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 $\hat{Z} = \frac{1}{L} \sum w^i$, 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 INFEERENCE 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. Functors provide a grammar and type system which take inspiration from modern autodifferentiation libraries. Expressions in functors denote discrete factors, Gaussian factors, point mass, delayed values, function application, substitution, marginalization, and plated products. Types encapsulate tensor dimensions, which permit functors 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 Šćibior 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. Šcibior et al. (2017); Šcibior 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 Šcibior 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

\[

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) \\
\text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \text{dom}(\rho_2) = \text{dom}(\tau_2) \quad \tau_3 = \tau_2 + \tau_1 \Rightarrow [\text{extend}(p, f)(c_0)](\tau_3) = [f(c_1)](\tau_2) [p(c_0)](\tau_1)
\]

In this rule, we omit subscripts \([\cdot]\), 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 \(\tau_3 = \tau_2 + \tau_1\) by combining disjoint traces \(\tau_1\) and \(\tau_2\) 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 tranformation \(\text{target}(q)\). 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

\[
\begin{align*}
p &= \text{target}(p) & q' &= \text{target}(q) & q'_1 &= \text{target}(q_1) & q'_2 &= \text{target}(q_2) \\
p &= \text{target}(\text{propose}(p, q)) & q' &= \text{target}(\text{resample}(q)) & \text{compose}(q'_2, q'_1) &= \text{target}(\text{compose}(q_2, q_1))
\end{align*}
\]

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

\[
[f(c_0)](\tau_3) = [f(c_1)](\tau_2) [p(c_0)](\tau_1)
\]

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 \(\text{extend}(p, f)\)

\[
\text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \quad \tau_1 = \tau_1 + \tau_2 \Rightarrow [\text{compose}(q'_2, q'_1)(c_0)](\tau_3) = [q'_2(c_1)](\tau_3) [q'_1(c_0)](\tau_1)
\]
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
\[
\begin{align*}
  &c_1, \tau_1, \rho_1, w_1 \leadsto p(c_0)[\tau_0] \\
  &c_2, \tau_2, \rho_2, w_2 \leadsto f(c_1)[\tau_0] \\
  &\text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \\
  &\text{dom}(\rho_2) = \text{dom}(\tau_2) \\
  &c_1, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2 \leadsto \text{extend}(p, f)(c_0)[\tau_0]
\end{align*}
\]

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
\[
\begin{align*}
  &c, \tau, \rho, w \leadsto \text{target}(q)(c_0)[\tau_0] \\
  &c, \tau, \rho, w \leadsto q(c_0)[\tau_0]
\end{align*}
\]

As in the previous section, the transformed programs define a grammar \( q' ::= p | \text{compose}(q'_1, q'_2) \). 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
\[
\begin{align*}
  &c_1, \tau_1, \rho_1, w_1 \leadsto q'_1(c_0)[\tau_0] \\
  &c_2, \tau_2, \rho_2, w_2 \leadsto q'_2(c_1)[\tau_0] \\
  &\text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \\
  &c_1, \tau_1 \oplus \tau_2, \rho_1 \oplus \rho_2, w_1 \cdot w_2 \leadsto \text{compose}(q'_2, q'_1)(c_0)[\tau_0]
\end{align*}
\]

Note that the operational semantics do not rely on evaluation under substitution of inference programs \( q \), since conditional evaluation is only ever 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
\[
\begin{align*}
  &c_1, \tau_1, \rho_1, w_1 \leadsto q(c_0)[\tau_0] \\
  &\llbracket p(c_0)[\tau_0] \rrbracket_p(\tau_1) = p_q[\tau_0](\tau_1; c_0) = \prod_{\alpha \in \text{dom}(\tau_1) \setminus \text{dom}(\tau_0)} \rho_1(\alpha)
\end{align*}
\]

D EVALUATION IN CONTEXT

To perform variational inference, we need to update a variational objective \( L \) each time we evaluate a propose combinator. For this purpose we define an evaluation in the context of a user-defined objective function \( \ell : (\rho_q, \rho_p, w, v) \to \mathbb{R} \). For this purpose we introduce the notation
\[
\langle L, \ell, (c, \tau, \rho, w) \rangle \leadsto \langle L', \ell, q(c') \rangle.
\]

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

The base case for primitive programs is trivial, we simply perform a traced evaluation and leave the loss \( L = L' \) invariant
\[
\langle L, \ell, (c, \tau, \rho, w) \rangle \leadsto \langle L, \ell, \ell(c') \rangle
\]

The semantics of evaluation in context for \( \text{compose}, \text{extend}, \text{and resample} \) are similarly trivial, in the sense that they only serve to thread their input \( L \) 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
\[
\begin{align*}
  &L_2 = \ell(p_1, p_2, w_1, w_2 / u_1) + L_1 \\
  &\langle L_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle \leadsto \langle L_0, \ell, q(c_0) \rangle \\
  &c_2, \tau_2, \rho_2, w_2 \leadsto p(c_0)[\tau_1] \\
  &c_3, \tau_3, \rho_3, w_3 \leadsto \text{marginal}(p)(c_0)[\tau_2] \\
  &u_1 = \prod_{\alpha \in \text{dom}(\tau_2) \setminus \text{dom}(\tau_1)} \rho_1(\alpha)
\end{align*}
\]

\[
\begin{align*}
  &\langle L_2, \ell, (c_3, \tau_3, \rho_3, w_2 / w_1 / u_1) \rangle \leadsto \langle L_0, \ell, \text{propose}(p, q)(c_0) \rangle
\end{align*}
\]
\begin{align*}
\langle L_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle & \sim \langle L_0, \ell, q_1(c_0) \rangle \quad \langle L_2, \ell, (c_2, \tau_2, \rho_2, w_2) \rangle \sim \langle L_1, \ell, q_2(c_1) \rangle \\
\text{dom}(\rho_1) \cap \text{dom}(\rho_2) & = \emptyset
\end{align*}

\begin{align*}
\langle L_2, \ell, (c_2, \tau_2 \oplus \tau_1, \rho_2 \oplus \rho_1, w_2 \cdot w_1) \rangle & \sim \langle L_0, \ell, \text{compose}(q_2, q_1)(c_0) \rangle \\
\langle L_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle & \sim \langle L_0, \ell, p(c_0) \rangle \quad \langle L_2, \ell, (c_2, \tau_2, \rho_2, w_2) \rangle \sim \langle L_1, \ell, f(c_1) \rangle
\end{align*}

\begin{align*}
\text{dom}(\rho_1) \cap \text{dom}(\rho_2) & = \emptyset \quad \text{dom}(\rho_2) = \text{dom}(\tau_2)
\end{align*}

\begin{align*}
\langle L_2, \ell, (c_2, \tau_2 \oplus \tau_1, \rho_2 \oplus \rho_1, w_2 \cdot w_1) \rangle & \sim \langle L_0, \ell, \text{extend}(p, f)(c_0) \rangle \\
\langle L_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle & \sim \langle L_0, \ell, q(c_0) \rangle \quad \langle L_2, \ell, (c_2, \tau_2, \rho_2, w_2) \rangle \sim \langle L_1, \ell, f(c_1) \rangle
\end{align*}

\begin{align*}
\langle L_1, \ell, (\vec{c}_2, \vec{\tau}_2, \vec{\rho}_2, w_2) \rangle & \sim \langle L_0, \ell, \text{resample}(q)(c_0) \rangle
\end{align*}

Figure 1: Operational semantics for evaluating \texttt{compose}, \texttt{extend}, and \texttt{resample} combinators in the context of a loss function \(\ell\) and accumulated loss \(L\).

\section{Proper Weighting of Programs}

\textbf{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 \([p_2]_\gamma\) when evaluation of \(p_1\) is strictly properly weighted for \([p_1]_\gamma\).

\textbf{Proof}. Recall from Section 3.4, that the program \(p_2\) denotes a composition

\begin{equation}
\begin{align*}
\text{dom}(\rho_2) & = \text{dom}(\tau_2)
\end{align*}
\end{equation}

\begin{align*}
\langle L_2, \ell, (c_2, \tau_2 \oplus \tau_1, \rho_2 \oplus \rho_1, w_2 \cdot w_1) \rangle & \sim \langle L_0, \ell, \text{extend}(p, f)(c_0) \rangle \\
\langle L_1, \ell, (c_1, \tau_1, \rho_1, w_1) \rangle & \sim \langle L_0, \ell, q(c_0) \rangle \quad \langle L_2, \ell, (c_2, \tau_2, \rho_2, w_2) \rangle \sim \langle L_1, \ell, f(c_1) \rangle
\end{align*}

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

\begin{align*}
\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 + \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 + \tau_f), \\
& = \int d\tau_2 \gamma_2(\tau_2; c_0) h(\tau_2).
\end{align*}

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

\qed

\textbf{Theorem 1} (Strict proper weighting of target programs). Evaluation of a target program \(p\) is (strictly) properly weighted for the unnormalized density \([p]_\gamma\), that it denotes.

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

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

\textbf{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 \([q_2]_\gamma\) when evaluation of \(q_1\) is strictly properly weighted for \([q_1]_\gamma\).
Proof. In Section 3.4, we defined the operational semantics for the resample combinator as

\[
\tilde{c}_1, \tilde{r}_1, \tilde{\rho}_1, \tilde{w}_1 \sim q_1(c_0) \quad \tilde{a}_1 \sim \text{RESAMPLE}(\tilde{w}_1) \quad \tilde{c}_2, \tilde{r}_2, \tilde{\rho}_2 = \text{REINDEX}(\tilde{a}_1, \tilde{c}_1, \tilde{r}_1, \tilde{\rho}_1) \quad \tilde{w}_2 = \text{MEAN}(\tilde{w}_1).
\]

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. \( \tilde{r} = [\tau_1, \ldots, \tau_L] \), which is also the dimension along which resampling is performed, but this is not a requirement in the underlying implementation.

Let \( q_2(c_0) \) denote the unnormalized density of the program. Our goal is to demonstrate that outgoing samples are individually strictly properly weighted for the unnormalized density \( \gamma(\cdot; c_0) \),

\[
E_{q_2(c_0)} [w^k_2 h(\tau^k_2)] = \int d\tau' \gamma(\tau'; c_0) h(\tau'),
\]

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

\[
E_{q_1(c_0)} [w^i_1 h(\tau^i_1)] = \int d\tau' \gamma(\tau'; c_0) h(\tau').
\]

The resample combinator randomly selects ancestor indices \( \tilde{a}_1 \sim \text{RESAMPLE}(\tilde{w}_1) \). Informally, this procedure selects \( a^i_1 = k \) with probability proportional to \( w^i_1 \). More formally, this procedure must satisfy

\[
E_{\text{RESAMPLE}(\tilde{w}_1)} [I[a^i_1 = k]] = \frac{w^i_1}{\sum_{i=1}^L w^i_1}.
\]

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

\[
c^i_2 = c^i_1, \quad \tau^i_2 = \tau^i_1, \quad \rho^i_2 = \rho^i_1, \quad w^i_2 = \frac{1}{L} \sum_{i'=1}^L w^i_1.
\]

Strict proper weighting now follows directly from definitions

\[
E_{q_2(c_0)} [w^k_2 h(\tau^k_2)] = E_{q_2(c_0)} \left[ \left( \frac{1}{L} \sum_{i'} w^i_{1'} \right) h(\tau^i_1) \right]
= E_{q_1(c_0)} \left[ \left( \frac{1}{L} \sum_{i'} w^i_{1'} \right) E_{\text{RESAMPLE}(\tilde{w}_1)} [h(\tau^i_1)] \right]
= E_{q_1(c_0)} \left[ \left( \frac{1}{L} \sum_{i'} w^i_{1'} \right) \sum_k E_{\text{RESAMPLE}(\tilde{w}_1)} [I[a^i_1 = k]] h(\tau^k_1) \right]
= E_{q_1(c_0)} \left[ \frac{1}{L} \sum_{i'} w^i_{1'} \sum_k \frac{w^k_1}{\sum_{i'=1}^L w^i_1} h(\tau^k_1) \right]
= E_{q_1(c_0)} \left[ \frac{1}{L} \sum_k w^k_1 h(\tau^k_1) \right] = \frac{1}{L} \sum_k E_{q_1(c_0)} [w^k_1 h(\tau^k_1)].
\]

Lemma 3 (Strict proper weighting of the compose combinator). Evaluation of the program \( q_3 = \text{compose}(q_1, q_2) \) is strictly properly weighted for its unnormalized density \( [\text{compose}(q_1, q_2)] \gamma \) when evaluation of \( q_1 \) and \( q_2 \) is strictly properly weighted for the unnormalized densities \( [q_1] \gamma \) and \( [q_2] \gamma \).
Proof. In Section 3.4, we defined the operational semantics for the compose combinator as
\[
\begin{align*}
&c_1, \tau_1, \rho_1, w_1 \sim q_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \sim q_2(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \\
&\tau_3 = \tau_2 \oplus \tau_1 \quad \rho_3 = \rho_2 \oplus \rho_1 \quad w_3 = w_2 \cdot w_1,
\end{align*}
\]
and its denotation as the product of conditional densities
\[
\begin{align*}
&c_1, \tau_1, \rho_1, w_1 \sim q_1(c_0) \quad c_2, \tau_2, \rho_2, w_2 \sim q_2(c_1) \quad \text{dom}(\rho_1) \cap \text{dom}(\rho_2) = \emptyset \\
&\left[\text{compose}(q_2, q_1)(c_0)\right](\tau_1 \oplus \tau_2) = \left[q_2(c_1)\right](\tau_2) \cdot \left[q_1(c_0)\right](\tau_1).
\end{align*}
\]
Let \([q_0]_\gamma = \gamma_3\) denote the unnormalized density of the composition. We will show that, for any measurable function \(h(\tau_3)\),
\[
E_{q_3(c_0)}[w_3 h(\tau_3)] = Z_3(c_0) \int d\tau_3 \gamma_3(\tau_3; c_0) h(\tau_3).
\]
We can express the expectation with respect to the program \(q_3\) as a nested expectation with respect to \(q_1\) and \(q_2\)
\[
E_{q_3(c_0)}[w_3 h(\tau_3)] = E_{q_1(c_0)}[w_1 E_{q_2(c_1)}[w_2 h(\tau_2 \oplus \tau_1)]].
\]
Let \([q_1]_\gamma = \gamma_1, [q_2]_\gamma = \gamma_2\) of the inputs and their composition. By the induction hypothesis, we can express rewrite both expectations as integrals with respect to \(\gamma_1\) and \(\gamma_2\). Strict proper weighting follows from definitions
\[
E_{q_3(c_0)}[w_3 h(\tau_3)] = \int d\tau_1 \gamma_1(\tau_1; c_0) \int d\tau_2 \gamma_2(\tau_2; c_1) h(\tau_1 \oplus \tau_2; c_0)
\]
\[
= \int d\tau_1 \int d\tau_2 \gamma_1(\tau_1; c_0) \gamma_2(\tau_2; c_1) h(\tau_1 \oplus \tau_2; c_0)
\]
\[
= \int d\tau_3 \gamma_3(\tau_3; c_0) h(\tau_3; c_0).
\]
\]
\]

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

Proof. Recall from Section 3.4 that that the operational semantics for propose are
\[
\begin{align*}
&c_1, \tau_1, \rho_1, w_1 \sim q(c_0) \quad c_2, \tau_2, \rho_2, w_2 \sim p(c_0)[\tau_1] \quad c_3, \tau_3, \rho_3, w_3 \sim \text{marginal}(p)(c_0)[\tau_2] \\
&u_1 = \prod_{\alpha \in \text{dom}(\rho_1) \setminus (\text{dom}(\tau_1) \setminus \text{dom}(\tau_2))} \rho_1(\alpha)
\end{align*}
\]
Our aim is to demonstrate that evaluation of \texttt{propose}(p, q) is strictly properly weighted for
\[
[q_{\text{propose}(p, q)}(c_0)]_\gamma = [\text{marginal}(p)(c_0)]_\gamma = \gamma_p(\cdot; c_0).
\]
This is to say that, for any measurable \(h(\tau_3)\)
\[
E_{q(c_0)} \left[\frac{w_2 w_1}{u_1} h(\tau_3)\right] = \int d\tau_3 \gamma_p(\cdot; c_0) h(\tau_3).
\]
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 \([q(c_0)]_\gamma = \gamma_q(\cdot; c_0),\)
\[
\begin{align*}
E_{q_2(c_0)} \left[\frac{w_2 w_1}{u_1} h(\tau_3)\right] &= E_{q_1(c_0)} \left[w_1 E_{p(c_0)[\tau_1]} \left[\frac{w_2}{u_1} h(\tau_3)\right]\right] \\
&= \int d\tau_1 \gamma_q(\tau_1; c_0) E_{p(c_0)[\tau_1]} \left[\frac{w_2}{u_1} h(\tau_3)\right].
\end{align*}
\]
We use $[p(c_0)] = \tilde{\gamma}_p(\cdot ; c_0)$ 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 $u_2/u_1$ with the relevant extended-space densities, and simplify the resulting expressions

$$\int d\tau_1 \gamma_1(\cdot; c_0) \mathbb{E}_p(c_0)[\gamma_1(\cdot; c_0)] = \int d\tau_1 \gamma_1(\tau_1; c_0) \mathbb{E}_p(c_0)[\tau_1]$$

By induction on the grammar for $q$.

**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.

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

**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

$$\gamma_q(\cdot; c_0, \phi), \quad \gamma_p(\cdot; c_0, \theta), \quad \gamma_{q_2}(\cdot; c_0),$$

with parameters $\theta$ and $\phi$ respectively. Notice that the target program $p$ and the inference program $q_2$ denote the same density $[p(c_0)] = [q_2(c_0)] = \gamma$, and hence, as a result of Theorem 1, the evaluation of $q_2$ is strictly properly weighted for $\gamma_p$.

Applying definition 1 we can now write

$$Z_{\rho_2} = \mathbb{E}_{\pi(p; c_0)}[h(\tau)] = \mathbb{E}_{q_2(c_0)}[w_2 h(\tau_2)],$$

for any measurable function $h$. Given evaluations $c_1, \tau_1, \rho_1, c_1', \tau_2, \rho_2, w_2 \sim q_1(c_0)$ 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),

$$\mathcal{L} = \mathbb{E}_{q_2(c_0)}[\log w_2] \leq \log (\mathbb{E}_{q_2(c_0)}[w_2]) = \log (Z_{\rho_2} + \mathbb{E}_{\pi(p; c_0)}[1]) = \log Z_{\rho_2},$$
using the constant function $h(\tau) = 1$. The gradient of this bound

$$\nabla_\theta \mathbb{E}_{q_2}(c_0) [\log w_2] = \mathbb{E}_{q_1}(c_0) \left[ \nabla_\theta \log w_1 + \nabla_\theta \mathbb{E}_p(c_0)_{\tau_1} \left[ \log \frac{\tilde{\gamma}_p(\tau_2; c_0, \theta) p_q(\tau_2; \tau_1; c_0, \phi)}{\gamma_0(\tau_1; c_0, \phi) p_p(\tau_1; (\tau_2; c_0, \theta))} \right] \right]$$

$$= \mathbb{E}_{q_1}(c_0) \left[ \nabla_\theta \mathbb{E}_p(c_0)_{\tau_1} \left[ \log \frac{\tilde{\gamma}_p(\tau_2; c_0, \theta) p_q(\tau_2; \tau_1; c_0, \phi)}{p_p(\tau_1; (\tau_2; c_0, \theta))} \right] \right]$$

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 $\tilde{\tau}_1, \tilde{\tau}_2, \tilde{w}_2 \sim p[\tau_1](c_0)$ do not depend on $\theta$, as all variables are generated from the inference program, and the prior term $p_{\tau_1}(\tilde{\tau}_2; c_0) = 1$. As a result we can move the gradient operator inside the inner expectation,

$$\mathbb{E}_{q_1}(c_0) \left[ \nabla_\theta \mathbb{E}_p(c_0)_{\tau_1} \left[ \log \frac{\tilde{\gamma}_p(\tau_2; c_0, \theta) p_q(\tau_2; \tau_1; c_0)}{\gamma_0(\tau_1; c_0, \phi) p_p(\tau_1; (\tau_2; c_0, \theta))} \right] \right] = \mathbb{E}_{q_1}(c_0) \left[ \mathbb{E}_p(c_0)_{\tau_1} \left[ \nabla_\theta \log \tilde{\gamma}_p(\tau_2; c_0, \theta) p_q(\tau_2; \tau_1; c_0) \right] \right]$$

$$= \mathbb{E}_{q_2}(c_0) \left[ \nabla_\theta \log \tilde{\gamma}_p(\tau_2; c_0, \theta) \right].$$

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 $p_{\tau_1}(\tau_1; c_0) = 1$. 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 $\theta$,

$$\mathbb{E}_{q_2}(c_0) \left[ \nabla_\theta \log \gamma_f(\tau_2; c_0, \theta) \right].$$

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

$$\nabla_\phi \mathbb{E}_{q_2}(c_0) [\log w_2] = \mathbb{E}_{q_2}(c_0) \left[ \nabla_\phi \log w_2 \right]$$

$$= \mathbb{E}_{q_2}(c_0) \left[ \frac{\partial \log w_2}{\partial \tilde{\tau}_2} \frac{\partial \tilde{\tau}_2}{\partial \phi} + \frac{\partial \log w_2}{\partial \phi} \right]$$

$$= \mathbb{E}_{q_2}(c_0) \left[ \frac{\partial \log w_2}{\partial \tilde{\tau}_2} \frac{\partial \tilde{\tau}_2}{\partial \phi} + \frac{\partial \log p_q(\tau_2; \tau_1; c_0, \phi)}{\partial \phi} \right]$$

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 $q_1 = f$ 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 $\phi$,

$$\mathbb{E}_{q_2}(c_0) \left[ \frac{\partial}{\partial \tilde{\tau}_2} \log \left( \frac{\tilde{\gamma}_p(\tilde{\tau}_2; c_0, \theta)}{p_f(\tilde{\tau}_1; c_0, \phi)} \right) \frac{\partial \tilde{\tau}_2}{\partial \phi} - \frac{\partial}{\partial \phi} \log p_f(\tau_1; c_0, \phi) \right]$$

$$= \mathbb{E}_{q_2}(c_0) \left[ \frac{\partial}{\partial \tilde{\tau}_2} \log \left( \frac{\tilde{\gamma}_0(\tilde{\tau}_2; c_0, \theta)}{p_f(\tilde{\tau}_1; c_0, \phi)} \right) \frac{\partial \tilde{\tau}_2}{\partial \phi} - \frac{\partial}{\partial \phi} \log p_f(\tau_1; c_0, \phi) \right]$$

$$= -\nabla_\phi \text{KL}(p_f||\pi_p),$$

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

$$\mathbb{E}_{q_2}(c_0) \left[ \frac{\partial}{\partial \tilde{\tau}_2} \log \left( \frac{\gamma_p(\tilde{\tau}_2; c_0, \theta)}{p_f(\tilde{\tau}_1; c_0, \phi)} \right) \frac{\partial \tilde{\tau}_2}{\partial \phi} - \frac{\partial}{\partial \phi} \log p_f(\tau_1; c_0, \phi) \right] = \mathbb{E}_{q_2}(c_0) \left[ \frac{\partial}{\partial \tilde{\tau}_2} \log \left( \frac{\gamma_p(\tilde{\tau}_2; c_0, \theta)}{p_f(\tilde{\tau}_1; c_0, \phi)} \right) \frac{\partial \tilde{\tau}_2}{\partial \phi} \right],$$

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

\[
\nabla_\theta \log Z_p(c_0, \theta) = \frac{1}{Z_p(c_0, \theta)} \nabla_\theta \int d\tau \gamma_p(\tau; c_0, \theta)
\]

\[
= \frac{1}{Z_p(c_0, \theta)} \int d\tau \gamma_p(\tau; c_0, \theta) \nabla_\theta \log \gamma_p(\tau; c_0, \theta)
\]

\[
= \int d\tau \pi_p(\tau; c_0, \theta) \nabla_\theta \log \gamma_p(\tau; c_0, \theta)
\]

\[
= E_{\pi_p(\cdot; c_0, \theta)}[\nabla_\theta \log \gamma_p(\tau; c_0, \theta)].
\]

Notice that here we compute the gradient w.r.t. the non-extended density \(\gamma_p\), 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 \(c_2, \tau_2, \rho, w_1, w_2 \sim q_2(c_0)\) we can derive a costistent self-normalized estimator

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

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

\[
-\nabla_\phi KL(\pi_p || \pi_q) = E_{\pi_p(\cdot; c_0, \theta)}[\nabla_\phi \log \pi_q(\tau; c_0, \phi)]
\]

\[
= E_{\pi_p(\cdot; c_0, \theta)}[\nabla_\phi \log \pi_q(\tau; c_0, \phi)] - \nabla_\phi Z_q(c_0, \phi)
\]

\[
= E_{\pi_p(\cdot; c_0, \theta)}[\nabla_\phi \log \gamma_q(\tau; c_0, \phi)] - E_{\pi_q(\cdot; c_0, \phi)}[\nabla_\phi \log \gamma_q(\tau; c_0, \phi)]
\]

\[
= Z_p^{-1}(c_0, \theta)E_{q_2}[w_2 \nabla_\phi \log \gamma_q(\tau_2; c_0, \phi)] - Z_q^{-1}(c_0, \phi)E_{q_1}[w_1 \nabla_\phi \log \gamma_q(\tau_1; c_0, \phi)]
\]

\[
\approx \sum_l \left( \frac{w_2^l}{\sum_{l'} w_2^{l'}} - \frac{w_1^l}{\sum_{l'} w_1^{l'}} \right) \nabla_\phi \log \gamma_q(\tau_1^l; c_0, \phi).
\]

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

\[
E_{\pi_q(\cdot; c_0, \phi)}[\nabla_\phi \log \gamma_q(\tau; c_0, \phi)] = E_{\pi_q(\cdot; c_0, \phi)}[\nabla_\phi \log \pi_q(\tau; c_0, \phi)] = 0,
\]

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 \(\gamma_K\), 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:

\[
[q_0(c) = \text{Normal}(\mu = 0, \sigma = 5) \quad q_k(c) = q_1(c)^{-\beta_k} \gamma(c)^{\beta_k}, \quad \beta_k = \frac{k-1}{K-1}, \quad \text{for } k = 1 \ldots K
\]

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(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(µ(1), σ(1)), "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")
    return s.factor(beta * (log_gamma(x) - q0.log_prob(x)))

Figure 2: models defined for Annealed Variational Inference

x = Linear 50. ReLU(c)  µ_k = Linear 2(x) + c  cov_k = DiagEmbed . 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 ˆ 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. Our 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 x1: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^what_d and a time-dependent position variable z^where_d,t. The deep generative model takes the form

\[
\begin{align*}
    z^\text{what}_d &\sim \text{Normal}(0, I), \\
    x_t &\sim \text{Bernoulli}(\sigma\left(\sum_d \text{ST}(g(\bar{z}^\text{what}_d), \bar{z}^\text{where}_d)\right)), \\
    z^\text{where}_{d,1} &\sim \text{Normal}(0, I), \\
    z^\text{where}_{d,t} &\sim \text{Normal}(\bar{z}^\text{where}_{d,t-1}, \sigma_0^2 I), \\
    z^\text{where}_{d,t} &\sim \text{Normal}(z^\text{where}_{d,t-1}, \sigma_0^2 I), \\
    d &= 1, 2, ..., D, \\
    t &= 1, 2, ..., T.
\end{align*}
\]
where $\sigma_0 = 0.1$ and $z^{\text{what}}_d \in \mathbb{R}^{10}$, $z^{\text{where}}_d \in \mathbb{R}^2$. 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

$$\{z^{\text{what}}_{1:D}\}, \{z^{\text{where}}_{1:D,1}\}, \{z^{\text{where}}_{1:D,2}\}, \ldots, \{z^{\text{where}}_{1:D,T}\}$$

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

$$p_0(x_{1:T} \mid z^{\text{what}}_1, z^{\text{where}}_1) = \prod_{t=1}^{T} \text{Bernoulli}(x_t \mid \sigma\left(\sum_d \text{ST}(g_\theta(z^{\text{what}}_d), z^{\text{where}}_{d,t})\right))$$

Given each object feature $z^{\text{what}}_d$, the APG sampler reconstruct a $28 \times 28$ object image using a MLP decoder, the architecture of which is

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

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

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

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

$$q_\phi(z^{\text{where}}_{1:D,t} \mid x_t) = \prod_{d=1}^{D} \text{Normal}(z^{\text{where}}_{d,t} \mid \tilde{\mu}^{\text{where}}_{d,t}, \tilde{\sigma}^{\text{where}}_{d,t} 2I), \quad \text{for } t = 1, 2, \ldots, T,$$
$$q_\phi(z^{\text{where}}_{1:D,t} \mid x_t, z^{\text{what}}_1) = \prod_{d=1}^{D} \text{Normal}(z^{\text{where}}_{d,t} \mid \tilde{\mu}^{\text{where}}_{d,t}, \tilde{\sigma}^{\text{where}}_{d,t} 2I), \quad \text{for } t = 1, 2, \ldots, T,$$
$$q_\phi(z^{\text{what}}_{1:D} \mid x_{1:T}, z^{\text{where}}_1) = \prod_{d=1}^{D} \text{Normal}(z^{\text{what}}_d \mid \tilde{\mu}^{\text{what}}_d, \tilde{\sigma}^{\text{what}}_d 2I).$$

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

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

For the position features, the proposal $q_\phi(z^{\text{where}}_{1:D,t} \mid x_t)$ and proposal $q_\phi(z^{\text{where}}_{1:D,t} \mid x_t, z^{\text{what}}_1)$ share the same network, but contain different pre-steps where we compute the input of that network. The initial proposal $q_\phi(z^{\text{where}}_{1:D,t} \mid x_t)$ will convolve the frame $x_t$ with the mean image of the object dataset; The conditional proposal $q_\phi(z^{\text{where}}_{1:D,t} \mid x_t, z^{\text{what}}_1)$ will convolve the frame $x_t$ with each reconstructed object image $g_\theta(z^{\text{what}}_d)$. We perform convolution sequentially by looping over all objects $d = 1, 2, \ldots, D$. Here is pseudocode of both pre-steps:

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

$$\tilde{\mu}^{\text{where}}_{d,t}, \log \tilde{\sigma}^{\text{where}}_{d,t} 2 \leftarrow f_\phi(x_{d,t}^{\text{conv}}), \quad d = 1, 2, \ldots, D.$$
Algorithm 1 Convolution Processing for $g_\phi(x_{\text{where}}^{1:D,t} | x_t)$

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: \hspace{1em} $x_{\text{conv}}^{d,t} \leftarrow \text{Conv2d}(x_t)$ with kernel $mm$, stride = 1, no padding.
4: end for
5: **Output** Convolved features $\{x_{\text{conv}}^{d,t} \in \mathbb{R}^{4761}\}_{d=1}^D$

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

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

<table>
<thead>
<tr>
<th>Encoder $f_\phi^L(\cdot)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> $x_{\text{conv}}^{d,t} \in \mathbb{R}^{4761}$</td>
</tr>
<tr>
<td>FC 200. ReLU. FC $2 \times 100$. ReLU. FC $2 \times 2$.</td>
</tr>
</tbody>
</table>

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 $28 \times 28$ subframe according to $z_{\text{where}}^{d,t}$ using the spatial transformer ST as

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

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 $\{z_{\text{what}}^{d}\}_{d=1}^D$, i.e. the vector-valued means $\{\tilde{\mu}_{\text{what}}^{d}\}_{d=1}^D$ and the logarithms of the diagonal covariances $\{\log \tilde{\sigma}_{\text{what}}^{d}\}_{d=1}^D$. The architecture of this network is

<table>
<thead>
<tr>
<th>Encoder $f_\phi^G(\cdot)$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Input</strong> $x_{\text{crop}}^{d,t} \in \mathbb{R}^{784}$</td>
</tr>
<tr>
<td>FC 400. ReLU. FC 200. ReLU. \rightarrow Neural Sufficient Statistics $T_\phi^G(x_{\text{crop}}^{d,t}) \in \mathbb{R}^{200}$</td>
</tr>
<tr>
<td>Intermediate Input $\sum_{t=1}^T T_\phi^G(x_{\text{crop}}^{d,t}) \rightarrow FC 2 \times 10$.</td>
</tr>
</tbody>
</table>

**REFERENCES**


