Automatic inference for higher-order probabilistic programs

Brooks Paige

Lady Margaret Hall

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Trinity 2016

Abstract

Probabilistic models used in quantitative sciences have historically co-evolved with methods for performing inference: specific modeling assumptions are made not because they are appropriate to the application domain, but because they are required to leverage existing software packages or inference methods. The intertwined nature of modeling and computational concerns leaves much of the promise of probabilistic modeling out of reach for data scientists, forcing practitioners to turn to off-the-shelf solutions. The emerging field of *probabilistic programming* aims to reduce the technical and cognitive overhead for writing and designing novel probabilistic models, by introducing a specialized programming language as an abstraction barrier between modeling and inference.

The aim of this thesis is to develop inference algorithms that scale well and are applicable to broad model families. We focus particularly on methods that can be applied to models written in general-purpose *higher-order* probabilistic programming languages, where programs may make use of recursion, arbitrary deterministic simulation, and higher-order functions to create more accurate models of an application domain. In a probabilistic programming system, probabilistic models are defined using a modeling language; a backend implements generic inference methods applicable to any model written in this language. Probabilistic programs — models — can be written without concern for how inference will later be performed.

We begin by considering several existing probabilistic programming languages, their design choices, and tradeoffs. We then demonstrate how programs written in higher-order languages can be used to define coherent probability models, describing possible approaches to inference, and providing explicit algorithms for efficient implementations of both classic and novel inference methods based on and extending sequential Monte Carlo. This is followed by an investigation into the use of variational inference methods within higher-order probabilistic programming languages, with application to policy learning, adaptive importance sampling, and amortization of inference.

Acknowledgements

This thesis is only possible because Frank Wood kidnapped me and brought me to England, and I'd like to thank him for that, and for being an encouraging and supportive supervisor throughout this entire process. I would additionally like to thank my parents for dealing so well with the fact that I suddenly disappeared to England, and also for being encouraging and supportive throughout this entire process; and my brother for always hosting me back home in NYC and for visiting me in Oxford longer than anyone else. I also would particularly like to thank Jan Willem van de Meent, whose collaboration, feedback, and friendship throughout this DPhil was invaluable.

This thesis also would not be the same without the input and assistance of my many additional collaborators and colleagues at Oxford and abroad; David Tolpin, Yee Whye Teh, Arnaud Doucet, Dino Sejdinovic, Hongseok Yang, Yura Perov, Christian Naesseth, Tom Rainforth, Fredrik Lindsten, Tuan-Anh Le, Stefan Webb, Rob Cornish, Jeff Siskind, Siddharth N, Lloyd Elliott, Seth Flaxman, Lawrence Murray, as well as everyone else who has passed through Frank's lab. I would also like to thank Zsófia Boda for ensuring that I at least sometimes left the office.

Additionally, I would like to thank EPSRC, the Bracken Fund, and the DARPA PPAML program for supporting me financially during the course of my studies in Oxford.

Publications and Collaborations

Portions of this thesis are based on previously published work by the author, performed in collaboration with others. On a chapter-by-chapter basis:

- Chapters 1–3, as well as Chapter 8, are wholly my own presentation and are not directly derived from any particular existing published work.
- Chapter 4 is largely based on Paige and Wood [2014], published at ICML 2014, presented here with expanded context and updated discussion.
- Chapter 5 previously appeared as Paige, Wood, Doucet, and Teh [2014] at NIPS 2014, and is presented with minor modifications and additional discussion here. The overall idea and algorithm was developed jointly in discussions between all four authors. In particular, the partially-deterministic branching rule in Section 5.3.2 was proposed by Yee Whye Teh, and the theoretical properties stated in Section 5.3.4 are due to Arnaud Doucet. The approach to implementation described in Section 5.4, as well as all source code including the probabilistic C implementation, all experiments and figures, and the overall presentation and written content are my own work.
- The application to policy search in Section 6.3 is based on van de Meent, Paige, Tolpin, and Wood [2016], published at AISTATS 2016. Chapter 6 overall provides an extended presentation of my particular contribution to this collaboration. The formulation and implementation of variational inference in Anglican is my own work, as is the presentation here.
- Chapter 7 previously appeared as Paige and Wood [2016] at ICML 2016, and is presented here in a slightly extended format.

All source code, implementation, and experimental figures are solely the work of the author, with the single exception of the Canadian traveller problem case study shown in Figures 6.6 and 6.7, which was implemented in conjunction with Jan Willem van de Meent and David Tolpin.

The approach taken in Chapter 3 and throughout in formulating probabilistic program execution as the interaction between continuations and an inference backend is informed by discussions with Frank Wood, David Tolpin, Jan Willem van de

Meent, Hongseok Yang, and others, and is loosely based on the approach we took in both Tolpin, van de Meent, Paige, and Wood [2015] and van de Meent, Paige, Tolpin, and Wood [2016].

Contents

List of Figures **ix**

1 Programs as probabilistic models **1**
1.1 Generative models in machine learning 2
1.2 Probabilistic programming goals 8
1.3 Challenges for probabilistic programming 10

2 Language expressivity and model complexity **13**
2.1 Probabilistic programming language design 14
2.2 First-order languages 16
2.2.1 Bugs and Jags: Gibbs sampling 17
2.2.2 Stan: Hamiltonian Monte Carlo 18
2.2.3 Infer.NET: Message passing 19
2.3 Higher-order languages 20
2.3.1 Execution-based semantics 21
2.3.2 Modeling capabilities 23
2.3.3 Inference challenges 24

3 Inference over partial program executions **26**
3.1 An interface for inference engines 27
3.2 Probability of a program execution 31
3.3 A first simulation-based inference engine 35
3.3.1 Importance sampling 36
3.3.2 As a probabilistic programming backend 39
3.4 Partitioning the program execution trace 40
3.5 A single-site Metropolis-Hastings algorithm 43
3.5.1 As a probabilistic programming backend 45
3.5.2 A database of random choices 47
3.6 Particle MCMC algorithms 48
3.6.1 Advanced particle MCMC methods 50
3.7 Do probability distributions defined in this manner behave as we would expect? 51

4 Efficient implementation of continuations **57**
4.1 Leveraging existing compilers and operating systems 58
4.1.1 Related work 60
4.2 C as a probabilistic programming language 60
4.3 Inference Algorithms in Probabilistic C 65
4.3.1 Sequential Monte Carlo 66
4.3.2 Particle Metropolis-Hastings 69
4.3.3 Particle Gibbs 69
4.4 Experimental validation 72
4.4.1 Comparative performance of inference engines 73
4.4.2 Performance characteristics across multiple cores 75
4.5 Discussion 75

5 Asynchronous anytime sequential Monte Carlo **79**
5.1 Related work 80
5.2 Background and notation 81
5.2.1 Resampling and degeneracy 82
5.2.2 Synchronization and limitations 83
5.3 The particle cascade 83
5.3.1 Local branching decisions 83
5.3.2 Variance reduction 85
5.3.3 Computing expectations and marginal likelihoods 86
5.3.4 Theoretical properties, unbiasedness, and consistency 86
5.4 Active bounding of memory usage 87
5.5 Particle cascade experiments 89
5.5.1 Performance and scalability 92
5.6 Discussion 93

6 Variational inference in higher-order languages **95**
6.1 Variational Bayes 97
6.1.1 Stochastic gradient variational inference 99
6.2 Variational probabilistic programs 100
6.2.1 Computing the gradient during program execution 102
6.2.2 Implementation challenges in higher-order languages 104
6.3 Maximum likelihood estimation and black-box policy learning . . . 107
6.3.1 Empirical Bayes estimation 107
6.3.2 Planning as inference 112
6.4 Different types of reparameterization 116
6.4.1 Automatic differentiation and reparameterization 117
6.4.2 Model parameterization and approximation quality 118
6.5 Variational inference and optimal importance sampling proposals . 119
6.6 Discussion 121

7 Amortized inference: compiling away runtime costs of inference 123
7.1 Preliminaries . . . 125
7.1.1 Sequential Monte Carlo for Bayesian networks . . . 126
7.1.2 Target densities and proposal kernels . . . 127
7.1.3 Neural autoregressive distribution estimation . . . 129
7.2 Approach . . . 130
7.2.1 Defining the inverse model . . . 130
7.2.2 Learning a family of approximating densities . . . 132
7.2.3 Joint conditional neural density estimation . . . 134
7.2.4 Training the neural network . . . 136
7.3 Examples . . . 137
7.3.1 Inverting a single factor . . . 137
7.3.2 A hierarchical Bayesian model . . . 139
7.3.3 Factorial hidden Markov model . . . 140
7.4 Discussion . . . 141

8 Future directions for probabilistic programming 144

Bibliography 147

List of Figures

1.1 A generative model 4
1.2 A Bayes net for a medical diagnosis system 6
1.3 A comparison: programming, statistics, and probabilistic programming 9

3.1 Random variables and dependency structure in a program execution 32
3.2 A sequential decomposition of the program execution 41
3.3 A Poisson sampling algorithm 52
3.4 Execution diagram for Poisson sampler 53

4.1 Probabilistic C: A Gaussian with unknown mean 61
4.2 Probabilistic C: A hidden Markov model 62
4.3 Probabilistic C: Infinite mixture of Gaussians 64
4.4 Comparison of compiled and interpreted implementations 74
4.5 Comparison of compiled inference algorithms 74
4.6 Effect of system architecture on runtime performance 75

5.1 Particle cascade: convergence per sample 90
5.2 Particle cascade: convergence per unit time 91
5.3 Particle cascade: average compute time across architectures 91
5.4 Particle count stability 93

6.1 A conditioned Dirichlet-multinomial model in Anglican 96
6.2 A posterior distribution as an unconditioned program 96
6.3 Learning parameters of a distribution 103
6.4 Empirical Bayes estimation 110
6.5 Comparison of variational inference and empirical Bayes procedures 111
6.6 The Canadian traveller problem 114
6.7 Convergence for CTP domains of 20 and 50 nodes 115

7.1 Graphical model and inverse for non-conjugate regression model . . 132
7.2 Graphical model and inverse for hierarchical Poisson model 133
7.3 Graphical model and inverse for factorial HMM 135
7.4 Structure of conditional neural density estimator 136
7.5 Learned proposals for non-conjugate polynomial regression 138

7.6 Convergence rates of marginal likelihood estimates 140
7.7 Effective sample size comparison for factorial HMM 141

1

Programs as probabilistic models

Creating simplified models of complex phenomena is a quintessentially human activity. We live and act in a world we cannot ever fully observe or understand: tomorrow's weather, what our flatmates really think about us, and even basic laws of physics are all fundamentally unknowable. In the face of this hopeless uncertainty, we turn to abstractions. A model isolates the fundamental variables which we believe important to a particular system, and hypothesizes a particular relationship between them. Then, to test these hypotheses, we collect data, and see whether these models are consistent with the world around us.

A *probabilistic model* or statistical model characterizes these relationships in terms of probabilities: it codifies a set of assumptions about the underlying process which produces the data we collect, specifying distributions over the possible values of unknown latent quantities and over the variation in the observed data. The probabilistic model is an idealized approximation to this process, a simulator which can be used to mimic observed behavior or data. A good probabilistic model can be used to generate data whose distribution is qualitatively close to that generated by the unknown simulation process in the real world. *Inference* in a probabilistic model is the act of reasoning backwards from the data, through the model: given the data that we have, what have we learned about the world?

This thesis is about *probabilistic programming*: a paradigm in which we specify models as small computer programs defining stochastic simulations, with inference then performed automatically. Probabilistic models used in quantitative sciences have historically co-evolved with methods for performing inference: specific modeling assumptions are made not because they are appropriate to the application domain, but because they are required to leverage existing software packages or inference methods. The intertwined nature of modeling and computational concerns leaves much of the promise of probabilistic modeling out of reach for data scientists, forcing practitioners to turn to off-the-shelf solutions. Probabilistic programming aims to reduce the technical and cognitive overhead for writing and designing novel probabilistic models, by introducing a specialized programming language as an abstraction barrier between modeling and inference.

1.1 Generative models in machine learning

Many machine learning problems can be abstracted as the task of estimating some unknown quantity $\mathbf{x}$, given some collected data $\mathbf{y}$: given an email message, is it spam? Given a photograph, does it contain a human face? Given two separate scans of hand-written text, do they show the same letters? Were they written by the same person? In these settings, the data is something hard and concrete — an email, a photograph, scanned text, a collection of pixels — and we are trying to estimate some underlying property. What was the intention of this email author? Do these pixels represent a face?

A large class of machine learning and statistical methods pose this problem as one of estimating a functional relationship: the goal is to fit some function f such that $\mathbf{x} \approx f(\mathbf{y})$. Probabilistic variants of this approach choose this function to define a conditional distribution $p(\mathbf{x}|\mathbf{y})$, defining the probability that $\mathbf{x}$ takes a particular value given the observed data $\mathbf{y}$. This is fundamentally a data-driven approach: to estimate this function, we require labeled emails ("spam", "not spam"), or labeled photographs ("there is a face in this region of the image"). That is, to fit this function f, we need data not just for $\mathbf{y}$, but also many labeled examples of

the properties $\mathbf{x}$ we are going to want to one day predict. Many recent advances in machine learning applications come from taking data-driven approaches using complex neural network regression models, and successfully scaling them up to very large datasets. These methods are essentially black-box: the function $f_\theta(\mathbf{y})$ or distribution $p_\theta(\mathbf{x}|\mathbf{y})$ is chosen from a large parametric family with free parameters θ, and using training data pairs $\mathbf{x}, \mathbf{y}$ the parameters can be optimized via gradient descent. These parameters θ are not interpretable quantities we may wish to infer; we have no insight into why or how our predictor makes its decisions; and the discriminative model is designed to answer one question alone: it captures the relationship $p(\mathbf{x}|\mathbf{y})$. This is a form of supervised learning which can be incredibly effective at reaching high levels of predictive accuracy for a specific task at hand. Standardized datasets of $\mathbf{x}, \mathbf{y}$ pairs can be used for benchmarking: new methods can be evaluated impartially by considering how well the learnt function can predict the quantity of interest on held-out test data.

As a contrasting approach, generative models are specified by defining a joint probability distribution $p(\mathbf{x}, \mathbf{y})$, modeling both the latent variables $\mathbf{x}$ as well as the observed data $\mathbf{y}$ — instead of only modeling the conditional distribution of $\mathbf{x}$ given a fixed $\mathbf{y}$. Conceptually this model definition often can be broken down into defining two, different probability distributions. The first is the *prior* distribution $p(\mathbf{x})$, which characterizes the space of all possible values the latent variables $\mathbf{x}$ could take. Sometimes this expresses itself simply in terms of constraints: for example, if the latent variable $\mathbf{x}$ is known to be one of a finite set of possibilities, or known to be bounded within some interval. It can also express the degree of plausibility for certain results: if some latent quantity $\mathbf{x}$ generally lies near some particular quantity, the prior distribution can place higher probability mass near such values, while still permitting progressively less expected (but still plausible) values further away. This part of the model describes the possible values $\mathbf{x}$ could take, *independently of any data*, and does not depend on $\mathbf{y}$. The second is a conditional distribution $p(\mathbf{y}|\mathbf{x})$ which describes the probability of the observed data, given the values of the random

How do you recognize this is the number four?

How do you generate an image of the number four?

Figure 1.1: A hand-written digit from the MNIST dataset [LeCun et al., 1998]. (left) When you see this image, you immediately recognize it as the number four. However, the process by which you come to know it is a four is difficult to characterize. (right) In contrast, the process for *generating* an image resembling the one provided is conceptually straightforward: given you are drawing the number four, make a downward stroke and a curved stroke.

variables in $\mathbf{x}$. This conditional distribution describes the generative simulation process for creating the data $\mathbf{y}$ if the latent values $\mathbf{x}$ are already known.

An important intuition here is that it is in many cases much easier to describe the set of possibilities for the unknown $\mathbf{x}$, followed by a manner for simulating values of $\mathbf{y}$, than it is to describe the backwards process $p(\mathbf{x}|\mathbf{y})$ directly. This tends to be because in machine learning tasks we seek to invert a causal relationship: given a picture of a handwritten digit, what number was the person trying to write? In what sequence did the person draw these strokes of the pen? As humans we are good at answering these questions, but not very good at explaining *why* or *how* we go about answering them. On the other hand, describing the forward generative process is straightforward, at least for a human: first we choose a number (from, say, a uniform distribution over the digits $0, 1, \ldots, 9$) and then we put pen to paper and make a few simple strokes, as in Figure 1.1. A recently-proposed approach to identifying handwritten characters frames the recognition problem in precisely this manner [Lake et al., 2015], demonstrating an unprecedented human-like ability to produce meaningful representations of new characters after seeing only a single written example.

For a machine, answering questions like whether an email is spam or if an image contains a face requires learning a complex mapping function from $\mathbf{y}$ to $\mathbf{x}$. In situations where we have many training examples of $\mathbf{x}, \mathbf{y}$ pairs available to fit such a function, we can learn this directly from data; however, in other situations we

may find ourselves short on data but long on understanding of the specific subject domain or problem at hand. In cases like these, we can use domain knowledge to define a generative model that is capable of simulating both the unknown quantities $\mathbf{x}$ as well as the data $\mathbf{y}$ that we collect.

Generative models are then related back to the questions we are trying to answer using *Bayes' rule*, which makes explicit the relationship between model and inference. Bayes' rule relates two conditional probabilities, with

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{p(\mathbf{y})} = \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x})}{\int p(\mathbf{y}|\mathbf{x})p(\mathbf{x})\mathrm{d}\mathbf{x}}. \tag{1.1}$$

Thus, once one has defined a generative model — the distributions $p(\mathbf{x})$ and $p(\mathbf{y}|\mathbf{x})$ — all that remains to characterize the *posterior* distribution $p(\mathbf{x}|\mathbf{y})$, the relationship we are trying to learn, is the minor computational "detail" of evaluating Equation (1.1).

Why write generative models?

If we are interested in a relationship $p(\mathbf{x}|\mathbf{y})$, perhaps it sounds backwards to first take the time to define $p(\mathbf{y}|\mathbf{x})$ and $p(\mathbf{x})$. However, there are a number of arguments in favor of generative modeling, relative to purely data-driven approaches to fitting a black-box function relating $\mathbf{x}$ and $\mathbf{y}$.

Perhaps the best-known success stories of generative models are those of *graphical models*. An example of a toy graphical model used as part of a medical diagnostics system [Lauritzen and Spiegelhalter, 1988] is shown in Figure 1.2. In this setting there are many latent variables, which have a complex dependency structure. This sort of example highlights many of the advantages of specifying the model structure in a generative fashion. First, the overall model design is principled, not heuristic: top-down, not bottom up. This leads naturally to interpretable latent variables; all the values in our model have a sensible, coherent relationship enforced by the model structure imposed by the joint distribution $p(\mathbf{x}, \mathbf{y})$. This also leads to clear understanding of the assumptions being made by our estimator. Fitting a black-box function to estimate $p(\mathbf{x}|\mathbf{y})$ may perform oddly at corner cases, or in regimes which are not well-represented by the training data used to fit the model; on the other

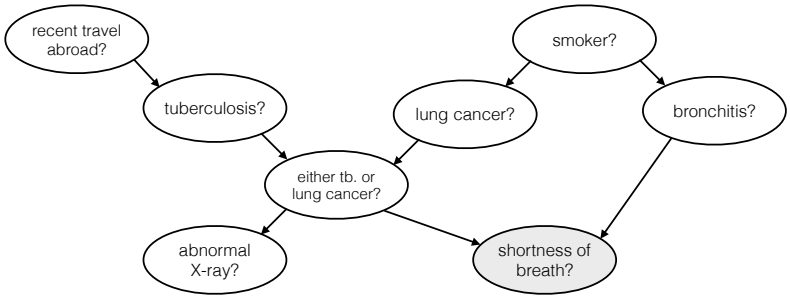


Figure 1.2: An example Bayesian network, taken from Lauritzen and Spiegelhalter [1988]. This network represents a hypothetical medical diagnosis system which could by used to diagnose the hypothetical cause — bronchitis, lung cancer, tuberculosis, or possible none of the above — for a patient who exhibits shortness of breath. Individual nodes in this graph represent binary random variables, and the direction of the arrows denotes the direction of influence. This generative model for shortness of breath can be used to answer a variety of queries: not just for predicting the cause, but also for determining which additional test or follow-up question would be most useful in confirming a diagnosis.

hand, since the generative model clarifies all relationships between observed and random variables, all modeling assumptions are entirely explicit rather than being somehow expressed by the presence or absence of specific training data. When inference in a generative model gives unexpected or undesirable results, this can be clearly attributed to incorrect modeling assumptions, rather than (say) the need for more data. Methods which instead look to approximate $p(\mathbf{x}|\mathbf{y})$ directly with a black-box function can be thought of as implicitly performing approximate posterior inference under some sort of joint distribution — however, this distribution and its assumptions are completely unknown.

There are numerous additional advantages: the framing of the relationship from data to latent variables as a posterior distribution means Bayesian methods naturally additionally characterize the uncertainty in estimates of the latent variables through the concentration of the posterior. The use of the generative model to provide structure yields more data-efficient inference than in equivalent discriminative modeling settings [Ng and Jordan, 2002], albeit potentially at a cost in predictive accuracy in the large-data regime. The graphical structure of graphical models lends itself to easy compositionality of simpler models into larger models. It

also lends itself to causal inference, and reasoning about the effect of possible interventions [Pearl, 1995].

Generative models and simulation models can be far more complex than graphical models such as that in Figure 1.2. Nonparametric models employ infinite dimensional parameter spaces; conditional distributions may be arbitrarily complex; variables may be linked through complex deterministic processes. Taken together, domain experts should be able to recreate arbitrarily high-fidelity models.

Why not write generative models?

Given these nice properties, one may ask why generative modeling is not the mainstay of machine learning methods. There two primary setbacks.

The first is that inference — computing the posterior distribution in Equation (1.1) — can only be performed exactly and efficiently in a very small set of models: simple models composed of conjugate distributions, or graphical models where each conditional distribution is either discrete or Gaussian. In all other cases, one must resort to approximation: exact inference in more complex models often requires evaluation of intractable integrals, or NP-hard computation requiring enumeration of exponentially large combinatorial spaces. There are vastly many different methods which have been employed for approximate inference in Bayesian models. Which methods perform well is often model-dependent, and even for methods which are thought to be generally applicable to a wide class of models, for each new generative model there are model-specific computations which need to be performed. The end result means implementing approximate computation as statistical software for a newly-posed generative model is typically a research task, not an engineering task.

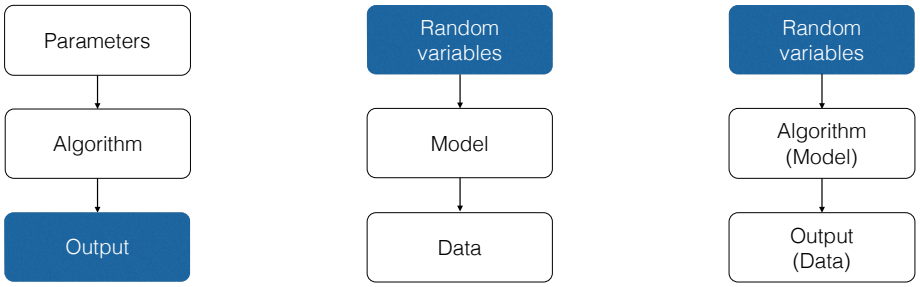
Second, specification of generative models is a significant practical hurdle. While a framework such as graphical models provides a manner for reasoning about assumptions in generative models, the representation is ultimately in terms of mathematical equations for different distribution types; this math then must be translated directly into model-specific inference code.

1.2 Probabilistic programming goals

Suppose you are convinced: you are a scientist, you have data, and you have a good idea how to describe the generative process as a probabilistic model $p(\mathbf{x}, \mathbf{y})$ which could have hypothetically generated the observed data. Where do you begin? How do you perform inference?

When it becomes time to sit down and write the inference algorithm code to sample from the posterior distribution $p(\mathbf{x}|\mathbf{y})$, somewhere much of the promise — ease of model understanding, and a focus on the generative process instead of the inverse — has been lost. To anyone not an expert in inference methods, the code describing the act of posterior approximation in a generative model is as opaque as a black-box model of $p(\mathbf{x}|\mathbf{y})$.

Probabilistic programming addresses this problem by taking the simulation aspect of generative models literally. A probabilistic programming language is a modeling language for generative models: a regular programming language, augmented with constructs for *sampling* random variables $\mathbf{x}$ from probability distributions, and constructs for *conditioning* the execution on values of other random variables $\mathbf{y}$. These two constructs, respectively, define the distributions $p(\mathbf{x})$ and $p(\mathbf{y}|\mathbf{x})$. The goal of probabilistic programming is to reduce the overall per-model coding burden. A current typical workflow involves first defining a generative model, typically as a set of equations describing a series of probability distributions over different random variables; then second, deciding what sort of inference algorithm may be appropriate for this model, a process which may involve model-specific derivations; and finally, hand-writing inference code, software which performs inference in this specific model, with this specific algorithm. In a probabilistic programming paradigm, the only code written is the definition of the generative model. A *compiler* or *inference engine* for a probabilistic programming language performs the inference automatically: given a probabilistic program — that is, source code for a stochastic simulator that defines a distribution $p(\mathbf{x}, \mathbf{y})$ — the engine produces code which then performs inference in this model. New inference techniques can be implemented as something akin to compiler optimizations.



(a) Forward Programming (b) Statistics (c) Probabilistic Programming

Figure 1.3: A comparison between goals and notation for (a) traditional programs, (b) statistical inference, and (c) probabilistic programs; the highlighted node in each column represents the quantity of interest. Program execution runs from top to bottom, whereas statistical inference runs from bottom to top. Probabilistic programming shares the same goal as machine learning and statistics — estimating posterior distributions of parameters, given observed data — but operates on models defined as algorithms with random choices, rather than on probability distributions directly. Based on a figure by Frank Wood.

This turns the traditional goal of computer programming on its head: we are used to thinking of a computer program as an algorithm which takes some inputs, and then returns some output. The salient bit is typically the output of the program — after all, we just went and wrote some complicated algorithm to generate that output, given our input! In contrast, in a probabilistic program, the algorithm is a simulator which functions as our probability model; we suppose that the data is hypothetical output from the program, and we are interested in running the program backwards to characterize the values or distributions over values of the inputs to the program. Figure 1.3 compares the goals of statistical inference and traditional "forward" computer programs.

A probabilistic programming engine performs Bayesian inference over computer program executions. The first immediate advantage of a probabilistic programming approach to machine learning is a separation of concerns: models are written independent of inference. The probabilistic programming language provides an abstraction boundary between model code and inference code, meaning someone designing a model can do so without regard for what inference algorithm will eventually be employed, and someone designing an inference algorithm can write it in such a manner that it can be automatically applied to any model written in the modeling language. A secondary but perhaps more important advantage is the role

such languages can play in the eventual democratization of access to machine learning and statistical methods. The number of people with training and background to write small computer programs which could potentially simulate their data is much larger than the number of people trained to define generative models mathematically, and then write custom inference code. In this sense, probabilistic programming opens up the possibility of using custom models to a vastly wider audience.

1.3 Challenges for probabilistic programming

Bayesian inference in general is a hard, unsolved problem, and one may wonder if we can implement a probabilistic programming system which performs inference generically over any model code a user may write. As it turns out, particle filtering, sequential Monte Carlo (SMC), and particle Markov chain Monte Carlo (PMCMC) methods for state-space models very naturally map onto the inference challenges defined by a large class of probabilistic programs, and we explore this relationship throughout this thesis.

Broadly, probabilistic programming languages take two distinct forms. "Higher-order" probabilistic programming languages such as Church [Goodman et al., 2008], Anglican [Wood et al., 2014], Probabilistic C [Paige and Wood, 2014], WebPPL [Goodman and Stuhlmüller, 2014], and Venture [Mansinghka et al., 2014], introduce no explicit constraints on the models which can be expressed, and are *universal* in the sense that they can be used to specify any computable distribution. In contrast, we will also look at "first-order" modeling languages which restrict the class of models that can be represented in exchange for being able to perform more efficient inference. Particularly well-known programming languages of this variety include BUGS [Spiegelhalter et al., 1996], which allows specifying a graphical model structure which can then be used for Gibbs sampling [Geman and Geman, 1984]; Stan [Stan Development Team, 2013], whose language restricts to differentiable models which admit Hamiltonian Monte Carlo [Neal, 2011] sampling; and Infer.NET [Minka et al., 2010], which restricts to exponential family models where efficient deterministic inference via expectation propagation [Minka, 2001] can be performed.

We will examine the tradeoffs made in language design, inference capabilities, and modeling expressivity in the following chapter.

This thesis hopes to outline and address several of the challenges towards designing and implementing usable probabilistic programming systems, and to inform the reader of the current state of the art and promising directions. We focus primarily on higher-order probabilistic languages; the space of models specifiable in these languages is a strict superset of the space of all probabilistic graphical models. Fundamental design challenges begin with defining an interface between the modeling language and inference algorithm; in Chapter 2 we will see that choices of inference algorithms and choices of modeling language design go hand-in-hand. In Chapter 3 we choose a simple fixed set of language constructs to define an explicit interface between models and inference, show how we can use this to define coherent probabilistic generative models, and outline a number of Monte Carlo inference algorithms in the literature which can be adapted to perform inference using this interface. The details of how a probabilistic programming language can be implemented in a computationally efficient manner by leveraging existing compiler and operating system capabilities, as well as explicit algorithms for inference, are outlined in Chapter 4. The relative advantages and weaknesses of these algorithms inspired the development of a novel asynchronous sequential Monte Carlo algorithm, which we present in Chapter 5.

In the subsequent following chapters, we then look to the future: can we design more computationally scalable approximate inference algorithms, which are still applicable to the broad class of models supported by our modeling languages? In Chapter 6, we design and implement a general-purpose variational Bayes algorithm, and demonstrate that it can be used for parameter estimation in complex structured models. Then in Chapter 7, we explore what additional computation can be performed "up front", prior to performing inference, when we have a generative model which we will like to use many times for inference in different data sets; we demonstrate an approach for offline learning of stochastic inverses which can be used to drastically accelerate importance sampling and sequential Monte Carlo

methods. We conclude by considering ways forward for inference and modeling in higher-order languages.

2

Language expressivity and model complexity

Probabilistic programming is a broad, umbrella term encompassing a variety of modeling languages, inferential methods, and goals. Differences in modeling language expressivity have consequences in terms of what probabilistic models are representable in the language; this in turn has consequences on which sorts of inference algorithms can be implemented and supported.

In this thesis we will make a distinction between *first-order* and *higher-order* probabilistic programming languages. The defining characteristic of a program written in a first-order probabilistic programming language is that its entire computation can be unrolled into a finite graph, by evaluating all possible branches of the program; in a higher-order language, this is not possible in general. In a functional programming language, including a construct which permits unbounded stochastic recursion (that is, functions whose recursion depth is itself random) yields a higher-order probabilistic programming language. For imperative or declarative programing languages which have a looping construct (e.g. a `for` loop), an equivalent condition is that the termination condition of the loop is itself a random variable. Higher-order probabilistic programming languages permit specification

of any computable probability distribution [Ackerman et al., 2011]; these are also sometimes referred to as *universal* probabilistic languages.

In terms of common statistical models, a Gaussian mixture model composed of a fixed, finite number of Gaussian distributions is a parametric model, and can be expressed in a first-order language. In contrast, a Dirichlet process infinite mixture of Gaussians [Rasmussen, 1999] is a Bayesian nonparametric model [Gershman and Blei, 2012] with an unbounded parameter space, and cannot be represented in a first-order probabilistic programming language without introducing an arbitrary truncation on the recursion depth — or equivalently, without introducing an upper bound on the number of components in the mixture. Similarly, in a first-order language we could not place a prior distribution on the number of clusters — for example, a mixture of K Gaussians where K is itself drawn from a Poisson distribution; the Poisson distribution has support over all positive integers. A higher-order language which permits unbounded recursion or loops with stochastic endpoints has no difficulty expressing such models.

2.1 Probabilistic programming language design

The differences between first-order and higher-order probabilistic programming languages largely can be attributed to design philosophy. Often, the more "restrictive" languages have been designed with the inference algorithm first: the language is defined to fit the model class for which the particular algorithm is efficient. In contrast, the higher-order languages have been designed "language first" — given a hypothetical Turing-complete modeling language, we can explore what inference algorithms can be designed to support it. If we wish to target this broadest-possible model family, what sort of inference methods are possible? Can we ever hope to make them efficient?

When designing a probabilistic programming system, the most challenging component is implementing an efficient inference operation for computing a representation of a conditional distribution. Performing automatic Bayesian inference in the very general context of arbitrary probabilistic programs poses a number

of challenges. Most modern state-of-the-art probabilistic programming systems use some form of Markov chain Monte Carlo (MCMC) inference to draw samples from the posterior distribution of some quantity of interest. First-order languages make an explicit trade-off between the modeling capabilities of the language, and the ease of performing inference.

All the probabilistic programming languages we will consider in this thesis include some fundamental concept of *random primitives*, or elementary random procedures, which allow drawing random variates from some externally defined set of distributions. These typically correspond to exponential family distributions commonly used as building blocks in larger probability models, such as the Bernoulli, Gaussian (normal), Poisson, and gamma distributions. Most languages provide a reasonably broad set of primitives, and provide tools for developers to implement additional random primitives. The primitives are characterized by the fact that they are implemented in some other language than the probabilistic programming language itself, typically in the host language or compilation target of the probabilistic programming language (for example, as low-level functions in C/C++).

Throughout this thesis, we will occasionally include illustrative example source code written in a Lisp variant which shares its general syntax with Clojure; in particular, code excerpts will all be valid expressions in the Anglican probabilistic programming language [Wood et al., 2014, Tolpin et al., 2016]. As a trivial example of an expression which can be only written in a higher-order language, consider the problem of drawing a sample from a geometric distribution, representing the number of failures before the first success in a sequence of Bernoulli trials. Most probabilistic programming languages that support discrete latent variables would already include a `geometric` random primitive, or if not, one could be created via tools for implementing new random primitives. However, what if we want to implement this not as a random primitive, but as a program in our probabilistic language? The direct way of sampling this is to repeatedly sample from a Bernoulli distribution until success, and return the total number of trials required. This

can be expressed by the following function `sample-geometric` in Anglican, which takes a single argument `p`:

```
(defm sample-geometric [p]
  (if (sample (flip p))
    0
    (+ 1 (sample-geometric p))))
```

Anglican differs from Clojure by providing a set of random primitives and a `sample` statement; the `flip` function is a constructor for a Bernoulli distribution over the values true and false; the expression `(flip p)` evaluates to a primitive distribution object from which we can draw a random variate. This function recurses to an arbitrary depth: although it halts with probability 1 for any $p \in (0, 1]$, one could not construct a finite graph outlining the computation for all possibilities. Even familiar simple distributions are sufficiently complex that we are unable to implement them in first-order probabilistic programming languages using only smaller building-blocks; in contrast, higher-order languages in theory only require a single primitive (for example, uniform on the $[0, 1]$ interval) to be able to draw samples from any such distribution: the expressivity of the probabilistic programming language is no less than that of the language in which the primitives are implemented.

2.2 First-order languages

Why develop a probabilistic programming language? One common motivation is reuse of inference code: a common modeling language can be used to avoid the time-consuming process of writing and debugging implementations of inference algorithms for every new model. Many of the more well-known probabilistic programming languages have been design for expressly this purpose. Given a high-quality, tested implementation of a popular inference algorithm, the design goal is to create a special-purpose language for defining precisely those models for which that inference algorithm is known to perform well. Coming from this direction, one starts from a simplest possible language, and gradually adds language complexity, prioritizing not breaking the inference engine: a good objective is to restrict the language to only permit writing models for which we can ensure we

will produce reasonable posterior estimates in finite time. We discuss three such languages now, looking at the relationship between language restrictions, modeling expressivity, and inference algorithms.

2.2.1 Bugs and Jags: Gibbs sampling

The declarative Bugs language [Spiegelhalter et al., 1996] is designed to be an efficient way of specifying a graphical model structure. The language exists as a way of compactly describing a probabilistic model which can be translated into a representation that the Bugs inference engine can use to automatically perform Gibbs sampling [Geman and Geman, 1984]. The Bugs language has strong restrictions which are required to ensure that only models compatible with the underlying inference engine can be expressed: there is no if–then–else construct, and any looping constructs require deterministic (rather than stochastic) endpoints. The language restrictions fall out naturally from the requirements of a model for Gibbs sampling to be possible: the model must have fixed finite-dimensional support, and a fixed dependency structure. For any given latent variable, we must be able to find (in graphical models terminology) its Markov blanket; that is, we must be able to statically identify which other latent variables directly influence (or are influenced) by it. Examples of models which cannot be expressed in the Bugs language include Bayesian nonparametric models [Gershman and Blei, 2012], which allow the number of instantiated latent variables to vary according to the data, as well as latent variable models such as the mixture of finite mixtures example described earlier, which yield a nondeterministic number of latent random variables as well as a nondeterministic dependency structure. This language has since been re-used with minor changes in Jags [Plummer, 2003], which also uses the compiled model as a basis for an alternative Gibbs sampler implementation.

For a deeper understanding of a first-order probabilistic programming language, it is helpful to imagine the compilation process. The compilation target for a model written in the Bugs language is literally a representation of a directed graphical model: once the model has been compiled from program code into the

appropriate data structure, then inference algorithms designed to operate on a graphical model representation no longer care in what language the model was originally specified. For example, the Biips probabilistic programming system [Todeschini et al., 2014] shares the Bugs modeling language, but instead uses sequential Monte Carlo and particle MCMC methods for inference [Doucet et al., 2001, Andrieu et al., 2010, Chopin et al., 2013]. However, the algorithms are all framed in terms of sequential Monte Carlo methods for inference in directed graphical models. Designing a sequential Monte Carlo algorithm for use on models specified in the Bugs language reduces to designing a sequential Monte Carlo algorithm for use on a graphical model data structure.

2.2.2 Stan: Hamiltonian Monte Carlo

The Stan project [Stan Development Team, 2013] aims to create a Bugs-like modeling environment that supports more scalable and computationally efficient inference, in particular targeting models for which gradient information can be computed. The default MCMC implementation is a variant of Hamiltonian Monte Carlo (HMC) [Neal, 2011], modified to choose its parameters adaptively and automatically [Hoffman and Gelman, 2014]. In some senses the language is more expressive than the Bugs language, but in others more restrictive; for example, it does not permit inference over discrete latent variables or combinatorial spaces, but it does allow unnormalized improper prior distributions for latent parameters. An if–then–else branching construct is provided, but requires the condition itself to be a function of the data, not of the latent variables.

The primary difference in Stan relative to Bugs is a change in the underlying representation of the model. The data structure necessary for HMC inference is not a graphical model, but rather a differentiable expression for the joint probability of all the latent variables. A model written in Stan compiles to a representation of the probability density, as well as a representation of its gradient via reverse-mode automatic differentiation [Carpenter et al., 2015]. These two functions are then repeatedly evaluated over the course of running the sampling algorithm. For

HMC, unlike for a Gibbs sampler, we do not need or want closed forms for specific full conditional distributions; nor do we explicitly need to know the dependency structure. This means deterministic transformations can be used more freely than in Bugs, without adverse effects on statistical efficiency, so long as those transforms are differentiable. The restriction on discrete latent variables also comes from the requirement that we compute the gradient of the joint probability of all the random variables in the model with respect to the latent variables — discrete latent variables may safely be observed, and models with discrete variables can also be expressed if the discrete variables can be marginalized out analytically. The Stan language exists as it does to support simple specification of models which are then appropriate for its highly-optimized HMC implementation, and as such it yields a very efficient sampler for the models which it supports.

2.2.3 Infer.NET: Message passing

Message passing methods provide an alternative to sampling methods used in Bugs and Stan. For discrete graphical models, or models containing only Gaussian latent variables and linear deterministic functions, belief propagation and other message passing algorithms provide an efficient means for computing posterior probabilities [Lauritzen and Spiegelhalter, 1988]. These algorithms take advantage of the fact that most probabilistic graphical models are not fully-connected graphs: the dependence structure is such that many latent variables only influence a handful of others. Message passing algorithms compute posterior probabilities and marginal likelihood estimates by combining estimates from neighboring cliques in a *factor graph*, a bipartite graph of latent variables and *factors*; the factors are functions of the adjacent latent variables which compute a component of the overall model probability, equivalent to the probability density functions in a directed model. For factor graphs which have no cycles and only discrete random variables then belief propagation (BP) [Lauritzen and Spiegelhalter, 1988] provides *exact* posterior marginal probabilities; in factor graphs with cycles, the algorithm is only approximate. When latent variables are neither discrete nor Gaussian, or if nonlinear deterministic functions are included

in a model with continuous latent variables, then further approximations are required: computing the messages to be passed along the local edges requires computing integration which is no longer possibly analytically. Expectation propagation (EP) approximates these integrals by projecting the messages into tractable exponential family distributions [Minka, 2001].

The Infer.NET modeling language [Minka et al., 2010] has been designed foremost as an interface to an underlying inference engine that performs expectation propagation. As such, the primitive distributions and deterministic functions available in the language are restricted to those for which the EP message passing algorithm can be performed: in particular, latent variable nodes are generally restricted to exponential families, and there are limitations on what deterministic functions are permitted. The compilation process for an Infer.NET program yields a factor graph representation; in contrast to the directed graphical models of Bugs, factor graphs are also capable of representing *undirected* graphical models, such as Markov random fields [Koller and Friedman, 2009].

Unlike the previous languages, branching on stochastic values is permitted by Infer.NET. The implementation uses the concept of *gates*, a modeling construct that permits consideration of multiple execution paths [Minka and Winn, 2009]. However, this does require the inference algorithm to consider all possible branches of every if statement, meaning expressing functions like `sample-geometric` is not possible. Moreover, computing the messages for EP requires implementing every deterministic operation as a factor, meaning it needs to integrate over the deterministic operation; adding new deterministic factors means deriving closed-form expressions for these integrals, performing an (expensive) sampling algorithm as an inner loop, or learning approximations to the messages [Heess et al., 2013, Eslami et al., 2014, Jitkrittum et al., 2015].

2.3 Higher-order languages

A common thread to the languages above is that the inference method came first: creating such a probabilistic programming language begins by choosing an inference

algorithm which can be considered fairly general-purpose, and then designing a (possibly restrictive) modeling language which allows the user to specify probabilistic models which can be handed over to such an inference engine. This modeling language is typically a "new" purpose-designed language, with its particular features tailored to align with the requirements of the underlying target inference engine; the language forms an abstraction barrier between model and inference, with the choice of inference engine showing its influence in the language design and structure.

The alternative approach to probabilistic programming we will explore through the rest of this thesis can be thought of as "model first", or "language first". In this philosophy, instead of beginning with an inference engine we know to be efficient and then asking how to build the language to support that inference engine, we begin with a Turing-complete general-purpose language we know can express any model we may ever wish to write, and then ask, in this context, whether it is possible to perform inference at all, let alone efficiently. To design a probabilistic programming language in this style, one typically begins from a standard, non-probabilistic language and augments it with primitive operations for sampling from common elementary probability distributions such as the Bernoulli, Poisson, or Gaussian distributions, and a primitive operation for conditioning the execution of the program on certain settings of random variables. Every possible execution of a program in such a probabilistic programming language can be assigned a probability by considering the values of the sampled and observed random variables.

2.3.1 Execution-based semantics

Probabilistic programming languages in this form descend broadly from the Church language [Goodman et al., 2008], which introduces what we will refer to as execution-based semantics. Models in Church are specified in a LISP-like language as stochastic programs with random elements, with program evaluation viewed as sampling. Conditioning in Church is performed using a `query` function, which takes as its arguments two LISP expressions: one denoting a generative model, and one denoting a Boolean predicate which evaluates to true or false; these expressions are evaluated

in a shared environment. Executing `query` returns a representation of the posterior distribution over evaluations of the model expression, conditioned on the event that the predicate expression evaluates to `true`. The simplest sort of implementation of `query` is a rejection sampler: execute the program (model) code, and then evaluate the predicate, retaining as samples only those executions for which the predicate holds. This obviously will not work in general, and a simple Metropolis-Hastings algorithm is also suggested; we will review this algorithm in the following chapter.

Subsequent developments in probabilistic programming systems have refined both modeling language and inference techniques. Compilation of model source code combined with careful bookkeeping for individual random choices [Wingate et al., 2011], allows for much more efficient implementations of `query`, both in constant-time speedup by removing interpreter overhead and in statistical efficiency by improving the Metropolis-Hastings proposal kernels.

Anglican [Wood et al., 2014, Tolpin et al., 2016] and Probabilistic C [Paige and Wood, 2014] explore the use of sequential Monte Carlo and particle filtering methods for performing inference in probabilistic programs; they also exhibit efficient compilation strategies, which we describe in Chapters 3 and 4. These languages also depart from the `query` semantics, instead offering an `observe` or `factor` function which can be freely interspersed with model code. Instead of conditioning on an event defined by a specific predicate, the `observe` or `factor` statements directly modify the probability of a given execution. Under these such semantics, evaluating program model code returns not only a value, but also a "weight" as defined by the `observe` or `factor` expressions. The semantics of this is that an expression such as `(observe (normal 0 1) x)` in Anglican would mean "condition on the event that a value drawn from a standard normal distribution is equal to `x`"; as a Boolean predicate, an equivalent formulation would be `(= (sample (normal 0 1)) x)`. Alternatively, a `factor` function takes as its argument a single numeric value, which can be interpreted as a log probability; in this example, its argument would be the log density function of a standard normal evaluated at `x`. The change from a single conditioning predicate to a set of `observe` statements does not fundamentally

change the expressivity class of the language, but instead makes models amenable to a wider variety of inference methods, by allowing algorithms to take advantage of the incremental nature in which conditioning statements arise during execution. WebPPL [Goodman and Stuhlmüller, 2014] is a compiled probabilistic programming language consisting of a pure functional subset of javascript, which has similar semantics and as an added bonus can even be run from within a web browser; it also includes further advances towards constructing efficient Metropolis-Hastings proposals [Ritchie et al., 2016], as well as sequential Monte Carlo methods.

The key aspect of this execution-based approach to probabilistic programming language design is that evaluation of the program is equivalent to drawing a sample from a probability distribution. Introducing a `query` operation or `factor` expressions then (semantically) allows conditioning. In languages such as Anglican, Probabilistic C, and WebPPL, the `factor` or `observe` statements can be used directly to define an unnormalized probability distribution over program executions; we will do so in the following chapter.

2.3.2 Modeling capabilities

Higher-order languages greatly ease specification of models which are otherwise quite painful to formalize. An illustrative example comes in modeling *theory of mind*, the act by which actors reason about the beliefs and actions of other actors, which can be expressed concisely via mutually recursive functions and nested conditioning statements [Stuhlmüller and Goodman, 2014].

In an example of a coordination game [Schelling, 1960], two actors Amy and Bob would like to meet up for a drink. They could meet either at the pub or the café, but have no way of communicating ahead of time which to choose. Suppose Amy has a mild preference for going to the pub. If each decides where to go independently, without taking each others preferences or reasoning into account, there is a large chance they will not successfully meet up.

However, suppose Bob knows that Amy would probably prefer the pub. Knowing this and wanting to meet up with Amy, he may be more likely to choose the pub

as well. Amy, in turn, could choose based not just on her preferences, but based on where she thinks Bob might go: if she knows that Bob knows that she prefers the pub, and believes Bob is trying to meet her, then her act of reasoning about what decision Bob would make would then *increase* the probability of her choosing the pub over the café from her baseline preference, and increase the probability of them meeting up successfully. This recursion can continue indefinitely.

Stuhlmüller and Goodman [2014] express a model much like this with two mutually recursive Church functions, one simulating Amy and one simulating Bob, each fewer than 6 lines of code. Both of these simulation functions call one another, reflecting the nested reasoning process. As the recursion depth of their nested reasoning increases, they find that the probability of the two meeting up successfully approaches 1; literally thinking "deeper" leads to a better overall outcome. Even the simple behaviors of these two actors are difficult to specify in any manner other than simulation.

2.3.3 Inference challenges

A fundamental challenge to inference in higher-order languages lies in the behavior when the program encounters control flow statements. As programmers, we are used to writing statements of the form "if A, then B; otherwise, C". In most standard programming language implementations, the semantics of this are such that *only one branch is ever evaluated* on a single execution of the program: we do not run the code in both B and C. This sort of short-circuiting means that we can safely write program code such as the `sample-geometric` function at the top of this chapter, without worrying that the program essentially defines an infinite loop with a probabilistic stopping condition. Efficient implementations of algorithms such as Gibbs sampling in Bugs and EP in Infer.NET require being able to statically determine the dependency structure of the random variables in the model. For functions such as `sample-geometric`, this is not straightforward.

Even if the dependency structure were known, and the number of random variables were fixed, allowing arbitrary program expressions still introduces challenges

for HMC and EP. Deterministic program code is not necessarily differentiable, and even if it is, if the language admits any sort of foreign function interface then taking gradients automatically through code written in an external language is typically impossible, precluding efficient HMC implementations. Similarly, to compute EP messages for probability models which include black-box deterministic functions one must resort to either importance sampling the messages, or training model-specific regressors.

Beyond challenges for specific inference algorithms, there are meta-challenges for implementing inference in higher-order languages: since we will not compile the program into a familiar graphical model representation, it is an open question what underlying representation or abstraction is appropriate for our models. That is: how should we design the interface at the boundary between modeling code and inference code? We suggest an interface and outline approaches to inference in the following chapter.

3

Inference over partial program executions

In this chapter we make explicit what is meant by performing Bayesian inference over the possible executions of computer programs. The first step of this is defining how to assign a probability to a program execution. In doing so, we relate the language constructs added to enable probabilistic programming to our typical understanding of probability models. We then outline some basic methods for performing inference. The discussion here particularly focuses on the issues specific to probabilistic programs; for a more general introduction to Bayesian inference in machine learning, refer to e.g. Bishop [2006].

Bayesian statistical approaches typically pose a model by specification of a *prior* distribution $p(\mathbf{x})$ over a set of latent variables $\mathbf{x}$ and a *likelihood* function or data distribution $p(\mathbf{y}|\mathbf{x})$ defining the conditional distribution of the data given the latent variables. Inferential goals involve constructing approximations to the *posterior* distribution $p(\mathbf{x}|\mathbf{y})$ of the latent variables given observed data, following Bayes' rule

$$p(\mathbf{x}|\mathbf{y}) = \frac{p(\mathbf{x})p(\mathbf{y}|\mathbf{x})}{p(\mathbf{y})} \tag{3.1}$$

where the posterior in general has no closed-form or analytic representation. We will refer to $\pi(\mathbf{x}) \equiv p(\mathbf{x}|\mathbf{y})$ as the *target density*; we also refer to the joint density taken as a function of $\mathbf{x}$ alone as the *unnormalized* target density $\gamma(\mathbf{x}) \equiv p(\mathbf{x}, \mathbf{y})$. Typically

we are not explicitly interested in approximating this conditional density $p(\mathbf{x}|\mathbf{y})$ itself, but rather computing posterior expectations of some test function $\psi(\mathbf{x})$, with

$$\mathbb{E}[\psi] = \int \psi(\mathbf{x})p(\mathbf{x}|\mathbf{y})\mathrm{d}\mathbf{x}. \tag{3.2}$$

This test function ψ may be the identity function, in which case this is simply the posterior expectation of the latent variables; it also may be chosen to compute higher-order moments, tail probabilities, or domain-specific loss functions. Throughout this thesis, when writing expectations we will generally be referring to expectations with respect to the posterior distribution $p(\mathbf{x}|\mathbf{y})$. When we mean an expectation with respect to a different distribution, or when the distribution in question is ambiguous, we will specify the particular distribution with a subscript, e.g. as $\mathbb{E}_{p(\mathbf{x}|\mathbf{y})}[\psi]$.

A fundamental aspect of generative models is that they allow simulation: we can simulate from $p(\mathbf{x})$ to sample a setting of the latent variables, and then in turn simulate from $p(\mathbf{y}|\mathbf{x})$ to generate synthetic data. A "good" generative model is one which produces synthetic data which resembles actual data which we have collected. With this in mind, we first consider what primitives we need to add to a programming language to create a "probabilistic" variant.

3.1 An interface for inference engines

A procedure in a deterministic (or "traditional") computer program performs some computation and then returns an output. A probabilistic programming language augments the deterministic programming language with two constructs for interacting with random variables [Gordon et al., 2014]:

- *sampling* a random value according to some distribution; and
- *conditioning* on the value of an observed random variable.

We will demonstrate this by introducing and implementing explicit keywords for `sample` and `observe` into an existing programming language.

In this discussion, we restrict ourselves to considering programs whose deterministic functions are *pure* — a pure function always returns the same output when

called on the same input. Such a requirement suggests functional programming languages are a natural fit for conversion to a probabilistic programming language. This is likely true at at least a conceptual level, and in this chapter we will focus on examples written in Anglican; in the following chapter we will revisit in the context of an imperative language. Regardless, in the spirit of functional programming we will consider every program to be an expression which has a value; equivalently, we could think of every program as being a function which when called has some return value. We will refer to this value as the *output* of the program.

A *side effect* in computer science, in this context, is the modification of some external state during the course of a program expression being evaluated. Typical side effects include printing content to the screen, editing a file on disk, or modifying a global variable. Requiring our deterministic code be composed of pure functions implies that the programs we will consider are generally free of side effects; in particular, their correct execution cannot rely on modification of global variables or external state, or they would not be pure. Our approach is to implement an engine for Bayesian inference in probabilistic programs through clever introduction of side effects into otherwise pure expressions.

Informally, one can imagine "running" a probabilistic program as an operation similar to running a deterministic program. Suppose we have some set of random primitives — distributions such as Gaussian, uniform, binomial, etc. — from which we can sample new random values, or condition on observed values. In Anglican, one can draw a value according to a standard normal distribution by calling `(sample (normal 0 1))` and can condition on it taking the value 1.1 with `(observe (normal 0 1) 1.1)`. The behavior of `sample` and `observe` within a single execution of a program is very straightforward: the `sample` statement takes a distribution as its argument and returns a value; the `observe` statement takes a distribution and a value as its arguments and returns nothing.

First, consider programs which only contain standard deterministic code, and `sample` statements (i.e., without `observe`). If we were to implement `sample` as a function which draws a random variate from the supplied distribution, then we can

execute this code as just a standard program; every time we run the program, we obtain a new set of random variates returned by the various `sample` statements, and a new return value for the overall program. Introducing the (pseudo-)random number generator into the language means that the probabilistic code is no longer pure; calling it multiple times yields multiple results. (Typically pseudo-random numbers are generated via a stateful computation, using algorithms such as the Mersenne twister [Matsumoto and Nishimura, 1998]; in this sense, drawing a random number modifies global state.) However, we know that the variation in the output of the program is due *only* to these `sample` statements. Furthermore, and somewhat crucially, the probabilistic program execution is completely deterministic given the values of the sampled latent variables during program execution — that is, given a sequence of sampled values $x_1, x_2, \ldots$ then the output of this probabilistic program is once again fully deterministic.

We suggest a separation between the deterministic program code (which we will refer to as $\mathcal{P}$) and the randomness introduced by `sample` by considering the probabilistic program to be executed in the context of some backend $\mathcal{B}$, and introduce the concept of a *program execution trace* which enumerates all random choices made during the course of executing the program. The pseudo-random number generator state, as well as any internal state used by the inference algorithm, will be encapsulated by the backend $\mathcal{B}$.

We will assume the program $\mathcal{P}$ can be paused mid-execution, and stored as a *continuation* object which represents the entire current state of the computation and which can be later resumed. Most of these algorithms will require multiple execution of continuations: that is, given a partially executed "copy" of the program, we can resume execution from that point one or more times. Details of how such computations can be implemented in practice are postponed to the subsequent Chapter 4.

We suppose that the program $\mathcal{P}$ is interrupted at each `sample` or `observe` statement (a "checkpoint"), at which point control is handed to the (stateful) backend. The program can then be resumed by calling $\mathcal{P}()$ again, possibly passing

it an argument. At a high level, the interaction between a backend $\mathcal{B}$ and the program $\mathcal{P}$ can be summarized as follows, for a single execution of $\mathcal{P}$:

1. Initialize by beginning execution of the program, calling $\mathcal{P}()$.

2. When execution of $\mathcal{P}$ encounters a `sample` statement, `observe` statement, or the end of the program, it yields control to the backend, and one of the three following operations occurs, depending on the case:

   - `sample`: $\mathcal{P}$ passes to $\mathcal{B}$ a tuple (f, θ) consisting of a distribution f and a parameter vector θ; the backend produces a value x. It then returns control to the program by calling the continuation $\mathcal{P}(x)$, providing the value x as an input to the rest of the program.
   - `observe`: $\mathcal{P}$ passes to $\mathcal{B}$ a tuple (g, ϕ, y) consisting of a distribution g, a parameter vector ϕ, and a observed value y. We then call $\mathcal{P}()$ with no arguments, continuing execution.
   - If $\mathcal{P}$ has terminated, it returns a value ψ to the backend, which can be any arbitrary (deterministic) function of the program trace.

This characterization of the interface between probabilistic programming language model code and an inference backend is derived from that developed in both Tolpin, van de Meent, Paige, and Wood [2015] and van de Meent, Paige, Tolpin, and Wood [2016].

Two things require further explanation. First, we would clarify that the "parameter vectors" of θ or ϕ are composed of the fully evaluated arguments to the distribution object; i.e. if the source code expression involved evaluated to `(sample (normal (+ 2 2) 5))`, at the sample checkpoint this would correspond to a distribution where f is the probability density functions of a normal distribution, and θ contains its evaluated arguments of mean 4 and standard deviation 5. Second, note that while the most "natural" action we would imagine the backend $\mathcal{B}$ taking at this juncture would be to draw a sample from the distribution $\mathcal{N}(4, 5)$, at this point we have not specified the behavior of the backend at the checkpoint — we

merely require that it always return some value. Different backends will implement different inference algorithms, and will have differing behavior at each checkpoint.

Also, although the interface for `sample` and `observe` describes the interaction between the program and the backend during a *single* execution, inference will require repeated execution of $\mathcal{P}$. This takes place in the context of a backend-specific outer loop, which repeatedly calls $\mathcal{P}$ and processes the returned values ψ.

3.2 Probability of a program execution

Suppose after a single execution of a probabilistic program in this manner, we encounter N `observe` statements and M `sample` statements. This yields sequences of tuples $\{(g_i, \phi_i, y_i)\}_{i=1}^{N}$ corresponding to the `observe` statements, and $\{(f_j, \theta_j)\}_{j=1}^{M}$ corresponding to the `sample` statements, with the associated sequence of sampled values (i.e. the program execution trace) $\mathbf{x} = \{x_j\}_{j=1}^{M}$. The probability of this program execution can be defined, up to an unknown normalizing constant, as a product of all random choices $\mathbf{x}$ and all observed values $\mathbf{y}$, with

$$\gamma(\mathbf{x}) \triangleq p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} g_i(y_i|\phi_i) \prod_{j=1}^{M} f_j(x_j|\theta_j). \tag{3.3}$$

Note that this ordering of random variables, as well as the cardinalities M and N, are not necessarily identical across different runs of the program. Despite that, we can still score in this manner any *single* execution: every set of sampled latent variables $\mathbf{x}$ uniquely defines a deterministic execution path of the program, and we define the probability of this program execution to be the product of the probability of all its random choices.

Obscured by the notation above is the dependency structure induced by the probabilistic program $\mathcal{P}$. Each parameter vector ϕ_i and θ_j are themselves deterministic functions of (potentially) every previous random choice in the program, as are the density functions g_i and f_j. Let n_i denote the total number of random values sampled prior to the i^{th} `observe` statement and the bold, subscripted value $\bar{\mathbf{x}}_j = x_1 \times \cdots \times x_j$ with a bar above it denote the concatenation of the first j sampled values in a partial program execution trace (with $\times$ a concatenation

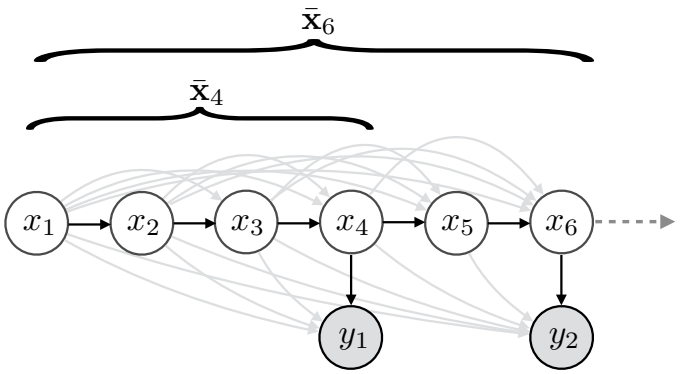


Figure 3.1: The random variables, and their dependencies, in a single program execution. When considering individual random choices $x_1, x_2, \ldots$, the dependencies are very long-range, potentially depending on the values of any and all previous random choices. However, if instead we define the model in terms of the global machine state, then we have a Markov model over $\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2, \ldots$.

operator). A version of Equation (3.3) which explicitly represents the dependency structure would have the form

$$\gamma(\mathbf{x}) = p(\mathbf{x}, \mathbf{y}) = \prod_{i=1}^{N} p(y_i|\bar{\mathbf{x}}_{n_i}) \prod_{j=1}^{M} p(x_j|\bar{\mathbf{x}}_{j-1}). \tag{3.4}$$

Here we see that the conditional distribution of each random choice x_j (corresponding to each `sample` statement) can depend on all previously sampled values $\bar{\mathbf{x}}_{j-1}$, and the conditional distribution of each observed random variable y_i (corresponding to each `observe` statement) can depend on all random choices $\bar{\mathbf{x}}_{n_i}$ which occur prior to the i^{th} `observe` statement. The nature of these dependences is such that not only can the parameters to the distributions change, but so can the type of distribution itself, and the domain of each x_j. Consider the following program excerpt:

```
(sample
  (if (sample (flip 0.5))
    (normal 0 1)
    (poisson 10)))
```

On every execution, this program contains exactly two `sample` checkpoints. The first of these, `(sample (flip 0.5))` has a deterministic distribution type (`flip`) and a deterministic parameter 0.5. However, a sample from `flip` can return either true or false; that is, the first latent random variable x_1 can take two possible values. At the second `sample` checkpoint, both the distribution type f (`normal` or

`poisson`) as well as the parameters θ ($[0, 1]$, or $[10]$) depend on the value of the first random variable. Note, however, that the distribution type and parameters are still deterministic given a setting of the latent random variables. We restrict the primitive distributions types in the language (here, we see `normal`, `poisson`, and `flip`) to be distributions for which we know, given their parameters, both how to sample from the conditional distribution $f_j(\cdot|\theta_j)$ and how to evaluate any $g_i(y_i|\phi_i)$. More complex distributions are created by combinations of primitive distributions and deterministic code. Figure 3.1 summarizes the notation for the individual random choices $x_1, x_2, \ldots$, the sequence of global agglomerative state variables $\bar{\mathbf{x}}_1, \bar{\mathbf{x}}_2, \ldots$, and the overall dependency structure.

In general, characterizing distributions such as $p(x_j|\bar{\mathbf{x}}_{j-1})$ may be quite difficult, particularly if the transformation from $\bar{\mathbf{x}}_{j-1}$ to θ is not a one-to-one function or its Jacobian is difficult to compute, as a change of variables is involved. As the parameters to each of these primitive distributions are defined by the execution of the program up to that point, changing even a single random choice can lead to changes in control flow which can change the total number of random choices in the program, as well as the types and parameters; effectively, this means the support of the latent variable space is difficult to fully characterize in any manner besides simulation. In the simple example above, for example, as the `normal` distribution has support over real numbers and the `poisson` distribution has support over nonnegative integers, the sequence of random choices $\mathbf{x} = [\mathsf{true}, 1]$ is in the support, as are $[\mathsf{false}, 1]$ and $[\mathsf{true}, -1]$; however, $[\mathsf{false}, -1]$ is not. For more complex programs, static determination of the support of the latent space can be arbitrarily hard. Nice pathological examples involve "trapdoor" hash functions used in cryptography, where pre-image computations are effectively impossible. For example, the `md5` function maps any input to a 128 bit message digest, and is designed so that small changes in its input yield large changes in output. Consider a program similar to the above, but with an intractable condition in the branching statement:

```
(let [z (sample (normal 0 1))
      h (md5 z)]
  ;; Is the first letter in the hashed string an "a"?
```

```
(if (= (get h 0) "a")
  (sample (normal 0 1))
  (sample (poisson 10)))
```

Answering a question such as "is $\mathbf{x} = [2.2, -1.0]$ in the support of this program" is effectively impossible without simulation: executing the program and evaluating `(md5 2.2)` and seeing whether the initial character is an `a`.

So far we have only defined a means of evaluating an unnormalized probability of a program trace. For posterior inference in a program with `sample` and `observe` statements, we wish to approximate expectations with respect to the normalized posterior probability distribution over program traces, which can be defined as

$$\pi(\mathbf{x}) \triangleq p(\mathbf{x}|\mathbf{y}) = \frac{\gamma(\mathbf{x})}{Z}, \qquad Z = p(\mathbf{y}) = \int \gamma(\mathbf{x}) d\mathbf{x}. \tag{3.5}$$

The normalizing constant Z is found by integrating over all possible program execution traces; the domain of integration here is the (potentially uncomputable) support of $\mathbf{x}$ described above, and so estimating this normalization will be challenging in general and will require approximation.

The program output, or the return value of the program, is also a deterministic function given the sampled values $\mathbf{x}$ in a program execution trace; we denote the output as $\psi(\mathbf{x})$. This allows us, in theory, to use the posterior distribution over traces $\pi(\mathbf{x})$ to characterize the distribution over program outputs given the observations $\mathbf{y}$; for example, the posterior mean can be found by

$$\mathbb{E}[\psi(\mathbf{x})] = \int \psi(\mathbf{x}) p(\mathbf{x}|\mathbf{y}) d\mathbf{x} = \int \psi(\mathbf{x}) \pi(\mathbf{x}) d\mathbf{x} = \frac{1}{Z} \int \psi(\mathbf{x}) \gamma(\mathbf{x}) d\mathbf{x}. \tag{3.6}$$

This characterization of a probabilistic program allows us to define models literally as source code for stochastic simulation, with the random elements controlled by `sample` and `observe` statements. The generative procedure uses `sample` to create random variables; synthetic data sets could be created simply by replacing any `observe` statement with `sample`, without changing the unnormalized distribution $p(\mathbf{x}, \mathbf{y})$.

Thus far we have assumed that latent variables $\mathbf{x}$ can be simulated easily, and the complications arise through conditioning on data $\mathbf{y}$. However, while (for example) any *directed* graphical model can be written naturally as a simulator, the same is

not true for undirected graphical models (e.g. Markov random fields). Undirected graphical models can be expressed in this framework by introducing additional pseudo-observations y_i on top of a generative process which simulates values of all latent variables $\mathbf{x}$: this can typically be achieved by selecting a subset of the factors in the factor graph to be used for simulation, with `observe` statements which represent not data, but rather the factors in the desired distribution on $\mathbf{x}$ not accounted for by the sampling process. For examples of sequential Monte Carlo applied to sampling from undirected graphical models, see Naesseth et al. [2014].

3.3 A first simulation-based inference engine

In Section 3.1 we defined the interface for interaction between a probabilistic program execution and an inference backend. However, in addition to leaving out any details of implementation, we did not describe what an inference backend ultimately produces: what sort of artifact represents a posterior distribution? At an abstract level, we can think of inference as an operation which takes a probability model and data as inputs — that is, a joint distribution $p(\mathbf{x}, \mathbf{y})$ and a particular setting of the random variables $\mathbf{y}$ — and returns some representation of the posterior distribution $p(\mathbf{x}|\mathbf{y})$. Program source code is a single concrete representation of the abstract concept of a probabilistic model $p(\mathbf{x}, \mathbf{y})$. In this chapter we consider inference algorithms which produce, as a posterior representation, a sequence of weighted pairs (ψ^k, w^k) for $k = 1, 2, \ldots$. Each weight $w^k \in \mathbb{R}^+$ is some positive value associated with the latent variable values $\mathbf{x}^k$, and the evaluated return expression $\psi^k \equiv \psi(\mathbf{x}^k)$; taking the first K weights, these can be normalized to sum to one as

$$W^k = \frac{w^k}{\sum_{\ell=1}^{K} w^\ell} \tag{3.7}$$

and we approximate expectations of the output values as weighted sum over sampled values

$$\mathbb{E}[\psi(\mathbf{x})] \approx \sum_{k=1}^{K} W^k \psi(\mathbf{x}^k). \tag{3.8}$$

A sequence of weighted values is just one possible (though convenient) posterior representation; alternatives are considered in Chapter 6. Broadly, a weighted set of evaluation points is a generic representation which encompasses output of MCMC algorithms, weighted sampling algorithms such as importance sampling and SMC, as well as alternative point selection schemes such as kernel herding [Chen et al., 2012] and sequential Bayesian quadrature [Huszár and Duvenaud, 2012].

3.3.1 Importance sampling

Importance sampling provides a baseline approach to approximating an intractable target distribution with samples drawn from a different, simpler distribution. We will look at a first, simple simulation-based inference engine based on likelihood weighting. This will not be acceptable as a general purpose algorithm for performing inference, but is easily understood and provides intuition for more complex approaches.

Given the definition of the probability of a program execution above, we can guarantee that we can always evaluate the unnormalized target $\gamma(\mathbf{x})$ pointwise, by forward execution of the program with fixed settings of latent random variables; we define this probability to be zero for $\mathbf{x}$ not in the support of the program. However, the posterior distribution $\pi(\mathbf{x})$ is intractable, in the sense that we are neither able to sample from it, evaluate it, nor compute its normalization Z directly.

Importance sampling proceeds by drawing candidate values from a different, tractable distribution $q(\mathbf{x})$, which is known variously as the *proposal distribution* or the *importance sampling distribution*. The fundamental idea is that posterior expectations of ψ can be expressed as expectations over $q(\mathbf{x})$, with

$$\mathbb{E}_{\pi}\Big[\psi(\mathbf{x})\Big] = \int \psi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} = \int \psi(\mathbf{x})\frac{\pi(\mathbf{x})}{q(\mathbf{x})}q(\mathbf{x})d\mathbf{x} = \mathbb{E}_{q}\left[\psi(\mathbf{x})\frac{\pi(\mathbf{x})}{q(\mathbf{x})}\right]. \tag{3.9}$$

A very simple way of proposing trace samples $\mathbf{x}^k$ is to run K independent copies of the program $\mathcal{P}$, yielding K traces, each sampled according to some sequence of M^k different densities $\{f_j^k, \theta_j^k\}_{j=1}^{M^k}$. To be clear, this means running each copy of the program entirely independently with the backend sampling proposing values of

x_j^k directly via the prior $p(\mathbf{x})$; each particle is simulated identically from

$$q(\mathbf{x}^k) = \prod_{j=1}^{M^k} f_j(x_j^k|\theta_j^k). \tag{3.10}$$

For each of these K traces $\mathbf{x}^k$, we can compute an associated unnormalized weight $w(\mathbf{x}^k)$ as

$$w(\mathbf{x}^k) = \frac{\gamma(\mathbf{x}^k)}{q(\mathbf{x}^k)} = \prod_{i=1}^{N^k} g_i^k(y_i^k|\phi_i^k) \tag{3.11}$$

where N^k denotes the number of `observe` statements yielding tuples $\{(g_i^k, \phi_i^k, y_i^k)\}_{i=1}^{N^k}$ for each of the K traces. This allows us to estimate integrals over $\psi(\mathbf{x})$ with respect to the unnormalized measure $\gamma(\mathbf{x})$ as

$$\bar{\psi}_K = \frac{1}{K}\sum_{k=1}^{K} w(\mathbf{x}^k)\psi(\mathbf{x}^k) \approx \int \psi(\mathbf{x})\gamma(\mathbf{x})d\mathbf{x}; \tag{3.12}$$

this is an unbiased estimate, as we have

$$\begin{aligned}
\mathbb{E}\left[\frac{1}{K}\sum_{k=1}^{K} w(\mathbf{x}^k)\psi(\mathbf{x}^k)\right] &= \frac{1}{K}\sum_{k=1}^{K}\int \psi(\mathbf{x}^k)\left[\prod_{i=1}^{N^k} g_i^k(y_i^k|\phi_i^k)\prod_{j=1}^{M^k} f_j^k(x_j^k|\theta_j^k)\right] dx_1 \ldots x_{M^k} \\
&= \frac{1}{K}\sum_{k=1}^{K}\int \psi(\mathbf{x}^k)\gamma(\mathbf{x}^k)d\mathbf{x}^k \\
&= \int \psi(\mathbf{x})\gamma(\mathbf{x})d\mathbf{x}
\end{aligned} \tag{3.13}$$

where the expectation in the left-hand side of Equation (3.13) is over all random choices made during the execution of the program. To estimate integrals with respect to the normalized measure $\pi(\mathbf{x})$, note that

$$\mathbb{E}[\psi(\mathbf{x})] = \int \psi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x} = \frac{1}{Z}\int \psi(\mathbf{x})\gamma(\mathbf{x})d\mathbf{x} = \frac{\int \psi(\mathbf{x})\gamma(\mathbf{x})d\mathbf{x}}{\int \gamma(\mathbf{x})d\mathbf{x}}. \tag{3.14}$$

We can estimate this by computing the ratio of separate estimators for the numerator and the denominator. In particular, the normalizing constant Z is also an expectation over $\gamma(\mathbf{x})$, which we can estimate as

$$\bar{Z}_K = \frac{1}{K}\sum_{k=1}^{K} w(\mathbf{x}^k) \approx \int \gamma(\mathbf{x})d\mathbf{x}. \tag{3.15}$$

Taken together, we can define an estimator with respect to $\pi(\mathbf{x})$ as

$$\widehat{\psi}_K = \frac{\sum_{k=1}^K w(\mathbf{x}^k)\psi(\mathbf{x}^k)}{\sum_{k=1}^K w(\mathbf{x}^k)} = \frac{\bar{\psi}_K}{\bar{Z}_K}. \tag{3.16}$$

This estimator can be compactly represented using normalized weights,

$$\mathbb{E}_\pi[\psi(\mathbf{x})] \approx \widehat{\psi}_K = \sum_{k=1}^K W^k \psi(\mathbf{x}^k); \qquad W^k = \frac{w(\mathbf{x}^k)}{\sum_{\ell=1}^K w(\mathbf{x}^\ell)}. \tag{3.17}$$

This inference algorithm as implemented in a probabilistic program returns the sequence of $(\psi(\mathbf{x}^k), w(\mathbf{x}^k))$ values from which we can later reconstruct $\widehat{\psi}_K$. Note that this estimator is biased for any finite K, but the bias drops off as order $O(1/K)$ [Liu, 2008]. Since each $\mathbf{x}^k$ is simulated independently, we have from the strong law of large numbers that [Owen, 2013]

$$\Pr\left(\lim_{K\to\infty} \widehat{\psi}_K = \int \psi(\mathbf{x})\pi(\mathbf{x})d\mathbf{x}\right) = 1. \tag{3.18}$$

While this will not be an efficient method for high-dimensional program traces, it illustrates a baseline "guess-and-check" method that can be used on effectively any program; "guess" by running the program forward, drawing a random value at each `sample` statement, and "check", probabilistically, by accumulating the probabilities at each `observe`.

Note that for this method to be effective, the "check" probabilities at each `observe` should not be too tightly peaked — in the limit of "hard" `observe` statements, where the conditional distributions $g_i(y_i|\phi_i)$ are point masses on single values (with positive probability only when, for example, $y_i = \phi_i$ exactly), then this algorithm reduces to a rejection sampling procedure which will fail completely on continuous-valued latent variables. The burden is on the programmer to avoid writing models which have hard constraints that are difficult to ever satisfy through simulation. For inference in models which are truly likelihood free — where the simulator is supposed to generate the data exactly — then rather than observing the data itself directly, the `observe` statement can be used to match summary statistics of the simulated data to summary statistics of the actual data. This approach is generally known as approximate Bayesian computation (ABC); for examples and discussion see e.g. Marin et al. [2012].

There is great flexibility in choice of proposal distribution $q(\mathbf{x})$, with the only firm requirements being (a) that we can evaluate $q(\mathbf{x})$ pointwise (at least, up to a constant), and (b) for all $\mathbf{x}$ such that $p(\mathbf{x}|\mathbf{y}) > 0$, then $q(\mathbf{x}) > 0$. The ideal (or optimal) proposal for estimating marginal distributions would be a data-driven proposal which samples directly from the posterior $p(\mathbf{x}|\mathbf{y})$; note that the ideal proposal varies when estimating expectations of different particular functions $\psi(\mathbf{x})$. Proposing from the prior distribution $p(\mathbf{x})$ itself, as we did here, is a natural and always-available choice (and it also simplifies the weight computations, as when $q(\mathbf{x}) \equiv p(\mathbf{x})$ then $w(\mathbf{x}) = p(\mathbf{y}|\mathbf{x})$) but often is very statistically inefficient, particularly as the dimensionality of $\mathbf{x}$ increases or if $p(\mathbf{y}|\mathbf{x})$ is tightly peaked. Choice of proposal distributions is discussed further in Section 6.5, and a procedure for learning them automatically is described in Chapter 7.

3.3.2 As a probabilistic programming backend

Now consider how we would implement this as an inference backend for a probabilistic programming language. We need to implement both `sample` and `observe` checkpoint functions; these are permitted to update and access global state variables. For importance sampling, consider a state with only a single field lp, which represents the log probability of a single, current program execution. We can then provide very simple implementations of the checkpoint functions. Each checkpoint takes one of two forms:

- (`sample`, f_j, θ_j): sample a value $x_j \sim f_j(x|\theta_j)$, and then call $\mathcal{P}(x_j)$;
- (`observe`, g_i, ϕ_i, y_i): update log probability $\mathsf{lp} = \mathsf{lp} + \log g_i(y_i|\phi_i)$, and call $\mathcal{P}()$.

An overall outer-loop function we will call `infer` takes the program $\mathcal{P}$ as input, and returns a posterior representation. This function needs to perform two tasks: it needs to initialize the global state (i.e., set lp), and it needs to emit weighted samples. An implementation for importance sampling proceeds by repeating, forever:

1. set $\mathsf{lp} = 0$;

2. generate a value ψ by executing the program $\mathcal{P}$ using a backend with the `sample` and `observe` implementations just defined above;

3. emit as a weighted sample $(\psi, \exp(\mathsf{lp}))$.

This generates an infinite sequence of weighted samples which can be used to approximate posterior expectations.

3.4 Partitioning the program execution trace

Suppose we assume that the number of calls to `observe` is fixed across different executions of the program; that is, the value of N is not itself random. This is not a strong restriction in practice, as in general we will have a generative model which is conditioned on a dataset $y_1, \ldots, y_N$, and thus any particular re-execution of the program on the same dataset will have N `observe` statements. This can also be extended to handle probabilistic models in which the number of actual observations generated is itself a random variable: for a fixed dataset with N data points, such models can be handled by including `observe` statements not only for the N points, but also for the value of N itself, explicitly conditioning on the fixed dataset size. The number M of latent variables x_j in each trace may still vary dramatically between executions. In this case the N observe statements then provide an explicit partitioning of the latent variables, which holds across all possible executions. This allows us to define a sequential Monte Carlo algorithm for probabilistic program inference, a refinement of the basic likelihood weighting or importance sampling method that can take advantage of incremental evidence when available to provide more efficient inference in higher-dimensional models.

A sequential Monte Carlo algorithm [Doucet et al., 2001] for inference in probabilistic programs is based on the ability to decompose the full program trace latent variables $\mathbf{x}$ into a product over incremental expansions of the program trace. We define $\mathbf{x}_i$ as the subspace of $\mathbf{x}$ which is sampled in between `observe` statements $i-1$ and i, that is, with $\mathbf{x}_{1:n} = \mathbf{x}_1 \times \cdots \times \mathbf{x}_n$ such that $\mathbf{x}_{1:N}$ denotes the full program trace, and with each $\mathbf{x}_i$ disjoint. Note that in this definition the

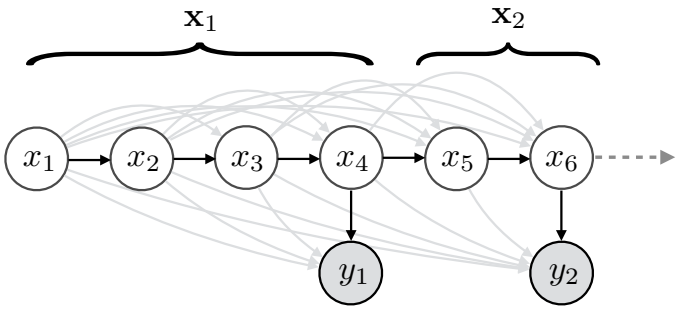


Figure 3.2: Using the observed random variables to define a sequential disjoint partitioning of the latent random choices. Contrast with the agglomerative grouping in Figure 3.1. Partitioning the latent variables into N disjoint grouping $\mathbf{x}_1, \ldots, \mathbf{x}_N$ makes defining a sequential Monte Carlo algorithm straightforward.

individual values x_j may be univariate, multivariate, or have any other structure as dictated by the primitive distribution type f_j; the use of bold indicates multiple individual `sample` statements, not the dimensionality of any particular one. While there are always N such $\mathbf{x}_i$, each may be of varying dimensionality on each execution, and there may also be some $\mathbf{x}_i = \emptyset$ if no new randomness is sampled between two subsequent `observe` statements. This notation is summarized in Figure 3.2.

We can thus define probability distributions $\gamma_1, \ldots, \gamma_N$ on a sequence of incremental program execution traces as

$$\gamma_n(\mathbf{x}_{1:n}) = \prod_{i=1}^{n} g(y_i|\mathbf{x}_{1:i})p(\mathbf{x}_i|\mathbf{x}_{1:i-1}), \tag{3.19}$$

with associated normalized incremental target densities

$$\pi_n(\mathbf{x}_{1:n}) = \frac{1}{Z_n}\gamma_n(\mathbf{x}_{1:n}) \tag{3.20}$$

where each Z_n is an unknown constant. Note that computing the density $p(\mathbf{x}_n|\mathbf{x}_{1:n-1})$ may well be impossible in general, but we can still draw samples from it via forward simulation of the program.

The sequential importance resampling algorithm initializes by executing K parallel copies of the program $\mathcal{P}$, and continuing execution until the first `observe` statement is encountered. Weights for each partial trace $\mathbf{x}_1^k$, for $k = 1, \ldots, K$, are then initialized to

$$w(\mathbf{x}_1^k) = g_1^k(y_1^k|\phi_1^k) \equiv p(y_1^k|\mathbf{x}_1^k). \tag{3.21}$$

Some of the sampled values $\mathbf{x}_1^k$ will be "better" than others, in the sense that they have higher weight. A resampling step now duplicates the more promising program execution traces, and discards those which are already very unlikely, by sampling *ancestor indices* a_1^k from a discrete distribution on W_1^k, where we define incremental normalized weights as

$$W_n^k = \frac{w(\mathbf{x}_n^k)}{\sum_{\ell=1}^K w(\mathbf{x}_n^\ell)}. \tag{3.22}$$

After resampling, all particles have equal weight.

We then continue executing the program from the program traces $\{\mathbf{x}_1^{a_1^k}\}_{k=1}^K$, until the next `observe` statement. In general, for $n > 1$, we then have

$$w(\mathbf{x}_n^k) = g_n^k(y_n^k|\phi_n^k) \equiv p(y_n^k|\mathbf{x}_{1:n-1}^{a_{n-1}^k} \times \mathbf{x}_n^k); \tag{3.23}$$

that is, each ϕ_n^k and new incremental partial trace $\mathbf{x}_n^k$, are attained by continuing program execution from the partial execution corresponding to the previous partial trace $\mathbf{x}_{1:n-1}^{a_{n-1}^k}$, defining

$$\mathbf{x}_{1:n}^k = \mathbf{x}_{1:n-1}^{a_{n-1}^k} \times \mathbf{x}_n^k. \tag{3.24}$$

After the first observation (and prior to resampling), the weighted set of samples $\{\mathbf{x}_1^k, W_1^k\}_{k=1}^K$ provides an approximation to the first intermediate density $\pi_1(\mathbf{x}_1)$ in the same manner as the likelihood weighting algorithm approximates the full $\pi(\mathbf{x})$. At subsequent stages, the resampling distribution $r_n(\cdot)$, for assigning $\{a_n^1, \ldots a_n^K\} \sim r_n(\cdot)$, is defined such that

$$r(a_n^k = k') = W_n^{k'}. \tag{3.25}$$

The unweighted distribution after resampling is still an approximation to $\pi_n(\mathbf{x}_n)$, in the sense that

$$\mathbb{E}\left[\frac{1}{K}\sum_{k=1}^K \psi(\mathbf{x}_{1:n}^{a_n^k})\right] = \frac{1}{K}\sum_{k=1}^K \mathbb{E}\left[\psi(\mathbf{x}_{1:n}^{a_n^k})\right] = \frac{1}{K}\sum_{k=1}^K W_n^k \psi(\mathbf{x}_{1:n}^k); \tag{3.26}$$

and the iterative resampling and simulating procedure yields an approximation to the full trace after reaching $\gamma_N = \gamma$. There are a number of different algorithms

for resampling the particle set, and manners of construction the distribution $r(a_n^k|\cdot)$ [Douc et al., 2005]; we will revisit resampling in detail in Chapter 5. The marginal likelihood $Z \equiv p(\mathbf{y}_{1:N})$ can be estimated recursively, with

$$Z = \prod_{n=1}^{N} Z_n. \tag{3.27}$$

The approximation we can compute through the unnormalized weights at each resampling point (i.e. at each `observe`) given by

$$\hat{Z} = \prod_{n=1}^{N} \hat{Z}_n = \prod_{n=1}^{N} \frac{1}{K} \sum_{k=1}^{K} w(\mathbf{x}_{1:n}^k) \tag{3.28}$$

is known to be unbiased [Del Moral, 2004].

Implementing this as a probabilistic programming backend is similar to importance sampling, except for two key differences. First, the outer loop `infer` method launches K copies of $\mathcal{P}$ at once. Then, at `observe` checkpoints, instead of updating the log weight, we wait until all K copies of the program have reached the `observe`, and then perform a resampling operation based on the values of $g_i^k(y_i^k|\phi_i^k)$. Explicit algorithms and implementations for this and other inference backends are deferred until the following chapter.

3.5 A single-site Metropolis-Hastings algorithm

The two sampling methods thus far — importance sampling, sequential Monte Carlo — are broadly referred to as particle-based methods, in that inference is performed by simulating and weighting a set of samples. It is also possible to define a more traditional Metropolis-Hastings (MH) sampler in this context [Goodman et al., 2008, Wingate et al., 2011], where a sequence of samples is generated by repeated application of a transition operator $A(\mathbf{x} \to \mathbf{x}')$ on a single current set of latent variables $\mathbf{x}$. While the focus of this thesis is on developing sequential Monte Carlo methods, we briefly describe this approach to MH inference here as an interlude for two reasons. First, we would like to demonstrate the generality of the language-inference interface described above, by showing that a more traditional Metropolis-Hastings algorithm can be effectively implemented in this context.

Second, considering the relative advantages and disadvantages of this inference algorithm motivates the particle MCMC approaches to inference we describe later.

What we will call a *single-site* MH sampler proposes at every iteration modification of a single random choice x_j in $\mathbf{x}$. In finite-dimensional models, this sort of scheme is often referred to as "Metropolis-in-Gibbs": an ideal Gibbs sampler would iterate through the random choices $x_1, \ldots, x_M$ and sample directly from the full conditional of every $p(x_j|\mathbf{x}_{\neg j}, \mathbf{y})$, where $\mathbf{x}_{\neg j}$ denotes the set of random choices $x_1, \ldots, x_M$ excluding x_j; however, when this full conditional distribution is not available in closed form, then one option is to apply a Metropolis-Hastings step on x_j (see analysis in e.g. Łatuszyński et al. [2013]). For some models — particularly for models where there is no *incremental evidence*, i.e. where there are no observations of $\mathbf{y}$ interspersed with samples of $\mathbf{x}$ — this can be advantageous relative to the particle-based methods, assuming one can design efficient transition kernels. For other models — particularly where there is tight correlation between random choices — this inference scheme can be problematic and fail to converge; such problems exist as well even for exact Gibbs sampling schemes, and in extreme cases can lead to reducible Markov chains which no longer admit the target density as their invariant distribution. In the context of probabilistic programs, due to its comparative ease of implementation, this method is known as *lightweight Metropolis-Hastings* (LMH) [Wingate et al., 2011].

A MH algorithm draws a sequence of dependent samples according to a target distribution $\pi(\mathbf{x})$ by using a proposal kernel $\kappa(\mathbf{x}'|\mathbf{x})$; given a current sample $\mathbf{x}$, we propose a new candidate sample $\mathbf{x}' \sim \kappa(\cdot|\mathbf{x})$ and compute an acceptance probability

$$A(\mathbf{x} \to \mathbf{x}') = \min\left(1, \frac{\pi(\mathbf{x}')\kappa(\mathbf{x}|\mathbf{x}')}{\pi(\mathbf{x})\kappa(\mathbf{x}'|\mathbf{x})}\right). \tag{3.29}$$

With probability $A(\mathbf{x} \to \mathbf{x}')$ we accept this proposal, and select $\mathbf{x}'$ as the next sample; otherwise, we select $\mathbf{x}$, repeating it as the next sample. Note that as the normalization constant Z for the density π is the same for both $\mathbf{x}$ and $\mathbf{x}'$, we can replace π with γ when computing this acceptance ratio. We can construct a sampler along these lines for sampling probabilistic program traces.

We initialize the algorithm by running a single execution of the probabilistic program, generating an initial trace $\mathbf{x}^0$, of length M^0. Now, given a trace $\mathbf{x}^k$, we define a proposal function $\kappa(\mathbf{x}'|\mathbf{x}^k)$ for sampling new candidate traces as follows. First: the trace $\mathbf{x}^k$ has length M^k, and we pick a single random choice x_ℓ^k by drawing ℓ uniformly from the set of integers $1, \ldots, M^k$. Then, we apply a reversible transition kernel $\kappa_\ell(x'_\ell|x_\ell^k)$ to propose a new value at that specific random choice. We now re-run the remainder of the program $\mathcal{P}$, starting with the partial program execution trace $\mathbf{x}'_\ell = \mathbf{x}_{\ell-1}^k \times x'_\ell$, simulating the rest of the program to generate a new proposal trace $\mathbf{x}'$ of length M'. This leads to an overall proposal density

$$\kappa(\mathbf{x}'|\mathbf{x}^k) = \frac{1}{M^k}\kappa_\ell(x'_\ell|x_\ell^k)\prod_{j=\ell+1}^{M'} f'_j(x'_j|\theta'_j) \tag{3.30}$$

which in turn leads to an acceptance probability for a proposed $\mathbf{x}^{k+1}$ of

$$A(\mathbf{x}^k \to \mathbf{x}') = \min\left(1, \frac{\gamma(\mathbf{x}')M^k\kappa_\ell(x_\ell^k|x'_\ell)\prod_{j=\ell+1}^{M^k} f_j^k(x_j^k|\theta_j^k)}{\gamma(\mathbf{x}^k)M'\kappa_\ell(x'_\ell|x_\ell^k)\prod_{j=\ell+1}^{M'} f'_j(x'_j|\theta'_j)}\right) \tag{3.31}$$

which defines a basic MCMC sampler targeting the space of program execution traces. A simple choice of proposal distribution $\kappa_\ell(x'_\ell|x_\ell^k)$ is the forward simulation density $f_\ell(x_\ell|\theta_\ell)$, in which case the overall acceptance ratio simplifies further to

$$A(\mathbf{x}^k \to \mathbf{x}') = \min\left(1, \frac{\gamma(\mathbf{x}')M^k\prod_{j=\ell}^{M^k} f_j(x_j^k|\theta_j^k)}{\gamma(\mathbf{x}^k)M'\prod_{j=\ell}^{M'} f'_j(x'_j|\theta'_j)}\right). \tag{3.32}$$

3.5.1 As a probabilistic programming backend

To write this basic LMH sampler as an inference backend, we need a sampler which keeps as its internal state a "tape" holding the sequence of all random choices made in a single execution. The simplest implementation works by re-executing the entire program for each new proposal, building up a second alternative sequence of random choices, while also storing all quantities necessary to compute the acceptance ratio in Equation (3.32). The overall outer `infer` loop repeatedly simulates new modified executions as candidates for updated values of `x`, along with computing the overall unnormalized probability `lp`, and a sequence `f` of individual proposal probabilities $f_j(x_j|\theta_j)$ for $j = 1, \ldots, M$.

First, we initialize by setting $\mathsf{lp} \leftarrow -\infty$, ensuring that the first MH iteration always accepts. Then, repeatedly,

1. set x' and f' to empty sequences, and lp' to 0;
2. set splice to be a splice point sampled uniformly from $\{1, \ldots, M\}$, where $M = |\mathsf{x}|$ (if x is empty, or on the first iteration, set $\mathsf{splice} = 0$);
3. execute the program $\mathcal{P}$ using a backend with the simple LMH `sample` and `observe` implementations we will define below, to generate a value ψ' and produce as side effects the sequence of random choices x', the joint probability $\mathsf{lp}' = \gamma(\mathsf{x}')$, and individual simulation probabilities $\mathsf{f}' = [f'_j(x'_j|\theta'_j)]_{j=1}^{M'}$;
4. with probability given by Equation (3.32) (which can be computed from $\mathsf{f}, \mathsf{f}', \mathsf{lp}$, and lp'), set $\psi \leftarrow \psi'$, $\mathsf{x} \leftarrow \mathsf{x}'$, $\mathsf{f} \leftarrow \mathsf{f}'$, and $\mathsf{lp} \leftarrow \mathsf{lp}'$;
5. emit as a weighted sample $(\psi, 1.0)$.

The `sample` and `observe` implementations accumulate probabilities (as in the previous importance sampling implementation), but now they instead update temporary values for the current proposal, and they also must store the sampled values for later use. With the above overall algorithm in mind, we define

- (`sample`, f'_j, θ'_j): if $j < \mathsf{splice}$, set $x'_j = x_j$ (i.e., reuse the stored value $\mathsf{x}(j)$); otherwise sample a fresh value $x'_j \sim f'_j(x|\theta'_j)$. Then append x'_j to x', append $\log f'_j(x'_j)$ to f', update log probability $\mathsf{lp}' = \mathsf{lp}' + \log f'_j(x'_j)$, and finally call $\mathcal{P}(x_j)$.
- (`observe`, g'_i, ϕ'_i, y'_i): update log probability $\mathsf{lp}' = \mathsf{lp}' + \log g'_i(y'_i|\phi'_i)$, and call $\mathcal{P}()$.

Note that the LMH algorithm emits a sequence of equally-weighted samples, but possibly emitting the same value ψ multiple times in a row.

3.5.2 A database of random choices

For high-dimensional problems, the basic Metropolis-Hasting algorithm that arises by proposing according to Equation (3.30) will still perform poorly, as after changing proposing a single value x'_ℓ we re-run the rest of the program. This can be made more efficient by re-using some of the sampled values in the remainder of the original trace, replacing the tape data structure in the algorithm above with a map.

Te be able to meaningfully re-use previously sampled values, we need to introduce a concept of an *address* space $\mathcal{A}$, which we use to uniquely label every random choice we `sample` during program execution [Wingate et al., 2011]. On the initial execution, for each random choice $x_j^0 \in \mathcal{X}_j$, we record a tuple $(\alpha, x_j^0) \in \mathcal{A} \times \mathcal{X}_j$. Then, when re-simulating the remainder of the program in the proposal in Equation (3.30), if we encounter a `sample` statement which has the same address α as a sampled value in the previous trace, and the distribution (or "type") of f is the same, then instead of re-simulating a new value x' we re-use the previous value associated with that α in the database. If the proposal is accepted, we update the random database, associating new values of x with each α, and removing from the database any tuples which do not exist in the updated trace. Note that just because the value is re-used does not mean that we can drop that term from the acceptance ratio in Equation (3.32) — in particular, even if some $x' = x^k$ at some address, the associated parameter vector $\theta' \neq \theta^k$ in general.

The use of a random database in reusing random choices across executions is described in detail in Wingate et al. [2011] and Wood et al. [2014]. Wingate et al. [2011] in particular includes concrete instructions on how to effectively define the address space. Most recent research into improving these lightweight MH samplers has been geared towards improving computational efficiency in evaluating the acceptance ratio: a naïve implementation will re-evaluate the entire unnormalized probability $\gamma(\mathbf{x}')$ at every proposal. However, if the dependency structure were fixed and known (as in a graphical model), allowing $\gamma(\mathbf{x}')$ to be written as a product over factors defined on small subsets of all random choices in $\mathbf{x}'$, then one would only need revaluate those few terms of γ which reference the entries

$\{x'_\ell\}$ modified by the proposal. Recent approaches to generating efficient MH proposals include Shred, a static compiler system which isolates subsets of the random variables which do not affect control flow [Yang et al., 2014], Venture, which updates a dynamic representation of the dependency graph after changing the value of individual latent random variables [Mansinghka et al., 2014], and C3, which uses a cache system to reduce computation in a way which mimics static dependency checking [Ritchie et al., 2016]. These efforts effectively aim to recover an idealized Metropolis-within-Gibbs sampler automatically, which would yield overall performance characteristics reminiscent of the inference engines for Bugs and Jags. These all introduce overhead, either in computation or language complexity. In Tolpin, van de Meent, Paige, and Wood [2015], we also seek to optimize the choice of which latent variable in $x_1, \ldots, x_M$ to update at each iteration by learning an output-sensitive distribution (i.e., dependent on the return functional ψ) for a randomized coordinate selection rule, resembling that of a random-scan Metropolis-within-Gibbs sampler [Łatuszyński et al., 2013].

3.6 Particle MCMC algorithms

Particle MCMC algorithms [Andrieu et al., 2010, Holenstein, 2009] use sequential Monte Carlo as a proposal distribution within an MCMC algorithm. The most basic application of particle MCMC to probabilistic programs is the particle independent Metropolis-Hastings (PIMH) algorithm. In this algorithm, we initialize an MCMC sampler by running SMC with K particles to create an initial set of K weighted execution traces $\{\mathbf{x}^{0,k}, W^{0,k}\}_{k=1}^K$ and compute its marginal likelihood estimate $\hat{Z}^0$ according to Equation (3.28). Then, for each "sweep" $s = 1, 2, \ldots$, we run a new SMC sampler to propose a candidate set of execution traces $\{\mathbf{x}^{\star,k}, W^{\star,k}\}_{k=1}^K$ with associated marginal likelihood estimate $\hat{Z}^\star$. The new candidate set of particles is accepted according to a probability

$$A^s_{PIMH} = \min\left(1, \frac{\hat{Z}^\star}{\hat{Z}^{s-1}}\right). \tag{3.33}$$

If accepted, then the next particle set and next marginal likelihood estimate is set to the proposed values; otherwise, the values from the previous iteration $s-1$ are repeated. In estimating expectations, the full set of particles can be used, with

$$\hat{\mathbb{E}}_{PIMH}[\psi(\mathbf{x})] = \frac{1}{S}\sum_{s=1}^{S}\sum_{k=1}^{K} W^{s,k}\psi(\mathbf{x}^{s,k}). \tag{3.34}$$

Algorithmically, this concatenates the output of the K particles from each of multiple executions of sequential Monte Carlo; with some probability we emit the K particles from the most recent sweep, and with some probability we emit the K particles from a retained sweep. Formal correctness of the PIMH algorithm is shown by considering it as a standard independent MH algorithm on an extended space of both the program traces, and the ancestor indices [Andrieu et al., 2010].

One obvious advantage of this algorithm is its *anytime* nature: running a single SMC iteration yields a K-sample approximation to the posterior, and then for each $s = 1, 2, \ldots$ we have $s \times K$ total samples. In that sense, this provides a principled method for combining multiple executions of sequential Monte Carlo. As we collect more and more samples, we see a corresponding reduction in error for estimates of posterior expectations. An alternative scheme can be constructed by replacing the MH accept-reject step with an additional importance weighting step; this can be understood as a Rao-Blackwellization over the accept-reject step. That is, we could run an algorithm in which we iteratively generate S particle sets; for each particle set $\{\mathbf{x}^{s,k}, W^{s,k}\}_{k=1}^{K}$ we can assign an additional (unnormalized) weight $\hat{Z}^s$. This iterated SMC estimator can then be defined using the normalized weights V_S^s after S particle sets have been generated, with

$$V_S^s = \frac{\hat{Z}^s}{\sum_{t=1}^{S}\hat{Z}^t} \tag{3.35}$$

$$\hat{\mathbb{E}}_{iSMC}[\psi(\mathbf{x})] = \sum_{s=1}^{S} V_S^s \sum_{k=1}^{K} W^{s,k} R(\mathbf{x}^{s,k}). \tag{3.36}$$

The iterated SMC estimator combines the multiple executions of SMC via importance sampling, rather than MCMC. Consistency of such an estimator is shown formally by characterization of this algorithm as a form of α-SMC [Whiteley et al., 2013].

3.6.1 Advanced particle MCMC methods

Potentially more interesting algorithms can be uncovered by considering other samplers which target the same extended space. In particular, the iterated conditional SMC (CSMC) or particle Gibbs (PG) algorithm [Andrieu et al., 2010] re-runs SMC repeatedly in a dependent manner. It is initialized with a single SMC run, in the same manner as PIMH. However, in subsequent MCMC iterations, a Gibbs step is taken. After the first SMC run, we sample a single execution trace $\mathbf{x}_{1:N}^{s,k}$ which will be denoted as a *retained particle* in execution $s+1$. As we run the next iteration of SMC, we simulate only $K-1$ new particles at each stage n, guaranteeing that the retained particle will persist until the end of the execution. Particle Gibbs was first suggested being applied to probabilistic programs in Wood et al. [2014].

In practice, particle Gibbs as implemented for probabilistic programs can sometimes mix poorly over random choices which occur earlier in the program execution trace. When applied to state space models as originally proposed in Andrieu et al. [2010], this can be sidestepped by partitioning from the rest of the model a set of "global" parameters, which then are sampled conditionally given a sample of the latent state space $\mathbf{x}$; in many common state-space models of interest, this is actually a conditional distribution available in closed form. In general for probabilistic programs, without performing code analysis we do not know up front which variables should be considered "global", and we certainly do not have a closed-form expression conditioned on the remaining latent variables in the program trace. This means effectively that global latent variables are treated indistinguishably from the rest of $\mathbf{x}$, often being absorbed into the first partitioned state $\mathbf{x}_1$.

Two different approaches have been employed to try to mitigate this degeneracy when running a particle Gibbs algorithm on probabilistic programs. One is the use of particle Gibbs with ancestor resampling [Lindsten et al., 2012], an algorithm which modifies the retained particle during the forward sweep. This can be implemented efficiently in probabilistic programs via a tracing interpreter to track the dependency structure [van de Meent et al., 2015], at the cost of a significant overhead relative to lightweight implementations of sequential Monte Carlo methods; unfortunately,

a naïve implementation of ancestor resampling in particle Gibbs must evaluate the full likelihood $\gamma(\mathbf{x})$ for every particle at every resample boundary, leading to an algorithm in which each sweep has a runtime cost which is quadratic in the number of `observe` statements. An alternate approach to prevent particle Gibbs from becoming "stuck" is to allow independent SMC runs to occasionally be "swapped in", providing an external source for generating new retained particles independent of the current particle Gibbs sweep. This algorithm can be implemented efficiently by running several independent SMC and conditional SMC sweeps in parallel, potentially on different physical hardware, and allowing the different particle systems to jointly choose retained particles at the end of each sweep, yielding the interacting particle MCMC algorithm [Rainforth, Naesseth, Lindsten, Paige, van de Meent, Doucet, and Wood, 2016].

3.7 Do probability distributions defined in this manner behave as we would expect?

So far, we have simply asserted that the probability distribution we have defined over the random variables in a program execution is consistent with our existing notions of a probability of a generative model, when the generative model is specified in a different representation (for example, as an explicit equation for the joint distribution $p(\mathbf{x}, \mathbf{y})$). For finite-dimensional models, with a fixed number of latent variables, it is fairly straightforward to convince oneself that this reduces to a standard definition of a joint probability distribution. However, particularly when considering the lightweight MH implementation, it is not immediately obvious that this leads to inference over the joint probability space in the way that we would expect. We demonstrate its correctness here in an explicit example.

Most any probabilistic programming language which permits discrete latent variables includes a primitive method for sampling from a Poisson distribution. However, for first-order languages it would be impossible to actually write the sampler for a Poisson distribution in the language itself. This is because the act of sampling a Poisson random variate requires executing a loop with a stochastic

```
(defm sample-poisson [rate]
  (let [L (exp (- rate))]
    (loop [M 0
           p 1]
      (let [M (+ M 1)
            u (sample (uniform-continuous 0 1))
            p (* p u)]
        (if (<= p L)
          (- M 1)
          (recur M p))))))
```

Figure 3.3: This Anglican code defines a recursive sampler for a Poisson random variate. We draw a sample by calling this function for a particular input rate, e.g. `(sample-poisson 4)`.

termination condition. In Figure 3.3 we show a program, again written in Anglican, which is itself a rejection sampler, using an algorithm of Knuth [1969] to sample a random variate k from a Poisson distribution with rate λ. The algorithm begins by initializing $L \leftarrow e^{-\lambda}$, $M \leftarrow 0$, and $p \leftarrow 1$, and then looping:

1. Update $M \leftarrow M + 1$
2. Sample $u \sim \text{Uniform}(0, 1)$
3. Update $p \leftarrow p \times u$
4. If $p \leq L$ return $M - 1$; otherwise repeat.

This algorithm increments M until eventually returning.

Note that each run of the program generates an execution trace with a possibly different amount of randomness; that is, sampling a value of $k = 10$ requires more random choices than a value of $k = 5$. If $\lambda = 4$, then under a Poisson distribution the probability masses $p(k = 3) = p(k = 4)$, but in this algorithm generating a $k = 4$ variate requires more random choices. Particularly vexing, each random choice is a draw from $\text{Uniform}(0, 1)$, meaning that if we label all the M random choices $u_1, \ldots, u_{k+1}$, then for each u_j we have $p(u_j) = 1$.

The execution path of this program is shown in Figure 3.4. Since the (unnormalized) probability of the program execution trace is defined as the product of the probabilities of all random choices made in the execution of the program, then each

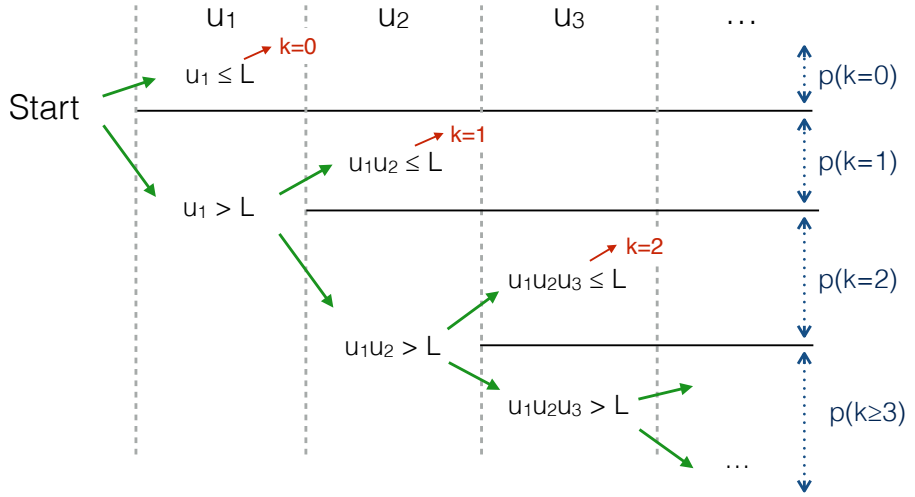


Figure 3.4: Diagram showing sequential steps during the execution of the sampler program. Execution flows to the right; each column shows a random draw $u_j \sim \text{Uniform}(0, 1)$. Green arrows represent the program branching that occurs at the end of each algorithm iteration. Red arrows denote program termination, with a given return value of k.

execution of the program has the same probability, regardless of k, with

$$p(\text{trace}) \propto \prod_{j=1}^{k+1} p(u_j) = 1 \tag{3.37}$$

for all $k = 0, 1, 2, \ldots$. The probability of sampling a particular value k is directly related to the length of the program execution trace — the algorithm is defined in such a way that, although each individual execution has equal probability, the number of execution traces which terminate after M total uniform draws is proportional to a Poisson distribution with rate λ.

The function defined in Figure 3.3 contains no conditioning statements, and as we know that the algorithm in `sample-poisson` returns a Poisson distributed random variate, importance sampling methods will trivially return a Poisson distributed value on every execution. That is, the output ψ from simply executing this code is Poisson distributed. However, it is less obvious whether it is reasonable to define our probability space as we have on the individual random choices u_j (i.e. the latent variables $\mathbf{x}$ in our generic notation above); in particular, we would like to convince ourselves that the sequence of dependent samples defined by LMH is also Poisson distributed, when using the proposal kernel defined in Section 3.5.

So, suppose we would like to use LMH to draw a dependent sequence of samples of `(sample-poisson 4)`. Here we will use the simplest variant of LMH, in which we re-sample all later random choices after modifying one (that is, we will not store random choices in a database for re-use). When running LMH, an initial sample is drawn by running the program forward once; this generates an initial sequence of uniform draws, and returns (as output) an initial Poisson-distributed value k. Subsequent samples are proposed by

1. selecting an individual random choice u_j from the uniform draws $u_1, \ldots, u_{k+1}$ instantiated in the current sample;

2. resampling a program suffix beginning from u_j, by drawing a new value u'_j, discarding all random choices $u_{j+1}, \ldots, u_{k+1}$, and continuing program execution from the point at which u_j is drawn; this creates a new sequence of random choices $u'_j, \ldots, u'_{k'+1}$ where k and k' are not necessarily the same;

3. accepting or rejecting the new proposed set of random choices with probability given by the acceptance ratio in Equation (3.32).

For this particular program, we can significantly simplify the expression for the acceptance ratio. Since there are no `observe` statements in this program, we have $p(\mathbf{y}|\mathbf{x}) = p(\mathbf{y}|\mathbf{x}') = 1$, and since all draws over the course of program execution are from a Uniform(0,1) density, we have $p(\mathbf{x}) = p(\mathbf{x}') = 1$; this implies that $\gamma(\mathbf{x}) = \gamma(\mathbf{x}')$ deterministically, and these terms cancel in Equation (3.32); furthermore, each individual term $f(x|\theta) = 1$. This yields a simplified acceptance probability $A(\mathbf{x} \to \mathbf{x}')$ of

$$A(\mathbf{x} \to \mathbf{x}') = \min\left(1, \frac{k+1}{k'+1}\right) = \min\left(1, \frac{M}{M'}\right), \tag{3.38}$$

which only depends on the number of random choices made in each execution, not their values.

Now consider what happens during a LMH proposal. Given a current M (i.e., the total number of random choices in the previous execution), we propose M' by

continuing the Poisson sampler routine starting from some sampled point j from $\{1, \ldots, M\}$. Effectively, this means the LMH transition first selects a minimum value which lower bounds k', and then resamples from the right tail of a Poisson distribution. Suppose Y is a Poisson random variable, with density $f(Y)$ and distribution function $F(Y)$. We can write out an explicit form for the right tail of the Poisson density $p(Y = k'|k' \geq j-1)$ for $j \geq 1$,

$$p(Y = k'|k' \geq j-1) = \frac{p(Y = k', k' \geq j-1)}{p(k' \geq j-1)} \tag{3.39}$$

$$= \frac{f(Y = k')\mathbb{I}_{[k' \geq j-1]}}{\int_{\ell=j-1}^{\infty} f(Y = \ell)dY} \tag{3.40}$$

$$= \frac{f(Y = k')\mathbb{I}_{[k' \geq j-1]}}{1 - F(j-1)}. \tag{3.41}$$

Since the j is proposed uniformly from $1, \ldots, k+1$, we can write out the proposal kernel κ which is implicitly used by LMH to propose a new value k' given a current value k, by marginalizing over the choice of j; however, we will write this out in terms of the total number of random choices left after making the transition:

$$\kappa(k'|k) = \sum_{j=1}^{k+1} p(Y = k'|k' \geq j-1)p(j|k) \tag{3.42}$$

$$= \frac{f(Y = k')}{k+1} \sum_{j=0}^{k} \frac{\mathbb{I}_{[k' \geq j]}}{1 - F(j)}. \tag{3.43}$$

So, for this particular model, LMH works by proposing a new quantity of total random choices from the kernel $\kappa(k'|k)$ in Eq. (3.43), and accepting it with probability $A(k \to k')$ in Eq. (3.38). Combining these, the overall transition probability $p(k \to k') = A(k \to k')\kappa(k'|k)$ can be written as a product of the density $f(\cdot)$ in the numerator of Eq. (3.43) and a piecewise-defined function $h(k, k')$, by considering the different cases in which k or k' is the greater value; we have

$$p(k \to k') = f(Y = k')h(k, k'), \tag{3.44}$$

with

$$h(k, k') = \begin{cases} \frac{1}{k'+1} \sum_{j=0}^{k} \frac{1}{1 - F(j)} & \text{if } k < k' \\ \frac{1}{k+1} \sum_{j=0}^{k} \frac{1}{1 - F(j)} & \text{if } k = k' \\ \frac{1}{k+1} \sum_{j=0}^{k'} \frac{1}{1 - F(j)} & \text{if } k > k'. \end{cases} \tag{3.45}$$

Note that the function $h(\cdot)$ is symmetric in its arguments; that is, $h(k, k') = h(k', k)$. To confirm that the stationary distribution of this operator is Poisson (i.e. has density $f(Y)$), we check that it satisfies the detailed balance equation

$$f(k')p(k' \to k) = f(k)p(k \to k'). \tag{3.46}$$

Substituting our expression in Eq. (3.44) above, we have

$$f(Y = k')f(Y = k)h(k', k) = f(Y = k)f(Y = k')h(k, k'). \tag{3.47}$$

Since $h(k, k') = h(k', k)$, these quantities are equal and detailed balance holds, so $f(Y)$ is indeed the stationary distribution of $p(k \to k')$, and thus the Markov chain of samples generated by LMH in this instance are appropriately Poisson distributed, despite the somewhat jarring definition of the "probability" of a single execution. Writing a Poisson sampler directly in the probabilistic programming language may not be an explicitly useful task — we rarely have situations in which we would like to (say) condition on particular states of the internal random choices, or report their marginal distributions. However, it provides a simple example demonstrating the need for languages which permit stochastic recursion: if we cannot even implement simple distributions without it, we would be hard-pressed to expect to write expressive models where we do wish to analyze or condition the random choices made during execution.

4

Efficient implementation of continuations

How do we go about implementing a simulation-based probabilistic programming system? The key difficulty is the need to store, pause, and later resume a partially-executed program $\mathcal{P}$. In an SMC algorithm, after each `observe` checkpoint, we will want to duplicate and continue to run multiple copies of executions corresponding to high-weight particles. In a lightweight MH algorithm, we will want to jump back to the location of a specific `sample` statement, and re-run the program from that point. For both, we will want to have some manner for explicitly referencing "the rest of the program", which is known as a *continuation*.

In any language, since the program execution is deterministic given a trace $\mathbf{x}$, we can write a simple-but-inefficient algorithm for running the continuations needed in these inference algorithms, by re-running the entire program for every continuation. That is, if in a LMH update we wish to modify the value of some x_j, to run the program $\mathcal{P}$ continuing from `sample` statement j we actually re-run the entire program from the beginning, using the existing values in the trace $x_1, \ldots, x_{j-1}$ to faithfully recreate whatever the internal memory state of the program $\mathcal{P}$ might be at the point of sampling x_j. This yields an algorithm in which every MCMC update has a runtime which is linear in the length of the trace. In SMC, after each resampling step, new copies of partial executions corresponding to an existing trace $\mathbf{x}_{1:n_j}$ can be created by running new executions of $\mathcal{P}$, re-using trace

values; here this leads to an algorithm with runtime quadratic in the length of the trace, which is particularly bad.

How can we write an SMC algorithm which runs in time linear in the length of the program? The key lies in an efficient implementation of re-invokable continuations. For common procedural programming languages, running multiple copies of a partially executed program requires creating copies of the entire internal memory state of the running program — all currently instantiated variables, the current stack trace, and so on. Fortunately, this does not have to be an immediate deep copy of the entire memory state — in particular, the behavior of the POSIX operating system primitive "fork" spawns new processes which duplicate the currently running process in constant time. Rather than copying over the entire memory space, it simply copies a high-level page table; the memory itself is only duplicated if necessary, in a lazy, copy-on-write fashion.

In pure functional languages, a code transformation to continuation passing style (CPS) allows explicit instantiation of a function which represents the rest of the computation [Appel, 2007, Goodman and Stuhlmüller, 2014]. So long as the language is restricted to prohibit mutation of shared referenced variables, then running multiple copies of the rest of the program corresponding to a particular trace can be as simple as calling the continuation multiple times.

It is also possible to implement a custom interpreter for a domain-specific language: the custom interpreter, rather than compiling the code, can choose to re-interpret from any particular point, and can explicitly model the state of the partially executed program. Unfortunately, this often leads to runtimes which are an enormous factor slower than a system which can leverage existing compilers to run model simulation in a lower-level, more performant language.

4.1 Leveraging existing compilers and operating systems

In this chapter, we develop an explicit formulation for how to implement re-invokable continuations in a manner based on efficient task copying. We will

see that this is orders of magnitude faster than implementations based on running a custom interpreter, and has additional benefits: we show that forward inference techniques such as sequential Monte Carlo and particle Markov chain Monte Carlo for probabilistic programming can be implemented in any programming language by creative use of standardized operating system functionality including processes, forking, mutexes, and shared memory. Exploiting this, we define and develop a probabilistic programming language intermediate representation language we call probabilistic C, which itself can be compiled to machine code by standard compilers and linked to operating system libraries yielding an efficient and portable probabilistic programming compilation target. This opens up a new hardware and systems research path for optimizing probabilistic programming systems. The implementation here was originally described in Paige and Wood [2014]; since this time, new probabilistic programming languages such as a probabilistic variant of Terra called Quicksand [Ritchie, 2014], and a probabilistic variant of Julia called Turing.jl have been introduced, with task-copying implementations of inference algorithms based on the approach here. We will discuss these, as well as the other use cases seen thus far for probabilistic C, at the end of the chapter.

Compilation is source to source transformation; we use the phrase *intermediate representation* to refer to a target language for a compiler. Here we introduce a C-language library that makes possible a C-language intermediate representation for probabilistic programming languages that can itself be compiled to executable machine code; we call this intermediate language *probabilistic C*. Probabilistic C can be compiled normally and uses only macros and POSIX operating system libraries [Open Group, 2004a] to implement general-purpose, scalable, parallel probabilistic programming inference. We characterize the performance of the intermediate representation by writing programs directly in probabilistic C and then testing them on computer architectures and programs that illustrate the capacities and trade-offs of both the forward inference strategies probabilistic C employs and the operating system functionality on which it depends.

Probabilistic C programs compile to machine executable meta-programs that perform inference over the original program via forward methods such as sequential Monte Carlo [Doucet et al., 2001] and particle MCMC variants [Andrieu et al., 2010]. Such inference methods can be implemented in a sufficiently general way so as to support inference over the space of probabilistic program execution traces using just POSIX operating system primitives. In doing so, we can efficiently leverage existing work by the operating systems developers in creating effective virtual memory management and process synchronization systems.

4.1.1 Related work

The characterization of probabilistic programming inference we consider here is the process of sampling from the a posteriori distribution of execution traces arising from stochastic programs constrained to reflect observed data. This is the view taken by the Church [Goodman et al., 2008], Venture [Mansinghka et al., 2014], WebPPL [Goodman and Stuhlmüller, 2014], and Anglican [Wood et al., 2014], programming languages among others. In such languages, models for observed data can be described purely in terms of a forward generative process.

Markov chain Monte Carlo (MCMC) is used by these systems to sample from the posterior distribution of program execution traces. Single-site Metropolis Hastings (MH) [Goodman et al., 2008] and particle MCMC (PMCMC) [Wood et al., 2014] are two such approaches. In the latter it was noted that a `fork`-like operation is a fundamental requirement of forward inference methods for probabilistic programming, where `fork` is the standard POSIX operating system primitive [Open Group, 2004b]. Kiselyov and Shan [2009] also noted that delimited continuations, a user-level generalization of `fork` could be used for inference, albeit in a restricted family of models.

4.2 C as a probabilistic programming language

Any program that makes a random choice over the course of its execution implicitly defines a prior distribution over its random variables; running the program can be

```
#include "probabilistic.h"

int main(int argc, char **argv) {

    double var = 2;
    double mu = normal_rng(1, 5);

    observe(normal_lnp(9, mu, var));
    observe(normal_lnp(8, mu, var));

    predict("mu,%f\n", mu);

    return 0;
}
```

Figure 4.1: This program performs posterior inference over the unknown mean `mu` of a Gaussian, conditioned on two data points. The `predict` directive formats output using standard `printf` semantics.

interpreted as drawing a sample from the prior. Inference in probabilistic programs involves conditioning on observed data, and characterizing the posterior distribution of the random variables given data. We introduce probabilistic programming capabilities into C by providing a library with two primary functions: `observe` which conditions the program execution trace given the log-likelihood of a data point, and `predict` which marks expressions for which we want posterior samples. Any random number generator and sampling library can be used for making random choices in the program, any numeric log likelihood value can be passed to an `observe`, and any C expression which can be printed can be reported using `predict`. The library includes a single macro which renames `main` and wraps it in another function that runs the original in an inner loop in the forward inference algorithms to be described. The only caveat is that when one uses a pseudo-random number generator, then when a running copy of the program calls `fork`, it also copies the internal state of the random number generator. We supply a library of functions to generate pseudo-random numbers from a variety of primitive distributions; this library is linked to the inference implementations in such a way that it is re-seeded after a `fork`.

Although C is a comparatively low-level language, it can nonetheless represent many well-known generative models concisely and transparently. Figure 4.1 shows a simple probabilistic C program for estimating the posterior distribution for

```
#include "probabilistic.h"
#define K 3
#define N 11

/* Markov transition matrix */
static double T[K][K] = { { 0.1,  0.5,  0.4 },
                          { 0.2,  0.2,  0.6 },
                          { 0.15, 0.15, 0.7 } };

/* Observed data */
static double data[N] = { NAN, .9, .8,  .7,   0, -.025,
                               -5, -2, -.1,   0,  0.13 };

/* Prior distribution on initial state */
static double initial_state[K] = { 1.0/3, 1.0/3, 1.0/3 };

/* Per-state mean of Gaussian emission distribution */
static double state_mean[K] = { -1, 1, 0 };

/* Generative program for a HMM */
int main(int argc, char **argv) {

    int states[N];
    for (int n=0; n<N; n++) {
        states[n] = (n==0) ? discrete_rng(initial_state, K)
                           : discrete_rng(T[states[n-1]], K);
        if (n > 0) {
            observe(normal_lnp(data[n], state_mean[states[n]], 1));
        }
        predict("state[%d],%d\n", n, states[n]);
    }

    return 0;
}
```

Figure 4.2: A hidden Markov model (HMM) with 3 underlying states and Gaussian emissions, observed at 10 discrete time-points. We observe 10 data points and predict the marginal distribution over the latent `state` at each time point.

the mean of a Gaussian, conditioned on two observed data points y_1, y_2, corresponding to the model

$$\mu \sim \mathcal{N}(1, 5), \qquad y_1, y_2 \overset{iid}{\sim} \mathcal{N}(\mu, 2). \tag{4.1}$$

We `observe` the data y_1, y_2 and `predict` the posterior distribution of μ. The functions `normal_rng` and `normal_lnp` in Figure 4.1 return (respectively) a normally-distributed random variate and the log probability density of a particular value, with mean and variance parameters `mu` and `var`. The `observe` statement requires

only the log-probability of the data points 8 and 9 conditioned on the current program state; no other information about the likelihood function or the generative process. In this program we predict the posterior distribution of a single value `mu`.

A hidden Markov model example is shown in Figure 4.2, in which $N = 10$ observed data points $y_{1:N}$ are drawn from an underlying Markov chain with K latent states, each with Gaussian emission distributions with mean μ_k, and a (known) $K \times K$ state transition matrix T, such that

$$z_0 \sim \text{Discrete}([1/K, \ldots, 1/K]) \tag{4.2}$$

$$z_n | z_{n-1} \sim \text{Discrete}(T_{z_{n-1}}) \tag{4.3}$$

$$y_n | z_n \sim \mathcal{N}(\mu_{z_n}, \sigma^2). \tag{4.4}$$

Bayesian nonparametric models can also be represented concisely; in Figure 4.3 we show a generative program for an infinite mixture of Gaussians. We use a Chinese restaurant process (CRP) to sequentially sample non-negative integer partition assignments z_n for each data point $y_1, \ldots, y_N$. For each partition, mean and variance parameters $\mu_{z_n}, \sigma^2_{z_n}$ are drawn from a normal-gamma prior; the data points y_n themselves are drawn from a normal distribution with these parameters, defining a full generative model

$$z_n \sim \text{CRP}(\alpha, z_1, \ldots, z_{n-1}) \tag{4.5}$$

$$1/\sigma^2_{z_n} \sim \text{Gamma}(1, 1) \tag{4.6}$$

$$\mu_{z_n} | \sigma^2_{z_n} \sim \mathcal{N}(0, \sigma^2_{z_n}) \tag{4.7}$$

$$y_n | z_n, \mu_{z_n}, \sigma^2_{z_n} \sim \mathcal{N}(\mu_{z_n}, \sigma^2_{z_n}). \tag{4.8}$$

This program also demonstrates the additional library function `memoize`, which can be used to implement stochastic memoization as described in Goodman et al. [2008].

Inference proceeds by drawing posterior samples from the space of program execution traces. We define an execution trace as the sequence of memory states (the entire virtual memory address space) that arises during the sequential step execution of machine instructions.

```
#include "probabilistic.h"
#define N 10

// Observed data
static double data[N] = { 1.0,  1.1,   1.2,
                          -1.0, -1.5,  -2.0,
                         0.001, 0.01, 0.005, 0.0 };

// Struct holding mean and variance parameters for each cluster
typedef struct theta {
    double mu;
    double var;
} theta;

// Draws a sample of theta from a normal-gamma prior
theta draw_theta() {
    double variance = 1.0 / gamma_rng(1, 1);
    return (theta) { normal_rng(0, variance), variance };
}

// Get the class id for a given observation index
static polya_urn_state urn;
void get_class(int *index, int *class_id) {
    *class_id = polya_urn_draw(&urn);
}

int main(int argc, char **argv) {
    double alpha = 1.0;
    polya_urn_new(&urn, alpha);

    mem_func mem_get_class;
    memoize(&mem_get_class, get_class, sizeof(int), sizeof(int));

    theta params[N];
    bool known_params[N] = { false };

    int class;
    for (int n=0; n<N; n++) {
        mem_invoke(&mem_get_class, &n, &class);
        if (!known_params[class]) {
            params[class] = draw_theta();
            known_params[class] = true;
        }
        observe(normal_lnp(data[n], params[class].mu,
                                    params[class].var));
    }

    // Predict number of classes
    predict("num_classes,%2d\n", urn.len_buckets);

    // Release memory; exit
    polya_urn_free(&urn);
    return 0;
}
```

Figure 4.3: A infinite mixture of Gaussians on the real line. Class assignment variables for each of the 10 data points are drawn following a Blackwell-MacQueen urn scheme to sequentially sample from a Dirichlet process.

The algorithms we propose for inference in probabilistic programs map directly onto standard computer operating system constructs, exposed in POSIX-compliant operating systems including Linux, BSD, and Mac OS X. The cornerstone of our approach is POSIX `fork` [Open Group, 2004b]. When a process forks, it clones itself, creating a new process with an identical copy of the execution state of the original process, and identical source code; both processes then continue with normal program execution completely independently from the point where `fork` was called. While copying program execution state may naïvely sound like a costly operation, this actually can be rather efficient: when `fork` is called, a lazy copy-on-write procedure is used to avoid deep copying the entire program memory. Instead, initially only the pagetable is copied to the new process; when an existing variable is modified in the new program copy, then and only then are memory contents duplicated. The overall cost of forking a program is proportional to the fraction of memory which is rewritten by the child process [Smith and Maguire, 1988].

Using `fork` we can branch a single program execution state and explore many possible downstream execution paths. Each of these paths runs as its own process, and will run in parallel with other processes. In general, multiple processes run in their own memory address space, and do not communicate or share state. We handle inter-process communication via a small shared memory segment; the details of what global data must be stored are provided later.

Synchronization between processes is handled via mutual exclusion locks (*mutex* objects). Mutexes become particularly useful for us when used in conjunction with a synchronized counter to create a *barrier*, a high-level blocking construct which prevents any process proceeding in execution state beyond the barrier until some fixed number of processes have arrived.

4.3 Inference Algorithms in Probabilistic C

We notate the probability of a program execution trace here as in the previous chapter. We first enumerate all N `observe` statements, and the associated observed data points $y_1, \ldots, y_N$. During a single run of the program, some total number

M random choices $x_1, \ldots, x_M$ are made. While M may vary between individual executions of the program, we require that the number of `observe` directive calls N is constant.

The observations y_n can appear at any point in the program source code and define a partition of the random choices $x_{1:M}$ into N subsequences $\mathbf{x}_{1:N}$, where each $\mathbf{x}_n$ contains all random choices made up to observing y_n but excluding any random choices prior to observation y_{n-1}. We can then define the probability of any single program execution trace

$$p(y_{1:N}, \mathbf{x}_{1:N}) = \prod_{n=1}^{N} g(y_n|\mathbf{x}_{1:n}) f(\mathbf{x}_n|\mathbf{x}_{1:n-1}) \tag{4.9}$$

In this manner, any model with a generative process that can be written in C code with stochastic choices can be represented in this sequential form in the space of program execution traces.

Each `observe` statement takes as its argument a floating point number expected to represent the log probability $\log g(y_n|\mathbf{x}_{1:n})$. Each quantity of interest in a `predict` statement corresponds to some function $\psi(\cdot)$ of all random choices $\mathbf{x}_{1:N}$ made during the execution of the program. Given a set of S posterior samples $\{\mathbf{x}_{1:N}^{(s)}\}$, we can approximate the posterior expectation of the `predict` value as

$$\mathbb{E}[\psi(\mathbf{x}_{1:N})] \approx \frac{1}{S} \sum_{s=1}^{S} \psi(\mathbf{x}_{1:N}^{(s)}). \tag{4.10}$$

4.3.1 Sequential Monte Carlo

Forward simulation-based algorithms are a natural fit for probabilistic programs: run the program and report executions that match the data. We now review sequential Monte Carlo with an eye towards how it can be implemented using only calls to `fork`. SMC approximates a target distribution $p(\mathbf{x}_{1:N}|y_{1:N})$ as a weighted set of K realized trajectories $\mathbf{x}_{1:N}^{\ell}$ such that

$$p(\mathbf{x}_{1:N}|y_{1:N}) \approx \sum_{\ell=1}^{K} w_N^{\ell} \delta_{\mathbf{x}_{1:N}^{\ell}}(\mathbf{x}_{1:N}). \tag{4.11}$$

For most probabilistic programs of interest, it will be intractable to sample from $p(\mathbf{x}_{1:N}|y_{1:N})$ directly. Instead, noting that (for $n > 1$) we have the recursive identity

$$p(\mathbf{x}_{1:n}|y_{1:n}) = p(\mathbf{x}_{1:n-1}|y_{1:n-1})g(y_n|\mathbf{x}_{1:n})f(\mathbf{x}_n|\mathbf{x}_{1:n-1}), \tag{4.12}$$

we sample from $p(\mathbf{x}_{1:N}|y_{1:N})$ by iteratively sampling from each $p(\mathbf{x}_{1:n}|y_{1:n})$, in turn, from 1 through N. At each n, we construct an importance sampling distribution by proposing from some distribution $q(\mathbf{x}_n|\mathbf{x}_{1:n-1}, y_{1:n})$; in probabilistic programs we find it convenient to propose directly from the executions of the program, i.e. each sequence of random variates $\mathbf{x}_n$ is jointly sampled from the program execution state dynamics

$$\mathbf{x}_n^\ell \sim f(\mathbf{x}_n|\mathbf{x}_{1:n-1}^{a_{n-1}^\ell}) \tag{4.13}$$

where a_{n-1}^ℓ is an "ancestor index," the particle index $1, \ldots, K$ of the parent (at time $n-1$) of $\mathbf{x}_n^\ell$. The unnormalized particle importance weights at each observation y_n are simply the `observe` data likelihood

$$\tilde{w}_n^\ell = g(y_{1:n}, \mathbf{x}_{1:n}^\ell) \tag{4.14}$$

which can be normalized as

$$w_n^\ell = \frac{\tilde{w}_n^\ell}{\sum_{\ell=1}^K \tilde{w}_n^\ell}. \tag{4.15}$$

After each step n, we now have a weighted set of execution traces which approximate $p(\mathbf{x}_{1:n}|y_{1:n})$. As the program continues, traces which do not correspond well with the data will have weights which become negligibly small, leading in the worst case to all weight concentrated on a single execution trace. To counteract this deficiency, we resample from our current set of K execution traces after each observation y_n, according to their weights w_n^ℓ. This is achieved by sampling a count O_n^ℓ for the number of "offspring" of a given execution trace ℓ to be included at time $n+1$; any sampling scheme must ensure $\mathbb{E}[O_n^\ell] = w_n^\ell$. Sampling offspring counts O_n^ℓ is equivalent to sampling ancestor indices a_n^ℓ. Program execution traces with no offspring are killed; program execution traces with more than one offspring are forked multiple times. After resampling, all weights $w_n^\ell = 1$.

Algorithm 1 Parallel SMC program execution

Require: N observations, K particles
launch K copies of the program (parallel)
for $n = 1 \dots N$ **do**
wait until all K reach `observe` y_n (barrier)
update unnormalized weights $\tilde{w}_n^{1:K}$ (serial)
if $ESS < \tau$ **then**
sample number of offspring $O_n^{1:K}$ (serial)
set weight $\tilde{w}_n^{1:K} = \frac{1}{K} \sum_{\ell=1}^{K} w_n^{\ell}$ (serial)
for $\ell = 1 \dots K$ **do**
fork or exit (parallel)
end for
else
set all number of offspring $O_n^{\ell} = 1$ (serial)
end if
continue program execution (parallel)
end for
wait until K program traces terminate (barrier)
`predict` from K samples from $\hat{p}(\mathbf{x}_{1:N}^{1:K} | y_{1:N})$ (serial)

We only resample if the effective sample size

$$ESS \approx \frac{1}{\sum_{\ell}(w_n^{\ell})^2} \tag{4.16}$$

is less than some threshold value τ; we choose $\tau = K/2$.

In probabilistic C, each `observe` statement forms a barrier: parallel execution cannot continue until all particles have arrived at the `observe` and have reported their current unnormalized weight. As execution traces arrive at the `observe` barrier, they take the number of particles which have already reached the current `observe` as a (temporary) unique identifier. Program execution is then blocked as the effective sample size is computed and the number of offspring are sampled. The number of offspring are stored in a shared memory block; when the number of offspring are computed, each particle uses the identifier assigned when reaching the `observe` barrier to retrieve (asynchronously) from shared memory the number of children to `fork`. Particles with no offspring wait for any child processes to complete execution, and terminate; particles with only one offspring do not `fork` any children but continue execution as normal.

The SMC algorithm is outlined in Algorithm 1, with annotations for which steps are executed in parallel, serially, or form a barrier. After a single SMC sweep is complete, we sample values for each `predict`, and then (if desired) repeat the process, running a new independent particle filter, to draw an additional batch of samples.

4.3.2 Particle Metropolis-Hastings

Particle Markov chain Monte Carlo, introduced in Andrieu et al. [2010], uses sequential Monte Carlo to generate high-dimensional proposal distributions for MCMC. The most simple formulation is the particle independent Metropolis-Hastings algorithm. After running a single particle filter sweep, we compute an estimate of the marginal likelihood,

$$\hat{Z} \equiv p(y_{1:N}) \approx \prod_{n=1}^{N} \left[\frac{1}{K} \sum_{\ell=1}^{K} w_n^{\ell} \right]. \tag{4.17}$$

We then run another iteration of sequential Monte Carlo which we use as a MH proposal; we estimate the marginal likelihood $\hat{Z}'$ of the new proposed particle set, and then with probability $\min(1, \hat{Z}'/\hat{Z})$ we accept the new particle set and output a new set of `predict` samples, otherwise outputting the same `predict` samples as in the previous iteration.

The inner loop of Algorithm 2 is otherwise substantially similar to SMC.

4.3.3 Particle Gibbs

Particle Gibbs is a particle MCMC technique which also has SMC at its core, with better theoretical statistical convergence properties than PIMH but additional computational overhead. We initialize particle Gibbs by running a single sequential Monte Carlo sweep, and then alternate between (1) sampling a single execution trace $\hat{\mathbf{x}}_{1:M}$ from the set of K weighted particles, and (2) running a "conditional" SMC sweep, in which we generate $K-1$ new particles in addition to the retained $\hat{\mathbf{x}}_{1:M}$.

The implementation based on operating system primitives is described in algorithms 3 and 4. The challenge here is that we must "retain" an execution trace, which we can later revisit to resume and branch arbitrarily many times. This

Algorithm 2 Parallel PIMH program execution

Require: M iterations, N observations, K particles
  for $m = 1 \dots M$ **do**
    launch K copies of the program (parallel)
    for $n = 1 \dots N$ **do**
      wait until all K reach an `observe` (barrier)
      update unnormalized weights $\tilde{w}_{1:K}$, proposal evidence $\hat{Z}'$ (serial)
      sample number of offspring $O_n^{1:K}$ (serial)
      set weight $\tilde{w}_n^{1:K} = 1$ (serial)
      for $\ell = 1 \dots K$ **do**
        fork or exit (parallel)
      end for
      continue program execution (parallel)
    end for
    wait until K program traces terminate (barrier)
    accept or reject new particle set (serial)
    `predict` from K samples from $\hat{p}(\mathbf{x}_{1:N}^{1:K} | y_{1:N})$ (serial)
    store current particle set $\mathbf{x}$ and evidence $\hat{Z}$ (serial)
    continue to next iteration (parallel)
  end for

is achieved by spawning off a "control" process at every observation point, which from then on manages the future of that particular execution state.

As before, processes arrive at an `observe` barrier, and when all particles have reached the observe we compute weights, and sample offspring counts O_n^ℓ. Particles with $O_n^\ell = 0$ terminate, but new child processes are no longer spawned right away. Instead, all remaining particles `fork` a new process whose execution path immediately diverges from the main codebase and enters the retain and branch loop in Algorithm 4. This new process takes responsibility for actually spawning the O_n^ℓ new children. The spawned child processes (and the original process which arrived at the `observe` barrier) wait (albeit briefly) at a new barrier marking the *end* of `observe` n, not continuing execution until all new child processes have been launched.

Program execution continues to the next `observe`, during which the retain / branch process waits until a full particle set reaches the end of the program. Once final weights $\tilde{w}_N^{1:K}$ are computed, we sample (according to weight) from the final particle set to select a single particle to retain during the next SMC iteration. When the particle is selected, a signal is broadcast to all retain /

Algorithm 3 Parallel Particle Gibbs program execution

Require: M iterations, N observations, K particles
  for $m = 1 \ldots M$ **do**
    $K' \leftarrow K$ if $m = 1$, otherwise $K - 1$
    launch S' copies of the program (parallel)
    for $n = 1 \ldots N$ **do**
      wait until all K' reach an `observe` (barrier)
      compute weights for all particles (serial)
      if $m > 1$ **then**
        signal num offspring to retained trace (serial)
      end if
      for $\ell = 1 \ldots K'$ **do**
        spawn retain / branch process [Algo. 4] (parallel)
      end for
      wait until K particles finish branching (barrier)
      continue program execution (parallel)
    end for
    wait until K program traces terminate (barrier)
    `predict` from K samples from $\hat{p}(\mathbf{x}_{1:N}^{1:K} | y_{1:N})$ (serial)
    select and signal particle to retain (serial)
    wait until N processes are ready to branch (barrier)
    continue to next iteration (parallel)
  end for

branch loops indicating which process ids correspond to the retained particle; all except the retained trace exit.

The retain / branch loop now goes into waiting again (this time for a branch signal), and we begin SMC from the top of the program. As we arrive at each observe n, we only sample $K - 1$ new offspring to consider: we guarantee that at least one offspring is spawned from the retained particle at n (namely, the retained execution state at $n + 1$). However, depending on the weights, often sampling offspring will cause us to want more than a single child from the retained particle. So, we signal to the retained particle execution state at time n the number of children to spawn; the retain / branch loop returns to its entry point and resumes waiting, to see if the previously retained execution state will be retained yet again.

Note that in particle Gibbs and PIMH, we must resample (select offspring and reset weights $w_n^\ell = 1$) after every observation in order to be able to properly align the retained particle on the next iteration through the program.

Algorithm 4 Retain and Branch inner loop

Require: input initial $C > 0$ children to spawn
  `is_retained` ← `false`
  while `true` **do**
    if $C = 0$ and not `is_retained` **then**
      discard this execution trace, exit
    else $\{C \geq 0\}$
      spawn C new children
    end if
    wait for signal which resets `is_retained`
    if `is_retained` **then**
      wait for signal which resets C
    else
      discard this execution trace, exit
    end if
  end while

4.4 Experimental validation

We now turn to benchmarking probabilistic C against existing probabilistic programming engines, and evaluate the relative strengths of the inference algorithms in Section 4.3. We find that compilation improves performance by approximately 100 times over interpreted versions of the same inference algorithm. We also find evidence that suggests that optimizing operating systems to support probabilistic programming usage could yield significant performance improvements as well.

The programs and models we use in our experiments are chosen to be sufficiently simplistic that we can compute the exact posterior distribution of interest analytically, allowing us to evaluate correctness of inference. Given the true posterior distribution p, we measure sampler performance by the KL-divergence $KL(\hat{p}||p)$, where $\hat{p}$ is our Monte Carlo estimate. The first benchmark program we consider is a hidden Markov model (HMM) very similar to that of Figure 4.2, where we predict the marginal distributions of each latent state. The HMM used in our experiments here is larger; it has the same model structure, but with $K = 10$ states and 50 sequential observations, and each state $k = 1, \ldots, 10$ has a Gaussian emission distribution with $\mu_k = k - 1$ and $\sigma^2 = 4$. The second benchmark is the CRP mixture of Gaussians program in Figure 4.3, where we predict the total number of distinct classes.

4.4.1 Comparative performance of inference engines

We begin by benchmarking against two existing probabilistic programming engines: the original interpreted variant of *Anglican*, as described in Wood et al. [2014], which also implements particle Gibbs but is an interpreted language based on Scheme, implemented in Clojure, and running on the JVM; and *probabilistic-js*[1], a compiled system implementing the inference approach in Wingate et al. [2011], which runs Metropolis-Hastings over each individual random choice in the program execution trace. (Note that this implementation of Anglican is now known as *interpreted-anglican*, and is no longer supported; Anglican has been replaced by a new compiled implementation [Tolpin et al., 2016], albeit one which is based around a CPS transformation rather than explicit task-copying.) The interpreted particle Gibbs engine is multithreaded, and we run it with 100 particles and 8 simultaneous threads; the Metropolis-Hastings engine only runs on a single core. In Figure 4.4 we compare inference performance in both of these existing engines to our particle Gibbs backend, running with 100 and 1000 particles, in an 8 core cloud computing environment on Amazon EC2, running on Intel Xeon E5-2680 v2 processors. Our compiled probabilistic C implementation of particle Gibbs runs over 100 times faster that the existing interpreted engine, generating good sample sets in on the order of tens of seconds.

The probabilistic C inference engine implements particle Gibbs, SMC, and PIMH sampling, which we compare in Figure 4.5 using both 100 and 1000 particles. SMC is run indefinitely by simply repeatedly drawing independent sets of particles.

Figures 4.4 and 4.5 plot wall clock time against KL-divergence. We use all generated samples as our empirical posterior distribution in order to produce as fair a comparison as possible. In all engines, results are reasonably stable across runs; the shaded band covers the 25th to 75th percentiles over multiple runs, with the median marked as a dark line. A sampler drawing from the target density will show approximately linear decrease in KL-divergence on these log-log plots; a steeper slope correspond to greater statistical efficiency. The methods

[1] https://github.com/dritchie/probabilistic-js

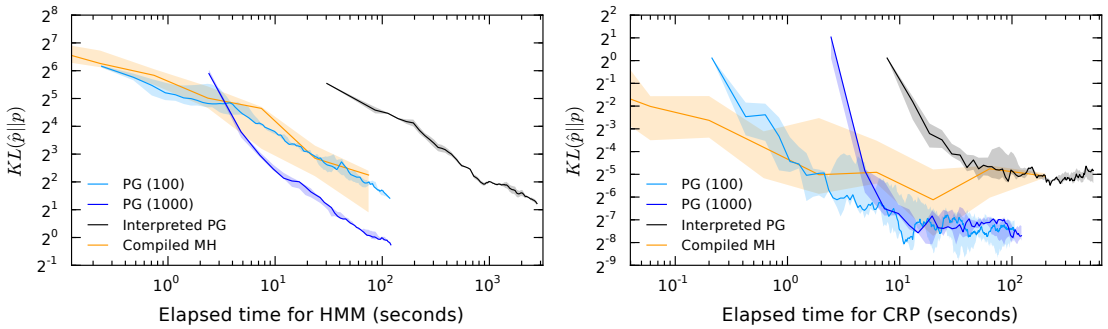


Figure 4.4: Performance plots for (left) HMM and (right) CRP models, run in an 8 core computing environment on Amazon EC2. In both models we see that compiling particle Gibbs, compared to running within an interpreter, leads to a large (approx. 100×) constant-factor speed increase. The Metropolis-Hastings sampler converges at a similar rate as particle Gibbs with 100 particles in the HMM, but appears to mix slower asymptotically for the CRP. In both models, increasing to 1000 particles in particle Gibbs yields somewhat faster convergence at the expense of a longer waiting time until the first sample set arrives.

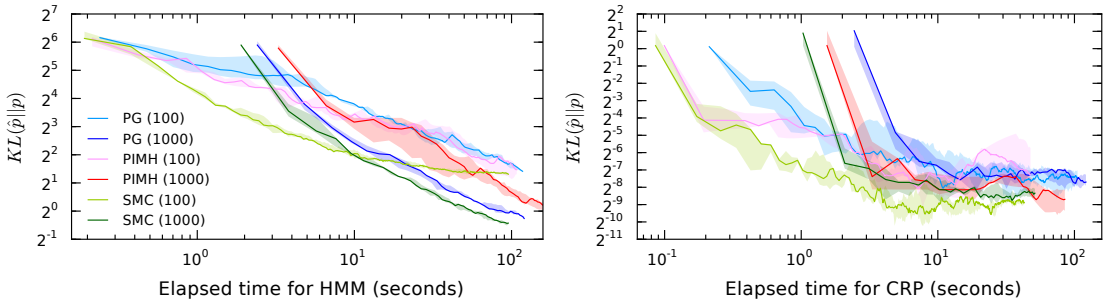


Figure 4.5: Comparison of SMC, PIMH, and PG for 100 and 1000 particles in (left) the HMM, and (right) the CRP. The relative computational and statistical efficiency of the PIMH and PG algorithms varies across models and number of particles. The SMC algorithm is quick to draw initial samples, but if run with too few particles can repeatedly fail to locate high-probability regions of the target density; we see this in its apparently poor behavior in the HMM with 100 particles. Colors are the same across plots.

based on sequential Monte Carlo do not provide any estimate of the posterior distribution until completing a single initial particle filter sweep; for large numbers of particles this may be a non-trivial amount of time. In contrast, the MH sampler begins drawing samples effectively immediately, although it may take a large number of individual steps before converging to the correct stationary distribution; individual Metropolis-Hastings samples are likely to be more autocorrelated, but producing each one is faster.

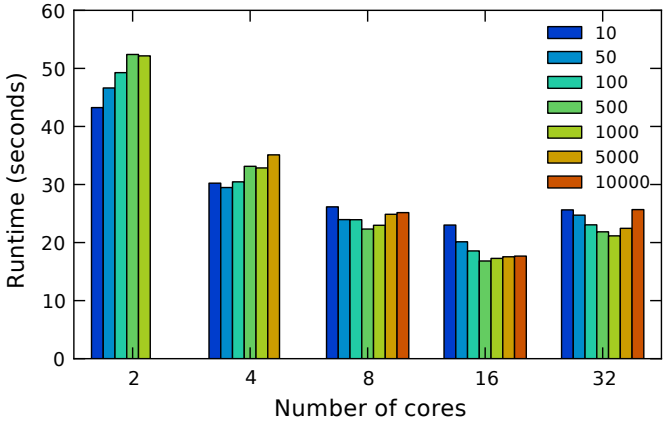


Figure 4.6: Effect of system architecture on runtime performance. Here we run the HMM code on EC2 instances with identical processors (horizontal axis) with varying number of particles (individual bars) and report runtime to produce 10,000 samples. Despite adding more cores, after 16 cores performance begins to degrade. Similarly, adding more particles eventually degrades performance for any fixed number of cores. In combination these suggest the availability of operating system optimizations that could improve overall performance.

4.4.2 Performance characteristics across multiple cores

As the probabilistic C inference engine offloads much of the computation to underlying operating system calls, we characterize the limitations of the OS implementation by comparing time to completion as we vary the number of cores. Tests for the hidden Markov model across core count (all on EC2, all with identical Intel Xeon E5-2680 v2 processors) are shown in Figure 4.6.

4.5 Discussion

Probabilistic C is a method for performing inference in probabilistic programs. Methodologically it derives from the forward methods for performing inference in statistical models based on sequential Monte Carlo and particle Markov chain Monte Carlo. We have shown that it is possible to efficiently and scalably implement this particular kind of inference strategy using existing, standard compilers and POSIX compliant operating system primitives.

What most distinguishes Probabilistic C from prior art is that it is highly compatible with modern computer architectures, all the way from operating systems

to central processing units (in particular their virtual memory operations), and, further, that it delineates a future research program for scaling the performance of probabilistic programming systems to large scale problems by investigating systems optimizations of existing computer architectures. Note that this is distinct but compatible with approaches to optimizing probabilistic programming systems by compilation optimizations, stochastic hardware, and dependency tracking with efficient updating of local execution trace subgraphs. It may, in the future, be possible to delineate model complexity and hardware architecture regimes in which each approach is optimal; we assert that, for now, it is unclear what those regimes are or will be.

Several interesting research questions remain: (1) Is it more sensible to write custom memory management and use threads than fork and processes as we have done? The main contribution of this chapter is to establish a probabilistic programming system implementation against a standardized, portable abstraction layer. It might be possible to eke out greater performance by capitalizing on the fact modern architectures are optimised for parallel threads more so than parallel processes; however, exploiting this would entail implementing memory management de facto equivalent in action to fork which may lead to lower portability. (2) Would shifting architectures to small page sizes help? There is a bias towards large page size support in modern computer architectures. It may be that the system use characteristics of probabilistic programming systems might provide a counterargument to that bias or inspire the creation of tuned end-to-end systems. Forking itself is lightweight until variable assignment which usually require manipulations of entire page tables. Large pages require large amounts of amortization in order to absorb the cost of copying upon stochastic variable assignment. Smaller pages could potentially yield higher efficiencies. (3) What characteristics of process synchronisation can be improved specifically for probabilistic programming systems? This is both a systems and machine learning question. From a machine learning perspective we believe it may be possible to construct efficient sequential Monte Carlo algorithms that do not synchronize individual threads at observe barriers and instead synchronize in a

queue. On a systems level it begs questions about what page replacement strategies to consider; perhaps entirely changing the page replacement schedule to reflect rapid process rather than thread multiplexing across cores.

Since 2014, only the first of these has been particularly addressed. Programming languages constructs such as delimited continuations [Felleisen, 1988] can be thought of as user level abstractions of fork, and might provide similar functionality at the user rather than system level in the context of statistically safer languages. The probabilistic programming language Turing.jl[2] is embedded in Julia, and implements a user-space task cloning procedure which lazily copies memory contents after cloning in a manner similar to that performed by the `fork` operation; however, each particle runs as a lightweight green thread, meaning that there is less overhead due to context-switching significantly less overall memory usage; in particular, there is not the stark drop-off in performance as the number of simultaneous processes becomes too large for the hardware.

Probabilistic C does not protect programmers from accidentally writing programs that are statistically incoherent. Many probabilistic programming languages (including Church, Anglican, and Venture) are nearly purely functional and as such disallow program variable value reassignment. This ensures a well-defined joint distribution on program variables. Probabilistic C offers no such guarantees. For this reason we do not recommend probabilistic C as a new probabilistic programming language per se — rather we describe it as an intermediate representation compilation target that implements a particular style of inference that is natively parallel and possible to optimize by system architecture choice. If optimizing system architecture is infeasible, then perhaps such a target can be implemented with a user-level implementation of "safe" forking. Since 2014, there has been some work in adapting existing compilers which use C as an intermediate language construct probabilistic variants by wrapping and exposing the probabilistic C functions for generating random numbers and weighting the execution trace; Jeff Siskind has used the probabilistic C library to implement a probabilistic scheme using the

[2]https://github.com/yebai/Turing.jl

Stalin Scheme-to-C compiler, a probabilistic ML using the MLton compiler, and a probabilistic variant of Haskell.

Finally, probabilistic C has seen some real-world usage. In Houlsby [2016], probabilistic C is used to infer posterior distributions over inputs to a undersea soil pressure modeling system used for monitoring offshore oil mobile jack-up rigs.

5

Asynchronous anytime sequential Monte Carlo

In this chapter we introduce a new sequential Monte Carlo algorithm we call the *particle cascade*, and implement it in probabilistic C. The particle cascade is an asynchronous, anytime alternative to traditional sequential Monte Carlo algorithms that is amenable to parallel and distributed implementations. It uses no barrier synchronizations which leads to improved particle throughput and memory efficiency. It is an anytime algorithm in the sense that it can be run forever to emit an unbounded number of particles while keeping within a fixed memory budget. The particle cascade provides an unbiased marginal likelihood estimator which can be straightforwardly plugged into existing pseudo-marginal methods.

The sequential Monte Carlo inference techniques we looked at in the previous chapter require blocking barrier synchronizations at resampling steps which limit parallel throughput and are costly in terms of memory. We introduce a new asynchronous anytime sequential Monte Carlo algorithm that has statistical efficiency competitive with standard SMC algorithms and has sufficiently higher particle throughput such that it is on balance more efficient per unit computation time. Our approach uses locally-computed decision rules for each particle that do not require block synchronization of all particles, instead only sharing of summary statistics

with particles that follow. In this new algorithm each resampling point acts as a queue rather than a barrier: each particle chooses the number of its own offspring by comparing its own weight to the weights of particles which previously reached the queue, blocking only to update summary statistics before proceeding.

An anytime algorithm is an algorithm that can be run continuously, generating progressively better solutions when afforded additional computation time. Traditional particle-based inference algorithms are not anytime in nature; all particles need to be propagated in lock-step to completion in order to compute expectations. Once a particle set runs to termination, inference cannot straightforwardly be continued by simply doing more computation. Particle Markov chain Monte Carlo methods (i.e. particle Metropolis Hastings and iterated conditional sequential Monte Carlo (iCSMC) [Andrieu et al., 2010]) for correctly merging particle sets produced by additional SMC runs are closer to anytime in nature but suffer from burstiness as big sets of particles are computed then emitted at once and, fundamentally, the inner-SMC loop of such algorithms still suffers the kind of excessive synchronization performance penalty that the particle cascade directly avoids. Our asynchronous SMC algorithm, the particle cascade, is anytime in nature. The particle cascade can be run indefinitely, without resorting to merging of particle sets.

5.1 Related work

Our algorithm shares a superficial similarity to Bernoulli branching numbers [Crisan et al., 1999] and other search and exploration methods used for particle filtering, where each particle samples some number of children to propagate to the next observation. Like the particle cascade, the total number of particles which exist at each generation is allowed to gradually increase and decrease. However, computing branching correction numbers is generally a synchronous operation, requiring all particle weights to be known in order to choose an appropriate number of offspring; nor are these methods anytime. Sequentially interacting Markov chain Monte Carlo [Brockwell et al., 2010, Moral and Doucet, 2010] is an anytime algorithm, which although conceptually similar to SMC has different synchronization properties.

Parallelizing the resampling step of sequential Monte Carlo methods has drawn increasing recent interest as the effort progresses to scale up algorithms to take advantage of high-performance computing systems and GPUs. Removing the global collective resampling operation [Murray et al., 2014] is a particular focus for improving performance.

Running arbitrarily many particles within a fixed memory budget can also be addressed by tracking random number seeds used to generate proposals, allowing particular particles to be deterministically "replayed" [Jun and Bouchard-Côté, 2014]. However, this approach is not asynchronous nor anytime.

5.2 Background and notation

We begin by briefly reviewing sequential Monte Carlo as generally formulated on state-space models. For the duration of this chapter, we use slightly different notation to better differentiate between particle weights at different stages of the algorithm; we introduce this notation here. Suppose we have a non-Markovian dynamical system with latent random variables $X_0, \ldots, X_N$ and observed random variables $Y_0, \ldots, Y_N$ described by the joint density

$$\begin{aligned} p(x_n | x_{0:n-1}, y_{0:n-1}) &= f(x_n | x_{0:n-1}) \\ p(y_n | x_{0:n}, y_{0:n-1}) &= g(y_n | x_{0:n}), \end{aligned} \tag{5.1}$$

where X_0 is drawn from some initial distribution $\mu(\cdot)$, and f and g are conditional densities.

Given observed values $Y_{0:N} = y_{0:N}$, the posterior distribution $p(x_{0:n}|y_{0:n})$ is approximated by a weighted set of K particles, with each particle k denoted $X^k_{0:n}$ for $k = 1, \ldots, K$. Particles are propagated forward from proposal densities $q(x_n|x_{0:n-1})$ and re-weighted at each $n = 1, \ldots, N$

$$X^k_n | X^k_{0:n-1} \sim q(x_n | X^k_{0:n-1}) \tag{5.2}$$

$$w^k_n = \frac{g(y_n | X^k_{0:n}) f(X^k_n | X^k_{0:n-1})}{q(X^k_n | X^k_{0:n-1})} \tag{5.3}$$

$$W^k_n = W^k_{n-1} w^k_n, \tag{5.4}$$

where w_n^k is the weight associated with observation y_n and W_n^k is the unnormalized weight of particle k after observation n. It is assumed that exact evaluation of $p(x_{0:N}|y_{0:N})$ is intractable and that the likelihoods $g(y_n|X_{0:n}^k)$ can be evaluated pointwise. In many complex dynamical systems, or in black-box simulation models, evaluation of $f(X_n^k|X_{0:n-1}^k)$ may be prohibitively costly or even impossible. As long as one is capable of simulating from the system, the proposal distribution can be chosen as $q(\cdot) \equiv f(\cdot)$, in which case the particle weights are simply $w_n^k = g(y_n|X_{0:n}^k)$, eliminating the need to compute the densities $f(\cdot)$.

The normalized particle weights $\bar{\omega}_n^k = W_n^k / \sum_{j=1}^K W_n^j$ are used to approximate the posterior

$$\hat{p}(x_{0:n}|y_{0:n}) \approx \sum_{k=1}^{K} \bar{\omega}_n^k \delta_{X_{0:n}^k}(x_{0:n}). \tag{5.5}$$

In the very simple sequential importance sampling setup described here, the marginal likelihood can be estimated by $\hat{p}(y_{0:n}) = \frac{1}{K} \sum_{k=1}^K W_n^k$.

5.2.1 Resampling and degeneracy

The algorithm described above suffers from a degeneracy problem wherein most of the normalized weights $\bar{\omega}_n^1, \ldots, \bar{\omega}_n^K$ become very close to zero for even moderately large n. Traditionally this is combated by introducing a resampling step: as we progress from n to $n+1$, particles with high weights are duplicated and particles with low weights are discarded, preventing all the probability mass in our approximation to the posterior from accumulating on a single particle. A resampling scheme is an algorithm for selecting the number of offspring particles M_{n+1}^k that each particle k will produce after stage n. Many different schemes for resampling particles exist; see [Douc et al., 2005] for an overview. Resampling changes the weights of particles: as the system progresses from n to $n+1$, each of the M_{n+1}^k children are assigned a new weight V_{n+1}^k, replacing the previous weight W_n^k prior to resampling. Most resampling schemes generate an unweighted set of particles with $V_{n+1}^k = 1$ for all particles. When a resampling step is added at every n, the marginal likelihood can be estimated by $\hat{p}(y_{0:n}) = \prod_{i=0}^n \frac{1}{K} \sum_{k=1}^K w_i^k$; this estimate of the marginal likelihood is unbiased [Del Moral, 2004].

5.2.2 Synchronization and limitations

Our goal is to scale up to very large numbers of particles, using a parallel computing architecture where each particle is simulated as a separate process or thread. In order to resample at each n we must compute the normalized weights $\bar{\omega}_n^k$, requiring us to wait until all individual particles have both finished forward simulation and computed their individual weight W_n^k before the normalization and resampling required for any to proceed. While the forward simulation itself is trivially parallelizable, the weight normalization and resampling step is a synchronous, collective operation. In practice this can lead to significant underuse of computing resources in a multiprocessor environment, hindering our ability to scale up to large numbers of particles.

Memory limitations on finite computing hardware also limit the number of simultaneous particles we are capable of running in practice. All particles must move through the system together, simultaneously; if the total memory requirements of particles is greater than the available system RAM, then a substantial overhead will be incurred from swapping memory contents to disk.

5.3 The particle cascade

The particle cascade algorithm we introduce addresses both these limitations: it does not require synchronization, and keeps only a bounded number of particles alive in the system at any given time. Instead of resampling, we will consider particle branching, where each particle may produce 0 or more offspring. These branching events happen asynchronously and mutually exclusively, i.e. they are processed one at a time.

5.3.1 Local branching decisions

At each stage n of sequential Monte Carlo, particles process observation y_n. Without loss of generality, we can define an ordering on the particles $1, 2, \ldots$ in the order they arrive at y_n. We keep track of the running average weight $\overline{W}_n^k$ of the first k

particles to arrive at observation y_n in an online manner

$$\overline{W}_n^k = W_n^k \qquad \text{for } k = 1, \tag{5.6}$$

$$\overline{W}_n^k = \frac{k-1}{k}\overline{W}_n^{k-1} + \frac{1}{k}W_n^k \qquad \text{for } k = 2, 3, \ldots. \tag{5.7}$$

The number of children of particle k depends on the weight W_n^k of particle k relative to those of other particles. Particles with higher relative weight are more likely to be located in a high posterior probability part of the space, and should be allowed to spawn more child particles.

In our online asynchronous particle system we do not have access to the weights of future particles when processing particle k. Instead we will compare W_n^k to the current average weight $\overline{W}_n^k$ among particles processed thus far. Specifically, the number of children, which we denote by M_{n+1}^k, will depend on the ratio

$$R_n^k = \frac{W_n^k}{\overline{W}_n^k}. \tag{5.8}$$

Each child of particle k will be assigned a weight V_{n+1}^k such that the total weight of all children $M_{n+1}^k V_{n+1}^k$ has expectation W_n^k.

There is a great deal of flexibility available in designing a scheme for choosing the number of child particles; we need only be careful to set V_{n+1}^k appropriately. Informally, we would like M_{n+1}^k to be large when R_n^k is large. If M_{n+1}^k is sampled in such a way that $\mathbb{E}[M_{n+1}^k] = R_n^k$, then we set the outgoing weight $V_{n+1}^k = \overline{W}_n^k$. Alternatively, if we are using a scheme which deterministically guarantees $M_{n+1}^k > 0$, then we set $V_{n+1}^k = W_n^k / M_{n+1}^k$.

A simple approach would be to sample M_{n+1}^k independently conditioned on the weights. In such schemes we could draw each M_{n+1}^k from some simple distribution, e.g. a Poisson distribution with mean R_n^k, or a discrete distribution over the integers $\{\lfloor R_n^k \rfloor, \lceil R_n^k \rceil\}$. For example, if using a discrete distribution over the two neighboring integers, i.e. a Bernoulli($R_n^k - \lfloor R_n^k \rfloor$) choice between $\{\lfloor R_n^k \rfloor, \lceil R_n^k \rceil\}$, we have

$$(M_{n+1}^k, V_{n+1}^k) = \begin{cases} (\lfloor R_n^k \rfloor, \overline{W}_n^k) \text{ w.p. } \lceil R_n^k \rceil - R_n^k, \\ (\lceil R_n^k \rceil, \overline{W}_n^k) \text{ w.p. } R_n^k - \lfloor R_n^k \rfloor \end{cases} \tag{5.9}$$

in which the outgoing weights are deterministically $\overline{W}_n^k$. This decision rule defines the same distribution on the number of offspring particles M_{n+1}^k as the well-known (and well-performing) systematic resampling procedure [Carpenter et al., 1999, Murray et al., 2014].

5.3.2 Variance reduction

One issue that arises in such approaches where the number of children for each particle is conditionally independent is that the variance of the total number of particles at each generation can grow faster than desirable. Suppose we start the system with K_0 particles. The number of particles at subsequent stages n is given recursively as $K_n = \sum_{k=1}^{K_{n-1}} M_n^k$. We would like to avoid situations in which the number of particles becomes too large, or collapses to 1.

Instead, we will allow M_n^k to depend on the number of children of previous particles at n, in such a way that we can stabilize the total number of particles in each generation. Suppose that we wish for the number of particles to be stabilized around K_0. After $k-1$ particles have been processed, we expect the total number of children produced at that point to be approximately $k-1$, so that if the number is less than $k-1$ we should allow particle k to produce more children, and vice versa. Similarly, if we already currently have more than K_0 children, we should allow particle k to produce fewer children.

We use a simple scheme which satisfies these criteria, where the number of particles is chosen at random when $R_n^k < 1$, and set deterministically when $R_n^k \geq 1$

$$(M_{n+1}^k, V_{n+1}^k) = \begin{cases} (0,0) \text{ w.p. } 1 - R_n^k, & \text{if } R_n^k < 1; \\ (1, \overline{W}_n^k) \text{ w.p. } R_n^k, & \text{if } R_n^k < 1; \\ (\lfloor R_n^k \rfloor, \frac{W_n^k}{\lfloor R_n^k \rfloor}) & \text{if } R_n^k \geq 1 \text{ and } \sum_{j=1}^{k-1} M_{n+1}^j > \min(K_0, k-1); \\ (\lceil R_n^k \rceil, \frac{W_n^k}{\lceil R_n^k \rceil}) & \text{if } R_n^k \geq 1 \text{ and } \sum_{j=1}^{k-1} M_{n+1}^j \leq \min(K_0, k-1). \end{cases} \tag{5.10}$$

As the number of particles becomes large, the estimated average weight closely approximates the true average weight.

Note the anytime nature of this algorithm — any given particle passing through the system needs only the running average $\overline{W}_n^k$ and the preceding child particle

counts $\sum_{j=1}^{k-1} M_{n+1}^j$ in order to make local branching decisions, not the previous particles themselves. Thus it is possible to run this algorithm for some fixed number of initial particles K_0, inspect the output of the completed particles which have left the system, and decide whether to continue by initializing additional particles.

5.3.3 Computing expectations and marginal likelihoods

Samples drawn from the particle cascade can be used to compute expectations in the same manner as usual; that is, given some function $\psi(\cdot)$, we normalize weights $\bar{\omega}_n^k = W_n^k / \sum_{j=1}^{K_n} W_n^j$ and approximate the posterior expectation by $\mathbb{E}[\psi(X_{0:n})|y_{0:n}] \approx \sum_{k=1}^{K_n} \bar{\omega}_n^k \psi(X_{0:n}^k)$.

We can also use the particle cascade to define an estimator of the marginal likelihood $p(y_{0:n})$,

$$\hat{p}(y_{0:n}) = \frac{1}{K_0} \sum_{k=1}^{K_n} W_n^k. \tag{5.11}$$

The form of this estimate is fairly distinct from the standard SMC estimators in Section 5.2. One can think of $\hat{p}(y_{0:n})$ as $\hat{p}(y_{0:n}) = \hat{p}(y_0) \prod_{i=1}^{n} \hat{p}(y_i|y_{0:i-1})$ where

$$\hat{p}(y_0) = \frac{1}{K_0} \sum_{k=1}^{K_0} W_0^k, \qquad \hat{p}(y_n|y_{0:n-1}) = \frac{\sum_{k=1}^{K_n} W_n^k}{\sum_{k=1}^{K_{n-1}} W_{n-1}^k} \text{ for } n \geq 1. \tag{5.12}$$

Note that the incrementally updated running averages $\overline{W}_n^k$ are very directly tied to the marginal likelihood estimate; that is, $\hat{p}(y_{0:n}) = \frac{K_n}{K_0} \overline{W}_n^k$.

5.3.4 Theoretical properties, unbiasedness, and consistency

Under weak assumptions we can show that the marginal likelihood estimator $\hat{p}(y_{0:n})$ defined in Eq. (5.11) is unbiased, and that both its variance and L2 errors of estimates of reasonable posterior expectations decrease in the number of particle initializations as $1/K_0$. Note that because the cascade is an anytime algorithm K_0 may be increased simply, without restarting inference. Detailed proofs are given in the supplemental material of Paige et al. [2014]; statements of the results are provided here. Denote by $B(E)$ the space of bounded real-valued functions on a space E, suppose each X_n is an $\mathcal{X}$-valued random variable, and assume

that $g(y_n|\cdot, y_{0:n-1})\colon \mathcal{X}^{n+1} \to \mathbb{R}$ is in $B(\mathcal{X}^{n+1})$ and strictly positive. For either the deterministic sampling scheme of Equation (5.10) or the stochastic resampling in Eq. (5.9), the algorithm provides an unbiased estimate of the normalizing constant:

Proposition 1 (Unbiasedness of marginal likelihood estimate) *For any* $K_0 \geq 1$ *and* $n \geq 0$

$$\mathbb{E}\left[\hat{p}(y_{0:n})\right] = p(y_{0:n}). \tag{5.13}$$

Under the Bernoulli$(R_n^k - \lfloor R_n^k \rfloor)$ version of the resampling rule in Eq. (5.9), further assume that the ordering in which particles arrive at each n is a random permutation of the particle index set. Then the following additional propositions characterizing convergence of the algorithm hold:

Proposition 2 (Variance of marginal likelihood estimate) *For any* $n \geq 0$*, there exists a constant* $a_n < \infty$ *such that for any* $K_0 \geq 1$

$$\mathbb{V}\left[\hat{p}(y_{0:n})\right] \leq \frac{a_n}{K_0}. \tag{5.14}$$

Proposition 3 (L2 error bounds) *For any* $n \geq 0$*, there exists a constant* $\overline{a}_n < \infty$ *such that for any* $K_0 \geq 1$ *and any* $\psi_n \in B\left(\mathcal{X}^{n+1}\right)$

$$\mathbb{E}\left[\left\{\left(\sum_{k=1}^{K_n} \bar{\omega}_n^k \psi_n(X_{0:n}^k)\right) - \int p(dx_{0:n}|y_{0:n})\psi_n(x_{0:n})\right\}^2\right] \leq \frac{\overline{a}_n}{K_0}\left\|\psi_n\right\|^2. \tag{5.15}$$

Additional results and proofs can be found in Paige et al. [2014].

5.4 Active bounding of memory usage

In an idealized computational environment, with infinite available memory, our implementation of the particle cascade could begin by launching (a very large number) K_0 particles simultaneously which then gradually propagate forward through the system. In practice, only some finite number of particles, probably much smaller than K_0, can be simultaneously simulated efficiently. Furthermore, the initial particles are not truly launched all at once, but rather in a sequence, introducing a dependency in the order in which particles arrive at each observation n.

Our implementation of the particle cascade addresses these issues by explicitly injecting randomness into the execution order of particles, and by imposing a machine-dependent hard cap on the number of simultaneous extant processes. This permits us to run our particle filter system indefinitely, for arbitrarily large and, in fact, growing initial particle counts K_0, on fixed commodity hardware.

Each particle in our implementation runs as an independent operating system process [Paige and Wood, 2014]. In order to efficiently run a large number of particles, we impose a hard limit ρ on the total number of particles which can simultaneously exist in the particle system; most of these will generally be sleeping processes. The ideal choice for this number will vary based on hardware capabilities, but in general should be made as large as possible.

Scheduling across particles is managed via a global *first-in random-out* process queue of length ρ; this can equivalently be conceptualized as a random-weight priority queue. Each particle corresponds to a single live process, augmented by a single additional control process which is responsible only for spawning additional initial particles (i.e. incrementing the initial particle count K_0). When any particle k arrives at any likelihood evaluation n, it computes its target number of child particles M_{n+1}^k and outgoing particle weight V_{n+1}^k. If $M_{n+1}^k = 0$ it immediately terminates; otherwise it enters the queue. Once this particle either enters the queue or terminates, some other process continues execution — this process is chosen uniformly at random, and as such may be a sleeping particle at any stage $n < N$, or it may instead be the control process which then launches a new particle. At any given time, there are some number of particles $K_\rho < \rho$ currently in the queue, and so the probability of resuming any particular individual particle, or of launching a new particle, is $1/(K_\rho + 1)$. If the particle released from the queue has exactly one child to spawn, it advances to the next observation and repeats the resampling process. If, however, a particle has more than one child particle to spawn, rather than launching all child particles at once it launches a single particle to simulate forward, decrements the total number of particles left to launch by one, and itself re-enters the queue. The system is initialized by seeding the system with a number of initial

particles $\rho_0 < \rho$ at $n = 0$, creating ρ_0 active initial processes. The ideal choice for the process count constraint ρ may vary across operating systems and hardware.

In the event that the process count is fully saturated (i.e. the process queue is full), then we forcibly prevent particles from duplicating themselves and creating new children. If we release a particle from the queue which seeks to launch $m > 1$ additional particles when the queue is full, we instead collapse all the remaining particles into a single particle; this single particle represents a virtual set of particles, but does not create a new process and requires no additional CPU or memory resources. We keep track of a particle count multiplier C_n^k that we propagate forward along with the particle. All particles are initialized with $C_0^k = 1$, and then when a particle collapse takes place, update their multiplier at $n+1$ to mC_n^k. This affects the way in which running weight averages are computed; suppose a new particle k arrives with multiplier C_n^k and weight W_n^k. We incorporate all these values into the average weight immediately, and update $\overline{W}_n^k$ taking into account the multiplicity, with

$$\overline{W}_n^k = \frac{k-1}{k + C_n^k - 1}\overline{W}_n^{k-1} + \frac{C_n^k}{k + C_n^k - 1}W_n^k \qquad \text{for } k = 2, 3, \ldots. \tag{5.16}$$

This does not affect the computation of the ratio R_n^k. We preserve the particle multiplier, until we reach the final $n = N$; then, after all forward simulation is complete, we re-incorporate the particle multiplicity when reporting the final particle weight $W_N^k = C_N^k V_N^k w_N^k$.

5.5 Particle cascade experiments

We report experiments on performing inference in two simple state space models, each with $N = 50$ observations, in order to demonstrate the overall validity and utility of the particle cascade algorithm. The first is a hidden Markov model (HMM) with 10 latent discrete states, each with an associated Gaussian emission distribution; the second a one-dimensional linear Gaussian model. Note that using these models means that we can compute posterior marginals at each n and the marginal likelihood $Z = p(y_{0:N})$ exactly.

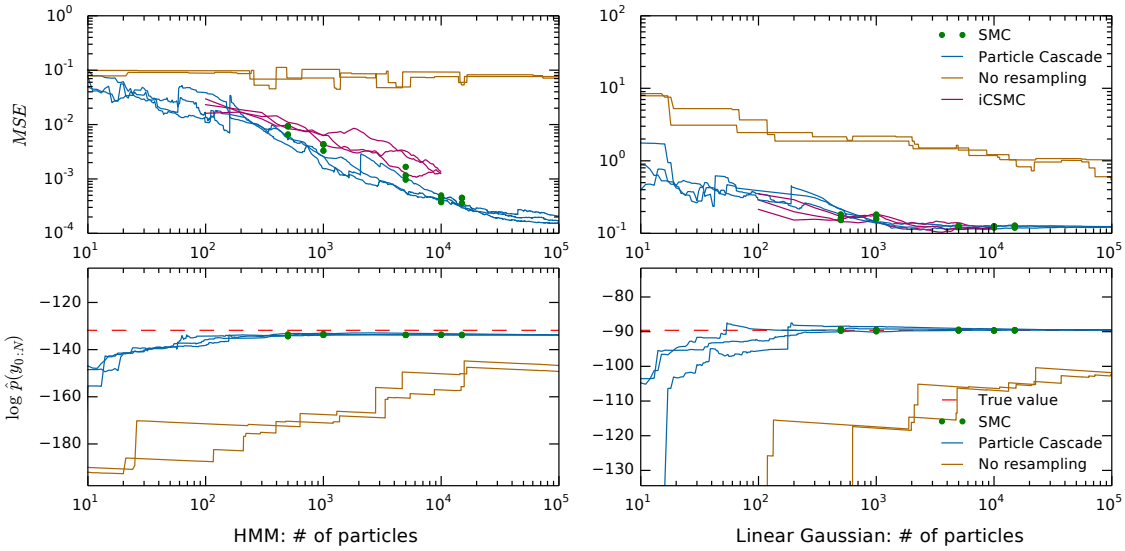


Figure 5.1: All results are reported over multiple independent replications, shown here as independent lines. (top) Convergence of estimates to ground truth vs. number of particles, shown as (left) MSE of marginal probabilities of being in each state for every observation n in the HMM, and (right) MSE of the latent expected position in the linear Gaussian state space model. (bottom) Convergence of marginal likelihood estimates to the ground truth value (marked by a red dashed line), for (left) the HMM, and (right) the linear Gaussian model.

These experiments are not designed to stress-test the particle cascade; rather, they are designed to show that performance of the particle cascade closely approximates that of fully synchronous SMC algorithms, even in a small-data small-complexity regime where we expect their performance to be very good. In addition to comparing to standard SMC, we also compare to a worst-case particle filter in which we never resample, instead propagating particles forward deterministically with a single child particle at every n. While the statistical (per-sample) efficiency of this approach is quite poor, it is fully parallelizable with no blocking operations in the algorithm at all, and thus provides a ceiling estimate of the raw sampling speed attainable in our overall implementation.

We also benchmark against what we believe to be the most practically competitive similar approach, iterated conditional SMC [Andrieu et al., 2010]. Iterated conditional SMC corresponds to the particle Gibbs algorithm in the case where parameter values are known; by using a particle filter sweep as a step within a

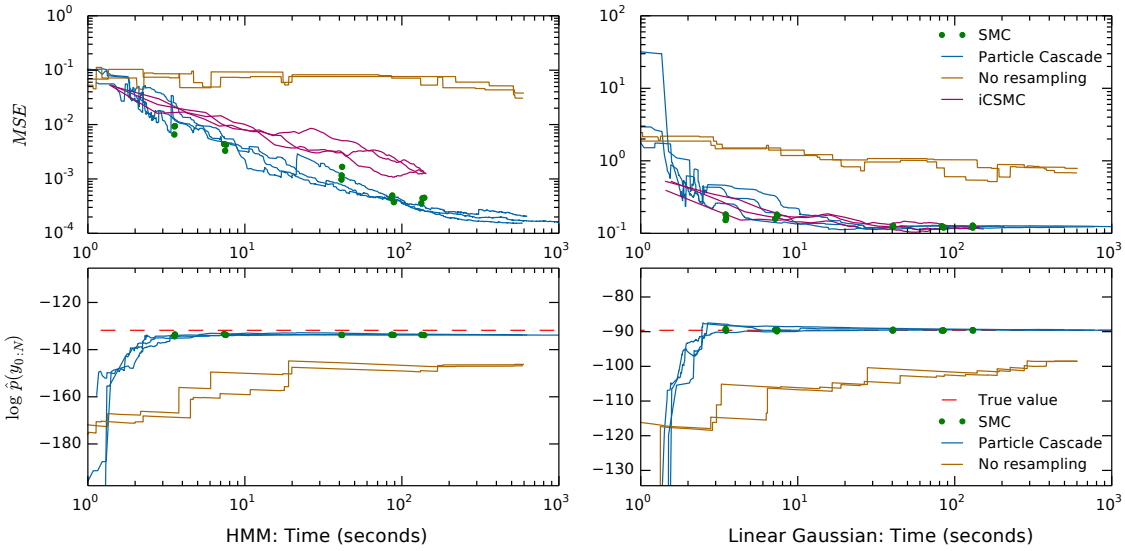


Figure 5.2: (top) Comparative convergence rates between SMC alternatives including our new algorithm, and (bottom) estimation of marginal likelihood, by time. Results are shown for (left) the hidden Markov model, and (right) the linear Gaussian state space model.

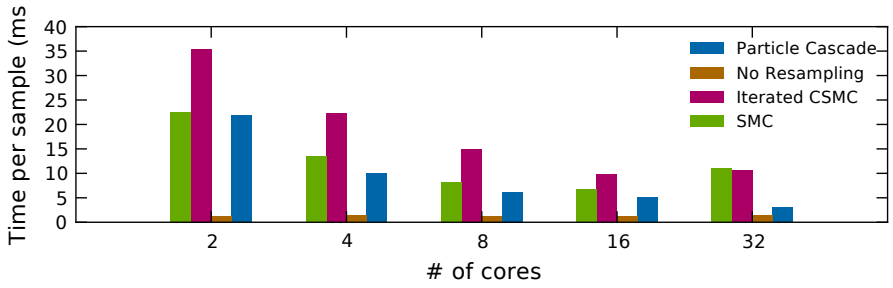


Figure 5.3: Average time to draw a single complete particle on a variety of machine architectures. Queueing rather than blocking at each observation improves performance, and appears to improve relative performance even more as the available compute resources increase. Note that this plot shows only average time per sample, not a measure of statistical efficiency. The high speed of the non-resampling algorithm is not sufficient to make it competitive with the other approaches.

larger MCMC algorithm, iCSMC provides a statistically valid approach to sampling from a posterior distribution by repeatedly running sequential Monte Carlo sweeps each with a fixed number of particles. One downside to iCSMC is that it does not provide an estimate of the marginal likelihood. In all benchmarks, we propose from the prior distribution, with $q(x_n|\cdot) \equiv f(x_n|x_{0:n-1})$; the SMC and iCSMC benchmarks use a multinomial resampling scheme.

On both these models we see the statistical efficiency of the particle cascade is

approximately in line with synchronous SMC, slightly outperforming the iCSMC algorithm and significantly outperforming the fully parallelized non-resampling approach. This suggests that the approximations made by computing weights at each n based on only the previously observed particles, and the total particle count limit imposed by ρ, do not have an adverse effect on overall performance. In Fig. 5.1 we plot convergence per particle to the true posterior distribution, as well as convergence in our estimate of the normalizing constant.

5.5.1 Performance and scalability

Although values will be implementation-dependent, we are ultimately interested not in per-sample efficiency but rather in our rate of convergence over time. We record wall clock time for each algorithm for both of these models; the results for convergence of our estimates of values and marginal likelihood are shown in Fig. 5.2. These particular experiments were all run on Amazon EC2, in an 8-core environment with Intel Xeon E5-2680 v2 processors. The particle cascade provides a much faster and more accurate estimate of the marginal likelihood than the competing methods, in both models. Convergence in estimates of values is quick as well, faster than the iCSMC approach. We note that for very small numbers of particles, running a simple particle filter is faster than the particle cascade, despite the blocking nature of the resampling step. This is due to the overhead incurred by the particle cascade in sending an initial flurry of ρ_0 particles into the system before we see any particles progress to the end; this initial speed advantage diminishes as the number of samples increases. Furthermore, in stark contrast to the simple SMC method, there are no barriers to drawing more samples from the particle cascade indefinitely. On this fixed hardware environment, our implementation of SMC, which aggressively parallelizes all forward particle simulations, exhibits a dramatic loss of performance as the number of particles increases from 10^4 to 10^5, to the point where simultaneously running 10^5 particles is simply not possible in a feasible amount of time.

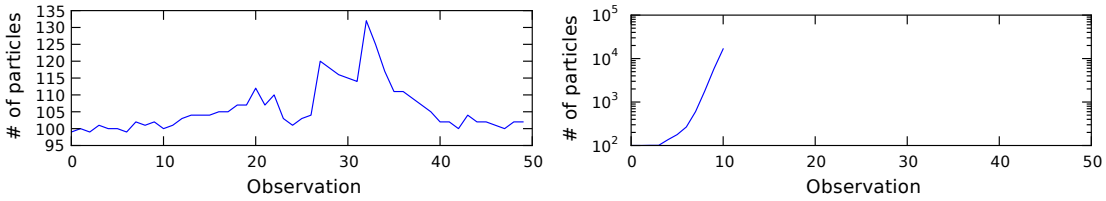


Figure 5.4: In this figure we demonstrate potential consequences when the ordering of the particles is not randomized, but rather dependent. comparing a best-case situation where the ordering of particles at n is completely independent of the ordering of particles at $n+1$, artificially subjecting the ordering of the particles to a random permutation, to a worst-case situation where the ordering of particles is completely preserved from n to $n+1$. We plot the number of particles K_n at each of $n = 1, \ldots, 50$ for a one-dimensional linear Gaussian model, initialized with 100 particles. (left) When the order of the particles arriving at each n is subject to a random permutation, then the number of particles is reasonably stable, staying at or near 100. (right) When the order of the particles arriving at each n is completely deterministic, then the total number of particles quickly explodes, in this case exceeding 15000 by $n = 11$. In practice, a naïve implementation of the incremental resampling scheme will have a very strong dependence in ordering across n — a particle which is one of the first to reach stage n is quite likely one of the first to reach stage $n+1$ as well.

We are also interested in how the particle cascade scales up to larger hardware, or down to smaller hardware. A comparison across five hardware configurations is shown in Fig. 5.3.

5.6 Discussion

The particle cascade has broad applicability to all SMC and particle filtering inference applications. For example, constructing an appropriate sequence of densities for SMC is possible in arbitrary probabilistic graphical models, including undirected graphical models; see e.g. the sequential decomposition approach of [Naesseth et al., 2014]. However, this is particularly motivated by SMC-based probabilistic programming systems [Wood et al., 2014, Paige and Wood, 2014]. The primary performance bottleneck in the inference algorithms of the preceding chapter was the barrier synchronization, something we have done away with entirely. What is more, while particle MCMC methods are particularly appropriate when there is a clear boundary that can be exploited between between parameters of interest and nuisance state variables, in higher-order probabilistic programming languages, we

often must resort to embedding the parameter values within the state trajectory itself, leaving no explicitly denominated global latent parameter variables. The particle cascade is particularly relevant in such situations. Finally, as the particle cascade yields an unbiased estimate of the marginal likelihood it can be plugged directly into PIMH, SMC^2 [Chopin et al., 2013], and other existing pseudo-marginal methods.

Future work on the particle cascade could help addresses thread or process scheduling. The randomized scheduling in our implementation here seeks to approximate randomizing the order in which all particles arrive at each individual `observe`, and works well in practice so long as the number of initial particles is sufficiently large. However, as we see in Figure 5.4, if the randomized particle ordering assumption is badly violated then the number of particles currently under execution can become impractically large, causing us to often hit our ceiling on the total number of particles.

6

Variational inference in higher-order languages

All the inference algorithms we have considered thus far compute Monte Carlo estimates of posterior expectations, by evaluating test functions at a set of sampled points: the posterior distribution is represented by a sequence of weighted samples. The primary goal of Chapters 3 through 5 has been to construct statistically and computationally efficient sampling algorithms, in hopes of producing statistically efficient sets of points in a reasonable amount of computation.

This is by no means the only possible representation of a posterior distribution, or of an approximation to the posterior. Consider the difference between *conditioned* and *unconditioned* probabilistic programs. Suppose we wish to perform posterior inference in the simple model shown in Figure 6.1, for Bayesian estimation of the parameter vector for a discrete distribution over the integers $[0, 1, 2, 3]$, given a few observations.

For this model, since the Dirichlet is the conjugate prior for the discrete (and multinomial) distributions, we can write down the posterior distribution *exactly*, itself expressed as a probabilistic program, but one which contains only `sample` statements (and no `observe` statements). The *unconditioned* program in Figure 6.2 represents the identical distribution over `p` as the program in Figure 6.1,

```
;; A probabilistic program with conditioning
(let [p (sample :p (dirichlet [1 1 1 1]))]
  (observe (discrete p) 1)
  (observe (discrete p) 1)
  (observe (discrete p) 0)
  (observe (discrete p) 0)
  (observe (discrete p) 3)
  (observe (discrete p) 3)
  (observe (discrete p) 0)
  (observe (discrete p) 0)
  (observe (discrete p) 0)
  (observe (discrete p) 1)
  p)
```

Figure 6.1: An Anglican program representing the posterior distribution of the parameter vector of a discrete distribution, given a few data points written inline. This is a *conditioned* probabilistic program, as it include `observe` statements. The distribution over the output `p` cannot be sampled from directly, without performing inference.

```
;; An equivalent unconditional probabilistic program
(let [p (sample (dirichlet [6 4 1 3]))]
  p)
```

Figure 6.2: This program defines the identical distribution over the return value `p` as the program in Figure 6.1, but does not contain any observe statements.

except that now we can sample directly by forward execution of the program. While we can not expect to write down compact representations of the posterior distribution as unconditioned probabilistic programs for any non-trivial models, this goal can motivate our approach: approximate inference in probabilistic programs is performed via program transformation, from a conditioned program to an unconditioned program. By analogy, very few models — only simple conjugate pair models like the Dirichlet-multinomial above — admit exact analytic forms of the posterior distribution.

Variational inference [Jordan et al., 1999, Wainwright and Jordan, 2008] turns the integration challenge central to Bayesian inference into an optimization problem, in which we learn the parameters of an analytic form which *approximately* represents the posterior. Instead of approximating the target distribution $\pi(\mathbf{x}) = p(\mathbf{x}|\mathbf{y})$ with a finite set of samples, variational approaches choose some approximating family of

probability distributions $q(\mathbf{x}|\lambda)$, where λ is a free parameter, and then optimizes the choice of parameter λ such that $q(\mathbf{x}|\lambda)$ well-approximates $\pi(\mathbf{x})$. Variational inference of this style applied to probabilistic programs yields a method in which the intractable distribution defined by a conditioned probabilistic program $\mathcal{P}$ is approximated by an alternative unconditioned probabilistic program $\mathcal{Q}_\lambda$. In turn, the program $\mathcal{Q}_\lambda$ defines a probability model $q(\mathbf{x}|\lambda)$ from which we can more efficiently draw samples and compute posterior expectations.

We begin by reviewing black-box variational inference [Ranganath et al., 2013, Wingate and Weber, 2013], which employs a generic stochastic gradient approach to optimizing the free parameters of $q(\mathbf{x}|\lambda)$. We demonstrate how it can be implemented as an inference backend for probabilistic programs, with some examples; we then discuss some of its advantages as well as the specific challenges faced for application to higher-order probabilistic programs. We close by noting the close connection between these variational inference approaches and methods for adaptive importance sampling [Cappé et al., 2008, Cornuet et al., 2012].

6.1 Variational Bayes

The first step in framing inference as optimization is to define an objective function, which captures a notion of similarity between the target posterior $p(\mathbf{x}|\mathbf{y})$ and the approximating distribution $q(\mathbf{x}|\lambda)$. The traditional approach is to consider a decomposition of the log marginal likelihood (or model evidence) $\log p(\mathbf{y})$, and use this to motivate minimizing the Kullback-Leibler (KL) divergence $D_{KL}(q||\pi)$. The KL divergence between two distributions is a non-negative quantity, which is zero only when the two distributions are pointwise equal almost everywhere. It is defined as

$$D_{KL}(q||\pi) = \int q(\mathbf{x}|\lambda) \log \frac{q(\mathbf{x}|\lambda)}{\pi(\mathbf{x})} \mathrm{d}\mathbf{x} \tag{6.1}$$

Note that this quantity, an expectation with respect to the distribution q, is not a metric as it is asymmetric in its arguments. Although this KL divergence is intractable (in particular, it requires pointwise evaluation of $\pi(\mathbf{x})$), we can

formulate a lower bound on the marginal likelihood which can be more convenient to work with, as

$$\begin{aligned} D_{KL}(q||\pi) &= \int q(\mathbf{x}|\lambda) \log \frac{q(\mathbf{x}|\lambda)}{\pi(\mathbf{x})} \mathrm{d}\mathbf{x} \\ &= \int q(\mathbf{x}|\lambda) \log \frac{q(\mathbf{x}|\lambda)p(\mathbf{y})}{p(\mathbf{x},\mathbf{y})} \mathrm{d}\mathbf{x} \\ &= \int q(\mathbf{x}|\lambda) \log p(\mathbf{y}) \mathrm{d}\mathbf{x} + \int q(\mathbf{x}|\lambda) \log \frac{q(\mathbf{x}|\lambda)}{p(\mathbf{x},\mathbf{y})} \mathrm{d}\mathbf{x} \\ &= \log p(\mathbf{y}) + \int q(\mathbf{x}|\lambda) \log \frac{q(\mathbf{x}|\lambda)}{p(\mathbf{x},\mathbf{y})} \mathrm{d}\mathbf{x} \end{aligned}$$

and thus we have the decomposition

$$\log p(\mathbf{y}) = D_{KL}(q||\pi) + \int q(\mathbf{x}|\lambda) \log \frac{p(\mathbf{x},\mathbf{y})}{q(\mathbf{x}|\lambda)} \mathrm{d}\mathbf{x}. \tag{6.2}$$

Now, since $\log p(\mathbf{y})$ is constant with respect to λ, and $D_{KL}(q||\pi)$ is non-negative, we define the evidence lower bound (ELBO) to be the second term

$$\mathcal{L}(\lambda) = \int q(\mathbf{x}|\lambda) \log \frac{p(\mathbf{x},\mathbf{y})}{q(\mathbf{x}|\lambda)} \mathrm{d}\mathbf{x} \tag{6.3}$$

which is a lower bound on the marginal likelihood, and as such maximizing $\mathcal{L}(\lambda)$ is then equivalent to minimizing $D_{KL}(q||\pi)$. The main convenience of this formulation is that the ELBO is not in any way a function of the intractable posterior distribution. It contains an expectation over the approximating family $q(\mathbf{x}|\lambda)$, not over the original model $\mathcal{P}$, and requires evaluating the unnormalized distribution $\gamma(\mathbf{x}) \equiv p(\mathbf{x},\mathbf{y})$, not the intractable posterior.

The second step is to define an approximating family $q(\mathbf{x}|\lambda)$. Historically this has been chosen to be a fully-factorized "mean field" approximation, in which the approximating distribution factorizes into a product over all individual latent variables as

$$q(\mathbf{x}|\lambda) = \prod_{j=1}^{M} q(x_j|\lambda_j), \tag{6.4}$$

since then, for models in the conjugate-exponential family [Wainwright and Jordan, 2008], it is possible to choose $q(\mathbf{x}|\lambda)$ such that the expectations over $\log \gamma(\mathbf{x})$ are tractable, and λ can be updated by an efficient coordinate ascent procedure. Recent

work in variational inference has aimed to improve the tightness of the lower bound by removing the factorization assumption in the approximating model, yielding a more expressive model family [Tran et al., 2015, Ranganath et al., 2016]. Models expressed as probabilistic programs will in general fall outside of the conjugate-exponential family, although we will still find it useful to make factorization assumptions in defining approximating programs $\mathcal{Q}_\lambda$.

6.1.1 Stochastic gradient variational inference

Stochastic gradient ascent provides a generic approach to maximizing the ELBO [Bottou, 2010, Hoffman et al., 2013], and we can adopt such methods here for a very broad class of probabilistic programs. The gradient of $\mathcal{L}$ is also an expectation over $q(\mathbf{x}|\lambda)$, and can be written as

$$\nabla_\lambda \mathcal{L}(\lambda) = \mathbb{E}_q \left[\log \left[\frac{p(\mathbf{x}, \mathbf{y})}{q(\mathbf{x}|\lambda)} \right] \nabla_\lambda \log q(\mathbf{x}|\lambda) \right]. \tag{6.5}$$

A Robbins-Munro stochastic approximation scheme [Robbins and Monro, 1951] for optimizing λ begins from an initial parameter estimate $\lambda^{(0)}$, which we then iteratively update with

$$\lambda^{(t+1)} = \lambda^{(t)} + \rho_t \nabla_\lambda \mathcal{L}(\lambda^{(t)}), \tag{6.6}$$

for $t = 1, 2, \ldots$ and where ρ_t form a sequence of step sizes such that

$$\sum_{t=1}^{\infty} \rho_t = \infty \qquad \text{and} \qquad \sum_{t=1}^{\infty} \rho_t^2 < \infty. \tag{6.7}$$

Following Hoffman et al. [2013], we use a step size schedule $\rho_t = \rho_0/(1+t)^{1/2}$ in all experiments, with a base learning rate of $\rho_0 = 0.1$. Ranganath et al. [2013] suggest a "black box" variational inference approach, approximating the gradient in Equation (6.5) with samples $\mathbf{x}^k \sim q(\mathbf{x}|\lambda)$ and constructing an estimator for the gradient of λ as

$$\hat{\nabla}_\lambda \mathcal{L}(\lambda) = \frac{1}{K} \sum_{k=1}^{K} \left[\log \left[\frac{p(\mathbf{x}^k, \mathbf{y})}{q(\mathbf{x}^k|\lambda)} \right] - \hat{\mathbf{b}} \right] \nabla_\lambda \log q(\mathbf{x}^k|\lambda) \tag{6.8}$$

where $\hat{\mathbf{b}}$ is a control variate, or baseline, which reduces the variance the estimator without changing its expectation [Paisley et al., 2012].

The appealing aspect of this estimator in Equation (6.8) is that it is incredibly general: it can be used for any $p(\mathbf{y}|\mathbf{x})$ and $p(\mathbf{x})$ which we can evaluate points wise, and it can be used for any differentiable approximating family q. In particular it requires no assumptions of exponential family distributions, or restrictions on how the joint $p(\mathbf{x}, \mathbf{y})$ is parameterized; it also does not require $p(\mathbf{x})$ or $p(\mathbf{x}, \mathbf{y})$ to be differentiable in any way. We merely need to be able to simulate from $q(\mathbf{x}|\lambda)$, evaluate $\nabla_\lambda \log q(\mathbf{x}|\lambda)$, and evaluate $p(\mathbf{x}, \mathbf{y})$.

A potential downside is that this gradient estimator often exhibits high variance. For the experimental validation in Ranganath et al. [2013], in addition to use of control variates a Rao-Blackwellization scheme in which gradients can be estimated by computing expectations with respect to only a small subset of the latent random variables; such a scheme is challenging to adopt here as the dependency structure between latent variables is neither known nor fixed.

6.2 Variational probabilistic programs

To implement variational inference in the probabilistic programming language framework of Chapter 3, we need to design a backend which computes a posterior approximation $q(\mathbf{x}|\lambda)$ as a side effect of program execution. Our main goal in this section is to show how a non-standard interpretation of the probabilistic program — as implemented through a custom inference backend — can compute and update a variational approximation through appropriate implementation of `sample` and `observe` checkpoints.

Procedurally, we learn an approximating program which is structurally identical to the original program, except that all `observe` statements are removed, and each `sample` statement draws from some new, alternative distribution. To do this, we need to introduce the concept of an *address* scheme, which we use to identify individual random choices consistently across multiple executions. While this can

be similar to the address schemes considered for lightweight MH, we will also permit address computations for individual random choices to be hand-specified functions.

Recall we defined the unnormalized trace probability defined by the original program in Equation (3.3) to be

$$\gamma(\mathbf{x}) = \prod_{i=1}^{N} g_i(y_i|\phi_i) \prod_{j=1}^{M} f_j(x_j|\theta_j). \tag{6.9}$$

As before for LMH, when executing the program assume each `sample` checkpoint passes to the backend not just a distribution f and a parameter θ, but also an address $\alpha \in \mathcal{A}$; instead of using the counter index j, we can refer to the distribution, parameter, and returned value as f_α, θ_α, and x_α respectively.

In programs that may not evaluate the same sequences of `sample` statements in every execution, the sequential index j is not a good identifier for a variable, since any branching statement that conditionally evaluates a `sample` statement affects the indices of all subsequent latent variables in the program. This means that a sample statement at a given point in the program source code may have a different sequential index j on different program executions. Intuitively, an address scheme should be designed such that `sample` statements which play the same "role" across multiple executions of the program should also share the same address. The minimal requirement for our address definitions is that random choices across multiple program execution traces which share the same address must share the same distribution type; this ensures that all random variables in the trace have the correct support, and that the learned parameters λ_α at each address always have the same dimension and type.

The approximating program Q_λ defines a new density

$$q(\mathbf{x}|\lambda) = \prod_{j=1}^{M} q_j(x_j|\lambda_j) \tag{6.10}$$

over the same space of latent random variables as the trace probability in Equation (3.3). A good choice of addresses means that if we learn parameters λ_α at each address, then the distribution

$$q(\mathbf{x}|\lambda) = \prod_{j=1}^{M} q_{\alpha_j}(x_{\alpha_j}|\lambda_{\alpha_j}) \tag{6.11}$$

will resemble the posterior $\pi(\mathbf{x})$, where we use α_j to denote the address of the j^{th} random choice during the execution that generated a particular $\mathbf{x}$. In terms of data structures, this can be conceptualized as a mapping: for each address α_j, we learn a parameter λ_{α_j} which hopefully leads to overall samples which are closer to the posterior than samples drawn using the parameter θ_j defined by the transition dynamics $f_j(x_j|\theta_j)$.

6.2.1 Computing the gradient during program execution

The key detail here is that since the gradient is an expectation over $q(\mathbf{x}|\lambda)$, we can estimate it through evaluation of the approximating $\mathcal{Q}_\lambda$; however, we also need to simultaneously compute $\gamma(\mathbf{x})$. Since both $\mathcal{P}$ and $\mathcal{Q}_\lambda$ contain the same fixed program structure, we can run them simultaneously, sampling according to distributions defined by the mapping in $\mathcal{Q}_\lambda$ while also evaluating the probabilities under $\mathcal{P}$.

A basic backend then could proceed by repeatedly running K executions of the program; each time we encounter a sample checkpoint, instead of sampling x_j from the program itself we sample from some variational density $q_j(x_{\alpha_j}|\lambda_{\alpha_j})$ for a current setting of λ; we initialize $\lambda_{\alpha_j} = \theta_{\alpha_j}$ such that on the first execution of the program we draw from the transition densities $f_j(x_j|\theta_j)$. Then, as the gradient in Equation (6.8) is an expectation over q, we can compute it componentwise as

$$\hat{\nabla}_{\lambda_\alpha}\mathcal{L}(\lambda) = \frac{1}{K_\alpha}\sum_{k=1}^{K_\alpha}\log\left[\frac{f_\alpha(x_\alpha^k|\theta_\alpha)}{q_\alpha(x_\alpha^k|\lambda_\alpha)} - \hat{b}_\alpha\right]\nabla_{\lambda_\alpha}\log q_\alpha(x_\alpha^k|\lambda_\alpha). \tag{6.12}$$

These K samples taken together give an estimate of the gradient $\nabla_\lambda\mathcal{Q}_\lambda$; we then update the parameters λ, and repeat. Note that each component is averaged over only the $K_\alpha \leq K$ executions of the program which sampled a random variate at the particular address α.

When evaluating the program $\mathcal{P}$, each `sample` checkpoint has an address α, and we need to store the proposal probability $\log q(x_\alpha|\lambda_\alpha)$, the model probability $\log f(x_\alpha|\theta_\alpha)$, and the local proposal gradient $\nabla_{\lambda_\alpha}\log q(x_\alpha|\lambda_\alpha)$; execution continues by passing a value sampled from $q(x_\alpha|\lambda_\alpha)$ to the continuation. At each `observe`

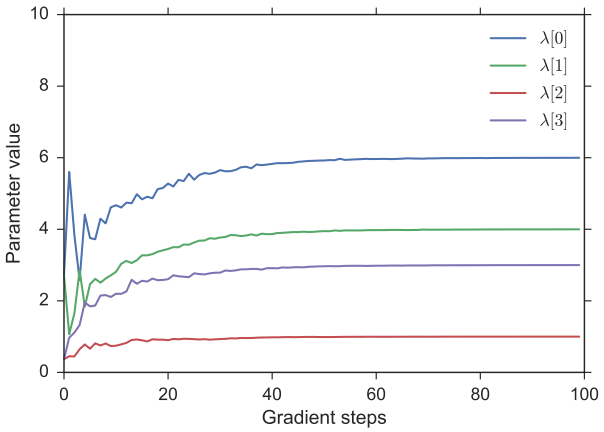


Figure 6.3: Variational Bayes estimation of values for the parameter λ governing the Dirichlet distirbution, for the Dirichlet-multinomial model shown in the text. In this model, the true posterior distribution is also Dirichlet, with parameters $\lambda = [6\ 4\ 1\ 3]$. We see the estimates quickly (< 1 second) converge to these values.

statement, we also need to store the component $g_i(y_i|\phi_i)$ which contributes to the overall model likelihood.

Relative to an existing inference engine for performing importance sampling, the primary additional burden of implementation is providing expressions for evaluating the gradients $\nabla_\lambda \log q(x|\lambda)$ for each primitive distribution type (e.g. `poisson`, `gamma`, `normal`) in our program. Note that many discrete distributions, including e.g. `poisson`, and categorical distributions, despite having discrete values x have a continuous parameter space λ. For purposes of optimization, it is often easier to transform the constrained parameters (such as the positive-valued standard deviation of a normal distribution) into an unconstrained space, and perform the gradient optimization there; this is equivalent to reparameterizing the primitive distributions to have unconstrained real-valued coordinates. As the KL divergence is insensitive to parameterization of the distributions, this only affects the optimization procedure, and does not change the model or the approximating distribution.

As a basic example and validation, we apply this variational inference engine to the Dirichlet-multinomial model exhibited at the beginning of this chapter in Figure 6.3. We find that this does indeed quickly converge to the true posterior parameters.

6.2.2 Implementation challenges in higher-order languages

There are two major challenges that arise when implementing variational inference in this setting, which ultimately limit its applicability as a general-purpose inference algorithm.

Choices of addressing schemes

In order for the learned approximations to have a reasonable interpretation, the addresses α which are used to index random choices across executions must do a reasonably good job of aligning random variables which fulfill the same semantic roles. In some sense, the burden on the addressing scheme for variational inference is actually much higher than for addressing schemes used for lightweight Metropolis-Hastings [Wingate et al., 2011]. For a single-site Metropolis-Hastings implementation, an addressing scheme which assigns the same address to rather "different" random variables in the trace, or one which fails to note that certain random choices may be rescored, will lead to a reduction in the acceptance rate of the MCMC algorithm, but outside of pathological cases does not affect the asymptotic behavior. Contrarily a poor choice of addressing scheme in the variational Bayes algorithm is fatal. The underlying posterior representation is a mapping from addresses α to distributions $q_\alpha(x_\alpha|\lambda_\alpha)$: if these addresses do not have a coherent semantic interpretation, then we cannot expect a coherent representation of the posterior distribution.

The choice of addresses used in the variational Bayes implementation in Anglican is a tuple (name type counter), where type refers to the parametric form of the distribution (e.g. Gaussian), and the counter is an automatically-incremented positive integer which enforces unique addresses for every random choice in a single execution, recording the number of total times sampling from a random choice with the same name and type. The name itself defaults to a unique identifier for the lexical position of the corresponding `sample` statement, however we permit specification of explicit addresses, with a syntax

```
(sample name-expr dist-expr)
```

where `dist-expr` is an expression which evaluates to a distribution object, and we evaluate `name-expr` to find the name for this particular sample statement. We use the ability to custom-label `sample` statements advantageously in an example in the following section.

However, this addressing scheme is not sufficient, nor could any be: when considering any model which admits a unbounded number of random choices, during training time (i.e., after finitely many executions of the program to estimate gradients) we explore only a finite number of possible executions. For example, the following code snippet would evaluate to a value `k` which is distributed according to a geometric distribution,

```
(loop [k 0]
  (if (sample :p (flip 0.9))
    k
    (recur (inc k))))
```

but in doing so samples at the address $\alpha = (\texttt{:p flip } k)$ for $k = 0, 1, \ldots$. Any approximation to the posterior which learns approximations at each address α will only learn approximations for some finite number of addresses k after finitely many gradient updates. However, when drawing samples from the approximating program, there is still positive probability of recursing down a branch of the program in which we sample from some never-before-seen $k + 1$. This issue is analogous to challenges to implementing variational inference in Bayesian nonparametric models, in which the number of latent parameters is stochastic; a standard approach is to truncate the approximating model $q(\cdot)$ to support only a (sufficiently large) finite number of the potentially infinite model parameters [Blei et al., 2006]. Here, this is equivalent to bounding runtime recursion depth; designing a principled general-purpose approach would be difficult. We can find mild relief in that branches of the program which have high posterior probability under $\mathcal{P}$ will tend to be well-explored during the gradient ascent procedure, and it is only in low-probability tails where we would encounter novel addresses when sampling from $\mathcal{Q}_\lambda$ after optimization.

Options for variance reduction

This Monte Carlo estimator of the gradient in Equation (6.12), known as the *score function estimator* or REINFORCE estimator [Williams, 1992, Schulman et al., 2015] is known to have high variance. By combining approaches to variance reduction, it can become feasible to use this estimator to perform stochastic gradient ascent on the ELBO. What kind of variance reduction is possible?

A first approach to variance reduction is the introduction of a *control variate* [Ross, 2006, Paisley et al., 2012]. In general, if we wish to compute the expectation of some function $h(\mathbf{x})$, then a control variate $b(\mathbf{x})$ is a function with known expectation, which we can use to modify the variance of the estimate of $h(\mathbf{x})$ without changing the expected value:

$$h'(\mathbf{x}) = h(\mathbf{x}) + a(b(\mathbf{x}) - \mathbb{E}[b(\mathbf{x})]). \tag{6.13}$$

In particular, if $\mathbb{E}[b(\mathbf{x})] = 0$, then

$$\mathbb{E}[h'(\mathbf{x})] = \mathbb{E}[h(\mathbf{x}) + ab(\mathbf{x})] = \mathbb{E}[h(\mathbf{x})]. \tag{6.14}$$

In this context, a good choice of control variate is the $b(\mathbf{x}) = \nabla_\lambda \log q(\mathbf{x}|\lambda)$, since

$$\begin{aligned}\mathbb{E}_q[\nabla_\lambda \log q(\mathbf{x}|\lambda)] &= \int q(\mathbf{x}|\lambda)\nabla_\lambda \log q(\mathbf{x}|\lambda)\mathrm{d}\mathbf{x} \\ &= \int \nabla_\lambda q(\mathbf{x}|\lambda)\mathrm{d}\mathbf{x} = \nabla_\lambda \int q(\mathbf{x}|\lambda)\mathrm{d}\mathbf{x} = 0.\end{aligned} \tag{6.15}$$

The variance of $h'(\mathbf{x})$ is minimized when choosing a to be

$$a = \frac{\mathrm{Cov}(h, b)}{\mathrm{Var}(b)} \tag{6.16}$$

which we can estimate with the same samples we are using to compute the expectation in Equation (6.8). This procedure is outlined in a very general setting in Schulman et al. [2015], and more specifically to this context in Ranganath et al. [2013] and van de Meent et al. [2016].

Unfortunately, other approaches for variance reduction commonly applied to this estimator of the ELBO are more challenging to adapt to this setting. The first is also used in Ranganath et al. [2013], and is a Rao-Blackwellization of the estimator:

in a graphical model there is a fixed dependency structure, which allows computing estimates of Equation (6.8) for each random choice x_j using only samples from those random variables which share a factor with x_j. Without a static dependency structure, this cannot be directly applied here.

One line of future work may involve seeing if common random numbers [Kleinman et al., 1999] can be employed for variance reduction in this context. The re-use of random numbers by fixing seeds across executions has been found to be useful in other gradient estimators based off of stochastic variants of finite differences [Spall, 1992] when approximating the gradients for Hamiltonian Monte Carlo [Meeds et al., 2015] and may also be useful here.

Finally, we note that the most effective way in which one reduces the variance of the estimator for the gradient of the ELBO is to use a different estimator [Kingma and Welling, 2014, Kucukelbir et al., 2015], which is possible for differentiable probability models with continuous random variables. We will discuss this later; first, we will highlight an application which demonstrates the utility of an automatic variational inference engine which can be applied to models which interleave random variables with complicated deterministic program code.

6.3 Maximum likelihood estimation and black-box policy learning

In this section, we show how this method can be modified to perform maximum likelihood estimation. In doing so we draw a connection between variational inference and policy search methods, and demonstrate a non-trivial application of this inference engine in Anglican.

6.3.1 Empirical Bayes estimation

In light of the challenges discussed in Section 6.2.2, consider for the moment a restricted but explicitly "safe" class of models in which there is a fixed number

of latent variables, all of which are *a priori* independent. In such a model, we can express the prior over these latent variables as

$$p(\mathbf{x}) = \prod_{j=1}^{M} f_j(x_j|\theta_j) \tag{6.17}$$

where each θ_j and the distribution types f_j are fixed, compile-time constants. Let $\lambda^{(t)}$ denote the setting of the variational parameters after t gradient steps, with $\lambda^{(0)} = [\theta_1, \ldots, \theta_M]$ being the compile-time constant values of the prior parameters. In this setting, consider the ELBO as defined above, as a function of both the variational parameters and the initial parameters (i.e., the model prior parameters):

$$\mathcal{L}(\lambda, \lambda^{(0)}) = \mathbb{E}_{q(\mathbf{x}|\lambda)}\left[\log \frac{p(\mathbf{x}, \mathbf{y}|\lambda^{(0)})}{q(\mathbf{x}|\lambda)}\right] = \mathbb{E}_{q(\mathbf{x}|\lambda)}\left[\log p(\mathbf{y}|\mathbf{x}) + \log \frac{p(\mathbf{x}|\lambda^{(0)})}{q(\mathbf{x}|\lambda)}\right]. \tag{6.18}$$

Since we are choosing our parametric family q to be of the same functional form as the prior — that is, the distribution type for $q_j(x_j|\lambda_j)$ is the same as the distribution type for $f_j(x_j|\theta_j)$ — then if we evaluate this at $\lambda = \lambda^{(0)}$, we have the approximating distribution $q(\mathbf{x}|\lambda^{(0)})$ equal to the prior distribution $p(\mathbf{x})$. In this case, the lower bound simplifies to

$$\begin{aligned}
\mathcal{L}(\lambda, \lambda^{(0)})\big|_{\lambda=\lambda^{(0)}} &= \mathbb{E}_{q(\mathbf{x}|\lambda)}\left[\log p(\mathbf{y}|\mathbf{x}) + \log \frac{p(\mathbf{x}|\lambda^{(0)})}{q(\mathbf{x}|\lambda)}\right]\Bigg|_{\lambda=\lambda^{(0)}} && (6.19)\\
&= \mathbb{E}_{q(\mathbf{x}|\lambda_0)}\left[\log p(\mathbf{y}|\mathbf{x})\right]; && (6.20)
\end{aligned}$$

that is, the log marginal likelihood $\log p(\mathbf{y})$. Similarly, the gradient of this lower bound simplifies to

$$\begin{aligned}
\mathcal{L}(\lambda, \lambda^{(0)})\big|_{\lambda=\lambda^{(0)}} &= \mathbb{E}_{q(\mathbf{x}|\lambda)}\left[\log \frac{p(\mathbf{y}|\mathbf{x})p(\mathbf{x}|\lambda^{(0)})}{q(\mathbf{x}|\lambda)} \nabla_\lambda \log q(\mathbf{x}|\lambda)\right]\Bigg|_{\lambda=\lambda^{(0)}} && (6.21)\\
&= \mathbb{E}_{q(\mathbf{x}|\lambda^{(0)})}\left[\log p(\mathbf{y}|\mathbf{x}) \nabla_\lambda \log q(\mathbf{x}|\lambda)\big|_{\lambda=\lambda^{(0)}}\right] && (6.22)\\
&= \int \nabla_\lambda q(\mathbf{x}|\lambda)|_{\lambda=\lambda^{(0)}} \log p(\mathbf{y}|\mathbf{x}) \mathrm{d}\mathbf{x} && (6.23)\\
&= \nabla_\lambda \mathbb{E}_{q(\mathbf{x}|\lambda)}\left[\log p(\mathbf{y}|\mathbf{x})\right]|_{\lambda=\lambda^{(0)}}. && (6.24)
\end{aligned}$$

This suggests we can modify the variational Bayes algorithm to compute maximum likelihood estimates by updating the prior parameters at each gradient step. That is, instead of following the stochastic gradient updates

$$\lambda^{(t+1)} = \lambda^{(t)} + \rho_t \nabla_\lambda \mathcal{L}(\lambda, \lambda^{(0)})|_{\lambda=\lambda^{(t)}}, \tag{6.25}$$

from Equation (6.6), we can update according to

$$\lambda^{(t+1)} = \lambda^{(t)} + \rho_t \nabla_\lambda \mathcal{L}(\lambda, \lambda^{(t)})|_{\lambda=\lambda^{(t)}}. \tag{6.26}$$

The only difference here is now, at each stochastic gradient iteration, we have modified the model prior parameters to be the currently learned estimate of the model posterior parameters: this effectively becomes an algorithm for learning parameters of the model $p(\mathbf{x})$, rather than an algorithm for learning a posterior approximation $q(\mathbf{x}|\lambda)$. Instead of treating the parameters $\lambda^{(0)} = [\theta_1, \ldots, \theta_M]$ as fixed prior parameters in the model, we treat $\lambda^{(0)}$ as an initialization for unknown model parameters, and run a series of gradient ascent steps which maximize the log marginal likelihood $\log p(\mathbf{y}) = \mathbb{E}[\log p(\mathbf{y}|\mathbf{x})]$. This is a form of maximum likelihood estimation also known as the "empirical Bayes" method for setting model hyperparameters; running this algorithm will aim to find a $\lambda^\star$

$$\lambda^\star = \underset{\lambda}{\operatorname{argmax}}\, \mathbb{E}_{q(\mathbf{x}|\lambda)}[\log p(\mathbf{y}|\mathbf{x})]. \tag{6.27}$$

In Figure 6.4 we show the estimated values of λ as we perform optimization, on the same model in Figure 6.1. At first glance, this figure may look unnerving — it appears to show the three of the components of the Dirichlet parameter diverging to infinity! However, consider what maximum likelihood estimation in a conjugate Dirichlet-multinomial model should yield as an estimate for the probability vector p: a vector representing the frequencies in which the different values occur in the data; for the data inlined in the program in Figure 6.1, that would be $p = [.5, .3, 0, .2]$. In Figure 6.5, we plot the mean and standard deviations of each dimension of p for the approximations $q(p|\lambda)$ learned under both the VB and the EB procedures. The mean of p converges to the Bayesian posterior mean and to the maximum likelihood estimate, respectively, under the two approaches. Additionally, we see that when running VB the variance of p converges to the true posterior variance. In contrast, as we run more gradient update steps of EB, we see that the approximating distribution q has ever lower variance, concentrating further on the maximum likelihood estimates. This is actually desirable behavior for this algorithm, and

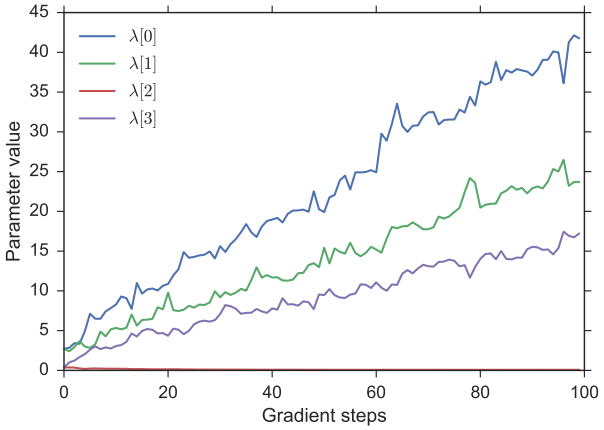


Figure 6.4: Empirical Bayes estimation of values for the parameter λ governing the Dirichlet distirbution. Unlike in the variational Bayes approximation to the posterior, these estimates to not level out a fixed value of the parameters.

explains the parameters in Figure 6.4 appearing not to converge — it is, rather, the ratio of those values over their sum which is converging as $q(p|\lambda)$ approaches a point mass on the maximum likelihood estimate of p.

Optimizing a subset of the random variables

This becomes more interesting when we consider a slightly expanded class of models, in which there are two types of random variables: in addition to the $\mathbf{x}$ we are interesting in estimating, we also may have "nuisance" variables $\mathbf{z}$. We keep the restriction on $\mathbf{x}$ that the values be *a priori* independent with deterministic prior parameters θ, but allow the conditional distribution of the additional latent variables $p(\mathbf{z}|\mathbf{x})$ to be unrestricted, allowing arbitrary dependencies and random dimensionality of $\mathbf{z}$. We can consider the probability distribution defined by such a program to factor according to

$$p(\mathbf{x}, \mathbf{y}, \mathbf{z}) = p_\theta(\mathbf{x}) p_\eta(\mathbf{z}|\mathbf{x}) p(\mathbf{y}|\mathbf{x}, \mathbf{z}). \tag{6.28}$$

The joint distribution over the latent nuisance variables $\mathbf{z}$ and the latent variables of interest $\mathbf{x}$ is given by

$$p(\mathbf{x}, \mathbf{z}) = \prod_{j=1}^{M} f_j(x_j|\theta_j) \prod_{j=1}^{M'} f'_j(z_j|\eta_j), \tag{6.29}$$

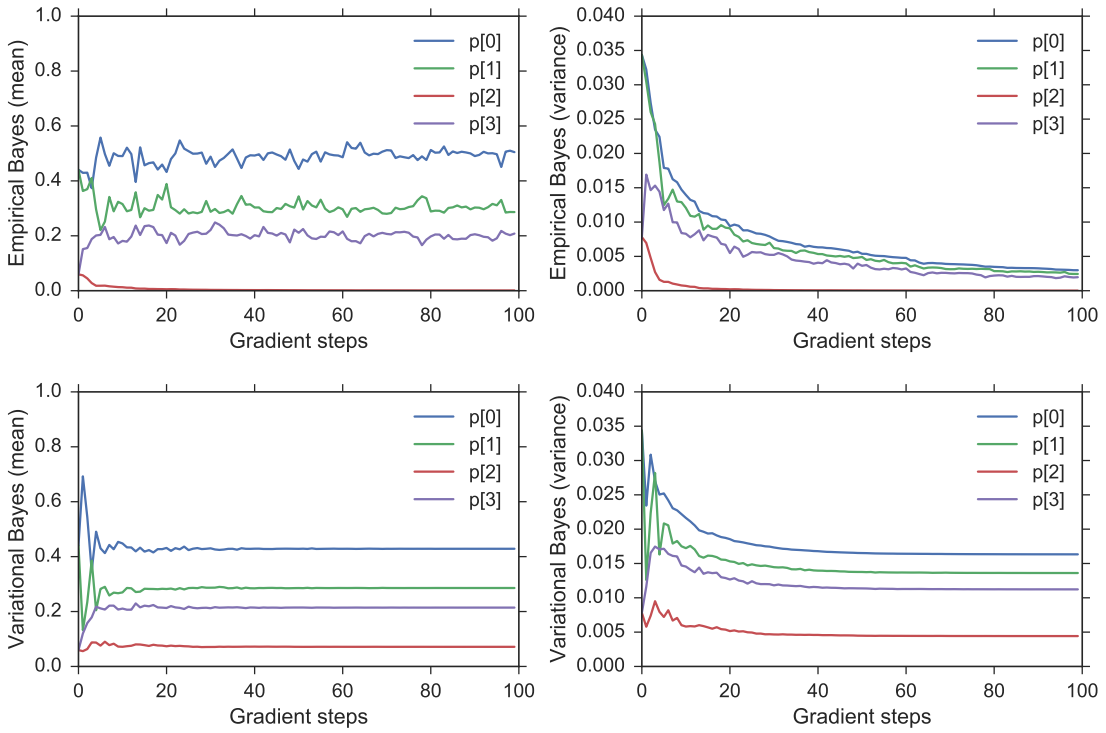


Figure 6.5: (Top) convergence of empirical Bayes estimation in the Dirichlet-multinomial model to the maximum likelihood estimate for the probability vector p; (Bottom) convergence of the variational Bayes estimation to the posterior distribution over p. Note that the mean of the empirical Bayes procedure is the maximum likelihood estimate of p, and the variance of the estimate decreases as we take more gradient steps; in contrast, both the mean and variance of p converge towards the posterior mean and variance when using the variational Bayes approach.

where we have defined M' to be the number of nuisance variables z_j, and f'_j with parameter η_j to be the parameter for the distribution over the nuisance variable defined by the program $\mathcal{P}$.

This partitioning of the latent space is particularly useful for describing problems in which one wishes to perform optimization over a noisy objective function. We can consider the nuisance variables $\mathbf{z}$ to be part of a likelihood function $p(\mathbf{y}|\mathbf{x})$ which cannot be evaluated directly, only estimated via simulation, as

$$p(\mathbf{y}|\mathbf{x}) = \int p(\mathbf{y}|\mathbf{x}, \mathbf{z}) p_\eta(\mathbf{z}|\mathbf{x}) \mathrm{d}\mathbf{z}. \tag{6.30}$$

The maximum likelihood estimation procedure then seeks to find a parameter

setting $\lambda^\star$ with

$$\begin{aligned}\lambda^\star &= \operatorname*{argmax}_\lambda \mathbb{E}_{q(\mathbf{x}|\lambda)}[\log p(\mathbf{y}|\mathbf{x})] \\ &= \operatorname*{argmax}_\lambda \mathbb{E}_{q(\mathbf{x}|\lambda)}[\log \int p(\mathbf{y}|\mathbf{z},\mathbf{x}) p_\eta(\mathbf{z}|\mathbf{x}) \mathrm{d}\mathbf{z}]. \end{aligned} \tag{6.31}$$

6.3.2 Planning as inference

A particular potent application is to *policy learning* in a reinforcement learning context [van de Meent, Paige, Tolpin, and Wood, 2016]. In planning under uncertainty, the objective is to find a policy that selects actions, given currently available information, in a way that maximizes expected reward. In general an *optimal policy* can neither be represented compactly (i.e., in a parameter-efficient manner) nor learned exactly. *Online* approaches, such as Monte Carlo tree search [Kocsis and Szepesvári, 2006] form one nonparametric class of policies for solving planning problems: at each decision point, these policies select actions by performing *rollouts*, simulating possible future outcomes and rewards. While online policies are often able to achieve near-optimal performance, they are computationally intensive and do not have compact parameterizations. In contrast, policy search methods (e.g. Deisenroth et al. [2011]) learn parameterized policies offline, which then can be used without performing rollouts at test time. This represents a tradeoff between computational efficiency at test time, relative to choosing a parameterization which may have insufficient complexity to represent the optimal policy.

By considering such planning problems in a "planning as inference" setting, we can use probabilistic programs to define parameter-efficient policies for reinforcement learning problems. In the context discussed above, we wish to learn *policy parameters* $\mathbf{x}$, while marginalizing over the uncertainty $\mathbf{z}$, which corresponds to noise over which the agent has has no control. Rather than a traditional likelihood function $p(\mathbf{y}|\mathbf{x},\mathbf{z})$, we define the likelihood to be the exponentiated of the reward $R(\mathbf{z},\mathbf{x})$; the reward is a function both of actions taken, and the actual state of the variables $\mathbf{z}$ representing the uncertainty in the world. In this sense, we define the overall unnormalized

probability of the program trace identically as before, where

$$\gamma(\mathbf{x}, \mathbf{z}) = \exp(R(\mathbf{x}, \mathbf{z}))p(\mathbf{x}, \mathbf{z}) = \prod_{i=1}^{N} \exp(R_i(\phi_i)) \prod_{j=1}^{M} f_j(x_j|\theta_j) \prod_{j=1}^{M'} f'_j(z_j|\eta_j). \quad (6.32)$$

The joint prior distribution $p(\mathbf{x}, \mathbf{z})$ is defined here exactly as in Equation (6.29); the only difference here is the likelihood term is replaced by a reward function $R(\mathbf{x}, \mathbf{z})$, which potentially decomposes further as a sum over individual incremental rewards $R_i(\phi_i)$. As in previous sections, the parameter vector ϕ_i for the component of the reward depends potentially on the values of both $\mathbf{x}$ and $\mathbf{z}$, and is determined by the execution of the program. By inserting this definition of the likelihood in Equation (6.31) for optimizing the parameters λ after marginalizing out nuisance variables $\mathbf{z}$, we see that this corresponds to

$$\lambda^\star = \operatorname*{argmax}_{\lambda} \mathbb{E}_{q(\mathbf{x}|\lambda)} \left[\mathbb{E}_{p_\eta(\mathbf{z}|\mathbf{x})}[R(\mathbf{x}, \mathbf{z})] \right] \quad (6.33)$$

in which we seek learn optimal policy parameters $\lambda^\star$ which maximize the reward in expectation over the nuisance variables $\mathbf{z}$.

Case study: the Canadian traveller problem

The discussion above was rather abstract, and we seek to clarify here by showing a non-trivial example of a parameterized policy whose structure is denoted by a probabilistic program. In the *Canadian traveller problem* (CTP) [Papadimitriou and Yannakakis, 1991], an agent must traverse a graph $G = (V, E)$ in which edges may be missing at random. It is assumed the agent knows the distance $d\colon E \to \mathbb{R}^+$ associated with each edge, as well as the probability $p\colon E \to (0, 1]$ that the edge is open, but on any particular trial the agent has no advance knowledge of which edges will be blocked. Solving this problem is NP-hard [Fried et al., 2013], and heuristic approaches (both online and offline) are used to solve problem instances [Eyerich et al., 2010].

This problem is an example of a deterministic partially-observable Markov decision process (POMDP), in which the initial state of the world is not known, and observations may be noisy, but the state transitions are deterministic. As the number

```
(defquery ctp
 "Probabilistic program representing an agent
  solving the Canadian Traveler Problem"
 [graph src tgt base-prob make-policy]
 (let [sub-graph
        (sample-weather graph base-prob src tgt)
       [path dist counts]
        (dfs-agent sub-graph src tgt (make-policy))]
  (factor (- dist))
  {:path path
   :distance dist
   :counts counts}))

(defm dfs-agent
 "Run depth-first-search from start to target,
  prioritizing edges according to policy"
 [graph start target policy]
   ... )
```

```
(defm make-random-policy
 "Policy: Select edge at random"
 []
 (fn policy [u vs]
  (sample
   (categorical
    (zipmap vs (repeat (count vs) 1.))))))

(defm make-edge-policy
 "Policy: learn priorities for each edge"
 []
 (let [Q (mem (fn [u v]
                (sample [u v]
                 (tag :policy
                  (gamma 1. 1.)))))]
  (fn policy [u vs]
   (argmax
    (zipmap vs (map (fn [v] (Q u v)) vs))))))
```

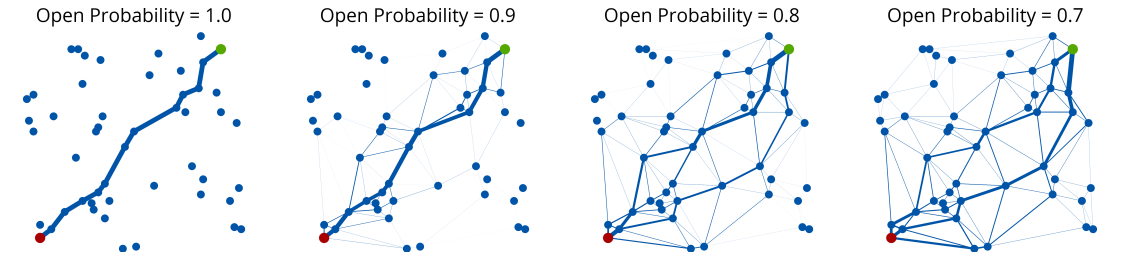


Figure 6.6: A Canadian traveler problem (CTP) implementation in Anglican. In the CTP, an agent must travel along a graph, which represents a network of roads, to get from the start node (green) to the target node (red). Due to bad weather some roads are blocked, but the agent does not know which in advance. Upon arrival at each node the agent observes the set of open edges. The function `dfs-agent` walks the graph by performing depth-first search, calling a function `policy` to choose the next destination based on the current and unvisited locations. The function `make-random-policy` returns a `policy` function that selects destinations uniformly at random, whereas `make-edge-policy` constructs a `policy` that selects according to sampled edge preferences `(Q u v)`. By learning a distribution on each value `(Q u v)` through gradient ascent on the marginal likelihood, we obtain a heuristic offline policy that follows the shortest path under the assumption that all as yet unseen edges are open, and explores more alternate routes as more edges are closed. Figure from van de Meent, Paige, Tolpin, and Wood [2016].

of steps grows in a sequential decision problem — here, as we take progressively more steps along the graph — the number of possible information states describing the world grows exponentially, meaning it is not in general possible to fully parameterize a distribution over actions in terms of a conditional probability table.

In Figure 6.6 we show the source code for a probabilistic program which describes the process of an agent solving the Canadian traveller problem. Given a problem instance — a graph G, the distances d, and the open probabilities p — we first sample a `sub-graph`, in which the weather (composed of nuisance random variables) causes some edges E to be closed. Then, we define an agent `dfs-agent`, an agent

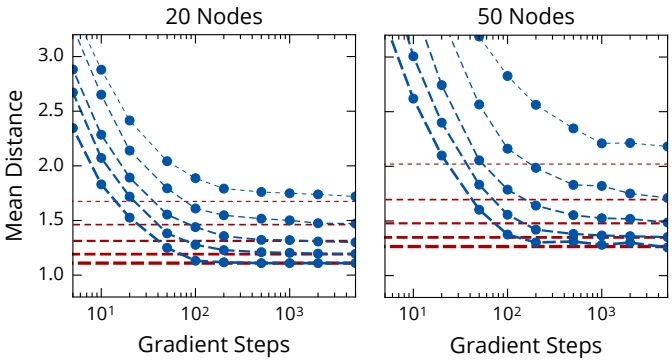


Figure 6.7: Convergence for CTP domains of 20 and 50 nodes. Blue lines show the mean traveled distance using the learned policy, averaged over 5 domains. Red lines show the mean traveled distance for the optimistic heuristic policy. Dash length indicates the fraction of open edges, which ranges from 1.0 to 0.6.

which attempts to travel from the start node (marked in green) to the end node (marked in red) following a depth-first search. A free parameter in a depth-first search is the *order* in which the various branches are explored, at each decision point; while the depth-first search will always eventually reach the end node, the total distance traveled depends on the order in which the agent tries different possible paths. This ordering is controlled by some policy for deciding in which order to consider the different paths; a baseline policy chooses which path to take at each branching point at random. The `edge-policy` learns parameters of a gamma distribution at each edge: we provide explicit names `[u v]` for addressing purposes, in order to align the sampled values from each different execution. Both the number and order of random choices is stochastic, as they depend on the graph traversal path taken by the agent. We define the overall reward to be the negative distance travelled; in the program in Figure 6.6 this is shown using the `(factor (- dist))` statement; `factor` is similar to `observe`, but directly adds a numeric log probability to the program trace rather than computing the log probability from a provided distribution object and value. Complete source code is available at `https://bitbucket.org/probprog/black-box-policy-search`.

The notation `(tag :policy (gamma 1. 1.))` annotates these random variables with a "tag" named `:policy`. This is the method by which nuisance variables and variables of interest are separated; at inference time, we declare that we wish

to perform the BBEB procedure only on those random variables tagged `:policy`, treating the rest as nuisance variables we should marginalize over. Note that in this example it is important to be able to separate these random variables: the policy which is optimal after marginalizing over possible weather is not at all the same as the the policy which is jointly optimal with the optimal weather. That is, were we to optimize all random variables together, we would learn a "policy" suggesting that we should always take the shortest path through the graph, and no roads should ever be blocked.

In the bottom of Figure 6.6, we can see that the learned policy behaves in a reasonable manner. When edges are open with high probability, the policy takes the shortest path from the start node to the target. As the fraction of closed (blocked) edges increases, the policy makes more frequent use of alternate routes. Note that in this problem instance, each edge has a fixed probability of being open, resulting in a preference for routes which traverse fewer edges.

Figure 6.7 shows convergence as a function of number of gradient steps; results are averaged across 5 different graph instances of 20 and 50 nodes respectively. We compare the learned policies to an optimistic policy, a heuristic which selects edges according to the shortest path under the assumption that all edges will be open. We see that the average distance traveled for the learned policy converges to close to that of the optimistic policy; the optimistic policy is a strong heuristic in these problem instances with close-to-optimal performance, so learning a policy which matches its performance is a positive result. In contrast, a baseline random policy which chooses any open edge will take much longer to reach a the goal node.

We present more results for this as well as for other POMDPs, using probabilistic programs to specify parameter-efficient policies, in van de Meent et al. [2016].

6.4 Different types of reparameterization

One potential source of confusion when discussing variational inference methods is what we mean by *reparameterization*. There are at least three distinct possibilities. First, there is the reparameterization of the proposal distribution described in

Section 6.2, for purposes of optimization within a black-box variational inference scheme. While this changes the space in which we take gradient steps, since we transform the variational parameters λ to the unconstrained space, make a gradient step, and then transform back, this only has the effect of easing optimization: if we did not transform to an unconstrained space, our optimization would have to handle gradient steps which may bring us outside the feasible set, via projection into the feasible set, reflection at boundaries, or other methods; optimizing on transformed versions of the random variables sidesteps this issue. These such transformations are similar to those performed by STAN for e.g. L-BFGS optimization, although here the transformations are performed on the hyperparameters rather than on the latent values themselves.

A second sort of reparameterization is popularized as the "reparameterization trick" [Kingma and Welling, 2014, Kucukelbir et al., 2015], and is used within the context of variational inference to reduce the variance of stochastic gradient estimators; we discuss the potential application of this here below. A third meaning of reparameterization which we encounter in this context is *model parameterization*: programs which are "identical", in the sense that they define the same overall probability distribution over the program output, can be expressed using radically internal program structure and latent variables. We will see that this degree of freedom in model parameterization can have surprising consequences on the quality of the learned approximating model.

6.4.1 Automatic differentiation and reparameterization

Variational inference as implemented in Stan (automatic differentiation variational inference, or ADVI) reports using only a single sample per gradient step. In contrast, in the policy learning example for the Canadian traveler problem, each gradient step required averaging across 1,000 samples. The reason for the greatly reduced variance in the ADVI gradients lies in the reparameterization of the model, which in turn is possible due to the existence of model gradients. Similarly, in the context

of variational autoencoders [Kingma and Welling, 2014], only a single sample is used for each stochastic gradient estimate of the ELBO.

In both cases, the key lies in re-expressing the sampled values x_j as transformed values drawn from some independent noise distribution: that is, by introducing a transformation $x_j = T_j(\epsilon_j, \lambda_j)$ where ϵ_j is drawn from some distribution $q_j(\epsilon_j)$ which does not depend on on the values of the parameters λ_j. While the exact manner of reparameterization and choice of $q(\epsilon)$ differs in the two approaches, in both cases the computation of the gradient of the ELBO now includes a term which depends on the gradient of the model itself, thanks to the chain rule: if generically we have $\mathbf{x} = T(\epsilon, \lambda)$ then we have

$$\nabla_\lambda \mathbb{E}_{q(\epsilon)}[\log \gamma(\mathbf{x})] = \mathbb{E}_{q(\epsilon)}[\nabla_\lambda \log \gamma(T(\epsilon, \lambda))] = \mathbb{E}_{q(\epsilon)}[\nabla_{\mathbf{x}} \log \gamma(\mathbf{x}) \nabla_\lambda T(\epsilon, \lambda)].$$

In contrast, when the ELBO is evaluated by sampling from $q(\mathbf{x}|\lambda)$ directly, as opposed to computed from samples of $q(\epsilon)$, we originally had

$$\nabla_\lambda \mathbb{E}_{q(\mathbf{x}|\lambda)}[\log \gamma(\mathbf{x})] = \mathbb{E}_{q(\mathbf{x}|\lambda)}[\log \gamma(\mathbf{x}) \nabla_\lambda \log q(\mathbf{x}|\lambda)].$$

The advantage of the second estimator is that it does not ask us to compute any gradients of $\log \gamma(\mathbf{x})$, permitting us to happily perform gradient ascent on non-differentiable probability models. The disadvantage is that it does not use any gradients of $\log \gamma(\mathbf{x})$! Including the actual model gradients when available yields much more sample-efficient estimators than weighting sampled values of the gradient of the proposal. This is discussed further, and hybrid methods are introduced, in Schulman et al. [2015].

6.4.2 Model parameterization and approximation quality

An additional question arises when writing probabilistic programs, where there are often many equivalent ways of expressing the same distribution. For example, consider the following very simple generative model:

$$a \sim \text{Normal}(0, 1) \qquad b \sim \text{Normal}(a, 1) \qquad c \sim \text{Normal}(b, 1)$$

where we observe $c = 2$, and want to estimate the posterior distribution $p(b|c)$. This can be performed analytically; we find that the posterior distribution is also a Gaussian, with mean $4/3$ and variance $2/3$. We can equivalently write this model as

$$a \sim \text{Normal}(0, 1) \qquad b' \sim \text{Normal}(0, 1) \qquad c \sim \text{Normal}(a + b', 1),$$

where the original value $b = a + b'$. This defines the same joint distribution over a, b, and c but now the variational inference procedure learns an approximation at $q(b')$ instead of at $q(b)$. Both programs as written in Anglican produce correct answers when running MCMC sampling. However, although variational inference estimates the mean of $4/3$ correctly when written in either form, the first program underestimates the variance of b, whereas the second program overestimates it.

This sort of behavior represents a new challenge when designing these models in such an unconstrained settings; if we write out the expression for the ELBO, we see that they are indeed different expressions for the lower bound in the two models. Possibly such behavior could be counteracted by using more expressive model families $q(\mathbf{x}|\lambda)$ to capture the posterior.

6.5 Variational inference and optimal importance sampling proposals

In higher-order languages which allow unbounded recursion, or branching on random choices, after finite number of stochastic gradient steps, there are still possible random choices which we have never seen during training. There are two concrete ways of dealing with this issue. One we used above in the context of policy search: instead of learning *all* the random choices, we learn a fixed-dimensional subset of the random choices, marginalizing over the rest. An alternative approach is to reinterpret the overall inference procedure: instead of treating it as a variational inference scheme where the learned program $\mathcal{Q}_\lambda$ is our posterior representation, we treat this as an adaptive importance sampling scheme where the learned parametric distribution forms a data-driven proposal distribution which we update online during inference. When encountering random choices at some "new" address α, we simply

sample from the distribution defined by $f_\alpha(x_\alpha|\lambda_\alpha)$. This then forms an iterated importance sampling algorithm, in the style of population Monte Carlo (PMC) [Cappé et al., 2004, Cappé et al., 2008] and adaptive multiple importance sampling (AMIS) [Cornuet et al., 2012]. At each iteration of the algorithm, we draw K values of latent variables from the approximating program $\mathcal{Q}_\lambda$, then use those values both to perform a gradient update on λ, and to estimate posterior expectations of the return value ψ. In the algorithm above, the gradient of the ELBO is an expectation over $q(\mathbf{x}|\lambda)$, In an improvement over the original likelihood weighting algorithm we described in Section 3.3, we can estimate posterior expectations of the return value $\psi(\mathbf{x})$ as

$$\mathbb{E}_p\left[\psi(\mathbf{x})\right] = \frac{1}{Z}\mathbb{E}_q\left[\frac{\gamma(\mathbf{x})\psi(\mathbf{x})}{q(\mathbf{x}|\lambda)}\right], \qquad Z = \mathbb{E}_q\left[\frac{\gamma(\mathbf{x})}{q(\mathbf{x}|\lambda)}\right]. \tag{6.34}$$

The partitioning of random variables used in isolating nuisance variables from policy parameters can be re-purposed to specify at which (any or all) addresses we will learn proposals.

If our end goal is use of $\mathcal{Q}_\lambda$ an importance sampling proposal density, then other objective functions than the ELBO may be preferable. Variational inference traditionally maximizes the evidence lower bound, a procedure equivalent to minimizing $D_{KL}(q||\pi)$. For importance sampling purposes, rather than minimizing this so-called exclusive divergence, we would instead prefer to minimize $D_{KL}(\pi||q)$, such that any approximating distribution $q(\mathbf{x})$ would satisfy $q(\mathbf{x}) > 0$ for any points where $p(\mathbf{x}|\mathbf{y}) > 0$. In such a scheme, instead of maximizing the ELBO we minimize an upper bound on the marginal likelihood, based on the alternative decomposition

$$\int p(\mathbf{x}|\mathbf{y}) \log\left[\frac{p(\mathbf{x},\mathbf{y})}{q(\mathbf{x}|\lambda)}\right] \mathrm{d}\mathbf{x} = \log p(\mathbf{y}) + D_{KL}(\pi||q) \geq \log p(\mathbf{y}). \tag{6.35}$$

This upper bound is a posterior expectation, and so inference is necessary to estimate this gradient. One possibility is to use the same weighted samples set we using to compute posterior estimates $\mathbb{E}[\psi]$ to estimate the gradient. However, for multimodal marginal distributions the learned proposal distributions try to place support over all possible modes, and we are restricted by our simplistic choice of proposal family

$q(\mathbf{x}|\lambda)$; adaptive importance sampling methods used in e.g. Cappé et al. [2008] use mixture models of multivariate Gaussian or multivariate Student-t distributions as the approximating family. Minka [2005] provides a comparison of different alpha-divergences (a generalization of the KL divergence), their approximation behavior, as well as their relative advantages as learned importance sampling proposals.

6.6 Discussion

We have shown that a variational Bayes algorithm can be implemented in a higher-order probabilistic programming language, and present an implementation in Anglican. Given the difficulties in constructing a low-variance estimate of the gradient of the ELBO, more work is required to determine exactly when such an algorithm is useful in practice. The most clear use is for situations where there is a low-dimensional parameter space but high model complexity and a high-fidelity model: in the Canadian traveller problem example, one of the explicit advantages of using a program as a policy means that the policy can be parameter-efficient, thanks to the structure given via the use of depth-first search within the policy.

Since variance reduction of the gradient is primarily performed by drawing additional samples, the overall procedure does not exhibit the clear computational efficiency relative to MCMC methods in models such as the matrix factorization examples benchmarking ADVI in Kucukelbir et al. [2015]. In other words, if the model can be written in such a manner that it is differentiable, and contains only a bounded number of continuous latent variables, then ADVI seems to be strictly superior. It is an open question (and likely model-specific) what the performance characteristics would look be for models written in an intermediate language, which permits (say) discrete latent variables, but still does not permit branching on stochasticity: stochastic gradients could be estimated through a mix of score function and pathwise derivative estimators following the procedure in Schulman et al. [2015].

More immediately promising and applicable is the use of a scheme like black-box variational inference in learning and updating proposal distributions within the context of a larger inference algorithm based on importance sampling. This successfully

sidesteps issues relating to random numbers of random choices occurring across program executions; and, furthermore, when using $\mathcal{Q}_\lambda$ as a proposal distribution (rather than directly as a posterior approximation) it is not necessary to successfully match the posterior — the learned distribution $\mathcal{Q}_\lambda$ only has to be mildly closer to the posterior than to the prior in order to lead to an inference algorithm which is more sample-efficient than proposing from $\mathcal{P}$, at the minor additional cost of computing expectations of gradients alongside estimates of the return value $\psi(\mathbf{x})$.

One weakness is that the proposal programs $\mathcal{Q}_\lambda$ learned here are still quite restrictive: the distribution $q(\mathbf{x}|\lambda)$ is assumed to fully factorize across dimensions of $\mathbf{x}$, with independent $q_j(x_j|\lambda_j)$. In the following chapter, we will relax that assumption and address learning a more expressive class of data-driven proposal distributions, parameterized by the output of a neural network.

7

Amortized inference: compiling away runtime costs of inference

In this chapter, we step back from the full generality of higher-order probabilistic programming languages to consider how to improve learning of proposal distributions for sequential Monte Carlo methods first in a slightly restricted setting; we target the model class supported by languages such as Bugs and Jags. Here the algorithm is framed in the notation of graphical models; however, procedures for compiling code written in the Bugs language to a graphical model structure for application of sequential Monte Carlo are described in detail in Todeschini et al. [2014].

We introduce a new approach for amortizing inference in directed graphical models by learning heuristic approximations to stochastic inverses, designed specifically for use as proposal distributions in SMC. We describe a procedure for constructing and learning a structured neural network which represents an inverse factorization of the graphical model, resulting in a conditional density estimator that takes as input particular values of the observed random variables, and returns an approximation to the distribution of the latent variables. This recognition model can be learned offline, independent from any particular dataset, prior to performing inference. The output of these networks can be used as automatically-learned high-quality

proposal distributions to accelerate sequential Monte Carlo across a diverse range of problem settings.

To recap: methods based on sequential Monte Carlo have shown themselves to provide state-of-the art results in applications far broader than the traditional use for filtering in state space models [Gordon et al., 1993, Pitt and Shephard, 1999], with diverse application to factor graphs [Naesseth et al., 2014], hierarchical Bayesian models [Lindsten et al., 2014], procedural generative graphics [Ritchie et al., 2015], and general probabilistic programs [Wood et al., 2014, Todeschini et al., 2014]. These are accompanied by complementary computational advances, including memory-efficient implementations [Jun and Bouchard-Côté, 2014], and highly-parallel variants [Murray et al., 2014, Paige et al., 2014].

All these algorithms, however, share the need for specifying a series of *proposal distributions*, used to sample candidate values at each stage of the algorithm. Sequential Monte Carlo methods perform inference progressively, iteratively targeting a sequence of intermediate distributions which culminates in a final target distribution. Well-chosen proposal distributions for transitioning from one intermediate target distribution to the next can lead to sample-efficient inference, and are necessary for practical application of these methods to difficult inference problems. Theoretically optimal proposal distributions [Doucet et al., 2000, Cornebise et al., 2008] are in general intractable, thus in practice implementing these algorithms requires either active (human) work to design an appropriate proposal distribution prior to sampling, or using an online estimation procedure to approximate the optimal proposal during inference (as in e.g. Van Der Merwe et al. [2000] or Cornebise et al. [2014] for state-space models). In many cases, a baseline proposal distribution which simulates from a prior distribution can be used, analogous to the so-called bootstrap particle filter for inference in state-space models; this is equivalent to our approach of proposing from incremental execution of the unconditioned model code $\mathcal{P}$ in higher-order probabilistic programming languages. However, when confronted with tightly peaked likelihoods (i.e. highly informative observations), proposing from the prior distribution may be arbitrarily statistically inefficient [Del Moral

and Murray, 2015]. Furthermore, for some choices of sequences of densities there is no natural prior distribution, or even it may not be available in closed form. All in all, the need to design appropriate proposal distributions is a real impediment to the automatic application of these SMC methods to new models and problems.

This chapter investigates how autoregressive neural network models for modeling probability distributions [Bengio and Bengio, 1999, Uria et al., 2013, Germain et al., 2015] can be leveraged to automate the design of model-specific proposal distributions for sequential Monte Carlo. We propose a method for learning proposal distributions for a given probabilistic generative model offline, prior to performing inference on any particular dataset. The learned proposals can then be reused as desired, allowing SMC inference to be performed quickly and efficiently for the same probabilistic model, but for new data — that is, for new settings of the observed random variables — once we have incurred the up-front cost of learning the proposals.

We thus present this work as an amortized inference procedure in the sense of Gershman and Goodman [2014], in that it takes a model as its input and generates an artifact which then can be leveraged for accelerating future inference tasks. Such procedures have been considered for other inference methods: learning idealized Gibbs samplers offline for models in which closed-form full conditionals are not available [Stuhlmüller et al., 2013], using pre-trained neural networks to inform local MCMC proposal kernels [Jampani et al., 2015, Kulkarni et al., 2015], and learning messages for new factors for expectation-propagation [Heess et al., 2013]. In the context of SMC, offline learning of high-quality proposal distributions provides a similar opportunity for amortizing runtime costs of inference, while simultaneously automating a currently-manual process. Source code for all experiments is available at `https://github.com/tbrx/compiled-inference`.

7.1 Preliminaries

A directed graphical model, or Bayesian network [Pearl and Russell, 1998], defines a joint probability distribution and conditional independence structure via a directed acyclic graph. For each x_i in a set of random variables $x_1, \ldots, x_N$, the network

structure specifies a conditional density $p_i(x_i|\text{PA}(x_i))$, where $\text{PA}(x_i)$ denotes the parent nodes of x_i. Inference tasks in Bayesian networks involve marking certain nodes as observed random variables, and characterizing the posterior distribution of the remaining latent nodes. The joint distribution over N latent random variables $\mathbf{x}$ and M observed random variables $\mathbf{y}$ is defined as

$$p(\mathbf{x}, \mathbf{y}) \triangleq \prod_{i=1}^{N} f_i\left(x_i|\text{PA}(x_i)\right) \prod_{j=1}^{M} g_j\left(y_j|\text{PA}(y_j)\right), \tag{7.1}$$

where f_i and g_j refer to the probability density or mass functions associated with the respective latent and observed random variables x_i, y_j. Posterior inference in directed graphical models entails using Bayes' rule to estimate the posterior distribution of the latent variables $\mathbf{x}$ given particular observed values $\mathbf{y}$; that is, to characterize the target density $\pi(\mathbf{x}) \equiv p(\mathbf{x}|\mathbf{y})$. In most models, exact posterior inference is intractable, and one must resort to either variational or finite-sample approximations. Note that relative to previous chapters, we have changed the definition of N and M, such that N now refers to the number of *latent* variables; due to a change in the way the sequence of target densities densities is defined in the next section, N still denotes the total number of stages in the sequential Monte Carlo algorithm.

7.1.1 Sequential Monte Carlo for Bayesian networks

Importance sampling methods approximate expectations with respect to a (presumably intractable) distribution $\pi(\mathbf{x})$ by weighting samples drawn from a (presumably simpler) proposal distribution $q(\mathbf{x})$. In graphical models, with $\pi(\mathbf{x}) \equiv p(\mathbf{x}|\mathbf{y})$, we define an unnormalized target density $\gamma(\mathbf{x}) \equiv p(\mathbf{x}, \mathbf{y})$ such that $\pi(\mathbf{x}) = Z^{-1}\gamma(\mathbf{x})$, where the normalizing constant Z is unknown.

The sequential Monte Carlo algorithms we consider [Doucet et al., 2001] for inference on an $N-$dimensional latent space $\mathbf{x}_{1:N}$ proceed by incrementally importance sampling a weighted set of K particles, with interspersed resampling steps to direct computation towards more promising regions of the high-dimensional space. We break the problem of estimating the posterior distribution of $\mathbf{x}_{1:N}$ into a series of simpler lower-dimensional problems by constructing an artificial sequence

of target densities $\pi_1, \ldots, \pi_N$ (and corresponding unnormalized densities $\gamma_1, \ldots, \gamma_N$) defined on increasing subsets $\mathbf{x}_{1:n}$, $n = 1, \ldots, N$, where the final $\pi_N \equiv \pi$ is the full target posterior of interest. At each intermediate density, the importance sampling density $q_{n+1}(x_{n+1}|x_{1:n})$ only needs to adequately approximate a low-dimensional step from $x_{1:n}$ to x_{n+1}.

Procedurally, we initialize at $n = 1$ by sampling K values of x_1 from a proposal density $q_1(x_1)$, and assigning each of these particles x_1^k an associated importance weight

$$w_1(x_1^k) = \frac{\gamma_1(x_1^k)}{q_1(x_1^k)}, \qquad W_1^k = \frac{w_1(x_1^k)}{\sum_{j=1}^K w_1(x_1^j)}. \tag{7.2}$$

For each subsequent $n = 2, \ldots, N$, we first resample the particles according to the normalized weights at W_{n-1}^k, preferentially duplicating high-weight particles and discarding those with low weight. To do this we draw particle ancestor indices $a_{n-1}^1, \ldots, a_{n-1}^K$ from a resampling distribution $r(\cdot|W_{n-1}^1, \ldots, W_{n-1}^K)$ corresponding to any standard resampling scheme [Douc et al., 2005]. We then extend each particle by sampling a value for $\mathbf{x}_n^k$ from the proposal kernel $q_n(x_n^k|\cdot)$, and update the importance weights

$$w_n(\mathbf{x}_{1:n}{}^k) = \frac{\gamma_n(\mathbf{x}_{1:n}^k)}{\gamma_{n-1}(\mathbf{x}_{1:n-1}^{a_{n-1}^k}) q_n(x_n^k|\mathbf{x}_{1:n-1}^{a_{n-1}^k})}, \tag{7.3}$$

$$W_n^k = \frac{w_n(\mathbf{x}_{1:n}^k)}{\sum_{j=1}^K w_n(\mathbf{x}_{1:n}^j)}. \tag{7.4}$$

We can approximate expectations with respect to the target density $\pi(\mathbf{x}_{1:N})$ using the SMC estimator

$$\hat{\pi}(\mathbf{x}_{1:N}^{1:K}) = \sum_{k=1}^K W_N^k \delta_{\mathbf{x}_{1:N}^k}(\mathbf{x}_{1:N}), \tag{7.5}$$

where $\delta(\cdot)$ is a Dirac point mass.

7.1.2 Target densities and proposal kernels

The choice of incremental target densities is application-specific; innovation in SMC algorithms often involves proposing novel manners for constructing sequences of

intermediate distributions. These incremental densities do not necessarily need to correspond to marginal distributions of full target. Particularly relevant recent work directed towards improving SMC inference in the same class of models we address includes the Biips ordering and arrangement algorithm [Todeschini et al., 2014], the divide and conquer approach [Lindsten et al., 2014], and heuristics for scoring orderings in general factor graphs [Naesseth et al., 2014, 2015]. All these methods provide a means for selecting a sequence of intermediate target densities — however, given a sequence of targets, one still must supply an appropriate proposal density.

The ideal choice for this proposal in general is found by proposing directly from the incremental change in densities [Doucet et al., 2000], with

$$q_n^{opt}(x_n|x_{1:n-1}) = \frac{\pi_n(x_{1:n})}{\pi_{n-1}(x_{1:n-1})} \propto \frac{\gamma_n(x_{1:n})}{\gamma_{n-1}(x_{1:n-1})}. \tag{7.6}$$

Using this proposal, each of the unnormalized weights in Equation (7.3) are independent of the sampled values of x_n^k. In practice this conditional density is nearly always intractable, and one must resort to approximation.

Adaptive importance sampling methods aim to learn the optimal proposal online during the course of inference, immediately prior to proposing values for the next target density. In both in the context of population Monte Carlo (PMC) Cappé et al. [2008] and sequential Monte Carlo [Cornebise et al., 2008, 2014, Gu et al., 2015], a parametric family $q(\mathbf{x}|\lambda)$ is proposed, with λ is a free parameter, and the adaptive algorithms aim to minimize either the reverse Kullback-Leibler (KL) divergence or Chi-squared distance between the approximating family $q(\mathbf{x}|\lambda)$ and the optimal proposal density. This can be optimized via stochastic gradient descent [Gu et al., 2015], or for specific forms of q by online Monte Carlo expectation maximization, both for population Monte Carlo [Cappé et al., 2008] and in state-space models [Cornebise et al., 2014]. Note that this is the reverse of the KL divergence traditionally used in variational inference [Jordan et al., 1999], and takes the form of an expectation with respect to the intractable target distribution.

7.1.3 Neural autoregressive distribution estimation

As a general model class for $q(\mathbf{x}|\cdot)$, we adapt recent advances in flexible neural network density estimators, appropriate for both discrete and continuous high-dimensional data. We focus particularly on the use of autoregressive neural network density estimation models [Bengio and Bengio, 1999, Larochelle and Murray, 2011, Uria et al., 2013, Germain et al., 2015] which model high-dimensional distribution by learning a sequence of one-dimensional conditional distributions; that is, learning each product term in

$$p(\mathbf{x}) = \prod_{n=1}^{N} p(x_n|x_1, \ldots, x_{n-1}), \tag{7.7}$$

typically with weight parameter sharing across densities.

We choose to adapt the masked autoencoder for distribution estimation (MADE) model [Germain et al., 2015], which fits an autoregressive model to binary data, with structure inspired by autoencoders. In its simplest form, a single-layer MADE model described on $N-$dimensional binary data $\mathbf{x} \in [0,1]^N$ has a hidden layer $\mathbf{h}(\mathbf{x})$ and output $\hat{\mathbf{x}}$ with

$$\mathbf{h}(\mathbf{x}) = \sigma_w(\mathbf{b} + (\mathbf{W} \odot \mathbf{M}_w)\mathbf{x}) \tag{7.8}$$

$$\hat{\mathbf{x}} = \sigma_v(\mathbf{c} + (\mathbf{V} \odot \mathbf{M}_v)\mathbf{h}(\mathbf{x})), \tag{7.9}$$

where $\mathbf{b}, \mathbf{c}, \mathbf{W}, \mathbf{V}$ are real-valued parameters to be learned, $\odot$ denotes elementwise multiplication, σ_w, σ_v are nonlinear functions, and $\mathbf{M}_w, \mathbf{M}_v$ are fixed binary masks. Critically, the construction of the masks is such that computing the network output for each $\hat{x}_n$ requires only the inputs $x_1, \ldots, x_{n-1}$, with the zeros in the masks dropping the connections. The masks are generated by assigning each unit in each hidden layer a number from $1, \ldots, N-1$, describing which of the dimensions $x_1, \ldots, x_{n-1}$ it is permitted to take as input; output units then are only permitted to take as input hidden nodes numbered lower than their output.

With a logistic function sigmoid as σ_v, then $\hat{x}_n$ can be interpreted as a probability $p(x_n|x_1, \ldots, x_{n-1})$, and to compute $\hat{x}_n$ one does not need supply any value as input to $\mathbf{h}(\mathbf{x})$ for the dimensions $x_n, \ldots, x_N$. That is, if one follows all connections "back"

through the network from $\hat{x}_n$ to the input $\mathbf{x}$, one would find only themselves at $x_1, \ldots, x_{n-1}$.

7.2 Approach

Our approach is two-fold. First, given a Bayesian network that acts as a generative model for our observed data $\mathbf{y}$ given latent variables $\mathbf{x}$, we construct a new Bayesian network which acts as a generative model for our latent $\mathbf{x}$, given observed data $\mathbf{y}$. This network is constructed such that the joint distribution of the new "inverse model", which we will refer to as $\tilde{p}(\mathbf{x}, \mathbf{y}) = \tilde{p}(\mathbf{y})\tilde{p}(\mathbf{x}|\mathbf{y})$, preserves the conditional dependence structure in the original model $p(\mathbf{x}, \mathbf{y}) = p(\mathbf{x})p(\mathbf{y}|\mathbf{x})$, but has a different factorization [Stuhlmüller et al., 2013].

Unfortunately, unlike the original forward model, the inverse model has conditional densities which we do not in general know how to normalize or sample from. However, were we to know the full conditional density of the inverse model $\tilde{p}(\mathbf{x}|\mathbf{y})$, then we could directly draw samples of $\mathbf{x}$ given a particular dataset $\mathbf{y}$.

Thus our second task is to learn approximations for the conditionals $\tilde{p}(x_i|\widetilde{\text{PA}}(\mathbf{x}_i))$, where $\widetilde{\text{PA}}(x_i)$ are parents of x_i in the inverse model. To do so we employ neural density estimators and design a procedure to train these "offline", in the sense that no real data is required.

7.2.1 Defining the inverse model

We begin by constructing an inverse model $\tilde{p}(\mathbf{x}, \mathbf{y})$ which admits the same distribution over all random variables as $p(\mathbf{x}, \mathbf{y})$, but with a different factorization. We first note that the directed acyclic graph structure of $p(\mathbf{x}, \mathbf{y})$ imposes a partial ordering on all random variables $\mathbf{x}$ and $\mathbf{y}$; we choose any single valid ordering arbitrarily, and define the sequences $x_1, \ldots, x_N$ and $y_1, \ldots, y_M$ such that for any x_i, $\text{PA}(x_i) \subseteq \{x_1, \ldots, x_{i-1}\} \cup \{y_j\}_{j=1}^{M}$, and for any y_j, $\text{PA}(y_j) \subseteq \{y_1, \ldots, y_{j-1}\} \cup \{x_i\}_{i=1}^{N}$.

Our goal here is to construct as simple as possible a distribution $\tilde{p}(\mathbf{x}|\mathbf{y})$ whose factorization does not introduce any new conditional independencies not also present in the original generative model. Consider two extremes: a fully factorized $\tilde{p}(\mathbf{x}|\mathbf{y}) \equiv$

$\prod_{i=1}^{N} \tilde{p}(x_i|\mathbf{y})$ which assumes all x_i are conditionally independent given $\mathbf{y}$ may be attractive for computational reasons, but fails to capture all the structure of the posterior; whereas a fully connected $\tilde{p}(\mathbf{x}|\mathbf{y}) \equiv \prod_{i=1}^{N} \tilde{p}(x_i|x_{1:i-1}, \mathbf{y})$ is guaranteed to capture all dependencies, but may be unnecessarily complex.

To define the approximating distribution at each x_i, we invert the dependencies on y_j, effectively running the generative model backwards. Following the heuristic algorithm of Stuhlmüller et al. [2013], we do this by literally constructing the dependency graph in reverse. Ordering the random variables $y_M, \ldots, y_1, x_N, \ldots, x_1$, we define a new parent set $\widetilde{\text{PA}}(x_i)$ for each x_i in the transformed model, with $\widetilde{\text{PA}}(x_i) \subseteq \{x_{i+1}, \ldots, x_N, y_1, \ldots, y_M\}$. Define the Markov blanket $\text{MB}(x_i)$ to be the set of all random variables which share a factor with x_i; that is, the union of the parents of x_i, the children of x_i, and the parents of the children of x_i. Then defining the parent sets in the transformed model as

$$\widetilde{\text{PA}}(x_i) = \text{MB}(x_i) \cap \{x_{i+1}, \ldots, x_N, y_1, \ldots, y_M\}$$
$$\widetilde{\text{PA}}(y_j) = \text{MB}(y_j) \cap \{y_{j+1}, \ldots, y_M\}$$

yields a model with the same local dependency structure as the original model $p(\mathbf{x}, \mathbf{y})$; however, now the sequence is reversed such that the observed values are inputs (i.e., $\widetilde{\text{PA}}(y_j) \cap \mathbf{x} = \emptyset$). The sequence under the new model, which we will refer to as $\tilde{p}(\mathbf{x}, \mathbf{y})$, factorizes naturally as $\tilde{p}(\mathbf{x}, \mathbf{y}) = \tilde{p}(\mathbf{x}|\mathbf{y})\tilde{p}(\mathbf{y})$; particularly important to us is the factorization of the conditional density $\tilde{p}(\mathbf{x}|\mathbf{y}) = \prod_{i=1}^{N} \tilde{p}(x_i|\widetilde{\text{PA}}(x_i))$.

This algorithm produces inverse graph structures which despite not being fully connected, preserve local conditional dependencies in the original graph:

Proposition 1. *Preservation of local conditional dependence.* Let x_A, x_B, x_C be latent or observed random variables in $p(\mathbf{x})$ with graph structure G, and with each of x_A, x_B, x_C adjacent to at least one of the others under G. Then let $\tilde{x}_A, \tilde{x}_B, \tilde{x}_C$ denote the corresponding random variables in the inverse model $\tilde{p}(\mathbf{x})$ with graph structure $\widetilde{G}$, constructed via the algorithm above. If $\tilde{x}_A$ and $\tilde{x}_B$ are conditionally independent

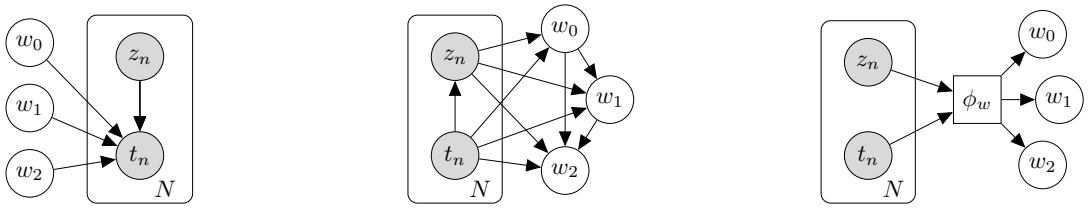


Figure 7.1: A non-conjugate regression model, as (left) a Bayes net representing a generative model for the data $\{t_n\}$; (middle) with dependency structure inverted, as a generative model for the latent variables w_0, w_1, w_2; (right) showing the explicit neural network structure of the learned approximation to the inverse conditional distribution $\tilde{p}(w_{0:2}|z_{1:N}, t_{1:N})$. New datasets $\{z_n, t_n\}_{n=1}^N$ can be input directly into the joint density estimator ϕ_w to estimate the posterior. Note that the ordering of the latent variables $w_{0:2}$ used in this example is chosen arbitrarily; any permutation of the latent variables would not change the overall structure of the inverse model.

given $\tilde{x}_C$ in the inverse model $\widetilde{G}$, they were also conditionally independent in the original model G; that is,

$$\tilde{x}_A \perp\!\!\!\perp \tilde{x}_B \Big| \tilde{x}_C \quad \Rightarrow \quad x_A \perp\!\!\!\perp x_B \Big| x_C.$$

Proof. Suppose we had a conditional dependence in G which was not preserved in $\widetilde{G}$, i.e. with $x_A \not\perp\!\!\!\perp x_B \Big| x_C$ but $\tilde{x}_A \perp\!\!\!\perp \tilde{x}_B \Big| \tilde{x}_C$. Without loss of generality assume $\tilde{x}_B$ was added to the inverse graph prior to $\tilde{x}_A$, i.e. $x_A \prec x_B$ in G. Note that $x_A \not\perp\!\!\!\perp x_B \Big| x_C$ can occur either due to a direct dependence between x_A and x_B, or, due to both $x_A, x_B \in \text{PA}(x_C)$; in either case, $x_B \in \text{MB}(x_A)$. Then when adding $\tilde{x}_B$ to the inverse graph $\widetilde{G}$ we are guaranteed to have $\tilde{x}_B \in \widetilde{\text{PA}}(\tilde{x}_A)$, in which case $\tilde{x}_A \not\perp\!\!\!\perp \tilde{x}_B$. □

Examples of generative models and their corresponding inverse models are shown in Figures 7.1–7.3. Note that as the topological sort of the nodes in the original generative model is not unique, neither is the inverse graphical model.

7.2.2 Learning a family of approximating densities

Following Cappé et al. [2008], learning proposals for importance sampling on $\pi(\mathbf{x})$ in a single-dataset setting (i.e., with fixed $\mathbf{y}$) entails proposing a parametric family $q(\mathbf{x}|\lambda)$, where λ is a free parameter, and then choosing λ to minimize

$$D_{KL}(\pi||q_\lambda) = \int \pi(\mathbf{x}) \log \left[\frac{\pi(\mathbf{x})}{q(\mathbf{x}|\lambda)} \right] \mathrm{d}\mathbf{x}. \tag{7.10}$$

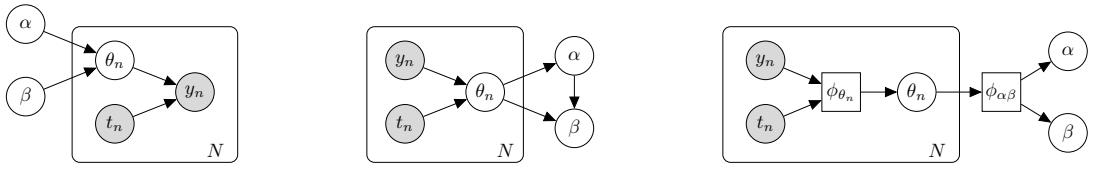


Figure 7.2: A hierarchical Bayesian model. (left) A generative model for the data $\{x_n\}$; (middle) with dependency structure inverted; (right) showing the two distinct joint neural conditional density estimators. Note in particular the inverse model still partially factorizes across the latent variables. The learned factor ϕ_{θ_n} is replicated N times in the inverse model, allowing re-use of weights, simplifying training.

This KL divergence between the true posterior distribution $\pi(\mathbf{x}) \equiv p(\mathbf{x}|\mathbf{y})$ and proposal distribution $q(\mathbf{x}|\lambda)$ is also known as the relative entropy criterion, and is a preferred objective function in situations in which the estimation goal construct a high-quality weighted sample representation, rather than to minimize the variance of a particular expectation [Cornebise et al., 2008].

In an amortized inference setting, instead of learning λ explicitly for a fixed value of $\mathbf{y}$, we learn a mapping from $\mathbf{y}$ to λ. More explicitly, if $\mathbf{y} \in \mathcal{Y}$ and $\lambda \in \vartheta$, then learning a deterministic mapping $\varphi : \mathcal{Y} \to \vartheta$ allows performing approximate inference for $p(\mathbf{x}|\mathbf{y})$ with only the computational complexity of evaluating the function φ. The tradeoff is that the training of φ itself may be quite involved.

We thus generalize the adaptive importance sampling algorithms by learning a family of distributions $q(\mathbf{x}|\mathbf{y})$, parameterized by the observed data $\mathbf{y}$. Suppose that $\lambda = \varphi(\eta, \mathbf{y})$, where the function φ is parameterized by a set of upper-level parameters η. We would like a choice of η which performs well across all datasets $\mathbf{y}$. We can frame this as minimizing the expected value of Eq. (7.10) under $p(\mathbf{y})$, suggesting an objective function $\mathcal{J}(\eta)$ defined as

$$\begin{aligned}\mathcal{J}(\eta) &= \int D_{KL}(\pi||q_\lambda)p(\mathbf{y})\mathrm{d}\mathbf{y} \\ &= \int p(\mathbf{y}) \int p(\mathbf{x}|\mathbf{y}) \log\left[\frac{p(\mathbf{x}|\mathbf{y})}{q(\mathbf{x}|\varphi(\eta,\mathbf{y}))}\right] \mathrm{d}\mathbf{x}\mathrm{d}\mathbf{y} \\ &= \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right] + const \end{aligned} \tag{7.11}$$

which has a gradient

$$\nabla_\eta \mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x},\mathbf{y})}\left[-\nabla_\eta \log q(\mathbf{x}|\varphi(\eta,\mathbf{y}))\right]. \tag{7.12}$$

Notice that these expectations in Equations (7.11) and (7.12) are with respect to the tractable joint distribution $p(\mathbf{x}, \mathbf{y})$. We can thus fit η by stochastic gradient descent, estimating the expectation of the gradient $\nabla_\eta \mathcal{J}(\eta)$ by sampling synthetic full-data training examples $\{\mathbf{x}, \mathbf{y}\}$ from the original model. This procedure can be performed entirely offline — we require only to be able to sample from the joint distribution $p(\mathbf{x}, \mathbf{y})$ to generate candidate data points (effectively providing infinite training data). In any directed graphical model this can be achieved by ancestral sampling, where in addition to sampling $\mathbf{x}$ we sample values of the as-yet unobserved variables $\mathbf{y}$. Furthermore, we do not need need to be able to compute gradients of our model $p(\mathbf{x}, \mathbf{y})$ itself — we only need the gradients of our recognition model $q(\mathbf{x}|\varphi(\eta, \mathbf{y}))$, allowing use of any differentiable representation for q. We choose the parametric family $q(\mathbf{x}|\lambda)$ and the transformation φ such that this inner gradient in Eq. (7.12) can be computed easily.

We can now use the conditional independence structure in our inverse model $\tilde{p}(\mathbf{x}, \mathbf{y})$ to break down $q(\mathbf{x}|\lambda)$, an approximation of $\tilde{p}(\mathbf{x}|\mathbf{y})$, into a product of smaller conditional densities each approximating $\tilde{p}(x_i|\widetilde{\mathrm{PA}}(x_i))$. The full proposal density $q(\mathbf{x}|\varphi(\eta, \mathbf{y}))$ can be decomposed as

$$q(\mathbf{x}|\varphi(\eta, \mathbf{y})) = \prod_{i=1}^{N} q_i(x_i|\varphi_i(\eta_i, \widetilde{\mathrm{PA}}(x_i))) \tag{7.13}$$

with the gradient similarly decomposing as

$$\nabla_{\eta_i} \mathcal{J}(\eta) = \mathbb{E}_{p(\mathbf{x},\mathbf{y})} \left[-\nabla_{\eta_i} \log q_i(x_i|\varphi_i(\eta_i, \widetilde{\mathrm{PA}}(x_i))) \right].$$

Each of these expectations requires only samples of the random variables in $\{x_i\} \cup \widetilde{\mathrm{PA}}(x_i)$, reducing the dimensionality of the joint optimization problem.

7.2.3 Joint conditional neural density estimation

We particularly wish to construct the inverse factorization $\tilde{p}(\mathbf{x}|\mathbf{y})$ (and our proposal model $q(\cdot)$) in such a way that we deal naturally with the presence of head-to-head nodes, in which one random variable may have a very large parent set. This situation is common in machine learning models: it is quite common to have generative

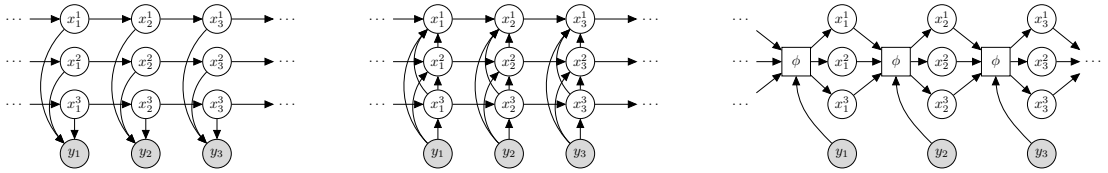


Figure 7.3: Factorial HMM. (left) The generative model consists D independent Markov models, with observed data y_t depending on the current state of each latent HMM. (middle) An inverse model obtained by reversing the order of the generative model at each t. Conditioned on the previous latent states at $t-1$ and the next observation y_t, all latent states at each t are dependent on one another and must be modeled jointly. (right) The repeated structure at each $t = 1, 2, \ldots$ means that the same learned conditional density network can be reused at every t.

models which factorize in the joint distribution, but have complex dependencies in the posterior; see for example the model in Figure 7.1.

We thus choose to treat all such situations in our inverse factorization — where a sequence of variables $\mathbf{x}' \subseteq \mathbf{x}$ are fully dependent on one another after conditioning on a shared set of parent nodes $\widetilde{\text{PA}}(\mathbf{x}')$ — as a single joint conditional density which we will approximate with an autoregressive density model. We extend MADE [Germain et al., 2015] to function as a conditional density estimator by allowing it to take $\widetilde{\text{PA}}(\mathbf{x}')$ as additional inputs, and constructing the masks such that these additional inputs are propagated through all hidden layers to all outputs, even for the very first dimension. As in MADE this can be achieved by labeling the hidden units with integers denoting which input dimensions they are allowed to accept. In contrast to the original MADE, we label hidden units with numbers from $0, \ldots, N-1$, where hidden units labeled 0 to take as input only the dimensions in $\widetilde{\text{PA}}(\mathbf{x}')$. For single-dimensional data, where $N = 1$, all hidden units are labeled 0 and all feed forward into the single output x_1, recovering a standard mixture density network [Bishop, 1994].

To model non-binary data, MADE can be extended by altering the output layer network to emit parameters of any univariate probability density function. We take the same approach by which RNADE [Uria et al., 2013] modifies the binary autoregressive distribution estimator NADE [Larochelle and Murray, 2011] to handle real-valued data, with an output layer that parameterizes a univariate

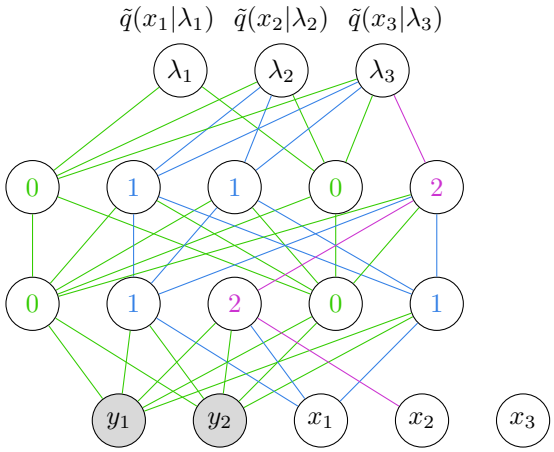


Figure 7.4: Structure of a conditional neural density estimator. The colors of the edges denote the influence of each particular random variable on the intermediate nodes, and on the overall networks generating each parameter. Green edges have inputs which only on the observed variables $\mathbf{y}$; the blue edges may also have x_1 as an input, and the purple edges additionally depend on the value of x_2. The output from each network is the parameters to a distribution governing the conditional distribution of each x_j. The integers on each hidden node denote the total number of latent variables which constitute input to that node; these integers are the sampled values used to construct the input masks.

mixture of D Gaussians for each dimension x_i conditioned on its parents. The probability of any particular x_i is given by

$$q(x_i|\varphi_i(\eta_i, \widetilde{\mathrm{PA}}(x_i))) = \sum_{d=1}^{D} \alpha_{i,d} \mathcal{N}(x_i|\mu_{i,d}, \sigma_{i,d}^2)$$

where $\mathcal{N}(\cdot)$ is the Gaussian probability density. This requires an output layer with $3 \times D$ dimensions, to predict D each of means $\mu_{i,d}$, standard deviations $\sigma_{i,d}$, weights $\alpha_{i,d}$; to enforce positivity of standard deviations we apply a softplus function to the raw network outputs, and a softmax function to ensure $\alpha_{i,\cdot}$ is a probability vector.

A schematic diagram illustrating the network connections in a conditional MADE is shown in Figure 7.4.

7.2.4 Training the neural network

Contrary to many standard settings in which one is limited by the amount of data present, we are armed with a sampler $p(\mathbf{x}, \mathbf{y})$ which allows us to generate effectively infinite training data. This could be used to sample a "giant" synthetic dataset, which we then use for mini-batch training via gradient descent; however, then we

must decide how large a dataset is required. Alternatively, we could sample a brand new set of training examples for every mini-batch, never re-using previous samples.

In testing we found that a hybrid training procedure, which samples new synthetic datasets based on performance on a held-out set of synthetic validation data, appeared more efficient than resampling a new synthetic dataset for each new gradient update. We perform mini-batch gradient updates on η using synthetic training data, while evaluating on the validation set. If the validation error increases, or after a set maximum number of steps, we draw new sets of both synthetic training and validation data from $p(\mathbf{x}, \mathbf{y})$.

In all experiments we use Adam [Kingma and Ba, 2015] with the suggested default parameters to update learning rates online, and use rectified linear activation functions.

7.3 Examples

7.3.1 Inverting a single factor

To illustrate the basic method for inverting factors, we consider a non-conjugate polynomial regression model, with global-only latent variables. The graphical model, its inversion, and the neural network structure are shown in Figure 7.1. Here we place a Laplace prior on the regression weights, and have Student-t likelihoods, giving us

$$\begin{aligned} w_d &\sim \text{Laplace}(0, 10^{1-d}) && \text{for } d = 0, 1, 2; \\ t_n &\sim \text{t}_\nu(w_0 + w_1 z_n + w_2 z_n^2, \epsilon^2) && \text{for } n = 1, \ldots, N \end{aligned}$$

for fixed $\nu = 4, \epsilon = 1$, and $z_n \in (-10, 10)$ uniformly. The goal is to estimate the posterior distribution of weights for the constant, linear, and quadratic terms, given any possible collected dataset $\{z_n, t_n\}_{n=1}^N$. In the notation of the preceding sections, we have latent variables $\mathbf{x} \equiv \{w_0, w_1, w_2\}$ and observed variables $\mathbf{y} \equiv \{z_n, t_n\}_{n=1}^N$.

Note particularly that although the original graphical model which expressed $p(\mathbf{y}|\mathbf{x})p(\mathbf{x})$ factorizes into products over y_n which are conditionally independent given $\mathbf{x}$, in the inverse model $\tilde{p}(\mathbf{x}|\mathbf{y})$ due to the explaining-away phenomenon all latent variables depend on all others: there are no latent variables which can be

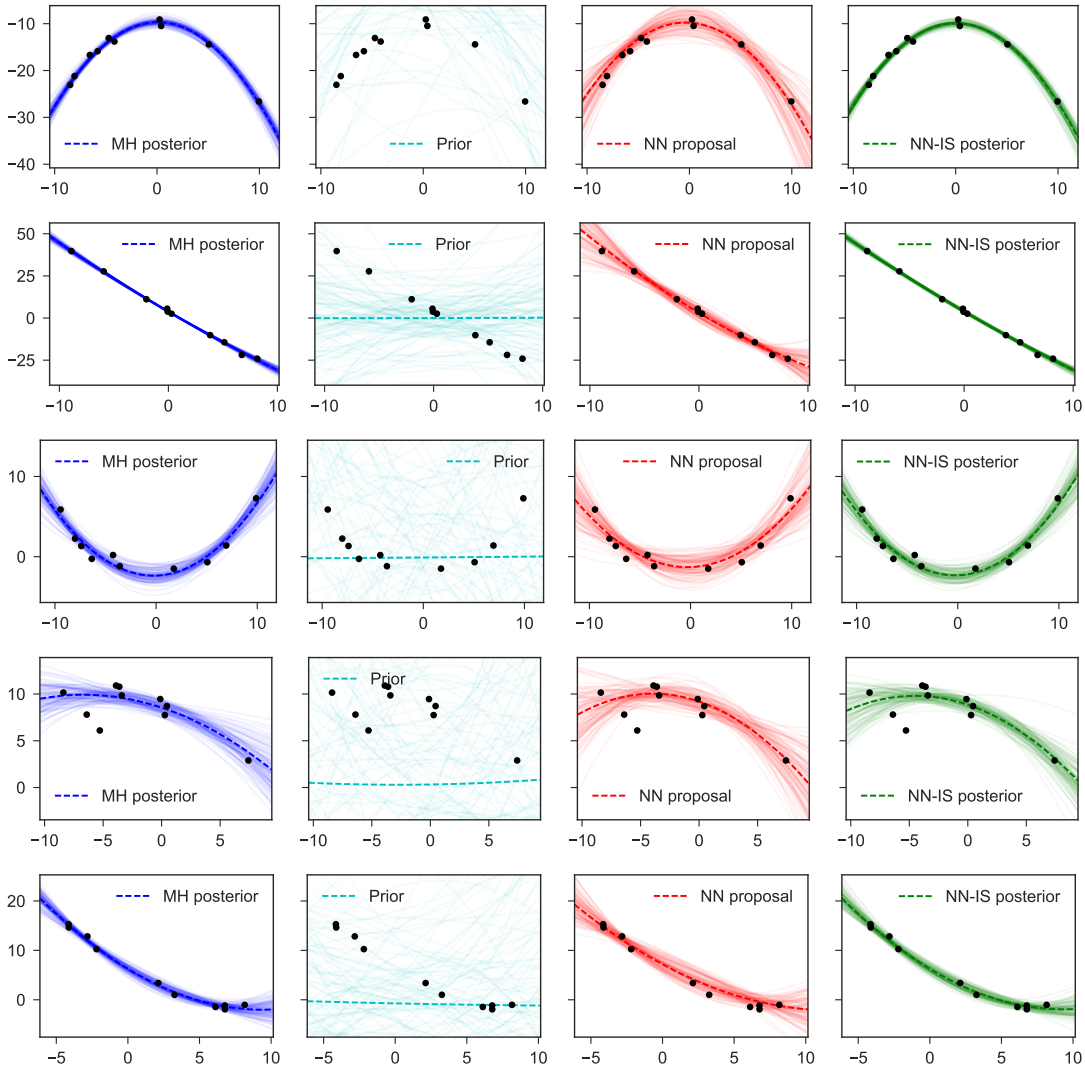


Figure 7.5: Representative output in the polynomial regression example. Plots show 100 samples each at 5% opacity, with the mean marked as a solid dashed line. These are all proposed using the same pre-trained neural network — not just the same neural network structure, but also identical learned weights. The MCMC posterior is generated by thinning 10000 samples by a factor 100, after 10000 samples of burnin. The neural network proposal yields estimated polynomial curves close to the true posterior solution, albeit slightly more diffuse.

d-separated from the observed $\mathbf{y}$, and all latent variables share $\mathbf{y}$ as parents. This means we fit as proposal only a single joint density $q(w_{0:2}|z_{1:N}, t_{1:N})$. Examples of representative output from this network are shown in Figure 7.5. The trained network used here 300 hidden units in each of two hidden layers, and a mixture of 3 Gaussians as each output.

7.3.2 A hierarchical Bayesian model

Consider as a new example a representative multilevel model where exact inference is intractable, a Poisson model for estimating failure rates of power plant pumps [George et al., 1993]. Given N power plant pumps, each having operated for t_n thousands of hours, we see x_n failures, following

$$\alpha \sim \text{Exponential}(1.0), \qquad \beta \sim \text{Gamma}(0.1, 1.0),$$
$$\theta_n \sim \text{Gamma}(\alpha, \beta), \qquad y_n \sim \text{Poisson}(\theta_n t_n).$$

The graphical model, an inverse factorization, and the neural network structure are shown in Figure 7.2. To generating synthetic training data, t_n are sampled *iid* from an exponential distribution with mean 50.

The repeated structure in the inverse factorization of this model allows us to learn a single inverse factor to represent the distribution $\tilde{p}(\theta_n|t_n, y_n)$ across all n. This yields a far simpler learning problem than were we forced to fit all of $\tilde{p}(\theta_{1:N}|t_{1:N}, y_{1:N})$ jointly. Further, the repeated structure allows us to use a divide-and-conquer SMC algorithm Lindsten et al. [2014] which works particularly efficiently on this model. Each of the N replicated structures are sampled in parallel with independent particle sets, weighted locally, and resampled; once all θ_n are sampled, we end by sampling α and β jointly, which need both be included in order to evaluate the final terms in the joint target density. We stress that there is no obvious baseline proposal density to use for a divide-and-conquer SMC algorithm, as neither the marginal prior nor posterior distributions over θ_n are available in closed form. Any usage of this algorithm requires manual specification of some proposal $q(\theta_n)$.

We test our proposals on the actual power pump failure data analyzed in George et al. [1993]. The relative convergence speeds of marginal likelihood estimators from importance sampling from prior and neural network proposals, and SMC with neural network proposals, are shown in Figure 7.6. To capture the wide tails of the broad gamma distributions, we use a mixture of 10 Gaussians here at each output node, and 500 hidden units in each of two hidden layers.

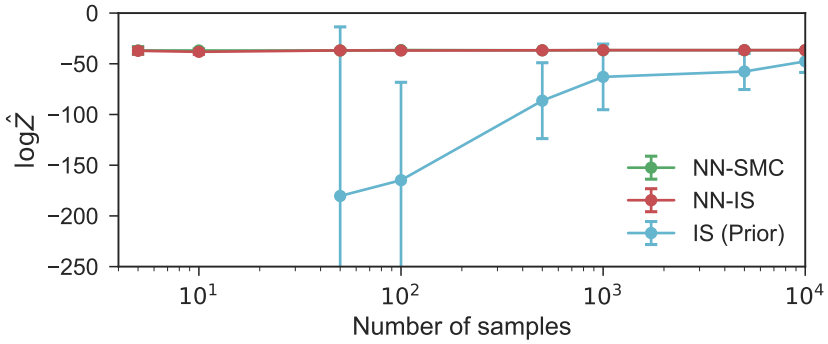


Figure 7.6: Convergence of marginal likelihood estimate as a function of number of particles, for likelihood-weighted importance sampling, neural network importance sampling, and a divide-and-conquer sequential Monte Carlo algorithm with neural network proposals. The SMC algorithm can achieve reasonable estimates of the normalizing constant with as few as 5 samples. Plot shows mean of 10 runs; error bars show two standard deviations.

7.3.3 Factorial hidden Markov model

Proposals can also be learned to approximate the optimal filtering distribution in models for sequential data; we demonstrate here on a factorial hidden Markov model [Ghahramani and Jordan, 1997], where each time step has a combinatorial latent space. The additive model we consider is inspired by the model studied in Kolter and Jaakkola [2012] for disaggregation of household energy usage; effective inference in this model is a subject of continued research. Some number of devices D are either in an active state, in which case each device i consumes μ^i units of energy, or it is off, in which case it consumes no energy. At each time step we receive a noisy observation of the total amount of energy consumed, summed across all devices. This model, whose graphical model structure is shown in Figure 7.3, can be represented as

$$x_t^i | x_{t-1}^i \sim \text{Bernoulli}(\theta^i[x_{t-1}^i])$$
$$y_t | x_t^1, \ldots, x_t^D \sim \mathcal{N}\Big(\textstyle\sum_{i=1}^D \mu^i x_t^i, \sigma^2\Big),$$

where θ^i represents the prior probability of devices switching on or off at each time increment. We design a synthetic example with $D = 20$, meaning each time step has $2^{20} \approx 100,000$ possible discrete states; the parameters μ^d are spread out from 30 to 500, with $\sigma = 20$. Each individual device has an initial probability 0.1 of being activated at $t = 1$, switching state at subsequent t with probability 0.05.

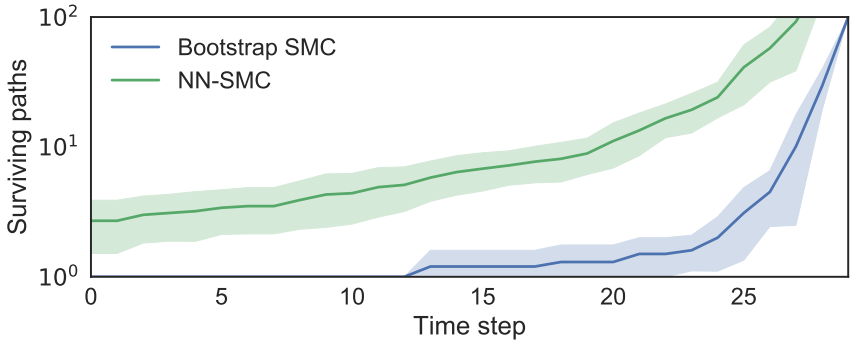


Figure 7.7: Learned proposals reduce particle degeneracy in the factorial HMM. Here we show the number of unique ancestries which survive over the course of 30 time steps, running 100 particles. Proposing from the transition dynamics nearly immediately degenerates to a single possible solution; the learned proposals increase the effective sample size at each stage and reduce the need for resampling. Plot shows mean and standard deviation over 10 runs.

As different combinations of devices can yield identical total energy usage it is impossible to disambiguate between different combinations of active devices from a single observation, meaning any successful inference algorithm must attempt to mix across many disconnected modes over time to preserve the multiple possible explanations. Synthetic data and example output of inference shown in Figure **??** demonstrates the algorithm successfully recovering differing interpretations. The effect of the learned proposals on the overall number of surviving particles is shown in Figure 7.7. Our proposal model uses D Bernoulli outputs in a 4-layer network, with 300 units per hidden layer; it takes as input the D latent states at the previous time $t-1$, as well as the current observation y_t.

7.4 Discussion

We present this work primarily as a manner by which we compile away application-time inference costs when performing SMC, and automating the manual task of designing proposal densities. However, in some situations direct sampling from the model may provide a satisfactory approximation even eschewing importance weighting steps; in such cases our approach can be viewed as a graphical-model-regularized algorithm for designing and training neural networks with interpretable structural representations. Rather than learning from data, the emulator model

is chosen to approximate the specified generative model, akin to the "sleep" cycle of the wake-sleep algorithm Hinton et al. [1995].

In contrast to variational autoencoders [Kingma and Welling, 2014], where one simultaneously learns parameters for both the inference network and generative model from data, we assume a known generative model with fixed parameters and structured, interpretable latent variables. This provides robustness to bias arising from training data which comes from an unrepresentative sample, and also allows us to apply our method in situations where a sufficiently large supply of exemplar data is unavailable. However, it does require placing trust in the generative model: in particular, it requires a generative model which could plausibly create the data we will later collect and condition on.

Beyond these differences, our choice of $D_{KL}(\pi||q)$, the same minimized by EP, leads to approximations more appropriate for SMC refinement than a variational Bayes objective function; see e.g. Minka [2005] for a discussion of "zero-forcing" behavior, and e.g. Cappé et al. [2008] for a discussion of pathological cases in learned importance sampling distributions.

Finally, we note there exist a few remaining challenges before this can be implemented automatically in a probabilistic programming system. First, for languages such as Anglican and probabilistic C, this is not immediately applicable, as in defining the proposals we have taken advantage of both the known model dependency structure, and the fixed dimensionality of the model. In theory the issue of varying model dimensionality could be addressed by replacing the deterministic sequence of conditional distributions with the output of a recurrent neural network, which takes current addresses as input; we leave this to future work.

However even for languages such as Bugs, where these assumptions hold, it remains an open question how to automatically choose an ordering over the latent variables for running an SMC algorithm. In order to find parameter efficient representations such as used in Poisson example, and in the factorial HMM, we also need a manner to find and identify repeated structures which can share a learned inverse. This is presumably possible through static analysis of Bugs

code (through looking at e.g. the definition of `for` loops), but defining a precise algorithm requires additional work.

8

Future directions for probabilistic programming

In this thesis we have seen that it is possible to define probability distributions as arbitrary stochastic simulation code, and then use model-agnostic implementations of inference algorithms to perform posterior inference automatically. The use of higher-order probabilistic programming languages allows us to express models of arbitrary complexity with a great degree of ease and familiarity.

However, we have also seen that performing inference in this setting is often more challenging than in restricted model classes where we may make additional assumption about the model (e.g. differentiability, or fixed dimensionality), and where we may have more information about model structure (e.g. factorization). In models where we have early, incremental access to feedback when executing the probabilistic program, then sequential Monte Carlo methods described are a natural fit and can be surprisingly performant for a general-purpose implementation; for some complex models, there is sometimes no other option available.

For other models, where little is gained from a sequential decomposition of the simulation, more work is required to improve statistical and computational efficiency. For SMC-based inference engines, the primary bottleneck appears to be

the difficulty in specification of efficient proposal kernels; online or offline schemes such as those presented in this thesis can help learn these proposals automatically.

Given the challenges we have faced, it is natural to wonder: is the extra expressivity afforded by higher-order probabilistic languages truly necessary to build meaningful probabilistic models? In the long run, and in many subject domains, the answer is probably yes. Many modeling problems are naturally framed compositionally, in which simpler models are built on top of each other in a way that is nondeterministic. This leads to an uncertainty in the total number of latent variables (or equivalently, in the size of the model); this observation has motivated not just Bayesian nonparametric methods mentioned already, but also recent developments in neural networks, such as the use of recurrent neural networks for attention modeling [Eslami et al., 2016]. Furthermore, many systems truly are discrete or combinatorial; for example, the Canadian traveller problem policy search example from Chapter 6, or the energy disaggregation example we considered in Chapter 7.

We then ask whether, if models are naturally framed compositionally, whether inference can be framed in such a manner as well. When writing non-probabilistic programs, we group source code into natural units of functions, classes, methods, packages, modules, and so on. The choices of structures lead to different programming design patterns — functional programming styles, object oriented programming, etc. What analogies do we need for probabilistic programs? How should we conceptually break down large probabilistic programs into smaller units? This is an open question, but likely necessary for scaling inference to larger and more complex models without restrictions on expressivity. In a system where the models can be constructed compositionally, it would be advantageous to also construct inference algorithms compositionally: sub-models which are amenable to message passing (or even exact inference) can be computed efficiently; sub-models which can be differentiated can use more efficient sampling proposals based on HMC or efficient reparameterized gradient estimators for variational inference; then, those

components which do not fit nicely into the domain of such inference algorithms can be addressed with more general methods discussed in this thesis.

It is also my hope that advances in amortized inference schemes such as in Chapter 7 can successfully yield efficient inference in universal probabilistic programming languages. To some degree, higher-order probabilistic programming languages are already efficient tools for model exploration. While the inference methods are often slow, if the slow procedure can be offset by up-front model compilation, then probabilistic programming languages can be used to define and test models on small data for rapid prototyping and for testing assumptions; once a practitioner is satisfied with a model, then it can be compiled into standalone statistical software which can be used to perform inference later on demand for new, larger data.

Bibliography

Nathanael L Ackerman, Cameron E Freer, and Daniel M Roy. Noncomputable conditional distributions. In *Logic in Computer Science (LICS), 2011 26th Annual IEEE Symposium on*, pages 107–116. IEEE, 2011.

Christophe Andrieu, Arnaud Doucet, and Roman Holenstein. Particle Markov chain Monte Carlo methods. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 72(3):269–342, 2010.

Andrew W Appel. *Compiling with continuations*. Cambridge University Press, 2007.

Yoshua Bengio and Samy Bengio. Modeling high-dimensional discrete data with multi-layer neural networks. In *Advances in Neural Information Processing Systems*, volume 99, pages 400–406, 1999.

Christopher M Bishop. Mixture density networks. Technical report, 1994.

Christopher M Bishop. *Pattern recognition and machine learning*. Springer, 2006.

David M Blei, Michael I Jordan, et al. Variational inference for dirichlet process mixtures. *Bayesian analysis*, 1(1):121–144, 2006.

Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

Anthony Brockwell, Pierre Del Moral, and Arnaud Doucet. Sequentially interacting Markov chain Monte Carlo methods. *Annals of Statistics*, 38(6):3387–3411, 2010.

O Cappé, a Guillin, J. M Marin, and C. P Robert. Population Monte Carlo. *Journal of Computational and Graphical Statistics*, 13(4):907–929, December 2004. ISSN 1061-8600.

Olivier Cappé, Randal Douc, Arnaud Guillin, Jean-Michel Marin, and Christian P Robert. Adaptive importance sampling in general mixture classes. *Statistics and Computing*, 18(4):447–459, 2008.

Bob Carpenter, Matthew D Hoffman, Marcus Brubaker, Daniel Lee, Peter Li, and Michael Betancourt. The stan math library: Reverse-mode automatic differentiation in c++. *arXiv preprint arXiv:1509.07164*, 2015.

James Carpenter, Peter Clifford, and Paul Fearnhead. An improved particle filter for non-linear problems. *IEE Proceedings - Radar, Sonar and Navigation*, 146(1): 2–7, Feb 1999.

Yutian Chen, Max Welling, and Alex Smola. Super-samples from kernel herding. *arXiv preprint arXiv:1203.3472*, 2012.

Nicolas Chopin, Pierre E Jacob, and Omiros Papaspiliopoulos. SMC2: an efficient algorithm for sequential analysis of state space models. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 75(3):397–426, 2013.

Julien Cornebise, Éric Moulines, and Jimmy Olsson. Adaptive methods for sequential importance sampling with application to state space models. *Statistics and Computing*, 18:461–480, 2008.

Julien Cornebise, Éric Moulines, and Jimmy Olsson. Adaptive sequential Monte Carlo by means of mixture of experts. *Statistics and Computing*, 24:317–337, 2014.

Jean Cornuet, Jean-Michel Marin, Antonietta Mira, and Christian P Robert. Adaptive multiple importance sampling. *Scandinavian Journal of Statistics*, 39(4):798–812, 2012.

D. Crisan, P. Del Moral, and T. Lyons. Discrete filtering using branching and interacting particle systems. *Markov Process. Related Fields*, 5(3):293–318, 1999.

Marc Peter Deisenroth, Gerhard Neumann, and Jan Peters. A survey on policy search for robotics. *Foundations and Trends in Robotics*, 2(2011):1–142, 2011.

Pierre Del Moral. *Feynman-Kac Formulae – Genealogical and Interacting Particle Systems with Applications*. Probability and its Applications. Springer, 2004.

Pierre Del Moral and Lawrence M Murray. Sequential Monte Carlo with highly informative observations. *SIAM/ASA Journal on Uncertainty Quantification*, 3 (1):969–997, 2015.

Randal Douc, Olivier Cappé, and Eric Moulines. Comparison of resampling schemes for particle filtering. In *In 4th International Symposium on Image and Signal Processing and Analysis (ISPA)*, pages 64–69, 2005.

Arnaud Doucet, Simon Godsill, and Christophe Andrieu. On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and computing*, 10(3): 197–208, 2000.

Arnaud Doucet, Nando De Freitas, Neil Gordon, et al. *Sequential Monte Carlo methods in practice*. Springer New York, 2001.

S. M. Ali Eslami, Daniel Tarlow, Pushmeet Kohli, and John Winn. Just-in-time learning for fast and flexible inference. In *Advances in Neural Information Processing Systems 27*, pages 154–162. 2014.

SM Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, Koray Kavukcuoglu, and Geoffrey E Hinton. Attend, infer, repeat: Fast scene understanding with generative models. *arXiv preprint arXiv:1603.08575*, 2016.

Patrick Eyerich, Thomas Keller, and Malte Helmert. High-quality policies for the Canadian traveler's problem. In *AAAI*, 2010.

Mattias Felleisen. The theory and practice of first-class prompts. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 180–190. ACM, 1988.

Dror Fried, Solomon Eyal Shimony, Amit Benbassat, and Cenny Wenner. Complexity of Canadian traveler problem variants. *Theor. Comput. Sci.*, 487:1–16, 2013.

S. Geman and D. Geman. Stochastic relaxation, Gibbs distributions, and the Bayesian restoration of images. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6:721–741, 1984.

Edward I George, UE Makov, and AFM Smith. Conjugate likelihood distributions. *Scandinavian Journal of Statistics*, pages 147–156, 1993.

Mathieu Germain, Karol Gregor, Iain Murray, and Hugo Larochelle. MADE: masked autoencoder for distribution estimation. In *Proceedings of the 32nd International Conference on Machine Learning, ICML 2015*, pages 881–889, 2015.

Samuel J Gershman and David M Blei. A tutorial on Bayesian nonparametric models. *Journal of Mathematical Psychology*, 56(1):1–12, 2012.

Samuel J Gershman and Noah D Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the Thirty-Sixth Annual Conference of the Cognitive Science Society*, 2014.

Zoubin Ghahramani and Michael I Jordan. Factorial hidden Markov models. *Machine learning*, 29(2-3):245–273, 1997.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014. Accessed: 2016-8-27.

Noah D. Goodman, Vikash K. Mansinghka, Daniel M. Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence (UAI2008)*, pages 220–229, 2008.

Andrew D Gordon, Thomas A Henzinger, Aditya V Nori, and Sriram K Rajamani. Probabilistic programming. In *Proceedings of the on Future of Software Engineering*, pages 167–181. ACM, 2014.

Neil J Gordon, David J Salmond, and Adrian FM Smith. Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F (Radar and Signal Processing)*, 140(2):107–113, 1993.

Shixiang Gu, Zoubin Ghahramani, and Richard E Turner. Neural adaptive sequential Monte Carlo. In *Advances in Neural Information Processing Systems 28*, 2015.

Nicolas Heess, Daniel Tarlow, and John Winn. Learning to pass expectation propagation messages. In *Advances in Neural Information Processing Systems 26*, pages 3219–3227. 2013.

Geoffrey E Hinton, Peter Dayan, Brendan J Frey, and Radford M Neal. The "wake-sleep" algorithm for unsupervised neural networks. *Science*, 268(5214):1158–1161, 1995.

Matthew D Hoffman and Andrew Gelman. The No-U-turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15(1):1593–1623, 2014.

Matthew D. Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic variational inference. *Journal of Machine Learning Research*, 14:1303–1347, 2013.

Roman Holenstein. *Particle Markov Chain Monte Carlo*. PhD thesis, The University of British Columbia, 2009.

Guy Houlsby. Interactions in offshore foundation design. *Géotechnique*, 66(10): 791–825, 2016.

Ferenc Huszár and David Duvenaud. Optimally-weighted herding is Bayesian quadrature. *Uncertainty in Artificial Intelligence (UAI)*, 2012.

Varun Jampani, Sebastian Nowozin, Matthew Loper, and Peter V Gehler. The informed sampler: A discriminative approach to Bayesian inference in generative computer vision models. *Computer Vision and Image Understanding*, 136:32–44, 2015.

Wittawat Jitkrittum, Arthur Gretton, Nicolas Heess, S.M. Ali Eslami, Balaji Lakshminarayanan, Dino Sejdinovic, and Zoltán Szabó. Kernel-based just-in-time learning for passing expectation propagation messages. In *Proceedings of the 31st Annual Conference on Uncertainty in Artificial Intelligence*, volume 31. AUAI Press, 2015.

Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. An introduction to variational methods for graphical models. *Machine learning*, 37(2):183–233, 1999.

Seong-Hwan Jun and Alexandre Bouchard-Côté. Memory (and time) efficient sequential monte carlo. In *Proceedings of the 31st international conference on Machine learning*, 2014.

Seong-Hwan Jun and Alexandre Bouchard-Côté. Memory (and time) efficient sequential Monte Carlo. In *Proceedings of the 31st international conference on Machine learning*, pages 514–522, 2014.

Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2015.

Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *Proceedings of the International Conference on Learning Representations (ICLR)*, 2014.

Oleg Kiselyov and Chung-chien Shan. Monolingual probabilistic programming using generalized coroutines. In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence (UAI2009)*, 2009.

Nathan L Kleinman, James C Spall, and Daniel Q Naiman. Simulation-based optimization with stochastic approximation using common random numbers. *Management Science*, 45(11):1570–1578, 1999.

Donald E. Knuth. *The Art of Computer Programming, Volume 2: Seminumerical Algorithms*. Addison-Wesley, 1969.

L. Kocsis and C. Szepesvári. Bandit based Monte-Carlo planning. In *European Conference on Machine Learning*, pages 282–293, 2006.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

J Zico Kolter and Tommi Jaakkola. Approximate inference in additive factorial HMMs with application to energy disaggregation. In *International conference on artificial intelligence and statistics*, pages 1472–1482, 2012.

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David M Blei. Automatic Variational Inference in Stan. *Neural Information Processing Systems*, 2015.

Tejas Dattatraya Kulkarni, Pushmeet Kohli, Joshua B Tenenbaum, and Vikash Kumar Mansinghka. Picture: a probabilistic programming language for scene perception. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266): 1332–1338, 2015.

Hugo Larochelle and Iain Murray. The neural autoregressive distribution estimator. In *International Conference on Artificial Intelligence and Statistics*, pages 29–37, 2011.

Krzysztof Łatuszyński, Gareth O Roberts, Jeffrey S Rosenthal, et al. Adaptive gibbs samplers and related mcmc methods. *The Annals of Applied Probability*, 23 (1):66–98, 2013.

Steffen L Lauritzen and David J Spiegelhalter. Local computations with probabilities on graphical structures and their application to expert systems. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 157–224, 1988.

Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11): 2278–2324, 1998.

Fredrik Lindsten, Michael I Jordan, and Thomas B Schön. Ancestor sampling for particle Gibbs. In F. Pereira, C.J.C. Burges, L. Bottou, and K.Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 2591–2599, 2012.

Fredrik Lindsten, Adam M Johansen, Christian A Naesseth, Bonnie Kirkpatrick, Thomas B Schön, John Aston, and Alexandre Bouchard-Côté. Divide-and-conquer with sequential Monte Carlo. *arXiv preprint arXiv:1406.4993*, 2014.

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

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

Jean-Michel Marin, Pierre Pudlo, Christian P Robert, and Robin J Ryder. Approximate bayesian computational methods. *Statistics and Computing*, pages 1–14, 2012.

Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 8(1):3–30, 1998.

Edward Meeds, Robert Leenders, and Max Welling. Hamiltonian ABC. *arXiv preprint arXiv:1503.01916*, 2015.

Thomas Minka. Expectation Propagation for approximate Bayesian inference. In *Proceedings of the 17th Conference on Uncertainty in Artificial Intelligence (UAI2001)*, pages 362–369, 2001.

Tom Minka. Divergence measures and message passing. Technical report, Microsoft Research, 2005.

Tom Minka and John Winn. Gates. In *Advances in Neural Information Processing Systems*, pages 1073–1080, 2009.

Tom Minka, J Winn, J Guiver, and D Knowles. Infer.NET 2.4, 2010. Microsoft Research Cambridge, 2010.

Pierre Del Moral and Arnaud Doucet. Interacting Markov chain Monte Carlo methods for solving nonlinear measured-valued equations. *Annals of Applied Probability*, 20(2):593–639, 2010.

Lawrence M. Murray, Anthony Lee, and Pierre E. Jacob. Parallel resampling in the particle filter. *arXiv preprint arXiv:1301.4019*, 2014.

Christian A. Naesseth, Fredrik Lindsten, and Thomas B. Schön. Sequential Monte Carlo for graphical models. In *Advances in Neural Information Processing Systems 27*. 2014.

Christian A. Naesseth, Fredrik Lindsten, and Thomas B. Schön. Towards automated sequential Monte Carlo for probabilistic Graphical Models. In *NIPS Workshop on Black Box Learning and Inference*. 2015.

Radford Neal. Exact MCMC using Hamiltonian dynamics. In Steve Brooks, Andrew Gelman, Galin L. Jones, and Xiao-Li Meng, editors, *Handbook of Markov Chain Monte Carlo*. Chapman and Hall/CRC, 2011.

Andrew Y. Ng and Michael I. Jordan. On discriminative vs. generative classifiers: A comparison of logistic regression and naive Bayes. In *Advances in Neural Information Processing Systems 14*, pages 841–848, 2002.

The Open Group. IEEE Std 1003.1, 2004 Edition. In *The Open Group Base Specifications Issue 6*, 2004a. URL `http://pubs.opengroup.org/onlinepubs/009695399`.

The Open Group. IEEE Std 1003.1, 2004 Edition. 2004b. URL `http://pubs.opengroup.org/onlinepubs/009695399/functions/fork.html`.

Art B. Owen. *Monte Carlo theory, methods and examples.* 2013.

Brooks Paige and Frank Wood. A compilation target for probabilistic programming languages. In *Proceedings of the 31st international conference on Machine learning*, volume 32 of *JMLR: W&CP*, pages 1935–1943, 2014.

Brooks Paige and Frank Wood. Inference networks for sequential Monte Carlo in graphical models. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *JMLR: W&CP*, pages 3040–3049, 2016.

Brooks Paige, Frank Wood, Arnaud Doucet, and Yee Whye Teh. Asynchronous anytime sequential Monte Carlo. In *Advances in Neural Information Processing Systems 27*, pages 3410–3418, 2014.

John Paisley, David Blei, and Michael Jordan. Variational Bayesian inference with stochastic search. In *Proceedings of the 29th International Conference on Machine Learning (ICML-12)*, pages 1367–1374. Omnipress, July 2012.

Christos H. Papadimitriou and Mihalis Yannakakis. Shortest paths without a map. *Theor. Comput. Sci.*, 84(1):127–150, July 1991.

Judea Pearl. Causal diagrams for empirical research. *Biometrika*, 82(4):669–688, 1995.

Judea Pearl and Stuart Russell. *Bayesian networks.* Computer Science Department, University of California, 1998.

M. K. Pitt and N. Shephard. Filtering via simulation: auxiliary particle filter. *Journal of the American Statistical Association*, 94:590–599, 1999.

Martyn Plummer. JAGS: A Program for Analysis of Bayesian Graphical Models Using Gibbs Sampling. *Proceedings of the 3rd International Workshop on Distributed Statistical Computing (DSC 2003). March*, pages 20–22, 2003.

Tom Rainforth, Christian A Naesseth, Fredrik Lindsten, Brooks Paige, Jan-Willem van de Meent, Arnaud Doucet, and Frank Wood. Interacting particle Markov chain Monte Carlo. In *Proceedings of the 33rd International Conference on Machine Learning*, volume 48 of *JMLR: W&CP*, pages 2616–2625, 2016.

Rajesh Ranganath, Sean Gerrish, and David M Blei. Black box variational inference. *arXiv preprint arXiv:1401.0118*, 2013.

Rajesh Ranganath, Dustin Tran, and David M Blei. Hierarchical variational models. In *Proceedings of The 33rd International Conference on Machine Learning*, pages 324–333, 2016.

Carl Edward Rasmussen. The infinite Gaussian mixture model. In *NIPS*, volume 12, pages 554–560, 1999.

Daniel Ritchie. Quicksand: A lightweight implementation of probabilistic programming for procedural modeling and design. 2014.

Daniel Ritchie, Ben Mildenhall, Noah D Goodman, and Pat Hanrahan. Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. *ACM Transactions on Graphics (TOG)*, 34(4):105, 2015.

Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. C3: Lightweight incrementalized mcmc for probabilistic programs using continuations and callsite caching. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 28–37, 2016.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

Sheldon M Ross. Simulation. *Burlington, MA: Elsevier*, 2006.

Thomas C Schelling. The strategy of conflict. *Cambridge, Mass*, 1960.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

Jonathan M. Smith and Gerald Q. Maguire, Jr. Effects of copy-on-write memory management on the response time of UNIX fork operations. *COMPUTING SYSTEMS*, 1(3):255–278, 1988.

James C Spall. Multivariate stochastic approximation using a simultaneous perturbation gradient approximation. *IEEE transactions on automatic control*, 37(3):332–341, 1992.

David Spiegelhalter, Andrew Thomas, Nicky Best, and Wally Gilks. Bugs 0.5: Bayesian inference using Gibbs sampling manual (version ii). *MRC Biostatistics Unit, Institute of Public Health, Cambridge, UK*, 1996.

The Stan Development Team. Stan modeling language user's guide and reference manual. *http://mc-stan.org/*, 2013.

Andreas Stuhlmüller and Noah D Goodman. Reasoning about reasoning by nested conditioning: Modeling theory of mind with probabilistic programs. *Cognitive Systems Research*, 28:80–99, 2014.

Andreas Stuhlmüller, Jacob Taylor, and Noah Goodman. Learning stochastic inverses. In *Advances in Neural Information Processing Systems 26*, pages 3048–3056. 2013.

Adrien Todeschini, François Caron, Marc Fuentes, Pierrick Legrand, and Pierre Del Moral. Biips: software for Bayesian inference with interacting particle systems. *arXiv preprint arXiv:1412.3779*, 2014.

David Tolpin, Jan-Willem van de Meent, Brooks Paige, and Frank Wood. Output-sensitive adaptive Metropolis-Hastings for probabilistic programs. In *Machine Learning and Knowledge Discovery in Databases, ECML PKDD 2015*, pages 311–326. Springer International Publishing, 2015.

David Tolpin, Jan Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language Anglican. *arXiv preprint arXiv:1608.05263*, 2016.

Dustin Tran, David Blei, and Edo M Airoldi. Copula variational inference. In *Advances in Neural Information Processing Systems*, pages 3564–3572, 2015.

Benigno Uria, Iain Murray, and Hugo Larochelle. RNADE: The real-valued neural autoregressive density-estimator. In *Advances in Neural Information Processing Systems*, pages 2175–2183, 2013.

Jan-Willem van de Meent, Hongseok Yang, Vikash Mansinghka, and Frank Wood. Particle Gibbs with ancestor sampling for probabilistic programs. In *Proceedings of the Eighteenth International Conference on Artificial Intelligence and Statistics*, volume 38 of *JMLR: W&CP*, pages 986–994, 2015.

Jan Willem van de Meent, Brooks Paige, David Tolpin, and Frank Wood. Black-box policy search with probabilistic programs. In *Proceedings of the 19th International conference on Artificial Intelligence and Statistics*, volume 41 of *JMLR: W&CP*, pages 1195–1204, 2016.

Rudolph Van Der Merwe, Arnaud Doucet, Nando De Freitas, and Eric Wan. The unscented particle filter. In *Advances in Neural Information Processing Systems*, pages 584–590, 2000.

Martin J Wainwright and Michael I Jordan. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends in Machine Learning*, 1(1–2): 1–305, 2008.

Nick Whiteley, Anthony Lee, and Kari Heine. On the role of interaction in sequential Monte Carlo algorithms. *arXiv preprint arXiv:1309.2918*, 2013.

Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.

David Wingate and Theophane Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, 2013.

David Wingate, Andreas Stuhlmueller, and Noah D Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *Proceedings of the 14th international conference on Artificial Intelligence and Statistics*, page 131, 2011.

Frank Wood, Jan Willem van de Meent, and Vikash Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, 2014.

Lingfeng Yang, Pat Hanrahan, and Noah D Goodman. Generating efficient mcmc kernels from probabilistic programs. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 1068–1076, 2014.