Deep Amortized Inference for Probabilistic Programs

Daniel Ritchie
Stanford University

Paul Horsfall
Stanford University

Noah D. Goodman
Stanford University

Abstract

Probabilistic programming languages (PPLs) are a powerful modeling tool, able to represent any computable probability distribution. Unfortunately, probabilistic program inference is often intractable, and existing PPLs mostly rely on expensive, approximate sampling-based methods. To alleviate this problem, one could try to learn from past inferences, so that future inferences run faster. This strategy is known as *amortized inference*; it has recently been applied to Bayesian networks [28, 22] and deep generative models [20, 15, 24]. This paper proposes a system for amortized inference in PPLs. In our system, amortization comes in the form of a parameterized *guide program*. Guide programs have similar structure to the original program, but can have richer data flow, including neural network components. These networks can be optimized so that the guide approximately samples from the posterior distribution defined by the original program. We present a flexible interface for defining guide programs and a stochastic gradient-based scheme for optimizing guide parameters, as well as some preliminary results on automatically deriving guide programs. We explore in detail the common machine learning pattern in which a 'local' model is specified by 'global' random values and used to generate independent observed data points; this gives rise to amortized local inference supporting global model learning.

1 Introduction

Probabilistic models provide a framework for describing abstract prior knowledge and using it to reason under uncertainty. Probabilistic programs are a powerful tool for probabilistic modeling. A probabilistic programming language (PPL) is a deterministic programming language augmented with random sampling and Bayesian conditioning operators. Performing inference on these programs then involves reasoning about the space of executions which satisfy some constraints, such as observed values. A universal PPL, one built on a Turing-complete language, can represent any computable probability distribution, including open-world models, Bayesian non-parameterics, and stochastic recursion [6, 19, 33].

If we consider a probabilistic program to define a distribution $p(\mathbf{x}, \mathbf{y})$, where $\mathbf{x}$ are (latent) intermediate variable and $\mathbf{y}$ are (observed) output data, then sampling from this distribution is easy: just run the program forward. However, computing the posterior distribution $p(\mathbf{x}|\mathbf{y})$ is hard, involving an intractable integral. Typically, PPLs provide means to approximate the posterior using Monte Carlo methods (e.g. MCMC, SMC), dynamic programming, or analytic computation.

These inference methods are expensive because they (approximately) solve an intractable integral from scratch on every separate invocation. But many inference problems have shared structure: it is reasonable to expect that computing $p(\mathbf{x}|\mathbf{y}_1)$ should give us some information about how to compute $p(\mathbf{x}|\mathbf{y}_2)$. In fact, there is reason to believe that this is how people are able to perform certain inferences, such as visual perception, so quickly—we have perceived the world many times before, and can leverage that accumulated knowledge when presented with a new perception task [3]. This idea of using the results of previous inferences, or precomputation in general, to make later inferences more efficient is called *amortized inference* [3, 28].

Learning a generative model from many data points is a particularly important task that leads to many related inferences. One wishes to update global beliefs about the true generative model from individual data points (or batches of data points). While many algorithms are possible for this task, they all require some form of 'parsing' for each data point: doing posterior inference in the current generative model to guess values of local latent variable given each observation. Because this local parsing inference is needed many many times, it is a good candidate for amortization. It is plausible that learning to do local inference via amortization would support faster and better global learning, which gives more useful local inferences, leading to a virtuous cycle.

This paper proposes a system for amortized inference in PPLs, and applies it to model learning. Instead of computing $p(\mathbf{x}|\mathbf{y})$ from scratch for each $\mathbf{y}$, our system instead constructs a program $q(\mathbf{x}|\mathbf{y})$ which takes $\mathbf{y}$ as input and, when run forward, produces samples distributed approximately according to the true posterior $p(\mathbf{x}|\mathbf{y})$. We call q a *guide program*, following terminology introduced in previous work [11]. The system can spend time up-front constructing a good approximation q so that at inference time, sampling from q is both fast and accurate.

There is a huge space of possible programs q one might consider for this task. Rather than posing the search for q as a general program induction problem (as was done in previous work [11]), we restrict q to have the same control flow as the original program p, but a different data flow. That is, q samples the same random choices as p and in the same order, but the data flowing into those choices comes from a different computation. In our system, we represent this computation using neural networks. This design choice reduces the search for q to the much simpler continuous problem of optimizing the weights for these networks, which can be done using stochastic gradient descent.

Our system's interface for specifying guide programs is flexible enough to subsume several popular recent approaches to variational inference, including those that perform both inference and model learning. To facilitate this common pattern we introduce the `mapData` construct which represents the boundary between global "model" variables and variables local to the data points. Our system leverages the independence between data points implied by `mapData` to enable mini-batches of data and variance reduction of gradient estimates. We evaluate our proof-of-concept system on a variety of Bayesian networks, topic models, and deep generative models.

Our system has been implemented as an extension to the WebPPL probabilistic programming language [7]. Its source code can be found in the WebPPL repository, with additional helper code at `https://github.com/probmods/webppl-daipp`.

2 Background

2.1 Probabilistic Programming Basics

For our purposes, a probabilistic program defines a generative model $p(\mathbf{x}, \mathbf{y})$ of latent variables $\mathbf{x}$ and data $\mathbf{y}$. The model factors as:

$$p(\mathbf{x}, \mathbf{y}) = p(\mathbf{y}|\mathbf{x}) \prod_i p(x_i|\mathbf{x}_{<i}) \tag{1}$$

The prior probability distribution $p(\mathbf{x})$ decomposes as a product of conditionals $p_i(x_i|\mathbf{x}_{<i})$, one for each random choice x_i in the program. The use of $\mathbf{x}_{<i}$ indicates that a random choice can potentially depend on any or all previous choices made by the program. $p(\mathbf{y}|\mathbf{x})$ is the likelihood of the data and need not be a proper probability distribution (i.e. unnormalized factors are acceptable). Note that $\mathbf{x}$ can vary in length across executions: a probabilistic program can sample a variable number of random variables.

Our system is implemented in the probabilistic programming language WebPPL, which we use for examples throughout this paper [7]. WebPPL is a PPL embedded in Javascript; that is, it adds sampling, conditioning, and inference operators to a purely-functional subset of Javascript. The following example program illustrates its basic features:

```
var model = function() {
   var x = sample(Bernoulli({p: 0.75}));
   var mu = x ? 2 : 0;
   observe(Gaussian({mu: mu, sigma: 1}), 0.5);
   return x;
};
```

```
7
8  Infer({method: 'MCMC'}, model);
```

This program uses MCMC to compute an approximate posterior distribution over the return value of the function `model`. `model` is a simple generative model with one latent Bernoulli variable (`x`) and one observed Gaussian variable, which in this example is observed to have the value `0.5`. The mean of the observed Gaussian variable (`mu`) is dependent on the value of `x`. Since `model` returns `x`, the result of this program is the posterior marginal distribution over the variable `x`. In the rest of this paper, we will build on this language, adding guide programs, amortized inference, and model-learning constructs.

2.2 Inference as Optimization: Variational Inference

Instead of approximating the posterior $p(\mathbf{x}|\mathbf{y})$ with a collection of samples, one could instead try to approximate it via a parameterized distribution $q_{\mathbf{y}}(\mathbf{x};\phi)$ which is itself easy to sample from. This is the premise behind variational inference [13]. The goal is to find parameters ϕ such that $q_{\mathbf{y}}(\mathbf{x};\phi)$ is as close as possible to $p(\mathbf{x}|\mathbf{y})$, where closeness is typically measured via KL-divergence.

To use variational inference, one must first choose a parameterized family of distributions q; one common choice is the *mean-field family*:

$$q^{\mathbf{MF}}_{\mathbf{y}}(\mathbf{x};\phi) = \prod_i q(x_i;\phi_i)$$

This is a fully-factored distribution: it approximates the true posterior as an independent product of parameterized marginals, one for each random variable. Several existing general-purpose variational inference systems use this scheme [32, 18]. This is easy to work with, but it does not capture any of the dependencies between variables that may occur in the true posterior. This limitation is often acceptable because $q_{\mathbf{y}}$ is defined relative to a particular observation set $\mathbf{y}$, and thus the parameters are re-optimized for each new $\mathbf{y}$. Thus, while this scheme provides an alternative to Monte Carlo methods (e.g. MCMC) that can be faster and more reliable, it still solves each inference problem from scratch.

2.3 Amortized (Variational) Inference

As mentioned in Section 1, *amortized inference* refers to the use of previous inference solutions (or other pre-computation) to solve subsequent inference problems faster. There exists experimental evidence that people leverage experience from prior inference tasks when asked to solve related ones [3]. This idea has inspired research into developing amortized inference systems for Bayesian networks [28, 22]. These systems model $p(\mathbf{x}|\mathbf{y})$ by inverting the network topology and attempting to learn the local conditional distributions of this inverted graphical model.

Amortized inference can also be achieved through variational inference. Instead of defining a parametric family $q_{\mathbf{y}}(\mathbf{x};\phi)$ which is specific to a given $\mathbf{y}$, we can instead define a general family $q(\mathbf{x}|\mathbf{y};\phi)$ which is conditional on $\mathbf{y}$; that is, it takes $\mathbf{y}$ as input. In this setting, the mean field family no longer applies, as the factors of q must now be functions of $\mathbf{y}$. However, we can extend the mean field family to handle input data by using neural networks (or other 'side computations'):

$$q(\mathbf{x}|\mathbf{y};\phi) = \prod_i q(x_i;\mathrm{NN}_i(\mathbf{y};\phi))$$

Here, the parameters of each local conditional in the guide q are computed via a neural network function NN_i of $\mathbf{y}$. This variational family supports amortized inference: one can invest time up-front optimizing the neural network weights such that $q(\mathbf{x}|\mathbf{y};\phi)$ is close to $p(\mathbf{x}|\mathbf{y})$. When given a never-before-seen $\mathbf{y}$, the guide program forwards $\mathbf{y}$ through the trained networks for fast inference. Several recent approaches to 'neural variational inference' use some instantiation of this design pattern [20, 24, 15].

Such neural guide families are easy to express in our extensions of WebPPL. Our system also allows generalizations of this pattern, such as providing neural nets with previously-made random choices as additional input:

$$q(\mathbf{x}|\mathbf{y};\phi) = \prod_i q(x_i;\mathrm{NN}_i(\mathbf{y},\mathbf{x}_{<i};\phi))$$

Here, $\mathbf{x}_{<i}$ are the random choices made before choice i is sampled. Such guide families have the potential to capture posterior dependencies between latent variables

2.4 Variational Model Learning

The amortized variational inference framework presented above can also be used to learn the parameters of the generative model p. If p is also parameterized, i.e. $p(\mathbf{x}|\mathbf{y};\theta)$, then its parameters θ can be optimized along with the parameters ϕ of the guide program [20, 24, 15].

Our system supports learning generative model parameters in addition to guide parameters. In the PPL setting it is natural to think of this as a particular model pattern in which there are global parameters or random choices that affect a local 'observation model', which in turn is assumed to generate each data point independently; we call this the mapData model pattern. We will show below how it is easy to use this pattern to do (regularized) maximum-likelihood learning, variational Bayes, or even different methods within the same model.

3 Specifying Guide Programs

In this section, we describe language extensions to WebPPL that allow for the specification of guide programs. We focus for now on manually-specified guide programs. In Section 7, we build on this interface to automatically derive guide programs.

3.1 Sampling with Guide Distributions

Previously, we mentioned that our system restricts guide programs q to have the same control flow as the original program p, meaning that the guide program samples the same variables in the same order. Our implementation enforces this restriction by defining the guide program inline with the regular program. At each `sample` statement, in addition to the distribution that the program p samples from, one can also specify what distribution the guide program q should sample from. For example, using the simple program from Section 2.1:

```
var model = function() {
   var x = sample(Bernoulli({p: 0.75}), {
      guide: Bernoulli({p: 0.475})
   });
   var mu = x ? 2 : 0;
   observe(Gaussian({mu: mu, sigma: 1}), 0.5);
   return x;
};
```

In this example, the guide samples from a Bernoulli with a different success probability `p`. This particular value happpens to give the true posterior for this program, since this toy problem is simple enough to solve in closed form. We note that the guide distribution need not be of the same family as the prior distribution; we will see later how this property can be useful.

3.2 Declaring Optimizable Parameters

In real problems, we will not know the optimal guides *a prior* and will instead want to learn guides by specifying guide distributions with tunable parameters:

```
var x = sample(Gaussian({mu: 0, sigma: 1}), {
   guide: Gaussian({mu: paramScalar('guideMu'), sigma: softplus(paramScalar('guideSigma'))})
});
```

Here, `paramScalar(name)` declares an optimizable, real-valued parameter named `name`; there are analogous functions `paramVector`, `paramMatrix`, and `paramTensor` for declaring vector, matrix, and tensor-valued parameters. Since the standard deviation `sigma` of the Gaussian guide distribution must be positive, we use the `softplus`[1] function to map the unbounded value returned by `paramScalar` to $\mathbb{R}^+$; our system includes similar transforms for parameters with other domains (e.g. `sigmoid` for parameters defined over the interval $[0, 1]$). Parameters must be named so they can be disambiguated by the optimization engine.

Using variational parameters directly as the guide distribution parameters (as done above) results in a mean field approximation for the variable `x`, as mentioned in Section 2.2. We can also compute the guide parameters via a neural network:

[1] $\text{softplus}(x) = \log(\exp(x) + 1)$

```
// Observed value
var y = 0.5;

// Neural net setup
var nIn = 1;
var nHidden = 3;
var nOut = 2;

var model = function() {
   // Neural net params
   var W1 = paramMatrix(nHidden, nIn, 'W1');
   var b1 = paramVector(nHidden, 'b1');
   var W2 = paramMatrix(nOut, nHidden, 'W2');
   var b3 = paramVector(nOut, 'b2');

   // Use neural net to compute guide params
   var nnInput = Vector([y]);
   var nnOutput = linear(sigmoid(linear(nnInput, W1, b1)), W2, b2);

   var x = sample(Gaussian({mu: 0, sigma: 1}), {
      guide: Gaussian({mu: T.get(nnOutput, 0), sigma: softplus(T.get(nnOutput, 1))})
   });
   observe(Gaussian({mu: x, sigma: 0.5}), y);
   return x;
};
```

Explicitly declaring parameters for and defining the structure of large neural networks can become verbose, so we can instead use the adnn[2] neural net library to include neural nets in our programs:

```
// Observed value
var y = 0.5;

// Neural net setup
var guideNet = nn.mlp(1, [
   {nOut: 3, activation: nn.sigmoid},
   {nOut: 2}
], 'guideNet');

var model = function() {
   // Use neural net to compute guide params
   var nnInput = Vector([y]);
   var nnOutput = nnEval(guideNet, nnInput);

   var x = sample(Gaussian({mu: 0, sigma: 1}), {
      guide: Gaussian({mu: T.get(nnOutput, 0), sigma: softplus(T.get(nnOutput, 1))})
   });
   observe(Gaussian({mu: x, sigma: 0.5}), y);
   return x;
};
```

In this case, the `nn.mlp` constructor has created a `guideNet` object with its own parameters; these parameters are registered with the optimization engine when `nnEval` is called.

3.3 Iterating over Observed Data

The previous examples have thus far conditioned on a single observation. But real models condition on multiple observations. Our system expresses this pattern with the `mapData` function:

```
var obs = loadData('data.json');   // List of observations
var guideNet = nn.mlp(1, [
   {nOut: 3, activation: nn.sigmoid},
   {nOut: 2}
], 'guideNet');
var model = function() {
   var mu_x = 0;
   var sigma_x = 1;
   var sigma_y = 0.5;
   var latents = mapData({data: obs, batchSize: 100}, function(y) {
      var nnInput = Vector([y]);
      var nnOutput = nnEval(guideNet, nnInput);
      var x = sample(Gaussian({mu: mu_x, sigma: sigma_x}), {
         guide: Gaussian({mu: T.get(nnOutput, 0), sigma: softplus(T.get(nnOutput, 1))})
      });
```

[2] https://github.com/dritchie/adnn

```
16        observe(Gaussian({mu: x, sigma: sigma_y}), y);
17        return x;
18      });
19      return latents;
20    };
```

`mapData` operates much like `map` in a typical functional programming language, but it has two important features: (1) the optimization engine treats every execution of the mapped function as independent, and thus (2) the optimization engine can operate on stochastic mini-batches of the data, sized according to the `batchSize` option. Property (2) is clearly important for efficient, scalable optimization; we will see in Section 4 how property (1) can also be directly leveraged to improve optimization.

3.4 Defining Learnable Models

Thus far we have focused on defining parameterized guides for inference. Parameterized guides can also be used to make models learnable. The following three code blocks show possible replacements for line 7 of the previous example, replacing the hardcoded constant `mu_x = 0` with a learnable version:

```
// Maximum likelihood
var mu_x = sample(
   ImproperUniform(),
   { guide:
      Delta({v: paramScalar('mu_x')})
   });
// For convenience, equivalent to:
// modelParamScalar('mu_x');
```

```
// L2-regularized
// maximum likelihood
var mu_x = sample(
   Gaussian({mu: 0, sigma: 1}),
   { guide:
      Delta({v: paramScalar('mu_x')})
   });
```

```
// Variational Bayes
var mu_x = sample(
   Gaussian({mu: 0, sigma: 1}),
   { guide:
      Gaussian({
        mu: paramScalar('mu_x_m'),
        sigma: softplus(paramScalar('mu_x_s'))
      })
   });
```

The code in the left block results in maximum likelihood estimation. By using a Delta distribution as a guide, the inference engine will optimize for the single best parameter value (i.e. the center of the Delta distribution). Maximum likelihood behavior comes from using an improper uniform distribution (i.e. distribution with probability one everywhere) as a prior. This pattern is common enough that our system provides convenient shorthand for it (the `modelParam` functions). In the middle code of block, we demonstrate L2-regularized maximum likelihood learning by replacing the improper uniform prior with a Gaussian prior. The inference engine will still predict a point estimate for `mu_x`, but the Gaussian prior results in L2 regularization. Finally, the right block shows a variational Bayesian model: `mu_x` has a Gaussian prior distribution, and the guide samples `mu_x` from an approximate variational Gaussian posterior with optimizable parameters. This form learns a distribution over generative models, maintaining an estimate of uncertainty about the true model.

Note that we could have alternatively implemented maximum likelihood via a direct parameterization, e.g. `var mu_x = paramScalar('mu_x')`. However, this style results in p being parameterized in addition to q. This complicates both the implementation and the theoretical analyses that we show later in this paper. In contrast, our chosen scheme has only the guide parameterized; Learning the model is just part of learning the guide.

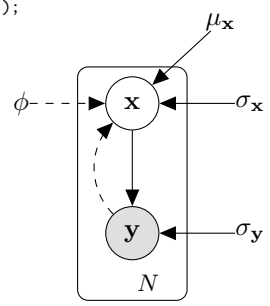
3.5 Examples: Simple Bayesian Networks

The example code we have built up in this section describes a Bayesian network with one continuous latent variable per continuous observation. Figure 1 Top shows the fully assembled code (using maximum likelihood estimation for the generative model parameters), along with a graphical model depiction using the notation of Kigma and Welling [15]. In this diagram, solid arrows indicate dependencies in the generative model given by the main program, and dashed arrows indicate dependencies in the guide program. ϕ is shorthand for all the neural network parameters in the guide program.

Figure 1 Bottom shows how to modify this code to instead have one discrete latent variable per observation; this is equivalent to a Gaussian mixture model. In this example, the `simplex` function maps a vector in $\mathbb{R}^{n-1}$ to the n-dimensional simplex (i.e. a vector whose entries sum to one). This process produces a vector of weights suitable for use as the component probabilities of a discrete random variable.

Figure 2 Top shows a slightly more complex Bayesian network with two latent continuous variables. Note that the guide program in this example predicts the two latent variables independently given the observation $\mathbf{y}$. In Figure 2 Bottom, we make some small changes to the code (lines 3 and 17,

```
var obs = loadData('data.json');
var guideNet = nn.mlp(1, [{nOut: 3, activation: nn.sigmoid}, {nOut: 2}], 'guideNet');
var model = function() {
   var mu_x = modelParamScalar('mu_x');
   var sigma_x = softplus(modelParamScalar('sigma_x'));
   var sigma_y = softplus(modelParamScalar('sigma_y'));
   var latents = mapData({data: obs}, function(y) {
      var nnInput = Vector([y]);
      var nnOutput = nnEval(guideNet, nnInput);
      var x = sample(Gaussian({mu: mu_x, sigma: sigma_x}), {
         guide: Gaussian({mu: T.get(nnOutput, 0),
                          sigma: softplus(T.get(nnOutput, 1))})
      });
      observe(Gaussian({mu: x, sigma: sigma_y}), y);
      return {x: x};
   });
   return latents;
};
```




```
var obs = loadData('data.json');
var nComps = 3;
var guideNet = nn.mlp(1, [{nOut: 3, activation: nn.sigmoid}, {nOut: nComps-1}], 'guideNet');
var model = function() {
   var theta_x = simplex(modelParamVector(nComps-1, 'theta_x'));
   var params_y = [
      {mu: modelParamScalar('mu_1'), sigma: softplus(modelParamScalar('s_1'))},
      {mu: modelParamScalar('mu_2'), sigma: softplus(modelParamScalar('s_2'))},
      {mu: modelParamScalar('mu_3'), sigma: softplus(modelParamScalar('s_3'))}
   ];
   var latents = mapData({data: obs}, function(y) {
      var nnInput = Vector([y]);
      var nnOutput = nnEval(guideNet, nnInput);
      var x = sample(Discrete({ps: theta_x}), {
         guide: Discrete(simplex(nnOutput))
      });
      observe(Gaussian(params_y[x]), y);
      return {x: x};
   });
   return latents;
};
```

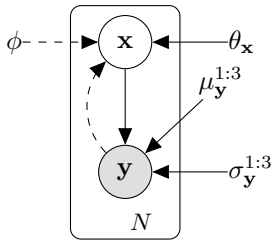


Figure 1: WebPPL code and corresponding graphical models for simple Bayesian networks with one latent variable per observation. *Top:* Continuous latent variable. *Bottom:* Discrete latent variable with 3 discrete values (i.e. a Gaussian mixture model with 3 mixture components).

highlighted in green) to instead have the guide program predict the second latent variable $\mathbf{x}_2$ as a function of the first latent variable $\mathbf{x}_1$. This small change allows the guide to capture a posterior dependency that was ingored by the first version.

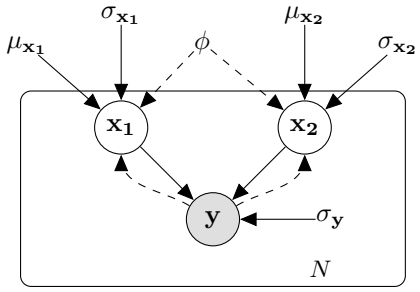
4 Optimizing Parameters

Now that we have seen how to author learnable guide programs, we will describe how to optimize the parameters of those programs.

4.1 ELBo: The Variational Objective

In Section 2.2, we mentioned that the goal of variational inference is to find values of the parameters ϕ for our guide program $q(\mathbf{x}|\mathbf{y};\phi)$ such that it is as close as possible to the true posterior $p(\mathbf{x}|\mathbf{y})$, where closeness is measured via KL-divergence. The KL-divergence between two general distributions is intractable to compute; however, some straightforward algebra produces an objective that is tractable

```
var obs = loadData('data.json');
var guideNet1 = nn.mlp(1, [{nOut: 3, activation: nn.sigmoid}, {nOut: 2}], 'guideNet1');
var guideNet2 = nn.mlp(1, [{nOut: 3, activation: nn.sigmoid}, {nOut: 2}], 'guideNet2');
var model = function() {
   var mu_x1 = modelParamScalar('mu_x1');
   var sigma_x1 = softplus(modelParamScalar('sigma_x1'));
   var mu_x2 = modelParamScalar('mu_x2');
   var sigma_x2 = softplus(modelParamScalar('sigma_x2'));
   var sigma_y = softplus(modelParamScalar('sigma_y'));
   var latents = mapData({data: obs}, function(y) {
      var nnInput1 = Vector([y]);
      var nnOutput1 = nnEval(guideNet1, nnInput1);
      var x1 = sample(Gaussian({mu: mu_x1, sigma: sigma_x1}), {
         guide: Gaussian({mu: T.get(nnOutput1, 0),
                          sigma: softplus(T.get(nnOutput1, 1))})
      });
      var nnInput2 = Vector([y]);
      var nnOutput2 = nnEval(guideNet2, nnInput2);
      var x2 = sample(Gaussian({mu: mu_x2, sigma: sigma_x2}), {
         guide: Gaussian({mu: T.get(nnOutput2, 0),
                          sigma: softplus(T.get(nnOutput2, 1))})
      });
      observe(Gaussian({mu: x1 + x2, sigma: sigma_y}), y);
      return {x1: x1, x2: x2};
   });
   return latents;
};
```




```
var obs = loadData('data.json');
var guideNet1 = nn.mlp(1, [{nOut: 3, activation: nn.sigmoid}, {nOut: 2}], 'guideNet1');
var guideNet2 = nn.mlp(2, [{nOut: 3, activation: nn.sigmoid}, {nOut: 2}], 'guideNet2');
var model = function() {
   var mu_x1 = modelParamScalar('mu_x1');
   var sigma_x1 = softplus(modelParamScalar('sigma_x1'));
   var mu_x2 = modelParamScalar('mu_x2');
   var sigma_x2 = softplus(modelParamScalar('sigma_x2'));
   var sigma_y = softplus(modelParamScalar('sigma_y'));
   var latents = mapData({data: obs}, function(y) {
      var nnInput1 = Vector([y]);
      var nnOutput1 = nnEval(guideNet1, nnInput1);
      var x1 = sample(Gaussian({mu: mu_x1, sigma: sigma_x1}), {
         guide: Gaussian({mu: T.get(nnOutput1, 0),
                          sigma: softplus(T.get(nnOutput1, 1))})
      });
      var nnInput2 = Vector([y, x1]);
      var nnOutput2 = nnEval(guideNet2, nnInput2);
      var x2 = sample(Gaussian({mu: mu_x2, sigma: sigma_x2}), {
         guide: Gaussian({mu: T.get(nnOutput2, 0),
                          sigma: softplus(T.get(nnOutput2, 1))})
      });
      observe(Gaussian({mu: x1 + x2, sigma: sigma_y}), y);
      return {x1: x1, x2: x2};
   });
   return latents;
};
```

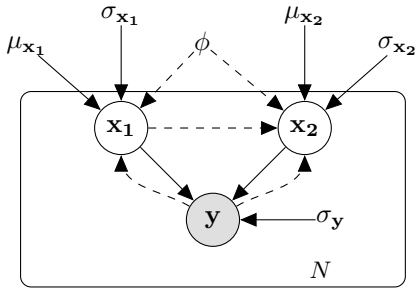


Figure 2: WebPPL code and corresponding graphical models for simple Bayesian networks with two latent variables per observation. *Top:* Guide program predicts the two latents independently. *Bottom:* Changing the guide program to treat the second latent variable as conditional on the first (green highlights show changes to the code).

(following the derivation of Wingate and Weber [32]):

$$\begin{aligned} D_{\mathrm{KL}}(q(\mathbf{x}|\mathbf{y};\phi)||p(\mathbf{x}|\mathbf{y})) &= \int_{\mathbf{x}} q(\mathbf{x}|\mathbf{y};\phi)\log\frac{q(\mathbf{x}|\mathbf{y};\phi)}{p(\mathbf{x}|\mathbf{y})} \\ &= \int_{\mathbf{x}} q(\mathbf{x}|\mathbf{y};\phi)\log\frac{q(\mathbf{x}|\mathbf{y};\phi)}{p(\mathbf{x},\mathbf{y})} + \log p(\mathbf{y}) \\ &= \mathbb{E}_q[\log(q(\mathbf{x}|\mathbf{y};\phi) - p(\mathbf{x},\mathbf{y}))] + \log p(\mathbf{y}) \\ &= \log p(\mathbf{y}) - \mathbb{E}_q[\log p(\mathbf{x},\mathbf{y}) - \log q(\mathbf{x}|\mathbf{y};\phi)] \\ &= \log p(\mathbf{y}) - \mathcal{L}(\mathbf{y},\phi) \geq 0 \end{aligned} \tag{2}$$

where the last inequality follows because KL-divergence is non-negative. This in turn implies that $\mathcal{L}(\mathbf{y},\phi)$ is a lower bound on the log marginal likelihood of the data (i.e. evidence) $\log p(\mathbf{y})$. Accordingly, $\mathcal{L}(\mathbf{y},\phi)$ is sometimes referred to as the 'Evidence Lower Bound', or ELBo [18]. Maximizing the ELBo corresponds to minimizing the KL-divergence. When $\log p(\mathbf{y}|\mathbf{x})$ can include unnormalized factors (as in our implementation), this is more properly called the variational free energy.

For an alternative derivation of the ELBo using Jensen's inequality, see Mnih and Gregor [20] and Jordan et al. [13, p. 213].

4.2 ELBo Gradient Estimators

Maximizing the ELBo requires estimating its gradient with respect to the parameters. There are two well-known approaches to performing this estimation:

Likelihood Ratio (LR) Estimator: In the general case, the gradient of the ELBo with respect to ϕ can be estimated by:

$$\begin{aligned} \nabla_\phi \mathcal{L}(\mathbf{y},\phi,\theta) &= \nabla_\phi \mathbb{E}_q[\log p(\mathbf{x},\mathbf{y};\theta) - \log q(\mathbf{x}|\mathbf{y};\phi)] \\ &= \mathbb{E}_q[\nabla_\phi \log q(\mathbf{x}|\mathbf{y};\phi)(\log p(\mathbf{x},\mathbf{y};\theta) - \log q(\mathbf{x}|\mathbf{y};\phi))] \end{aligned} \tag{3}$$

This estimator goes by the names 'likelihood ratio estimator' [5] and 'score function estimator' [2], and it is also equivalent to the REINFORCE policy gradient algorithm in the reinforcement learning literature [30]. The derivations of this estimator most relevant to our setting can be found in Wingate and Weber [32] and Mnih and Gregor [20]. Intuitively, each gradient update step using the LR estimator pushes the parameters ϕ in the direction $\nabla_\phi \log q(\mathbf{x}|\mathbf{y};\phi)$—that is, the direction that will maximize the probability of $\mathbf{x}$ under the guide. But since the goal of optimization is to learn an approximation to the true posterior, this update is weighted based on how probable $\mathbf{x}$ is under the joint distribution $p(\mathbf{x},\mathbf{y})$ (which is proportional to the true posterior). If $\mathbf{x}$ has high probability under the joint but low probability under the current guide, then the $(\log p(\mathbf{x},\mathbf{y}) - \log q(\mathbf{x}|\mathbf{y};\phi))$ term produces a large update (i.e. the guide should assign much higher probability to $\mathbf{x}$ than it currently does). If the joint and the guide assign equal probability to $\mathbf{x}$, then the update will have zero magnitude. If the joint assigns *lower* probability to $\mathbf{x}$ than the guide does, the resulting gradient update will move the parameters ϕ in the opposite direction of $\nabla_\phi \log q(\mathbf{x}|\mathbf{y};\phi)$.

The LR estimator is straightforward to compute, requiring only that $\log q(\mathbf{x}|\mathbf{y};\phi)$ be differentiable with respect to ϕ (the mean field and neural guide families presented in Section 2 satisfy this property). However, it is known to exhibit high variance. This problem is amenable to several variance reduction techniques, some of which we will employ later in this section.

Pathwise (PW) Estimator: Equation 3 suffers from high variance because the gradient can no longer be pushed inside the expectation: the expectation is with respect to q, and q depends on the parameters ϕ with respect to which we are differentiating. However, in certain cases, it is possible to re-write the ELBo such that the expectation distribution does not depend on ϕ. This situation occurs whenever the latent variables $\mathbf{x}$ can be expressed as samples from an un-parameterized distribution, followed by a parameterized deterministic transformation:

$$\mathbf{x} = g(\boldsymbol{\epsilon};\phi) \qquad \boldsymbol{\epsilon} \sim r(\cdot)$$

For example, sampling from a Gaussian distribution $\mathcal{N}(\mu,\sigma)$ can be expressed as $\mu + \sigma\cdot\epsilon$, where $\epsilon \sim \mathcal{N}(0,1)$. Continuous random variables which are parameterized by a location and a scale

parameter naturally support this type of transformation, and other types of continuous variables can often be well-approximated by deterministic transformations of unit normal variables [17].

Using this 'reparameterization trick' [15] allows the ELBo gradient to be rewritten as:

$$\begin{aligned}\nabla_\phi \mathcal{L}(\mathbf{y}, \phi, \theta) &= \nabla_\phi \mathbb{E}_q[\log p(\mathbf{x}, \mathbf{y}; \theta) - \log q(\mathbf{x}|\mathbf{y}; \phi)] \\ &= \nabla_\phi \mathbb{E}_r[\log p(g(\boldsymbol{\epsilon}; \phi), \mathbf{y}; \theta) - \log q(g(\boldsymbol{\epsilon}; \phi)|\mathbf{y}; \phi)] \\ &= \mathbb{E}_r[\nabla_\phi(\log p(g(\boldsymbol{\epsilon}; \phi), \mathbf{y}; \theta) - \log q(g(\boldsymbol{\epsilon}; \phi)|\mathbf{y}; \phi))]\end{aligned} \tag{4}$$

This estimator is called the 'pathwise derivative estimator' [4]. It transforms both the guide and target distributions into distributions over independent random 'noise' variables $\boldsymbol{\epsilon}$, followed by complex, parameterized, deterministic transformations. Given a fixed assignment to the noise variables, derivatives can propagate from the final log probabilities back to the input parameters, leading to much more stable gradient estimates than with the LR estimator.

4.3 Unified Gradient Estimator for Probabilistic Programs

A general probabilistic program makes many random choices, some of which are amenable to the reparameterization trick and others of which are not. Discrete random choices are never reparameterizable. Continuous random choices are reparameterizable if they can be expressed as a parameterized, deterministic transformation of a parameter-less random choce. In our implementation, every continuous random choice type either has this property or is well-approximated (and thus can be guided) by a random choice type that does (see Appendix A). Thus, for the rest of the paper, we will equate continuous with reparameterizable and discrete with non-reparameterizable. To optimize the ELBo for a probabilistic program p and an associated guide program q, we seek a single, unified gradient estimator that handles both discrete and continuous choices.

First, to simplify notation, we drop the dependency on ϕ from all derivations that follow. This is equivalent to making the parameters globally available, which is also true of our system (i.e. `paramScalar` and its ilk can be called anywhere).

Second, we assume that all random choices made by the guide program are first drawn from a distribution r and then transformed by a deterministic function g. Under this assumption, the distribution defined by a guide program factors as:

$$q(\mathbf{x}|\mathbf{y}) = \prod_i q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) \qquad \epsilon_i \sim r_i(\cdot)$$

As mentioned in Section 2.1, the length of $\boldsymbol{\epsilon}$ can vary across executions. However, since the guide program q by construction samples the same random choices in the same order as the target program p, q factors the same way as p (Equation 1) for any given execution. (Also note that the gradient with respect to parameters that affect only variables that are not created on a given execution will be zero, allowing us to simply ignore them.)

The distributions r_i and functions g_i have different meanings depending on whether the variable ϵ_i is continuous or discrete:

- Continuous ϵ_i: r_i is a unit normal or uniform distribution, and g_i is a parameterized transform. This is a direct application of the reparameterization trick. In this case, each local transformation g_i may also depend on the previous noise variables $\boldsymbol{\epsilon}_{<i}$, as choices occuring later in the program may be compound transformations of earlier choices.
- Discrete ϵ_i: $r_i = q_i$, and g_i is the identity function. This allows discrete choices to be represented in the reparameterization trick framework (without actually reparametrizing).

Given these assumptions, we can derive an estimator for the ELBo gradient:

$$\begin{aligned}\nabla_\phi \mathcal{L}(\mathbf{y}) &= \nabla_\phi \mathbb{E}_r[\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})] \\ &= \mathbb{E}_r[\nabla_\phi \log r(\boldsymbol{\epsilon}|\mathbf{y})(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})) + \nabla_\phi(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y}))] \\ &= \mathbb{E}_r[\underbrace{\nabla_\phi \log r(\boldsymbol{\epsilon}|\mathbf{y}) W(\boldsymbol{\epsilon}, \mathbf{y})}_{\text{LR term}} + \underbrace{\nabla_\phi \log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \nabla_\phi \log q(g(\boldsymbol{\epsilon})|\mathbf{y})}_{\text{PW term}}]\end{aligned} \tag{5}$$

where $W(\boldsymbol{\epsilon}, \mathbf{y}) = \log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})$; see Appendix B.1 for the derivation.

This estimator includes the LR and PW estimators as special cases. If all random choices are reparameterized (i.e. they are all continuous), then $r(\boldsymbol{\epsilon}|\mathbf{y})$ does not depend on ϕ, thus $\nabla_\phi \log r(\boldsymbol{\epsilon}|\mathbf{y})$ is zero and the LR term drops out, leaving only the PW term. If no random choices are reparameterized (i.e. they are all discrete), then the $\nabla_\phi \log q(g(\boldsymbol{\epsilon})|\mathbf{y})$ term drops out, using the identity $\mathbb{E}_f[\nabla \log f(x)] = 0$ (see Appendix B.2). The $\nabla_\phi \log p(g(\boldsymbol{\epsilon}), \mathbf{y})$ term is also zero, since only q and not p is dependent on ϕ, which leaves only the LR term.

While Equation 5 is a correct estimator for the ELBo gradient, like the LR estimator, the presence of discrete (i.e. non-reparameterized) random choices can lead to high variance. Thus we modify this estimator through three variance reduction techniques:

1. Replace $W(\boldsymbol{\epsilon}, \mathbf{y})$ with a separate $w_i(\boldsymbol{\epsilon}, \mathbf{y})$ for each factor $r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y})$ of $r(\boldsymbol{\epsilon}|\mathbf{y})$, as there exist independencies that can be exploited to reveal zero-expectation terms in $W(\boldsymbol{\epsilon}, \mathbf{y})$ for each i.
2. Subtract a constant 'baseline' term b_i from each $w_i(\boldsymbol{\epsilon}, \mathbf{y})$. This does not change the expectation, but it can reduce its variance, if designed carefully.
3. Factor $\nabla_\phi \log q(g(\boldsymbol{\epsilon})|\mathbf{y})$ in the PW term and remove factors corresponding to discrete (i.e. non-reparameterized) choices, since, as noted above, they have zero expectation.

Further details and correctness proofs for these three steps can be found in Appendix B. Applying them leads to the following estimator, which is the estimator actually used by our system:

$$\begin{aligned}
\nabla_\phi \mathcal{L}(\mathbf{y}) &= \mathbb{E}_r\Big[\sum_i \nabla_\phi \log r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y})(w_i(\boldsymbol{\epsilon}, \mathbf{y}) - b_i) + \nabla_\phi \log p(g(\boldsymbol{\epsilon}), \mathbf{y}) \;\; - \sum_{i \in \mathcal{C}} \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) \Big] \\
&= \mathbb{E}_r\Big[\sum_{i \in \mathcal{D}} \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y})(w_i(\boldsymbol{\epsilon}, \mathbf{y}) - b_i) + \nabla_\phi \log p(g(\boldsymbol{\epsilon}), \mathbf{y}) \;\; - \sum_{i \in \mathcal{C}} \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) \Big] \\
&= \mathbb{E}_r\Big[\nabla_\phi \log p(g(\boldsymbol{\epsilon}), \mathbf{y}) \;\; - \sum_i \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) v_i(\boldsymbol{\epsilon}, \mathbf{y}) \Big] \\
v_i(\boldsymbol{\epsilon}, \mathbf{y}) &= \begin{cases} -(w_i(\boldsymbol{\epsilon}, \mathbf{y}) - b_i) & \text{if } i \in \mathcal{D} \\ 1 & \text{if } i \in \mathcal{C} \end{cases}
\end{aligned} \tag{6}$$

where $\mathcal{C}$ and $\mathcal{D}$ are the sets of indices for all continuous and discrete random choices in the program execution, respectively. In the second line, we used the fact (mentioned above) that $\nabla_\phi \log r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y}) = 0$ for reparameterized (continuous) choices, and that $r = q$ and g is the identity for non-reparameterized (discrete) choices.

Equation 6 is similar to the 'surrogate loss function' gradient estimator of Schulman et al. [26], as an execution of a probabilistic program corresponds to one of their stochastic computation graphs. Their analysis is concerned with general stochastic objectives, however, while we focus particularly on the ELBo.

4.4 Optimization Interface

In Section 2.1, we showed how WebPPL programs use the `Infer` function to perform non-amortized inference on a `model` function. To optimize parameters for amortized inference, WebPPL provides an `Optimize` function with a similar interface:

```
var model = function() {
   // Use sample, guide, mapData, etc.
};

var params = Optimize(model, {
   steps: 100,
   optMethod: 'adam'
});
```

The code above performs 100 gradient update steps on `model` using the Adam stochastic optimization method [14]. The return value `params` of this function is a map from parameter names to optimized parameter values.

5 Using Learned Parameters

Given a set of learned parameters, our system can predict latent variables, generate synthetic data, or further refine parameters for a new dataset or a new model.

5.1 Predicting Latent Variables

A learned guide can be used to make inferences about new, never-before-seen observations. As an example, we'll use the Gaussian mixture model program in Figure 1 Bottom and show how to predict cluster assignments for new observations. Note that the observations used by the program are assigned to the `obs` variable, which is then passed to `mapData`. WebPPL is a purely functional language, so it does not support assigning a new dataset to `obs`. However, it does provide a special `globalStore` object whose fields can be re-assigned. With this in mind, we modify the Gaussian mixture model program as follows[3]:

```
globalStore.data = loadData('data.json');
// Set up guide neural net
var model = function() {
   // Set up generative parameters
   var latents = mapData({data: globalStore.data}, function(y) {
      // Guided sample latents, observe data points
   });
   return latents;
};
```

Now we can easily swap datasets using `globalStore.data`. Given a set of learned parameters `params` for this program, we can obtain a sample prediction for the latent variables for a new dataset:

```
globalStore.data = loadData('data_test.json');  // Load new test data set

// Forward sample from the guide
sample(Infer({method: 'forward', guide: true, params: params}, model));

// Use the guide as a Sequential Monte Carlo importance sampler
sample(Infer({method: 'SMC', particles: 100, params: params}));
```

We can make predictions either by running the guide program forward, or if the true posterior is very complex and the learned guide only partially approximates it, we can use the guide program as an importance sampler within Sequential Monte Carlo.

5.2 Generating Synthetic Data

Forward sampling from the guide can also be used to generate synthetic data from the learned distribution. If we make a slightly modified version of the Gaussian mixture model (call it `modelGen`) that samples data instead of observing it, we can used forward sampling with the optimized parameters `params` to synthesize new data points:

```
var modelGen = function() {
   var theta_x = simplex(modelParamVector(nComps-1, 'theta_x'));
   var params_y = [
      {mu: modelParamScalar('mu_1'), sigma: softplus(modelParamScalar('s_1'))},
      {mu: modelParamScalar('mu_2'), sigma: softplus(modelParamScalar('s_2'))},
      {mu: modelParamScalar('mu_3'), sigma: softplus(modelParamScalar('s_3'))}
   ];
   var x = sample(Discrete({ps: theta_x}));
   return sample(Gaussian(params_y[x]));
};

sample(Infer({method: 'forward', guide: true, params: params}, modelGen));
```

5.3 Further Optimization

A set of learned parameters `params` can also be passed back into `Optimize` for further optimization:

[3] An alternative to using `globalStore` for mutation would be to re-create the `model` function, closing over the test data instead of the training data.

```
var newParams = Optimize(model, {
   steps: 100,
   optMethod: 'adam',
   params: params
});
```

This can be useful for e.g. fine-tuning existing parameters for a new dataset. Indeed, `model` does not even need to be the same program that was originally used to learn `params`; it just needs to declare some parameters (via `paramScalar` etc.) with the same names as parameters in `params`. This can be useful for, for example, making a modification to an existing model without having to re-train its guide program from scratch, or for bootstrap training from a simpler model to a more complex one.

6 Experiments

Having detailed how to specify and optimize guide programs in our system, in this section, we experimentally evaluate how well programs written in our system can learn generative models and approximate posterior samplers. Unless stated otherwise, we use the following settings for the experiments in this section:

- The Adam optimization method [14] with $\alpha = 0.1$, $\beta_1 = 0.9$, and $\beta_2 = 0.999$.
- One sample from q per optimization step to estimate the expectation in Equation 6.

6.1 Gaussian Mixture Model

We first consider the simple Gaussian mixture model program from Figure 1 Bottom. This program samples discrete random choices, so its gradient estimator will include an LR term. Alternatively, we could re-write the program slightly to explicitly marginalize out the discrete random choices; see Appendix C.1. Marginalizing out of these choices leads to a tighter bound on the marginal log likelihood, so we would expect this version of the program to achieve a higher ELBo. As an extra benefit, the gradient estimator for this program then reduces to the PW estimator, which will have lower variance. These benefits come at the cost of amortized inference, however, as this version of the program does not have a guide which can predict the latent cluster assignment given an observed point. We also consider a non-amortized, mean field version of the program for comparison.

Figure 3 illustrates the performance of these programs after training for 200 steps on a synthetic datset of 100 points. On the left, we show how the ELBo changes during optimizaton. As expected, ELBo progress asymptotes at a higher value for the marginalized model. On the right, we show the estimated negative log likelihood of a separate synthetic test set under each program after optimization. Here, we also include the true model (i.e. the model used to synthesize the test/training data) for comparsion. As suggested by its optimization performance, the model with discrete choices marginalized out performs best. Note that the amortized guide program slightly out-performs the mean field guide program, indicating that the generalization provided by amortization has benefits for training generative models, in addition to enabling fast predictions of latent variables for previously-unseen observations.

6.2 QMR-DT

We next consider a more complicated Bayesian network model based on the QMR-DT medical diagnosis network [27]. QMR-DT is a bipartite graphical model with one layer of nodes corresponding to latent causes (e.g. diseases, in the medical setting) and a second layer of observed effects (e.g. symptoms). All nodes are binary (i.e. Bernoulli), and the cause nodes are connected to the effects via directed noisy-or links. Appendix C.2 shows our implementation.

Our amortized guide program for this model uses a neural network to jointly predict the probabilities of all latent cause variables given a set of observed effects. Since the QMR-DT model contains a large number of discrete random variables, we expect the variance reduction strategies introduced in Section 4.3 to have significant effect. Thus, we consider training this guide program with no variance reduction (*Amortized*, step size 10^{-5}), with per-choice likelihood ratio weights, (+ *local weights*, step size 10^{-3}), and with both per-choice weights and baselines (+ *baselines*, step size 10^{-2}). As a point of reference, we also include a mean field model (step size 10^{-2}) which uses all variance reduction strategies. Data for our experiments is sampled from a randomly-generated graph with 200

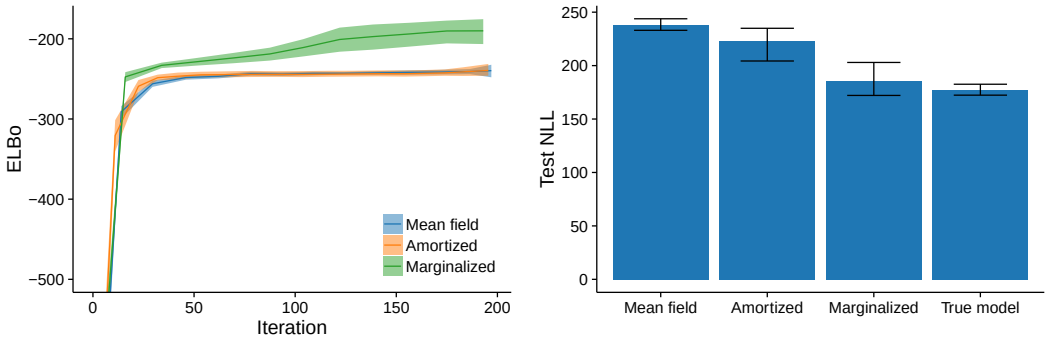


Figure 3: Performance of simple Gaussian mixture model program. *(Left)* ELBo optimization progress during training. The optimization objective for the marginalized model is a tighter bound on the marginal log likelihood and thus has a higher asymptote. *(Right)* Negative log-likelihood of a held-out test set.

causes and 100 effects. We sample 1000 observations for the training set and an additional 100 for a held-out test set.

Figure 4 shows the results of our experiments. The left plot shows optimization progress under each condition. Without any of the variance reduction strategies, gradients are extremely noisy and optimization makes almost no progress. Using local, per-variable likelihood ratio weights allows optimization to mke progress, and adding per-variable baselines further boosts performance. Though it uses all variance reduction strategies, the mean field model trains significantly more slowly than the variance-reduced amortized models. This happens because the mean field model has separate parameters for each training observation, rather than a single parameter set shared by a neural network, i.e. it has many more parameters that are each updated by very few gradient steps. Amortization thus both facilitates fast posterior prediction and exhibits faster training due to parameter sharing.

We next evaluate the guide program's posterior prediction ability. We use the learned guide to sample latent causes given an observed set of effects from the test set, sample effects given those causes, and then record what percentage of active effects in the test set observation are correctly predicted by the effects 'hallucinated' from our model. Specifically, if $\mathbf{e}$ is a vector of effect variables of length N, then the metric we use is:

$$\min(F(\mathbf{e}_{\text{true}}, \mathbf{e}_{\text{sampled}}), F(\mathbf{e}_{\text{sampled}}, \mathbf{e}_{\text{true}}))$$

$$F(\mathbf{e}_1, \mathbf{e}_2) = \frac{1}{\sum_i^N \mathbf{e}_1(i)} \sum_{i, \mathbf{e}_1(i)=1} \mathbb{1}\{\mathbf{e}_2(i) = 1\}$$

where $\mathbf{e}_{\text{true}}$ are effects from the test set and $\mathbf{e}_{\text{sampled}}$ are hallucinated from the model. Figure 4 Right plots the average F score over 100 runs, where we compare our amortized guide program against using the prior program to sample latent causes. The learned guide program correctly predicts more than twice as many active effects.

In addition to the guide program described above, which predicts all latent causes jointly given the observed effects, we also experimented with 'factored' guide programs which predict each latent cause one-by-one given the observed effects. We consider a guide that predicts each latent cause independently (*Factored*), as well as a guide that introduces dependencies between all latent causes via a recurrent neural network (*Factored + GRU*). The recurrent network receives as input the value of each latent cause as it is sampled, maintaining a persistent hidden state that, in theory, can capture the values of all latent causes sampled thus far. We use the gated recurrent unit (GRU) architecture for its ability to capture a longer-range dependencies [1], with a hidden state of dimension 20. The code for these programs is shown in Appendix C.2. We use stepsize 0.01 for both during optimization.

Figure 5 compares these guide programs with the joint guide used earlier in this section. While the independent factored guide performs slightly less well than the joint guide, adding the recurrent neural network to capture posterior dependencies improves performance to slightly better than the joint guide. One caveat is that the *Factored + GRU* guide takes significantly longer to train in our

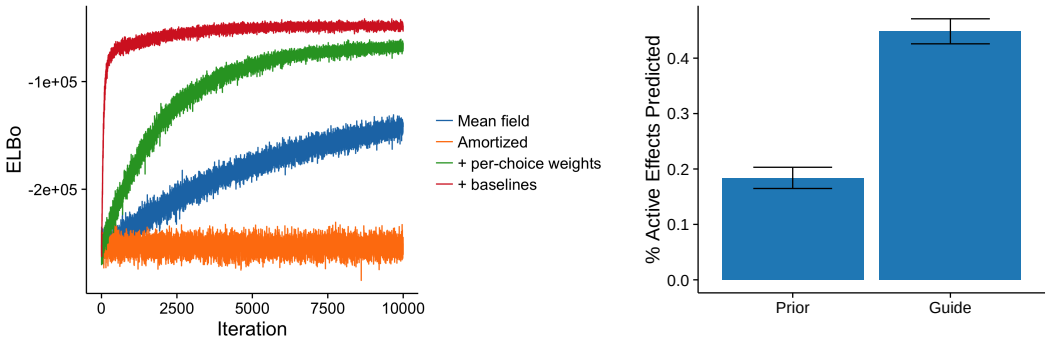


Figure 4: Performance of a QMR-DT model. *(Left)* ELBo optimization progress during training. *(Right)* Percentage of test set active effects correctly predicted using latent causes sampled from either the prior or the guide program.

implementation due to the recurrent network computation. Later in the paper, we discuss how this guide structure might point the way toward automatically deriving guide programs.

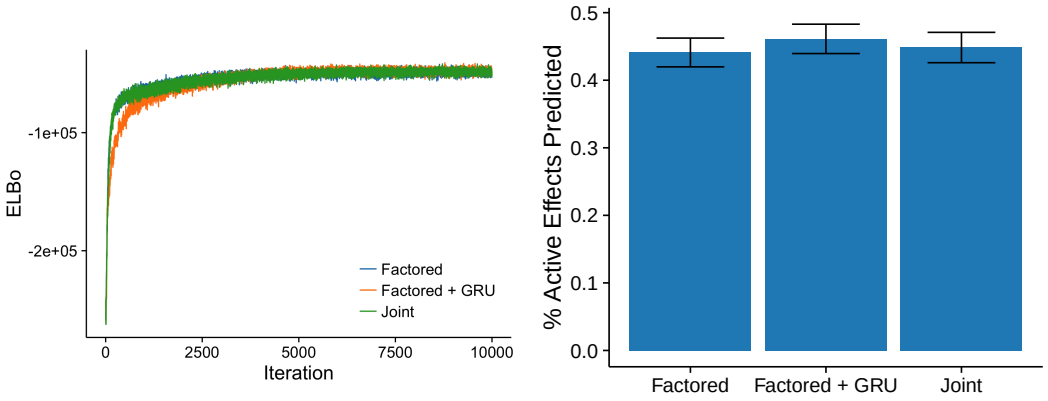


Figure 5: Experimenting with factored guide programs for QMR-DT. *(Left)* ELBo optimization progress during training. *(Right)* Percentage of test set active effects correctly predicted using latent causes sampled from either the prior or the guide program.

6.3 Latent Dirichlet Allocation

We also used our system to implement amortized inference in Latent Dirichlet Allocation topic models, over a data set of abstracts taken from the Stanford Computation and Cognition Lab's publication page.[4]

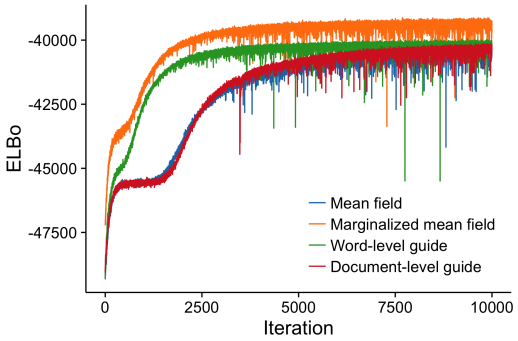
We experimented with two different amortized guide programs. The first is local to each word in a document (*Word-level guide*): it learns to predict the latent topic for each word, given the word and the latent topic distribution for the document. The second is local to each document (*Document-level guide*): it learns to predict the latent topic distribution for the document, given the words in the document and the latent word distributions for each topic. These two guides support amortized inference at different granularities and thus have different parameter sharing characteristics, which may lead to different learning behavior. For comparison, we also included two non-amortized conditions: a mean field model, and a mean field model with the latent choice of topic per word marginalized out (*Marginalized mean field*). Code for all of these programs can be found in Appendix C.3. We use five topics and learning rate 0.01 in all experiments.

[4]Thanks to Robert Hawkins for creating this dataset.

Figure 6.3 shows the results of these experiments. In the optimization progress plot on the left, we see that the marginalized mean field model achieve the highest ELBo. This is consistent with the results from the Gaussian mixture model experiments: marginalizing out latent variables when possible leads to a tighter bound on the marginal log likelihood.

Of our two amortized guides, the word-level guide performs better (and nearly as well as the marginalized model), likely due to increased parameter sharing. The document-level guide performs at least as well as the mean field model while also being able to efficiently predict topic distributions for previously-unseen documents.

Figure 6.3 Right shows the top ten highest probability words in each inferred topic for the marginalized mean field model. From left to right, these topics appear to be about experiments, pragmatic language models, knowledge acquisition, probabilistic programming languages, and a grab bag of remaining topics.



Topic 1	Topic 2	Topic 3	Topic 4	Topic 5
model	model	loop	probabilistic	attempt
causal	language	learning	models	inference
dynamic	pragmatic	abstract	programming	novel
judgments	abstract	extension	church	decisions
evidence	interpretation	bayesian	abstract	required
abstract	inference	knowledge	using	models
actions	knowledge	human	inference	behavior
pedagogical	words	causal	stochastic	bayesian
people	form	theories	language	cognition
participants	using	theory	model	expressive

Figure 6: Performance of Latent Dirichlet Allocation models. *(Left)* ELBo optimization progress during training. The optimization objective for the marginalized model is a tighter bound on the marginal log likelihood and thus has a higher asymptote. *(Right)* Top ten highest probability words in each inferred topic for the best-performing model.

6.4 Neural Generative Models: Variational Autoencoder & Sigmoid Belief Network

Our system naturally supports generative models which use neural network components. Two prominent examples of models in this class include the Variational Autoencoder (VAE) and Sigmoid Belief Networks (SBN). Both models sample latent variables from a multivariate distribution and then transform the result via a neural network to produce observed variables, often in the form of an image. The VAE uses a latent multivariate Gaussian distribution, whereas the SBN uses a latent multivariate Bernoulli.

Appendix C shows implementations of these models in our system. Our VAE implementation follows the original description of the model by Kingma and Welling [15], and our SBN implementation follows that of Mnih and Gregor [20]. The VAE uses a 20-dimensional latent code, and the SBN uses a single layer of 200 hidden variables. Our system cannot express the two-layer SBN of Mnih and Gregor, because its guide model samples the latent variables in the reverse order of the generative model.

Figure 7 Left shows results of training these models on the MNIST dataset, using Adam with a step size of 0.001. While both models train quickly at first, the SBN's training slows more noticeably than the VAE's due to its discrete nature. It takes more than three times as many iterations to achieve the same ELBo. In Figure 7 Right, we qualitatively evaluate both models by using them to reconstruct images from the MNIST test set. We use the guide program to sample latent variables conditional on the images in the "Target" column (i.e. the 'encoder' phase). We then transform these latent variables using the generative model's neural networks (i.e. the 'decoder' phase) to produce the reconstructed images in the "VAE" and "SBN" columns. As suggested by their training behavior, the VAE is able to generate higher-quality reconstructions after less training.

Our optimization exhibits some differences from the previous work. For the VAE, Kingma and Welling [15] exploit the closed-form solution of the KL divergence between two Gaussians to create

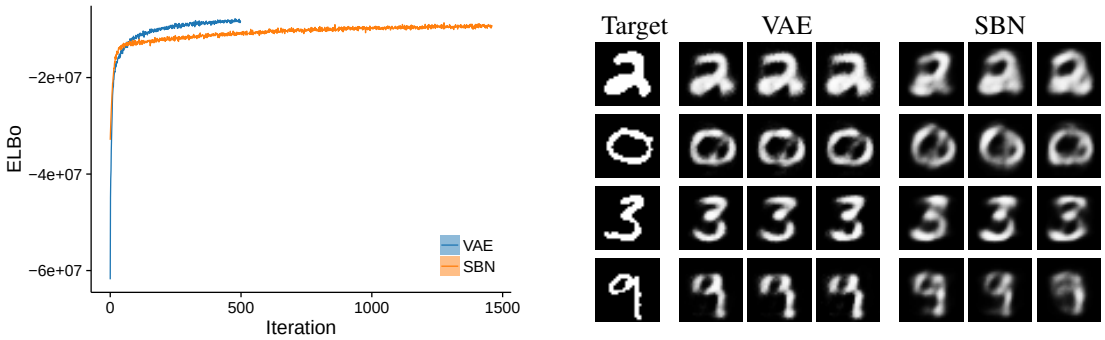


Figure 7: Evaluating the Variational Autoencoder (VAE) and Sigmoid Belief Network (SBN) programs on the MNIST dataset. *(Left)* ELBo optimization progress during training. *(Right)* Reconstructing the images in the "Target" column using both models.

an even lower-variance estimator of the ELBo gradient. We use a more general formulation, but our system can still successfully train the model. For the SBN, Mnih and Gregor [20] use neural networks to compute the per-variable baselines b_i in Equation 6, whereas we use a simpler approach (see Appendix B). However, the key point is that each of these models was described in a simple WebPPL program with neural guides and optimized by the default system, without the need for additional implementation efforts.

7 Deriving Guides Automatically

Thus far, we have shown how we can succesfully create and train guide programs for several types of generative models. However, writing guide programs can sometimes be tedious and repetitive; for example, note the large amount of shared structure between the guides shown in Figures 1 and 2. Furthermore, it is not always obvious how to write a good guide program. In Figure 2, knowledge of the structure of this very simple generative model led us to add a direct dependency between the two latent variables in the guide. For general programs—especially large, complex ones—it will not always be clear what these dependencies are or how to capture them with a guide.

This section describes our early experience with automatically deriving guide programs. We first describe how our system provides sensible default behavior that can make writing some guides less cumbersome. We then outline how the system might be extended to automatically derive guides for any program using recurrent neural networks.

7.1 Mean Field by Default

If a call to `sample` is not provided with an explicit guide distribution, our system automatically inserts a mean field guide. For example, the code `sample(Gaussian({mu: 0, sigma: 1}))` results in:

```
sample(Gaussian({mu: 0, sigma: 1}), {
   guide: Gaussian({mu: paramScalar(<auto_name>), sigma: softplus(paramScalar(<auto_name>))})
})
```

where parameter bounding transforms such as `softplus` are applied based on bounds metadata provided with each primitive distribution type. We use reparameterizable guides for continuous distributions (see Appendix A).

Since this process declares new optimizable parameters automatically, we must automatically generate names for these parameters. Our system names parameters according to where they are declared in the program execution trace, using the same naming technique as is used for random choices in probabilistic programming MCMC engines [31]. Since the names of these parameters are tied to the structure of the program, they cannot be re-used by other programs (as in the 'Further Optimization' example of Section 5.3).

7.2 Beyond Mean Field: Automatic Factored Guides with Recurrent Networks

In Section 6.2, we experimented with a factored guide program for the QMR–DT model. We think that this general style of guide—predicting each random choice in sequence, conditional on the hidden state of a recurrent neural network—might be generalized to an automatic guide for any program, as any probabilistic program can be decomposed into a sequence of random choices. In our QMR-DT experiments, we used a separate neural network (with separate parameters) to predict each latent variable (i.e. random choice). For complex models and large data sets, this approach would lead to a computationally unfeasible explosion in the number of parameters. Furthermore, it is likely that the prediction computations for many random choices in the program are related. For example, in the QMR-DT program, latent causes that share many dependent effects may be well predicted by the same or very similar networks.

Given these insights, we imagine a universally-applicable guide that uses a single prediction network for all random choices, but to which each random choice provides an additional identifying input. These IDs should be elements in a vector space, such that more 'similar' random choices have IDs which are close to one another for some distance metric in the vector space. One possible way to obtain such IDs would be to learn an embedding of the program-structural addresses of each random choice [31]. These might be learned in an end-to-end fashion by making them learnable parameter vectors in the overall variational optimization (i.e. letting closeness in the embedding space be an emergent property of optimizing our overall objective).

8 Conclusion

In this paper, we presented a system for amortized inference in probabilistic programs. Amortization is achieved through parameterized *guide programs* which mirror the structure of the original program but can be trained to approximately sample from the posterior. We introduced an interface for specifying guide programs which is flexible enough to reproduce state-of-the-art variational inference methods. We also demonstrated how this interface supports model learning in addition to amortized inference. We developed and proved the correctness of an optimization method for training guide programs, and we evaluated its ability to optimize guides for Bayesian networks, topic models, and deep generative models.

8.1 Future Work

There are many exciting directions of future work to pursue in improving amortized inference for probabilistic programs. The system we have presented in this paper provides a platform from which to explore these and other possibilities:

More modeling paradigms In this paper, we focused on the common machine learning modeling paradigm in which a global generative model generates many IID data points. There are many other modeling paradigms to consider. For example, time series data is common in machine learning applications. Just as we developed `mapData` to facilitate efficient inference in IID data models, we might develop an analogous data processing function for time series data (i.e. `foldData`). Using neural guides with such a setup would permit amortized inference in models such as Deep Kalman Filters [16]. In computer vision and computer graphics, a common paradigm for generative image models is to factor image generation into multiple steps and condition each step on the partially-generated image thus far [29, 25]. Such 'yield-so-far' models should also be possible to implement in our system.

Better gradient estimators While the variance reduction strategies employed by our optimizer make inference with discrete variables tractable, it is still noticeably less efficient then with purely continuous models. Fortunately, there are ongoing efforts to develop better, general-purpose discrete estimators for stochastic gradients [10, 21]. It should be possible to adapt these methods for probabilistic programs.

Automatic guides As discussed in Section 7, we believe that automatically deriving guide programs using recurrent neural networks may soon be possible. Recent enhancements to recurrent networks may be necessary to make this a reality. For example, the external memory of the Neural Turing Machine may be better at capturing certain long-range posterior dependencies [8]. We might also

draw inspiration from the Neural Programmer-Interpreter [23], whose stack of recurrent networks which communicate via arguments might better capture the posterior dataflow of arbitrary programs.

Other learning objectives In this paper, we focused on optimizing the ELBo. If we flip the direction of KL divergence in Equation 2, the resulting functional is in *upper* bound on the log marginal likelihood of the data—an 'Evidence Upper Bound,' or EUBo. Computing the EUBo and its gradient requires samples from the true posterior and is thus unusable in many applications, where the entire goal of amortized inference is to find a way to tractably generate such samples. However, some applications can benefit from it, if the goal is to speed up an existing tractable inference algorithm (e.g. SMC [25]), or if posterior execution traces are available through some other means (e.g. input examples from the user). There may also be less extreme ways to exploit this idea for learning. For example, in a `mapData`-style program, we might interleave normal ELBo updates with steps that hallucinate data from the posterior predictive (using a guide for global model parameters) and train the local guide to correctly parse these 'dreamed-up' examples. Such a scheme bears resemblance to the wake-sleep algorithm [12].

Control flow While our system's one-to-one mapping between random choices in the guide and in the target program makes the definition and analysis of guides simple, there are scenarios in which more flexibility is useful. In some cases, one may want to insert random choices into the guide which do not occur in the target program (e.g. using a compound distribution, such as a mixture distribution, as a guide). And for models in which there is a natural hierarchy between the latent variables and the observed variables, having the guide run 'backwards' from the observed variables to the top-most latents has been shown to be useful [28, 22, 20]. It is worth exploring how to support these (and possibly even more general) control flow deviations in a general-purpose probabilistic programming inference system.

Acknowledgments

This material is based on research sponsored by DARPA under agreement number FA8750-14-2-0009. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of DARPA or the U.S. Government.

References

[1] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation. In *EMNLP 2014*.

[2] M. C. Fu. Gradient Estimation. *Handbooks in operations research and management science*, 13, 2006.

[3] Sam Gershman and Noah D. Goodman. Amortized Inference in Probabilistic Reasoning. In *Proceedings of the Thirty-Sixth Annual Conference of the Cognitive Science Society*, 2014.

[4] P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Springer Science & Business, 2003.

[5] P. W. Glynn. Likelihood ratio gradient estimation for stochastic systems. *Communications of the ACM*, 33(10), 1990.

[6] Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: a language for generative models. In *UAI 2008*.

[7] Noah D. Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2015-12-23.

[8] Alex Graves, Greg Wayne, and Ivo Danihelka. Neural Turing Machines. *CoRR*, arXiv:1410.5401, 2014.

[9] E. Greensmith, P. L. Bartlett, and J. Baxter. Variance reduction techniques for gradient estimates in reinforcement learning. *The Journal of Machine Learning Research*, 5, 2004.

[10] Shixiang Gu, Sergey Levine, Ilya Sutskever, and Andriy Mnih. MuProp: Unbiased Backpropagation for Stochastic Neural Networks. In *ICLR 2016*.

[11] Georges Harik and Noam Shazeer. Variational Program Inference. *CoRR*, arXiv:1006.0991, 2010.

[12] GE Hinton, P Dayan, BJ Frey, and RM Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268, 1995.

[13] M. Jordan, Z. Ghahramani, T. Jaakkola, and L. Saul. Introduction to variational methods for graphical models. *Machine Learning*, 37, 1999.

[14] Diederik P. Kingma and Jimmy Ba. Adam: A Method for Stochastic Optimization. In *ICLR 2015*.

[15] Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. In *ICLR 2014*.

[16] Rahul G. Krishnan, Uri Shalit, and David Sontag. Deep Kalman Filters. *CoRR*, arXiv:1511.05121, 2015.

[17] Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic Variational Inference in Stan. In *NIPS 2015*.

[18] J. Manning, R. Ranganath, K. Norman, and D. Blei. Black Box Variational Inference. In *AISTATS 2014*.

[19] Vikash K. Mansinghka, Daniel Selsam, and Yura N. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *CoRR*, arXiv:1404.0099, 2014.

[20] Andriy Mnih and Karol Gregor. Neural Variational Inference and Learning in Belief Networks. In *ICML 2014*.

[21] Andriy Mnih and Danilo J. Rezende. Variational inference for Monte Carlo objectives. In *ICML 2016*.

[22] B. Paige and F. Wood. Inference Networks for Sequential Monte Carlo in Graphical Models. In *ICML 2016*.

[23] Scott Reed and Nando de Freitas. Neural Programmer-Interpreters. In *ICLR 2016*.

[24] Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *ICML 2014*.

[25] Daniel Ritchie, Anna Thomas, Pat Hanrahan, and Noah D. Goodman. Neurally-Guided Procedural Models: Amortized Inference for Procedural Graphics Programs using Neural Networks. In *NIPS 2016*.

[26] John Schulman, Nicolas Hess, Theophane Weber, and Pieter Abbeel. Gradient Estimation Using Stochastic Computation Graphs. In *NIPS 2015*.

[27] M. Shwe, B. Middleton, D. Heckerman, M. Henrion, E. Horvitz, H. Lehmann, and G. Cooper. Probabilistic diagnosis using a reformulation of the INTERNIST-1/QMR knowledge base. I. The probabilistic model and inference algorithms. *Methods of Information in Medicine*, 30.

[28] Andreas Stuhlmüller, Jessica Taylor, and Noah D. Goodman. Learning Stochastic Inverses. In *NIPS 2013*.

[29] Aaron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional Image Generation with PixelCNN Decoders. *CoRR*, arXiv:1606.05328, 2016.

[30] Ronald J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine Learning*, 8, 1992.

[31] David Wingate, Andreas Stuhlmüller, and Noah D. Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *AISTATS 2011*.

[32] David Wingate and Theophane Weber. Automated Variational Inference in Probabilistic Programming. In *NIPS 2012 Workshop on Probabilistic Programming*.

[33] F. Wood, J. W. van de Meent, and V. Mansinghka. A New Approach to Probabilistic Programming Inference. In *AISTATS 2014*.

A Appendix: Reparameterizations

Examples of primitive random choice distributions that can be reparameterized via a location-scale transform:

Distribution	$\boldsymbol{\epsilon \sim r(\cdot)}$	$\boldsymbol{g(\epsilon)}$
Gaussian(μ, σ)	Gaussian(0, 1)	$\mu + \sigma \cdot \epsilon$
LogitNormal(μ, σ)	Gaussian(0, 1)	sigmoid$(\mu + \sigma \cdot \epsilon)$
LogisticNormal($\boldsymbol{\mu}$, $\boldsymbol{\sigma}$)	Gaussian($\mathbf{0}$, $\mathbf{1}$)	simplex$(\boldsymbol{\mu} + \boldsymbol{\sigma} \cdot \epsilon)$
InverseSoftplusNormal(μ, σ)	Gaussian(0, 1)	softplus$(\mu + \sigma \cdot \epsilon)$
Exponential(λ)	Uniform(0, 1)	$-\log(\epsilon)/\lambda$
Cauchy(x_0, γ)	Uniform(0, 1)	$x_0 + \gamma \cdot \tan(\pi \cdot (\epsilon - 0.5))$

Examples of primitive distributions that do not have a location-scale transform but can be guided by a reparameterizable approximating distribution:

Distribution	**Guide Distribution**
Uniform	LogitNormal
Beta	LogitNormal
Gamma	InverseSoftplusNormal
Dirichlet	LogisticNormal

B Appendix: Gradient Estimator Derivations & Correctness Proofs

B.1 Derivation of Unified Gradient Estimator (Equation 5)

$$
\begin{aligned}
\nabla_\phi \mathcal{L}(\mathbf{y}) &= \nabla_\phi \mathbb{E}_r[\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})] \\
&= \nabla_\phi \int_{\boldsymbol{\epsilon}} r(\boldsymbol{\epsilon}|\mathbf{y})(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})) \\
&= \int_{\boldsymbol{\epsilon}} \nabla_\phi r(\boldsymbol{\epsilon}|\mathbf{y})(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})) + r(\boldsymbol{\epsilon}|\mathbf{y})\nabla_\phi(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})) \\
&= \int_{\boldsymbol{\epsilon}} r(\boldsymbol{\epsilon}|\mathbf{y})\nabla_\phi \log r(\boldsymbol{\epsilon}|\mathbf{y})(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})) + r(\boldsymbol{\epsilon}|\mathbf{y})\nabla_\phi(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})) \qquad (7) \\
&= \mathbb{E}_r[\nabla_\phi \log r(\boldsymbol{\epsilon}|\mathbf{y})(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y})) + \nabla_\phi(\log p(g(\boldsymbol{\epsilon}), \mathbf{y}) - \log q(g(\boldsymbol{\epsilon})|\mathbf{y}))]
\end{aligned}
$$

Line 7 makes use of the identity $\nabla f(x) = f(x)\nabla \log f(x)$.

B.2 Zero Expectation Identities

In what follows, we will make frequent use of the following:

Lemma 1. *If $f(x)$ is a probability distribution, then:*

$$\mathbb{E}_f[\nabla \log f(x)] = 0$$

Proof.

$$\mathbb{E}_f[\nabla \log f(x)] = \int_x f(x) \nabla \log f(x) = \int_x \nabla f(x) = \nabla \int_x f(x) = \nabla 1 = 0$$

□

Lemma 2. *For a discrete random choice ϵ_i and a function $f(\boldsymbol{\epsilon}_{<i}, \mathbf{y})$:*

$$\mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) f(\boldsymbol{\epsilon}_{<i}, \mathbf{y})] = 0$$

Proof.

$$\begin{aligned}
\mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) f(\boldsymbol{\epsilon}_{<i}, \mathbf{y})] &= \int_{\boldsymbol{\epsilon}} r(\boldsymbol{\epsilon}|\mathbf{y}) \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) f(\boldsymbol{\epsilon}_{<i}, \mathbf{y}) \\
&= \int_{\boldsymbol{\epsilon}} r(\boldsymbol{\epsilon}|\mathbf{y}) \nabla_\phi \log r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y}) f(\boldsymbol{\epsilon}_{<i}, \mathbf{y}) \\
&= \int_{\boldsymbol{\epsilon}_{<i}} r(\boldsymbol{\epsilon}_{<i}|\mathbf{y}) f(\boldsymbol{\epsilon}_{<i}, \mathbf{y}) \sum_{\epsilon_i} r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y}) \nabla_\phi \log r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y}) \int_{\boldsymbol{\epsilon}_{>i}} r(\boldsymbol{\epsilon}_{>i}|\boldsymbol{\epsilon}_{\leq i}, \mathbf{y}) \\
&= \int_{\boldsymbol{\epsilon}_{<i}} r(\boldsymbol{\epsilon}_{<i}|\mathbf{y}) f(\boldsymbol{\epsilon}_{<i}, \mathbf{y}) \cdot \mathbb{E}_r[\nabla_\phi \log r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y})] \cdot 1 \\
&= \int_{\boldsymbol{\epsilon}_{<i}} r(\boldsymbol{\epsilon}_{<i}|\mathbf{y}) f(\boldsymbol{\epsilon}_{<i}, \mathbf{y}) \cdot 0 = 0
\end{aligned}$$

where the last line makes use of Lemma 1. □

B.3 Variance Reduction Step 1: Zero Expectation W Terms

In this section, we show that for each random choice i, we can remove terms from $W(\boldsymbol{\epsilon}, \mathbf{y})$ to produce $w_i(\boldsymbol{\epsilon}, \mathbf{y})$. Specifically, we prove that while $w_i(\boldsymbol{\epsilon}, \mathbf{y}) \neq W(\boldsymbol{\epsilon}, \mathbf{y})$, we still have $\mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) w_i(\boldsymbol{\epsilon}, \mathbf{y})] = \mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) W(\boldsymbol{\epsilon}, \mathbf{y})]$.

To enable the computation of each w_i, our system builds a directed acyclic dependency graph as the program executes. The graph is constructed as follows:

- **On program start:** Create a root node `root`. Set `prev = root`. This holds the previous node and will be used to assign node parents.
- **On `sample` or `observe`:** Create a new node `node` representing this random choice/observation. Set `node.parent = prev`. Update `prev = node`.
- **On `mapData` begin:** Create two new nodes `split` and `join`. These nodes will delimit the beginning and ending of the `mapData` iteration. Push `split, join` onto a stack `mapDataStack`. This stack keeps track of `split, join` nodes when there are nested calls to `mapData`.
- **On `mapData` iteration begin:** Retrieve `split, join = top(mapDataStack)`. Update `prev = split`. This step reflects the independence assumptions of `mapData`: an iteration of `mapData` does not depend on any previous iterations, so there are no such edges in the graph. Instead, each iteration of `mapData` points back to the beginning of the `mapData` call.
- **On `mapData` iteration end:** Retrieve `split, join = top(mapDataStack)`. Add `prev` to `join.parents`. This step connects the last random choice in each `mapData` iteration to the `join` node.
- **On `mapData` end:** Retrieve `split, join = top(mapDataStack)`. Update `prev = join`. This step acknowledges that any subsequent computation may depend on the `mapData` as a whole.

In this graph, all nodes correspond to random choices or observations, except for the special `mapData` nodes `split` and `join`. When there are no calls to `mapData`, the graph has a linear topology, where each node is connected via a parent edge to the previously-sampled/observed node. `mapData` calls introduce fanout-fanin subgraphs: the `split` node fans out into separate linear chains for each `mapData` iteration, and the last nodes of these chains fan in to the `join` node. Figure 8 shows the resulting graph for one execution of a simple example program.

```
var data = loadData('data.json');
var model = function() {
   var a = sample(Bernoulli({p: 0.5}));
   mapData({data: data}, function(y) {
      var x = sample(Gaussian({mu: a ? 0 : 5, sigma: 1}));
      observe(Gaussian({mu: x, sigma: 0.5}), y);
   });
   return a;
}
```

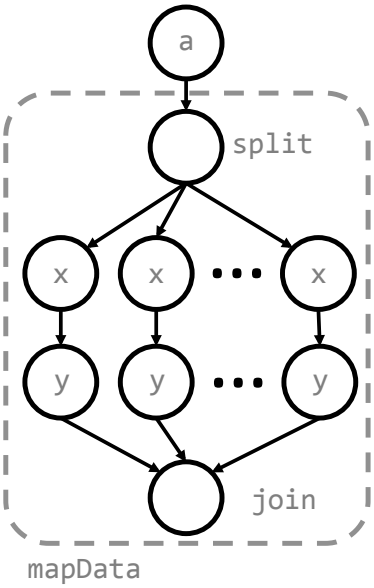


Figure 8: A conservative dependency graph *(Right)* resulting from one execution through a simple program *(Left)*.

By construction, this graph is an overly-conservative dataflow dependency graph: if random choice ϵ_a flows into random choice ϵ_b (or observation y_c), then a path $\epsilon_a \to \epsilon_b$ (or $\epsilon_a \to y_c$) exists in the graph. The converse is not necessarily true (i.e. there can exist edges between nodes that have no dataflow dependency). Note also that, by construction, the existence of a path $\epsilon_a \to \epsilon_b$ implies that ϵ_b was sampled after ϵ_a in the program execution order.

From the perpective of a random choice node ϵ_i, the graph nodes can be partitioned into the following subsets:

- $\mathcal{D}_i$: the nodes "downstream" from ϵ_i (i.e. the set of all nodes d for which an edge $\epsilon_i \to d$ exists.
- $\mathcal{U}_i$: the nodes "upstream" from ϵ_i (i.e the set of all nodes u for which an edge $u \to \epsilon_i$ exists.
- $\mathcal{C}_i$: the set of nodes which are in neither $\mathcal{D}_i$ nor $\mathcal{U}_i$.

Figure 9 illustrates these partitions on an example graph. For convenience, we also define $\mathcal{B}_i \equiv \mathcal{U}_i \cup \mathcal{C}_i$.

For any given random choice ϵ_i, these partitions allows us to factor $W(\boldsymbol{\epsilon}, \mathbf{y})$ into three terms:

$$\begin{aligned} W_i(\boldsymbol{\epsilon}, \mathbf{y}) &= \log \frac{p(g(\boldsymbol{\epsilon}), \mathbf{y})}{q(g(\boldsymbol{\epsilon})|\mathbf{y})} \\ &= \log \frac{p(\mathcal{B}_i)}{q(\mathcal{B}_i)} + \log \frac{p(g(\epsilon_i)|\mathcal{B}_i)}{q(g(\epsilon_i)|\mathcal{B}_i)} + \log \frac{p(\mathcal{D}_i|\epsilon_i, \mathcal{B}_i)}{q(\mathcal{D}_i|\epsilon_i, \mathcal{B}_i)} \\ &= w_i(\mathcal{B}_i) + w_i(\epsilon_i, \mathcal{B}_i) + w_i(\mathcal{D}_i, \epsilon_i, \mathcal{B}_i) \end{aligned}$$

In the remainder of this section, we prove that the $w_i(\mathcal{B}_i)$ term can be safely removed. Specifically, we prove the following:

Theorem 1. *For a discrete random choice ϵ_i:*

$$\mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) w_i(\mathcal{B}_i)] = 0$$

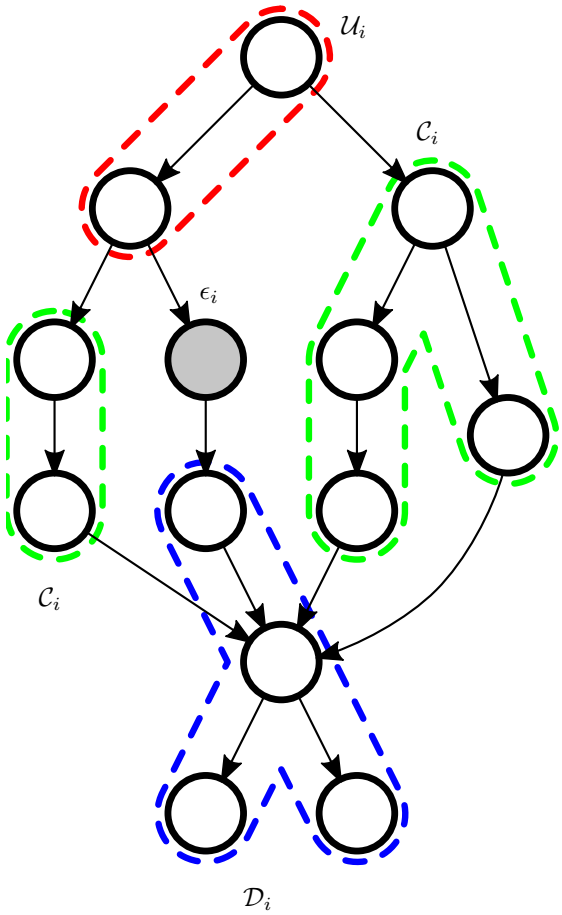


Figure 9: Partitioning a dependency graph into multiple node subsets from the perspective of a random variable node ϵ_i.

In proving this theorem, the following two lemmas will be useful:

Lemma 3.

$$\boldsymbol{\epsilon}_{<i} \cap \mathcal{D}_i = \varnothing$$

Proof. By definition, $\boldsymbol{\epsilon}_{<i}$ are all random choices which occur before ϵ_i in program execution order. Also by definition, for all $d \in \mathcal{D}_i$, there exists a path $\epsilon_i \to d$. As noted above, by construction, this implies that d was created after ϵ_i in program execution order. Thus d cannot be in $\boldsymbol{\epsilon}_{<i}$. □

Lemma 4.

$$q(g_i(\epsilon_i)|\mathcal{B}_i) \equiv q(g_i(\epsilon_i)|\mathcal{U}_i \cup \mathcal{C}_i) = q(g_i(\epsilon_i)|\mathcal{U}_i)$$

Proof. By construction there is no directed path $\epsilon_i \to \mathcal{C}_i$, or vice versa. Further, all nodes that are upstream of both ϵ_i and $\mathcal{C}_i$ are in $\mathcal{U}_i$ (because all nodes that are upstream of ϵ_i are in $\mathcal{U}_i$ by definition). It follows that these sets are conditionally independent:

$$q(g_i(\epsilon_i), \mathcal{C}_i|\mathcal{U}_i) = q(g_i(\epsilon_i)|\mathcal{U}_i)q(\mathcal{C}_i|\mathcal{U}_i).$$

Using this, the result is immediate:

$$q(g_i(\epsilon_i)|\mathcal{U}_i \cup \mathcal{C}_i) = \frac{q(g_i(\epsilon_i), \mathcal{C}_i|\mathcal{U}_i)}{q(\mathcal{C}_i|\mathcal{U}_i)} = q(g_i(\epsilon_i)|\mathcal{U}_i).$$

□

A corollary of this lemma is that $q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) = q(g_i(\epsilon_i)|\mathcal{U}_i)$, since $\boldsymbol{\epsilon}_{<i}$ must be a subset of $\mathcal{B}_i$. We can now prove Theorem 1:

$$\begin{aligned}
\mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) w_i(\mathcal{B}_i)] &= \int_{\boldsymbol{\epsilon}} r(\boldsymbol{\epsilon}|\mathbf{y}) \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) w_i(\mathcal{B}_i) \\
&= \int_{\mathcal{B}_i} r(\mathcal{B}_i) w_i(\mathcal{B}_i) \sum_{\epsilon_i} r(\epsilon_i|\mathcal{B}_i) \int_{\mathcal{D}_i} r(\mathcal{D}_i|\epsilon_i, \mathcal{B}_i) \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) \\
&= \int_{\mathcal{B}_i} r(\mathcal{B}_i) w_i(\mathcal{B}_i) \sum_{\epsilon_i} r(\epsilon_i|\mathcal{B}_i) \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) \int_{\mathcal{D}_i} r(\mathcal{D}_i|\epsilon_i, \mathcal{B}_i) \qquad (8) \\
&= \int_{\mathcal{B}_i} r(\mathcal{B}_i) w_i(\mathcal{B}_i) \sum_{\epsilon_i} q(g_i(\epsilon_i)|\mathcal{B}_i) \nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}), \mathbf{y}) \cdot 1 \qquad (9) \\
&= \int_{\mathcal{B}_i} r(\mathcal{B}_i) w_i(\mathcal{B}_i) \sum_{\epsilon_i} q(g_i(\epsilon_i)|\mathcal{U}_i) \nabla_\phi \log q(g_i(\epsilon_i)|\mathcal{U}_i) \qquad (10) \\
&= \int_{\mathcal{B}_i} r(\mathcal{B}_i) w_i(\mathcal{B}_i) \mathbb{E}_q[\nabla_\phi \log q(g_i(\epsilon_i)|\mathcal{U}_i)] \\
&= \int_{\mathcal{B}_i} r(\mathcal{B}_i) w_i(\mathcal{B}_i) \cdot 0 = 0 \qquad (11)
\end{aligned}$$

Line 8 uses Lemma 3. Line 9 uses the fact that ϵ_i is discrete. Line 10 uses Lemma 4 and its corollary. Finally, line 11 uses Lemma 1.

We should note that similar methods to our removal of terms from W have been used by prior work to reduce variance in LR gradient estimators. Rao-blackwellization, using the Markov blanket of a node in a graphical model, produces a similar estimator [18]. In deep generative models, a similar technique allows the derivation of lower-variance estimators for each layer of the deep generative network [20]. Finally, less conservative (i.e. exact) dependency graphs can be used to reduce gradient variance in stochastic computation graphs [26].

B.4 Variance Reduction Step 2: Baselines

Next, we prove that subtracting a constant baseline term b_i from every w_i does not change the expectation in Equation 6:

$$\begin{aligned}
\mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i}))(w_i(\boldsymbol{\epsilon}, \mathbf{y}) - b_i)] &= \mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i})) w_i(\boldsymbol{\epsilon}, \mathbf{y})] - \mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i})) b_i] \\
&= \mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g(\boldsymbol{\epsilon}_{<i})) w_i(\boldsymbol{\epsilon}, \mathbf{y})]
\end{aligned}$$

Where the last step makes use of Lemma 2.

In our system, we use $b_i = \mathbb{E}[w_i]$, which we estimate with of a moving average of the samples used to compute gradients. While this choice of b_i is not guaranteed to reduce variance, it often does in practice, and previous systems for variational inference and reinforcement learning have exploited it [18, 26, 9]. Another option is to *learn* b_i, for example as a neural net function of $\mathbf{y}$ [20]. The proof above also permits b_i to be a function of $\boldsymbol{\epsilon}_{<i}$ (i.e. all previous random choices), which could reduce variance further by tracking posterior dependencies. This is a promising avenue for future work.

B.5 Variance Reduction Step 3: Zero Expectation q Factors

Finally, we prove that we can remove any factors corresponding to discrete (i.e. non-reparameterized choices) from the $\nabla_\phi \log q(g(\boldsymbol{\epsilon})|\mathbf{y})$ term in Equation 5 without changing its expectation:

$$\mathbb{E}_r[\nabla_\phi \log q(g_i(\epsilon_i)|g_{<i}(\boldsymbol{\epsilon}_{<i}), \mathbf{y})] = \mathbb{E}_r[\nabla_\phi \log r(\epsilon_i|\boldsymbol{\epsilon}_{<i}, \mathbf{y})] = 0$$

where we have used Lemma 2 and the fact that $r = q$ and g is the identity for discrete random choices.

C Appendix: Example Programs

C.1 Gaussian mixture model

With discrete choices marginalized out:

```
var obs = loadData('data.json');
var nComps = 3
var model = function() {
   var theta_x = simplex(modelParamVector(nComps-1, 'theta_x'));
   var params_y = [
      {mu: modelParamScalar('mu_y_1'), sigma: softplus(modelParamScalar('sigma_y_1'))},
      {mu: modelParamScalar('mu_y_2'), sigma: softplus(modelParamScalar('sigma_y_2'))},
      {mu: modelParamScalar('mu_y_3'), sigma: softplus(modelParamScalar('sigma_y_3'))}
   ];
   mapData({data: obs}, function(y) {
      // Explicitly sum out latent mixture component
      var scores = mapIndexed(function(i, muSigma) {
         var w = T.get(theta_x, i);
         return Gaussian(muSigma).score(y) + Math.log(w);
      }, params_y);
      // Summed-out likelihod
      factor(logsumexp(scores));
   });
};
```

C.2 QMR-DT

Program with joint guide:

```
var graph = loadQMRGraph('qmr_graph.json');
var data = loadData('qmr_data.json');

var noisyOrProb = function(symptomNode, diseases) {
   var cp = product(map(function(parent) {
      return diseases[parent.index] ? (1 - parent.prob) : 1;
   }, symptomNode.parents));
   return 1 - (1-symptomNode.leakProb)*cp;
};

var guideNet = nn.mlp(graph.numSymptoms, [
   {nOut: graph.numDiseases, activation: sigmoid}
], 'guideNet');
var predictDiseaseProbs = function(symptoms) {
   return nneval(guideNet, Vector(symptoms));
};

var model = function() {
   mapData({data: data, batchSize: 20}, function(symptoms) {
      var predictedProbs = predictDiseaseProbs(symptoms);
      var diseases = mapIndexed(function(i, disease) {
         return sample(Bernoulli({p: disease.priorProb}), {
            guide: Bernoulli({p: T.get(predictedProbs, i)})
         });
      }, graph.diseaseNodes);

      mapData({data: symptoms}, function(symptom, symptomIndex) {
         var symptomNode = graph.symptomNodes[symptomIndex];
         observe(Bernoulli({p: noisyOrProb(symptomNode, diseases)}), symptom);
      });
   });
};
```

Program with factored guide. Note that the guide uses a separate neural network (with separate parameters) to predict each latent cause.

```
var graph = loadQMRGraph('qmr_graph.json');
var data = loadData('qmr_data.json');

var noisyOrProb = function(symptomNode, diseases) {
   var cp = product(map(function(parent) {
      return diseases[parent.index] ? (1 - parent.prob) : 1;
   }, symptomNode.parents));
   return 1 - (1-symptomNode.leakProb)*cp;
};
```

```
var predictNet = cache(function(i) {
   return nn.mlp(graph.numSymptoms, [
      {nOut: 1, activation: sigmoid}
   ], 'predictNet_'+i);
});
var predictDiseaseProb = function(symptoms, i) {
   return T.get(nneval(predictNet(i), Vector(symptoms)), 0);
};

var model = function() {
   mapData({data: data, batchSize: 20}, function(symptoms) {
      var diseases = mapIndexed(function(i, disease) {
         return sample(Bernoulli({p: disease.priorProb}), {
            guide: Bernoulli({p: predictDiseaseProb(symptoms, i)})
         });
      }, graph.diseaseNodes);

      mapData({data: symptoms}, function(symptom, symptomIndex) {
         var symptomNode = graph.symptomNodes[symptomIndex];
         observe(Bernoulli({p: noisyOrProb(symptomNode, diseases)}), symptom);
      });
   });
};
```

Program with factored guide with GRU.

```
var graph = loadQMRGraph('qmr_graph.json');
var data = loadData('qmr_data.json');
var gruHiddenDim = 20;

var noisyOrProb = function(symptomNode, diseases) {
   var cp = product(map(function(parent) {
      return diseases[parent.index] ? (1 - parent.prob) : 1;
   }, symptomNode.parents));
   return 1 - (1-symptomNode.leakProb)*cp;
};

var predictNet = cache(function(i) {
   return nn.mlp(graph.numSymptoms + gruHiddenDim, [
      {nOut: 1, activation: sigmoid}
   ], 'predictNet_'+i);
});
var predictDiseaseProb = function(symptoms, i) {
   var inputs = T.concat(Vector(symptoms), globalStore.gruHiddenState);
   return T.get(nneval(predictNet(i), inputs, 0);
};

// GRU cell: takes input + previous hidden state, produces new hidden state.
var gru = makeGRU(gruHiddenDim, 1, 'gru');
var updateGRUState = function(latentChoiceVal) {
   var inputs = T.concat(Vector([latentChoiceVal]), globalStore.gruHiddenState);
   globalStore.gruHiddenState = nneval(gru, inputs);
};

var model = function() {
   mapData({data: data, batchSize: 20}, function(symptoms) {
      globalStore.gruHiddenState = zeros([gruHiddenDim]);
      var diseases = mapIndexed(function(i, disease) {
         var x = sample(Bernoulli({p: disease.priorProb}), {
            guide: Bernoulli({p: predictDiseaseProb(symptoms, i)})
         });
         updateGRUState(x);
         return x;
      }, graph.diseaseNodes);

      mapData({data: symptoms}, function(symptom, symptomIndex) {
         var symptomNode = graph.symptomNodes[symptomIndex];
         observe(Bernoulli({p: noisyOrProb(symptomNode, diseases)}), symptom);
      });
   });
};
```

C.3 LDA

To simplify the programs in this section, we use the technique of Section 7.1 to insert a mean field `LogisticNormal` guide for any `Dirichlet` random choice that does not have an explicit guide declared.

Mean field model:

```
var model = function(corpus, vocabSize, numTopics, alpha, eta) {
  var topics = repeat(numTopics, function() {
    return sample(Dirichlet({alpha: eta}));
  });

  mapData({data: corpus}, function(doc) {

    var topicDist = sample(Dirichlet({alpha: alpha}));

    mapData({data: doc}, function(word) {
      var z = sample(Discrete({ps: topicDist}));
      var topic = topics[z];
      observe(Discrete({ps: topic}), word);
    });

  });

  return topics;
};
```

Marginalized mean field model:

```
var model = function(corpus, vocabSize, numTopics, alpha, eta) {

  var topics = repeat(numTopics, function() {
    return sample(Dirichlet({alpha: eta}));
  });

  mapData({data: corpus}, function(doc) {

    var topicDist = sample(Dirichlet({alpha: alpha}));

    forEach(doc, function(count, word) {

      if (count > 0) {
        // Sum over topic assignments/z.
        var prob = sum(mapN(function(z) {
          var zScore = Discrete({ps: topicDist}).score(z);
          var wgivenzScore = Discrete({ps: topics[z]}).score(word);
          return Math.exp(zScore + wgivenzScore);
        }, numTopics));

        factor(Math.log(prob) * count);
      }

    });

  });

  return topics;

};
```

Word-level guide:

```
var model = function(corpus, vocabSize, numTopics, alpha, eta) {

  var numHid = 50;
  var embedSize = 50;
  var embedNet = nn.mlp(vocabSize, [{nOut: embedSize, activation: nn.tanh}], 'embedNet');

  var net = nn.mlp(embedSize + numTopics, [
    {nOut: numHid, activation: nn.tanh},
    {nOut: numTopics}
  ], 'net');

  var wordAndTopicDistToParams = function(word, topicDist) {
    var embedding = nneval(embedNet, oneOfK(word, vocabSize));
    var out = nneval(net, T.concat(embedding, T.sub(topicDist, 1)));
    return {ps: softplus(tensorToVector(out))};
```

```
  };

  var topics = repeat(numTopics, function() {
    return sample(Dirichlet({alpha: eta}));
  });

  mapData({data: corpus}, function(doc) {

    var topicDist = sample(Dirichlet({alpha: alpha}));

    mapData({data: doc}, function(word) {
      var z = sample(Discrete({ps: topicDist}), {
        guide: Discrete(wordAndTopicDistToParams(word, topicDist))
      });
      var topic = topics[z];
      observe(Discrete({ps: topic}), word);
    });

  });

  return topics;
};
```

Document-level guide:

```
var nets = cache(function(numHid, vocabSize, numTopics) {
  var init = nn.constantparams([numHid], 'init');

  var ru = makeRNN(numHid, vocabSize, 'ru');

  var outputHidden = nn.mlp(numHid, [
    {nOut: numHid, activation: nn.tanh}
  ], 'outputHidden');

  var outputMu = nn.mlp(numHid, [
    {nOut: numTopics - 1}
  ], 'outputMu');

  var outputSigma = nn.mlp(numHid, [
    {nOut: numTopics - 1}
  ], 'outputSigma');

  return {
    init: init,
    ru: ru,
    outputHidden: outputHidden,
    outputMu: outputMu,
    outputSigma: outputSigma
  };
});

var model = function(data, vocabSize, numTopics, alpha, eta) {
  var corpus = data.documentsAsCounts;
  var numHid = 20;
  var nets = nets(numHid, vocabSize, numTopics);

  var guideParams = function(topics, doc) {
    var initialState = nneval(nets.init);
    var state = reduce(function(x, prevState) {
      return nneval(nets.ru, [prevState, x]);
    }, initialState, topics.concat(normalize(Vector(doc))));
    var hidden = nneval(nets.outputHidden, state);
    var mu = tensorToVector(nneval(nets.outputMu, hidden));
    var sigma = tensorToVector(softplus(nneval(nets.outputSigma, hidden)));
    var params = {mu: mu, sigma: sigma};
    return params;
  };

  var topics = repeat(numTopics, function() {
    return sample(Dirichlet({alpha: eta}));
  });

  mapData({data: corpus}, function(doc) {

    var topicDist = sample(Dirichlet({alpha: alpha}), {
      guide: LogisticNormal(guideParams(topics, doc))
    });

```

```
    mapData({data: countsToIndices(doc)}, function(word) {
      var z = sample(Discrete({ps: topicDist}));
      var topic = topics[z];
      observe(Discrete({ps: topic}), word);
    });

  });

  return topics;
};
```

C.4 Variational Autoencoder

In this example and the one that follows, `nnevalModel` evaluates a neural network while also placing an improper uniform prior over the network parameters. This allows neural networks to be used as part of learnable models.

```
// Data
var data = loadData('mnist.json');
var dataDim = 28*28;
var hiddenDim = 500;
var latentDim = 20;

// Encoder
var encodeNet = nn.mlp(dataDim, [
   {nOut: hiddenDim, activation: nn.tanh}
], 'encodeNet');
var muNet = nn.linear(hiddenDim, latentDim, 'muNet');
var sigmaNet = nn.linear(hiddenDim, latentDim, 'sigmaNet');
var encode = function(image) {
   var h = nneval(encodeNet, image);
   return {
      mu: nneval(muNet, h),
      sigma: softplus(nneval(sigmaNet, h))
   };
};

// Decoder
var decodeNet = nn.mlp(latentDim, [
   {nOut: hiddenDim, activation: nn.tanh},
   {nOut: dataDim, activation: nn.sigmoid}
], 'decodeNet');
var decode = function(latent) {
   return nnevalModel(decodeNet, latent);
};

// Training model
var model = function() {
   mapData({data: data, batchSize: 100}, function(image) {
      // Sample latent code (guided by encoder)
      var latent = sample(TensorGaussian({mu: 0, sigma: 1, dims: [latentDim, 1]}), {
         guide: DiagCovGaussian(encode(image))
      });

      // Decode latent code, observe binary image
      var probs = decode(latent);
      observe(MultivariateBernoulli({ps: probs}), image);
   });
}
```

C.5 Sigmoid Belief Network

```
// Data
var data = loadData('mnist.json');
var dataDim = 28*28;
var latentDim = 200;

// Encoder
var encodeNet = nn.mlp(dataDim, [
   {nOut: latentDim, activation: nn.sigmoid}
], 'encodeNet');
var encode = function(image) {
   return nneval(encodeNet, image)
};
```

```
13
14 // Decoder
15 var decodeNet = nn.mlp(latentDim, [
16    {nOut: dataDim, activation: nn.sigmoid}
17 ], 'decodeNet');
18 var decode = function(latent) {
19    return nnevalModel(decodeNet, latent);
20 };
21
22 // Training model
23 var priorProbs = Vector(repeat(latentDim, function() { return 0.5; }));
24 var model = function() {
25    mapData({data: data, batchSize: 100}, function(image) {
26       // Sample latent code (guided by encoder)
27       var latent = sample(MultivariateBernoulli({ps: priorProbs}), {
28          guide: MultivariateBernoulli({ps: encode(image)})
29       });
30
31       // Decode latent code, observe binary image
32       var probs = decode(latent);
33       observe(MultivariateBernoulli({ps: probs}), image);
34    });
35 }
```