Supplementary Material: Learning Proposals for Probabilistic Programs with Inference Combinators

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

[1]Khoury College of Computer Sciences, Northeastern University, Boston, Massachusetts, USA
[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 estimators[1], or by implementing doubly-reparameterized estimators (Tucker et al., 2019).

[1]Note 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 unsure 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 Hakaru (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 $\mathtt{p} ::= \mathtt{f} \mid \mathtt{extend(p,\ f)}$. For a primitive program f, the denotational semantics inherit trivially from the axiomatic denotational semantics of the modeling language. For a program extend(p, f) we define the density as the composition of densities of the inputs

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \mathtt{p}(c_0) \qquad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathtt{f}(c_1) \qquad \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset \qquad \mathrm{dom}(\rho_2) = \mathrm{dom}(\tau_2) \qquad \tau_3 = \tau_2 \oplus \tau_1}{[\![\mathtt{extend(p,\ f)}(c_0)]\!](\tau_3) = [\![\mathtt{f}(c_1)]\!](\tau_2)\ [\![\mathtt{p}(c_0)]\!](\tau_1)}$$

In this rule, we omit subscripts $[\![\cdot]\!]_\gamma$ and $[\![\cdot]\!]_p$, 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 $\mathtt{q} ::= \mathtt{p} \mid \mathtt{compose(q', q)} \mid \mathtt{resample(q)} \mid \mathtt{propose(p, q)}$. 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 tranformation target(q). This transformation replaces all sub-expressions of the form propose(p, q) with their targets p and all sub-expressions resample(q) with q. We define the transformation recursively

$$\frac{}{\mathtt{p} = \mathtt{target(p)}} \qquad \frac{}{\mathtt{p} = \mathtt{target(propose(p,q))}} \qquad \frac{\mathtt{q'} = \mathtt{target(q)}}{\mathtt{q'} = \mathtt{target(resample(q))}} \qquad \frac{\mathtt{q_1'} = \mathtt{target(q_1)} \qquad \mathtt{q_2'} = \mathtt{target(q_2)}}{\mathtt{compose(q_2', q_1')} = \mathtt{target(compose(q_2, q_1))}}$$

Since this transformation removes all instances of propose and resample forms, transformed programs $\mathtt{q'} = \mathtt{target(q)}$ define a simplified grammar $\mathtt{q'} ::= \mathtt{p} \mid \mathtt{compose(q_1',\ q_2')}$. We now define the denotational semantics for inference programs as

$$\frac{\mathtt{q'} = \mathtt{target(q)}}{[\![\mathtt{q}(c)]\!] = [\![\mathtt{q'}(c)]\!]}$$

The denotational semantics for a transformed program are trivially inherited from those for target programs when $\mathtt{q'} = \mathtt{p}$, whereas the denotational semantics for a composition compose(q2', q1') are analogous to those of an extension extend(p, f)

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \mathtt{q_1'}(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathtt{q_2'}(c_1) \quad \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset \quad \tau_3 = \tau_1 \oplus \tau_2}{[\![\mathtt{compose(q_2',\ q_1')}(c_0)]\!](\tau_3) = [\![\mathtt{q_2'}(c_1)]\!](\tau_2)\ [\![\mathtt{q_1'}(c_0)]\!](\tau_1)}$$

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

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

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

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

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

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

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

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

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 we define an evaluation in the context of a user-defined objective function $\ell : (\rho_q, \rho_p, w, v) \to \mathbb{R}$. For this purpose we introduce the notation

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

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

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

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 ctomputed is that of the propose combinator, for which we define the semantics

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

$$\frac{\langle \mathcal{L}_1, \ell, (c_1, \tau_1, \rho_1, w_1)\rangle \leftsquigarrow \langle \mathcal{L}_0, \ell, \mathsf{q_1(c_0)}\rangle \quad \langle \mathcal{L}_2, \ell, (c_2, \tau_2, \rho_2, w_2)\rangle \leftsquigarrow \langle \mathcal{L}_1, \ell, \mathsf{q_2(c_1)}\rangle \quad \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset}{\langle \mathcal{L}_2, \ell, (c_2, \tau_2 \oplus \tau_1, \rho_2 \oplus \rho_1,\ w_2 \cdot w_1)\rangle \leftsquigarrow \langle \mathcal{L}_0, \ell, \mathtt{compose(q_2,\ q_1)(c_0)}\rangle}$$

$$\frac{\langle \mathcal{L}_1, \ell, (c_1, \tau_1, \rho_1, w_1))\rangle \leftsquigarrow \langle \mathcal{L}_0, \ell, \mathsf{p(c_0)}\rangle \quad \langle \mathcal{L}_2, \ell, (c_2, \tau_2, \rho_2, w_2))\rangle \leftsquigarrow \langle \mathcal{L}_1, \ell, \mathsf{f(c_1)}\rangle \quad \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset \quad \mathrm{dom}(\rho_2) = \mathrm{dom}(\tau_2)}{\langle \mathcal{L}_2, \ell, (c_2, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2)\rangle \leftsquigarrow \langle \mathcal{L}_0, \ell, \mathtt{extend(p,\ f)(c_0)}\rangle}$$

$$\frac{\langle \mathcal{L}_1, \ell, (\vec{c}_1, \vec{\tau}_1, \vec{\rho}_1, \vec{w}_1)\rangle \leftsquigarrow \langle \mathcal{L}_0, \ell, \mathsf{q}(\vec{c}_0)\rangle \quad \vec{a}_1 \sim \textsc{resample}(\vec{w}_1) \quad \vec{c}_2, \vec{\tau}_2, \vec{\rho}_2 = \textsc{reindex}(\vec{a}_1, \vec{c}_1, \vec{\tau}_1, \vec{\rho}_1) \quad \vec{w}_2 = \textsc{mean}(\vec{w}_1)}{\langle \mathcal{L}_1, \ell, (\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2, \vec{w}_2)\rangle \leftsquigarrow \langle \mathcal{L}_0, \ell, \mathtt{resample(q)}(\vec{c}_0)\rangle}$$

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* $\mathsf{p}_2 = \mathtt{extend(p_1,\ f)}$ *is strictly properly weighted for its unnormalized density* $[\![\mathsf{p}_2]\!]_\gamma$ *when evaluation of* p_1 *is strictly properly weighted for* $[\![\mathsf{p}_1]\!]_\gamma$.

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

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \mathsf{p_1(c_0)} \quad c_f, \tau_f, \rho_f, w_f \leftsquigarrow \mathsf{f(c_1)} \quad \tau_2 = \tau_1 \oplus \tau_f \quad \rho_2 = \rho_1 \oplus \rho_f}{c_1, \tau_2, \rho_2, w_1 \leftsquigarrow \mathsf{p_2(c_0)} \quad [\![\mathsf{p}_2]\!]_\gamma(\tau_2; c_0) = [\![\mathsf{p}_1]\!]_\gamma(\tau_1; c_0) \cdot [\![\mathsf{f}]\!]_\gamma(\tau_f; c_1)} \tag{1}$$

Our induction hypothesis is that p_1 is strictly properly weighted for its density $[\![\mathsf{p}_1]\!]_\gamma := \gamma_1$. Since the primitive program `f` may only include unobserved variables, $w_f = 1$ and its evaluation is properly weighted relative to the prior density $[\![\mathsf{f}]\!]_\gamma = [\![\mathsf{f}]\!]_p := p_f$. Strict proper weighting with respect to $[\![\mathsf{p}_2]\!]_\gamma = \gamma_2$ follows directly from definitions

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

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* $[\![\mathsf{p}]\!]_\gamma$ *that it denotes.*

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

- *Base case*: $\mathsf{p} = \mathtt{f}$. This follows from Proposition 1.
- *Inductive case*: $\mathsf{p}_2 = \mathtt{extend(p_1,\ f)}$. This follows from Lemma 1.

□

Lemma 2 (Strict proper weighting of the resample combinator)**.** *Evaluation of a program* $\mathsf{q}_2 = \mathtt{resample(q_1)}$ *is strictly properly weighted for its unnormalized density* $[\![\mathsf{q}_2]\!]_\gamma$ *when evaluation of* q_1 *is strictly properly weighted for* $[\![\mathsf{q}_1]\!]_\gamma$.

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

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

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. $\vec{\tau} = [\tau^1, \ldots, \tau^L]$, which is also the dimension along which resampling is performed, but this is not a requirement in the underlying implementation.

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

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

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

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

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

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

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

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

Strict proper weighting now follows directly from definitions

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

□

Lemma 3 (Strict proper weighting of the compose combinator)**.** *Evaluation of the program* $\mathsf{q}_3 =$ `compose(`q_1`,` q_2`)` *is strictly properly weighted for its unnormalized density* $[\![$`compose(`q_1`,` q_2`)`$]\!]_\gamma$ *when evaluation of* q_1 *and* q_2 *is strictly properly weighted for the unnormalized densities* $[\![\mathsf{q}_1]\!]_\gamma$ *and* $[\![\mathsf{q}_2]\!]_\gamma$.

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

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \mathsf{q}_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathsf{q}_2(c_1) \quad \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset \quad \tau_3 = \tau_2 \oplus \tau_1 \quad \rho_3 = \rho_2 \oplus \rho_1 \quad w_3 = w_2 \cdot w_1}{c_2, \tau_3, \rho_3, w_3 \leftsquigarrow \texttt{compose}(\mathsf{q}_2, \mathsf{q}_1)(c_0)},$$

and its denotation as the product of conditional densities

$$\frac{c_1, \tau_1, \rho_1, w_1 \leftsquigarrow \mathsf{q}_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathsf{q}_2(c_1) \quad \mathrm{dom}(\rho_1) \cap \mathrm{dom}(\rho_2) = \emptyset}{[\![\texttt{compose}(\mathsf{q}_2, \mathsf{q}_1)(c_0)]\!](\tau_1 \oplus \tau_2) = [\![\mathsf{q}_2(c_1)]\!](\tau_2)\, [\![\mathsf{q}_1(c_0)]\!](\tau_1)}.$$

Let $[\![\mathsf{q}_3]\!]_\gamma = \gamma_3$ denote the unnormalized density of the composition. We will show that, for any measurable function $h(\tau_3)$,

$$\mathbb{E}_{\mathsf{q}_3(c_0)}\left[w_3\, h(\tau_3)\right] = Z_3(c_0) \int d\tau_3\, \gamma_3(\tau_3; c_0)\, h(\tau_3).$$

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

$$\mathbb{E}_{\mathsf{q}_3(c_0)}\left[w_3\, h(\tau_3)\right] = \mathbb{E}_{\mathsf{q}_1(c_0)}\left[w_1\, \mathbb{E}_{\mathsf{q}_2(c_1)}\left[w_2\, h(\tau_2 \oplus \tau_1)\right]\right],$$

Let $[\![\mathsf{q}_1]\!]_\gamma = \gamma_1$, $[\![\mathsf{q}_2]\!]_\gamma = \gamma_2$ 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

$$\begin{aligned}
\mathbb{E}_{\mathsf{q}_3(c_0)}\left[w_3\, h(\tau_3)\right] &= \int d\tau_1\, \gamma_1(\tau_1; c_0) \int d\tau_2\, \gamma_2(\tau_2; c_1) h(\tau_1 \oplus \tau_2; c_0) \\
&= \int d\tau_1 \int d\tau_2\, \gamma_1(\tau_1; c_0)\, \gamma_2(\tau_2; c_1)\, h(\tau_1 \oplus \tau_2; c_0) \\
&= \int d\tau_3\, \gamma_3(\tau_3; c_0)\, h(\tau_3; c_0).
\end{aligned}$$

□

Lemma 4 (Strict proper weighting of the propose combinator)**.** *Evaluation of a program* $\texttt{propose}(\mathsf{p}, \mathsf{q})$ *is strictly properly weighted for the unnormalized density* $[\![\texttt{propose}(\mathsf{p}, \mathsf{q})]\!]_\gamma$ *when evaluation of* q *is strictly properly weighted for the unnormalized density* $[\![\mathsf{q}]\!]_\gamma$.

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

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

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

$$[\![\texttt{propose}(\mathsf{p}, \mathsf{q})(c_0)]\!]_\gamma = [\![\texttt{marginal}(\mathsf{p})(c_0)]\!]_\gamma = \gamma_p(\cdot; c_0).$$

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

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

We start by expressing the expectation with respect to $\mathsf{q}_2(c_0)$ as an expectation with respect to $\mathsf{q}_1(c_0)$ and $\mathsf{p}(c_0)[\tau_1]$, and use the inductive hypothesis to express the first expectation as an integral with respect to $[\![\mathsf{q}(c_0)]\!]_\gamma = \gamma_q(\cdot; c_0)$,

$$\begin{aligned}
\mathbb{E}_{\mathsf{q}_2(c_0)}\left[\frac{w_2 w_1}{u_1}\, h(\tau_3)\right] &= \mathbb{E}_{\mathsf{q}_1(c_0)}\left[w_1\, \mathbb{E}_{\mathsf{p}(c_0)[\tau_1]}\left[\frac{w_2}{u_1}\, h(\tau_3)\right]\right] \\
&= \int d\tau_1\, \gamma_q(\tau_1; c_0)\, \mathbb{E}_{\mathsf{p}(c_0)[\tau_1]}\left[\frac{w_2}{u_1}\, h(\tau_3)\right].
\end{aligned}$$

We use $[\![\mathtt{p}(c_0)]\!] = \tilde{\gamma}_p(\cdot\,; c_0)$ to refer to the density that $\mathtt{p}$ 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

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

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 $\mathrm{dom}(\tau_2) \setminus \mathrm{dom}(\tau_3)$.

□

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

Proof. By induction on the grammar for $\mathtt{q}$.

- *Base case:* $\mathtt{q} = \mathtt{p}$. Theorem 1 above provides a proof.
- *Inductive case:* $\mathtt{q}_2 = \mathtt{resample}(\mathtt{q}_1)$. This follows from Lemma 2.
- *Inductive case:* $\mathtt{q}_3 = \mathtt{compose}(\mathtt{q}_1, \mathtt{q}_2)$. This follows from Lemma 3.
- *Inductive case:* $\mathtt{q}_2 = \mathtt{propose}(\mathtt{p}, \mathtt{q}_1)$. This follows from Lemma 4.

□

F GRADIENT COMPUTATIONS

F.1 STOCHASTIC VARIATIONAL INFERENCE (SVI).

Let $\mathtt{q}_2 = \mathtt{propose}(\mathtt{p}, \mathtt{q}_1)$ be a program in which the initial inference program $\mathtt{q}_1$, target program $\mathtt{p}$, and inference program $\mathtt{q}_2$ denote the densities

$$[\![\mathtt{q}_1(\mathtt{c}_0)]\!]_\gamma = \gamma_q(\cdot\,; c_0, \phi) \qquad [\![\mathtt{p}(\mathtt{c}_0)]\!]_\gamma = \tilde{\gamma}_p(\cdot\,; c_0, \theta), \qquad [\![\mathtt{q}_2(\mathtt{c}_0)]\!]_\gamma = \gamma_p(\cdot\,; c_0, \theta),,$$

with parameters θ and ϕ respectively. Notice that the target program $\mathtt{p}$ and the inference program $\mathtt{q}_2$ denote the same density $[\![\mathtt{p}(\mathtt{c}_0)]\!]_\gamma = [\![\mathtt{q}_2(\mathtt{c}_0)]\!]_\gamma$ and hence, as a result of Theorem 1, the evaluation of $\mathtt{q}_2$ is strictly properly weighted for γ_p. Applying definition 1 we can now write

$$Z_p(c_0; \theta)\mathbb{E}_{\pi_p(\cdot;\, c_0)}[h(\tau)] = \mathbb{E}_{\mathtt{q}_2(c_0)}[w_2 h(\tau_2)], \qquad c_2, \tau_2, \rho_2, w_2 \leftsquigarrow \mathtt{q}_2(\mathtt{c}_0),$$

for any measurable function h. Given evaluations $c_1, \tau_1, \rho_1, w_1 \sim \mathtt{q}_1(c_0)$ and $c_2', \tau_2', \rho_2', w_2' \sim \mathtt{p}[\tau_1](c_0)$ we can similarly compute a stochastic lower bound (Burda et al., 2016),

$$\mathcal{L} = \mathbb{E}_{\mathtt{q}_2(c_0)}\left[\log w_2\right] \leq \log\left(\mathbb{E}_{\mathtt{q}_2(c_0)}\left[w_2\right]\right) = \log\left(Z_p(c_0, \theta)\mathbb{E}_{\pi(\cdot;c_0)}\left[1\right]\right) = \log Z_p(c_0, \theta),$$

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

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

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. $\mathrm{dom}(\tilde{\tau}_2) \setminus \mathrm{dom}(\tau_1) = \emptyset$, the traces produced by the conditioned evaluation $\tilde{c}_2, \tilde{\tau}_2, \tilde{\rho}_2, \tilde{w}_2 \sim \mathsf{p}[\tau_1](c_0)$ 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,

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

If, additionally, all random variables in the inference program are reused in the target program, i.e. $\mathrm{dom}(\tau_1) \setminus \mathrm{dom}(\tilde{\tau}_2) = \emptyset$, 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. $\mathrm{dom}(\tilde{\tau}_2) = \mathrm{dom}(\tau_1)$, we recover the standard variational inference gradient w.r.t. the model parameters θ,

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

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

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

In the case where the set of random variables in the proposal and target program is the same, i.e. $\mathrm{dom}(\tilde{\tau}_2) = \mathrm{dom}(\tau_1)$ and $\mathsf{q}_1 = \mathsf{f}$ 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,

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

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

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

which can be approximated using reparameterized weights obtained by evaluating 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

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

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 $c_2^l, \tau_2^l, \rho_2^l, w_2^l \hookleftarrow \mathtt{q_2(c_0)}$ we can derive a costistent self-normalized estimator

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

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

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

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

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

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:

$$[\![\mathtt{q_0(c)}]\!] = \mathrm{Normal}(\mu = 0, \sigma = 5) \qquad [\![\mathtt{q}_k\mathtt{(c)}]\!]_\gamma = q_1(c)^{1-\beta_k}\gamma(c)_K^{\beta_k}, \qquad \beta_k = \frac{k-1}{K-1}, \qquad \text{for } k = 1 \ldots K$$

Implementations of these programs can be found in Figure 2. Additionally, each forward- and reverse- kernel program $\mathtt{q}_k\mathtt{(c)}$ 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(μ(k), σ(k))
    xs.append(s.sample(dist, str(k), count))
  return s, shuffle(cat(xs))

def q_0(s):
  c = s.sample(Normal(μ(k), σ(k)), "q_0")
  return s, c
```

```
η = ... # initialize neural network for kernel k
def q_k(s, c):
  c′ = s.sample(Normal(η_μ(k), η_σ(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) - q0.log_prob(x)))
  return x
```

Figure 2: models defined for Annealed Variational Inference

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

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(λ 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 kth object in the tth 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

$$z_d^{\text{what}} \sim \text{Normal}(0,\ I), \qquad z_{d,1}^{\text{where}} \sim \text{Normal}(0,\ I), \quad z_{d,t}^{\text{where}} \sim \text{Normal}(z_{d,t-1}^{\text{where}},\ \sigma_0^2 I),$$
$$x_t \sim \text{Bernoulli}\Big(\sigma\Big(\sum_d \text{ST}\big(g_\theta(z_d^{\text{what}}),\ z_{d,t}^{\text{where}}\big)\Big)\Big), \qquad d = 1, 2, ..., D, \qquad t = 1, 2, .., T.$$

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

$$\{z_{1:D}^{\text{what}}\}, \qquad \{z_{1:D,\,1}^{\text{where}}\}, \qquad \{z_{1:D,\,2}^{\text{where}}\}, \qquad \cdots, \qquad \{z_{1:D,\,T}^{\text{where}}\}$$

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 lr=$2e-4$.

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

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

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

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

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

$$q_\phi(z_{1:D,\,t}^{\text{where}} \mid x_t) = \prod_{d=1}^{D} \text{Normal}\Big(z_{d,t}^{\text{where}} \;\Big|\; \tilde{\mu}_{d,t}^{\text{where}}, \tilde{\sigma}_{d,t}^{\text{where}\,2} I\Big), \qquad \text{for } t = 1, 2, \ldots, T,$$

$$q_\phi(z_{1:D,\,t}^{\text{where}} \mid x_t, z_{1:D}^{\text{what}}) = \prod_{d=1}^{D} \text{Normal}\Big(z_{d,t}^{\text{where}} \;\Big|\; \tilde{\mu}_{d,t}^{\text{where}}, \tilde{\sigma}_{d,t}^{\text{where}\,2} I\Big), \qquad \text{for } t = 1, 2, \ldots, T,$$

$$q_\phi(z_{1:D}^{\text{what}} \mid x_{1:T}, z_{1:T}^{\text{where}}) = \prod_{d=1}^{D} \text{Normal}\Big(z_d^{\text{what}} \;\Big|\; \tilde{\mu}_d^{\text{what}}, \tilde{\sigma}_d^{\text{what}\,2} I\Big).$$

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 ˜ 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

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

For the position features, the proposal $q_\phi(z_{1:D,\,t}^{\text{where}} \mid x_t)$ and proposal $q_\phi(z_{1:D,\,t}^{\text{where}} \mid x_t, z_{1:D}^{\text{what}})$ share the same network, but contain different pre-steps where we compute the input of that network. The initial proposal $q_\phi(z_{1:D,\,t}^{\text{where}} \mid x_t)$ will convolve the frame x_t with the mean image of the object dataset; The conditional proposal $q_\phi(z_{1:D,\,t}^{\text{where}} \mid x_t, z_{1:D}^{\text{what}})$ 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 $d = 1, 2, ..., D$. Here is pseudocode of both pre-steps:

We employ a MLP encoder $f_\phi^{\text{L}}(\cdot)$ 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

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

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

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

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

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

Encoder $f^{\text{L}}_\phi(\cdot)$
Input $x^{\text{conv}}_{d,t} \in \mathbb{R}^{4761}$
FC 200. ReLU. FC 2×100. ReLU. FC 2×2.

The architecture of the MLP encoder $f^{\text{L}}_\phi(\cdot)$ 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 $z^{\text{where}}_{d,t}$ using the spatial transformer ST as

$$x^{\text{crop}}_{d,t} \leftarrow \text{ST}\big(x_t, z^{\text{where}}_{d,t}\big), \qquad\qquad d = 1, 2, \ldots, D, \quad t = 1, 2, \ldots, T.$$

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

Encoder $f^{\text{G}}_\phi(\cdot)$
Input $x^{\text{crop}}_{d,t} \in \mathbb{R}^{784}$
FC 400. ReLU. FC 200. ReLU. $\rightarrow$ Neural Sufficient Statistics $T^{\text{G}}_\phi(x^{\text{crop}}_{d,t}) \in \mathbb{R}^{200}$
Intermediate Input $\sum_{t=1}^{T} T^{\text{G}}_\phi(x^{\text{crop}}_{d,t}) \rightarrow$ 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 [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.

Burda, Yuri, Grosse, Roger, Salakhutdinov, Ruslan. "Importance Weighted Autoencoders." *International Conference on Representations*. 2016. arXiv: 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.

Caterini, Anthony L., Doucet, Arnaud, Sejdinovic, Dino. "Hamiltonian Variational Auto-Encoder" (May 2018). arXiv: 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.

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.

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. arXiv: 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.

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. arXiv: 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.

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.

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.

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.

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 [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.

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. Bejing, 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.

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.

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.

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. 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 [cs, stat].

Tucker, George, Lawson, Dieterich, Gu, Shixiang, Maddison, Chris J. "Doubly Reparameterized Gradient Estimators for Monte Carlo Objectives" (Oct. 2019). arXiv: 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 [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.

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.