})$, stride = 1, no padding.
 - 4: **end for**
 - 5: **Output** Convolved features $\{x_{d,t}^{\text{conv}} \in \mathbb{R}^{4761}\}_{d=1}^D$
-

Encoder $f_\phi^L(\cdot)$

Input $x_{d,t}^{\text{conv}} \in \mathbb{R}^{4761}$

FC 200. ReLU. FC 2×100 . ReLU. FC 2×2 .

The architecture of the MLP encoder $f_\phi^L(\cdot)$ is

For the object features, the APG sampler performs conditional updates in the sense that we crop each frame x_t into a 28×28 subframe according to $z_{d,t}^{\text{where}}$ using the spatial transformer ST as

$$x_{d,t}^{\text{crop}} \leftarrow \text{ST}(x_t, z_{d,t}^{\text{where}}), \quad d = 1, 2, \dots, D, \quad t = 1, 2, \dots, T.$$

we employ a MLP encoder $T_\phi^G(\cdot)$ that takes the cropped subframes as input, and predicts frame-wise neural sufficient statistics, which we will sum up over all the time steps.

Then we employ another network $f_\phi^G(\cdot)$ that takes the sums as input, and predict the variational parameters for object features $\{z_d^{\text{what}}\}_{d=1}^D$, i.e. the vector-valued means $\{\mu_d^{\text{what}}\}_{d=1}^D$ and the logarithms of the diagonal covariances $\{\log \tilde{\sigma}_d^{\text{what}^2}\}_{d=1}^D$. The architecture of this network is

Encoder $f_\phi^G(\cdot)$

Input $x_{d,t}^{\text{crop}} \in \mathbb{R}^{784}$
--

FC 400. ReLU. FC 200. ReLU. \rightarrow Neural Sufficient Statistics $T_\phi^G(x_{d,t}^{\text{crop}}) \in \mathbb{R}^{200}$

Intermediate Input $\sum_{t=1}^T T_\phi^G(x_{d,t}^{\text{crop}}) \rightarrow$ FC 2×10 .
--

REFERENCES

Baydin, Atılım Güneş, Le, Tuan Anh, Heinrich, Lukas, Gram-Hansen, Bradley, Schroeder de Witt, Christian, Bhimji, Wahid, Cranmer, Kyle, Wood, Frank. *Pyprob/Ppx*. pyprob. 2018.

Baydin, Atılım Güneş, Shao, Lei, Bhimji, Wahid, Heinrich, Lukas, Meadows, Lawrence F., Liu, Jialin, Munk, Andreas, Naderiparizi, Saeid, Gram-Hansen, Bradley, Louppe, Gilles, Ma, Mingfei, Zhao, Xiaohui, Torr, Philip, Lee, Victor, Cranmer, Kyle, Prabhat, Wood, Frank. “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, Eli, Chen, Jonathan P., Jankowiak, Martin, Obermeyer, Fritz, Pradhan, Neeraj, Karaletsos, Theofanis, Singh, Rohit, Szerlip, Paul, Horsfall, Paul, Goodman, Noah D. “Pyro: Deep Universal Probabilistic Programming” (Oct. 2018). arXiv: [1810.09538](https://arxiv.org/abs/1810.09538) [cs, stat].
- Borgström, Johannes, Gordon, Andrew D., Greenberg, Michael, Margetson, James, Van Gael, Jurgen. “Measure Transformer Semantics for Bayesian Machine Learning.” *Programming Languages and Systems*. Ed. by G. Barthe. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 77–96.
- Bornschein, Jörg, Bengio, Yoshua. “Reweighted Wake-Sleep.” *International Conference on Learning Representations* (2015). arXiv: [1406.2751](https://arxiv.org/abs/1406.2751).
- Burda, Yuri, Grosse, Roger, Salakhutdinov, Ruslan. “Importance Weighted Autoencoders.” *International Conference on Representations*. 2016. arXiv: [1509.00519](https://arxiv.org/abs/1509.00519).
- Carpenter, Bob, Gelman, Andrew, Hoffman, Matthew D., Lee, Daniel, Goodrich, Ben, Betancourt, Michael, Brubaker, Marcus, Guo, Jiqiang, Li, Peter, Riddell, Allen. “Stan : A Probabilistic Programming Language.” English. *Journal of Statistical Software* 76.1 (Jan. 2017). DOI: [10.18637/jss.v076.i01](https://doi.org/10.18637/jss.v076.i01).
- Caterini, Anthony L., Doucet, Arnaud, Sejdinovic, Dino. “Hamiltonian Variational Auto-Encoder” (May 2018). arXiv: [1805.11328](https://arxiv.org/abs/1805.11328) [cs, stat].
- Chopin, N. “A Sequential Particle Filter Method for Static Models.” *Biometrika* 89.3 (Aug. 2002), pp. 539–552. DOI: [10.1093/biomet/89.3.539](https://doi.org/10.1093/biomet/89.3.539).
- Clerc, Florence, Danos, Vincent, Dahlqvist, Fredrik. “Pointless Learning (long version).” *International Conference on Foundations of Software Science and Computation Structures (FoSSaCS)*. 679127. 2017, pp. 1–19.
- Cusumano-Towner, Marco F., Saad, Feras A., Lew, Alexander K., Mansinghka, Vikash 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](https://doi.org/10.1145/3314221.3314642).
- Dahlqvist, Fredrik, Silva, Alexandra, Danos, Vincent, Garnier, Ilias. “Borel Kernels and their Approximation, Categorically.” *Electronic Notes in Theoretical Computer Science* 341 (2018), pp. 91–119. DOI: [10.1016/j.entcs.2018.11.006](https://doi.org/10.1016/j.entcs.2018.11.006). arXiv: [1803.02651](https://arxiv.org/abs/1803.02651).
- De Raedt, Luc, Kimmig, Angelika, Toivonen, Hannu. “ProbLog: A Probabilistic Prolog and Its Application in Link Discovery.” *IJCAI International Joint Conference on Artificial Intelligence* (2007), pp. 2468–2473.
- Fong, Brendan. “Causal Theories: A Categorical Perspective on Bayesian Networks.” Master of Science in Mathematics and the Foundations of Computer Science. University of Oxford, 2013. arXiv: [1301.6201](https://arxiv.org/abs/1301.6201).
- Ge, Hong, Xu, Kai, Ghahramani, Zoubin. “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, Noah, Mansinghka, Vikash, Roy, Daniel M, Bonawitz, Keith, Tenenbaum, Joshua B. “Church: A Language for Generative Models.” *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*. 2008, pp. 220–229.
- Goodman, Noah D, Stuhlmüller, Andreas. *The Design and Implementation of Probabilistic Programming Languages*. 2014.
- Heunen, Chris, Kammar, Ohad, Staton, Sam, Yang, Hongseok. “A convenient category for higher-order probability theory.” *Proceedings - Symposium on Logic in Computer Science*. 2017. DOI: [10.1109/LICS.2017.8005137](https://doi.org/10.1109/LICS.2017.8005137). arXiv: [1701.02547](https://arxiv.org/abs/1701.02547).

- Hinton, G. E., Dayan, P., Frey, B. J., Neal, R. M. “The “Wake-Sleep” Algorithm for Unsupervised Neural Networks.” en. *Science* 268.5214 (May 1995), pp. 1158–1161. DOI: [10.1126/science.7761831](https://doi.org/10.1126/science.7761831).
- Hoffman, Matthew D. “Learning deep latent Gaussian models with Markov chain Monte Carlo.” *International conference on machine learning*. PMLR. 2017, pp. 1510–1519.
- Holtzen, Steven, Van den Broeck, Guy, Millstein, Todd. “Scaling Exact Inference for Discrete Probabilistic Programs.” *Proceedings of the ACM on Programming Languages* 4.OOPSLA (Nov. 2020), 140:1–140:31. DOI: [10.1145/3428208](https://doi.org/10.1145/3428208).
- Huang, Chin-Wei, Tan, Shawn, Lacoste, Alexandre, Courville, Aaron 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, Diederik P., Welling, Max. “Auto-Encoding Variational Bayes.” *International Conference on Learning Representations* (2013).
- Le, Tuan Anh, Igl, Maximilian, Rainforth, Tom, Jin, Tom, Wood, Frank. “Auto-Encoding Sequential Monte Carlo.” *International Conference on Learning Representations*. 2018. arXiv: [1705.10306](https://arxiv.org/abs/1705.10306).
- Le, Tuan Anh, Kosiorek, Adam R., Siddharth, N., Teh, Yee Whye, Wood, Frank. “Revisiting Reweighted Wake-Sleep for Models with Stochastic Control Flow.” *Uncertainty in Artificial Intelligence*. 2019.
- Lew, Alexander K., Cusumano-Towner, Marco F., Sherman, Benjamin, Carbin, Michael, Mansinghka, Vikash K. “Trace Types and Denotational Semantics for Sound Programmable Inference in Probabilistic Languages.” *ACM Principles of Programming Languages*. Vol. 4. January. 2020, pp. 1–31. DOI: [10.1145/3371087](https://doi.org/10.1145/3371087).
- Li, Yingzhen, Turner, Richard E, Liu, Qiang. “Approximate inference with amortised mcmc.” *arXiv preprint arXiv:1702.08343* (2017).
- Liu, Jun S. *Monte Carlo strategies in scientific computing*. Springer Science & Business Media, 2008.
- Maddison, Chris J, Lawson, John, Tucker, George, Heess, Nicolas, Norouzi, Mohammad, Mnih, Andriy, Doucet, Arnaud, Teh, Yee. “Filtering Variational Objectives.” *Advances in Neural Information Processing Systems 30*. Ed. by I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett. Curran Associates, Inc., 2017, pp. 6573–6583.
- Mansinghka, Vikash, Selsam, Daniel, Perov, Yura. “Venture: A Higher-Order Probabilistic Programming Platform with Programmable Inference.” *arXiv* (Mar. 2014), pp. 78–78.
- Masrani, Vaden, Le, Tuan Anh, Wood, Frank. “The Thermodynamic Variational Objective.” *Advances in Neural Information Processing Systems*. Ed. by H. Wallach, H. Larochelle, A. Beygelzimer, F. dAlché-Buc, E. Fox, R. Garnett. Vol. 32. Curran Associates, Inc., 2019.
- Minka, T, Winn, J, Guiver, J, Knowles, D. *Infer.NET 2.4, Microsoft Research Cambridge*. 2010.
- Murray, Lawrence M. “Bayesian State-Space Modelling on High-Performance Hardware Using LibBi” (June 2013). arXiv: [1306.3277](https://arxiv.org/abs/1306.3277) [stat].
- Murray, Lawrence M., Schön, Thomas B. “Automated Learning with a Probabilistic Programming Language: Birch.” *Annual Reviews in Control* 46 (Jan. 2018), pp. 29–43. DOI: [10.1016/j.arcontrol.2018.10.013](https://doi.org/10.1016/j.arcontrol.2018.10.013).
- Naesseth, Christian, Linderman, Scott, Ranganath, Rajesh, Blei, David. “Variational Sequential Monte Carlo.” en. *International Conference on Artificial Intelligence and Statistics*. PMLR, Mar. 2018, pp. 968–977.
- Naesseth, Christian, Lindsten, Fredrik, Schon, Thomas. “Nested Sequential Monte Carlo Methods.” *International Conference on Machine Learning*. 2015, pp. 1292–1301.

- Naesseth, Christian A., Lindsten, Fredrik, Schön, Thomas B. “Elements of Sequential Monte Carlo.” *arXiv:1903.04797 [cs, stat]* (Mar. 2019). (Visited on 12/16/2019).
- Narayanan, Praveen, Carette, Jacques, Romano, Wren, Shan, Chung-chieh, Zinkov, Robert. “Probabilistic Inference by Program Transformation in Hakaru (System Description).” *International Symposium on Functional and Logic Programming*. Springer, 2016, pp. 62–79.
- Obermeyer, Fritz, Bingham, Eli, Jankowiak, Martin, Phan, Du, Chen, Jonathan P. “Functional Tensors for Probabilistic Programming.” *arXiv preprint arXiv:1910.10775* (2019).
- Paige, Brooks, Wood, Frank. “A Compilation Target for Probabilistic Programming Languages.” *International Conference on Machine Learning (ICML)* 32 (Mar. 2014).
- Pfeffer, Avi. *Figaro: An Object-Oriented Probabilistic Programming Language*. Tech. rep. 9781577354260. 2009, pp. 1–9.
- Pfeffer, Avi, Lynn, Spencer K. “Scruff: A Deep Probabilistic Cognitive Architecture for Predictive Processing.” *Biologically Inspired Cognitive Architectures Meeting*. Springer, 2018, pp. 245–259.
- Plummer, Martyn. “JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling.” *Proceedings of the 3rd International Workshop on Distributed Statistical Computing*. Vol. 124. Vienna, Austria., 2003.
- Rainforth, Tom, Kosiorek, Adam, Le, Tuan Anh, Maddison, Chris, Igl, Maximilian, Wood, Frank, Teh, Yee Whye. “Tighter Variational Bounds Are Not Necessarily Better.” en. *International Conference on Machine Learning*. July 2018, pp. 4277–4285.
- Ranganath, Rajesh, Gerrish, Sean, Blei, David. “Black Box Variational Inference.” en. *Artificial Intelligence and Statistics*. Apr. 2014, pp. 814–822.
- Rezende, Danilo Jimenez, Mohamed, Shakir, Wierstra, Daan. “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. Beijing, China: PMLR, June 2014, pp. 1278–1286.
- Ritchie, Daniel, Horsfall, Paul, Goodman, Noah D. “Deep Amortized Inference for Probabilistic Programs.” en (Oct. 2016). *arXiv: 1610.05735 [cs, stat]*.
- Salimans, Tim, Kingma, Diederik, Welling, Max. “Markov chain monte carlo and variational inference: Bridging the gap.” *International Conference on Machine Learning*. 2015, pp. 1218–1226.
- Salvatier, John, Wiecki, Thomas V, Fonnesbeck, Christopher. “Probabilistic programming in Python using PyMC3.” *PeerJ Computer Science* 2 (2016), e55.
- Schulman, John, Heess, Nicolas, Weber, Theophane, Abbeel, Pieter. “Gradient Estimation Using Stochastic Computation Graphs.” *Advances in Neural Information Processing Systems* 28. Ed. by C. Cortes, N. D. Lawrence, D. D. Lee, M. Sugiyama, R. Garnett. Curran Associates, Inc., 2015, pp. 3528–3536.
- Ścibior, Adam, Kammar, Ohad, Ghahramani, Zoubin. “Functional Programming for Modular Bayesian Inference.” *Proc. ACM Program. Lang.* 2.ICFP (July 2018), 83:1–83:29. DOI: [10.1145/3236778](https://doi.org/10.1145/3236778).
- Ścibior, Adam, Kammar, Ohad, Vákár, Matthijs, Staton, Sam, Yang, Hongseok, Cai, Yufei, Ostermann, Klaus, Moss, Sean K., Heunen, Chris, Ghahramani, Zoubin. “Denotational Validation of Higher-Order Bayesian Inference.” *Proc. ACM Program. Lang.* 2.POPL (Dec. 2017), 60:1–60:29. DOI: [10.1145/3158148](https://doi.org/10.1145/3158148).
- Shan, Chung-chieh, Ramsey, Norman. “Exact Bayesian inference by symbolic disintegration.” *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 2017, pp. 130–144.

- Siddharth, N., Paige, Brooks, van de Meent, Jan-Willem, Desmaison, Alban, Goodman, Noah D., Kohli, Pushmeet, Wood, Frank, Torr, Philip. “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.
- Spiegelhalter, David J, Thomas, Andrew, Best, Nicky G, Gilks, Wally R. *BUGS: Bayesian Inference Using Gibbs Sampling, Version 0.50*. MRC Biostatistics Unit, Cambridge. 1995.
- Tolpin, David. *Infergo: Go Programs That Learn*. en. <http://infergo.org/>. 2018.
- Tolpin, David, van de Meent, Jan-Willem, Yang, Hongseok, Wood, Frank. “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](https://doi.org/10/ghxhzn).
- Toronto, Neil, McCarthy, Jay, Van Horn, David. “Running Probabilistic Programs Backwards.” *European Symposium on Programming Languages and Systems*. Vol. 3. 1. 2015, pp. 53–79. arXiv: [1412.4053](https://arxiv.org/abs/1412.4053). URL: <http://arxiv.org/abs/1412.4053>.
- Tran, Dustin, Hoffman, Matthew D, Moore, Dave, Suter, Christopher, Vasudevan, Srinivas, Radul, Alexey, Johnson, Matthew, Saurous, Rif A. “Simple, distributed, and accelerated probabilistic programming.” *Proceedings of the 32nd International Conference on Neural Information Processing Systems*. 2018, pp. 7609–7620.
- Tran, Dustin, Kucukelbir, Alp, Dieng, Adji B., Rudolph, Maja, Liang, Dawen, Blei, David M. “Edward: A Library for Probabilistic Modeling, Inference, and Criticism” (Oct. 2016). arXiv: [1610.09787](https://arxiv.org/abs/1610.09787) [cs, stat].
- Tucker, George, Lawson, Dieterich, Gu, Shixiang, Maddison, Chris J. “Doubly Reparameterized Gradient Estimators for Monte Carlo Objectives” (Oct. 2019). arXiv: [1810.04152](https://arxiv.org/abs/1810.04152) [cs, stat].
- van de Meent, Jan-Willem, Paige, Brooks, Tolpin, David, Wood, Frank. “An Interface for Black Box Learning in Probabilistic Programs.” *POPL Workshop on Probabilistic Programming Semantics*. 2016.
- “Black-Box Policy Search with Probabilistic Programs.” 2016, 1195–1204.
- van de Meent, Jan-Willem, Paige, Brooks, Yang, Hongseok, Wood, Frank. “An Introduction to Probabilistic Programming” (Sept. 2018). arXiv: [1809.10756](https://arxiv.org/abs/1809.10756) [cs, stat].
- Wang, Tongzhou, Wu, Yi, Moore, Dave, Russell, Stuart J. “Meta-learning MCMC proposals.” *Advances in Neural Information Processing Systems*. 2018, pp. 4146–4156.
- Wingate, David, Weber, Theo. “Automated Variational Inference in Probabilistic Programming” (2013), pp. 1–7. arXiv: [1301.1299](https://arxiv.org/abs/1301.1299).
- Wood, Frank, van de Meent, Jan-Willem, Mansinghka, Vikash. “A New Approach to Probabilistic Programming Inference.” *Artificial Intelligence and Statistics*. 2014, pp. 1024–1032.
- Wu, Hao, Zimmermann, Heiko, Sennesh, Eli, Le, Tuan Anh, van de Meent, Jan-Willem. “Amortized Population Gibbs Samplers with Neural Sufficient Statistics.” *International Conference on Machine Learning*. PMLR. 2020, pp. 10421–10431.
- Zimmermann, Heiko, Wu, Hao, Esmaeili, Babak, Stites, Sam, van de Meent, Jan-Willem. “Nested Variational Inference.” *3rd Symposium on Advances in Approximate Bayesian Inference* (2021).
- Zinkov, Robert, Shan, Chung-chieh. “Composing Inference Algorithms as Program Transformations.” *Uncertainty in Artificial Intelligence* (2017). arXiv: [1603.01882](https://arxiv.org/abs/1603.01882).