Probabilistic Programming for Deep Learning

Dustin Tran

Submitted in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy
under the Executive Committee
of the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2020

Abstract

Probabilistic Programming for Deep Learning

Dustin Tran

We propose the idea of *deep probabilistic programming*, a synthesis of advances for systems at the intersection of probabilistic modeling and deep learning. Such systems enable the development of new probabilistic models and inference algorithms that would otherwise be impossible: enabling unprecedented scales to billions of parameters, distributed and mixed precision environments, and AI accelerators; integration with neural architectures for modeling massive and high-dimensional datasets; and the use of computation graphs for automatic differentiation and arbitrary manipulation of probabilistic programs for flexible inference and model criticism.

After describing deep probabilistic programming, we discuss applications in novel variational inference algorithms and deep probabilistic models. First, we introduce the variational Gaussian process (VGP), a Bayesian nonparametric variational family, which adapts its shape to match complex posterior distributions. The VGP generates approximate posterior samples by generating latent inputs and warping them through random non-linear mappings; the distribution over random mappings is learned during inference, enabling the transformed outputs to adapt to varying complexity of the true posterior. Second, we introduce hierarchical implicit models (HIMS). HIMS combine the idea of implicit densities with hierarchical Bayesian modeling, thereby defining models via simulators of data with rich hidden structure.

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to acknowledge a number of people that supported this work.

First, I'd like to thank my advisor Prof. David Blei. I could not have hoped for an advisor with a greater overlap in research vision and clarity of thinking. Across our many works together, Dave gave me the opportunity to work with dozens of collaborators across university departments and outside university, across seniority levels, and across different yet similar research interests; this diversity made me a more successful researcher. I also admire that Dave is respectful, caring, and inviting: he shows by example how to be a better human being, and I've grown all the more for it.

I would like to sincerely thank Profs. Edo Airoldi and Finale Doshi-Velez during my formative years as I transitioned to a Ph.D. I especially thank my labmate Panos Toulis who graciously gave me experience on my first research publication. My earliest research projects could not have been successful without all their support.

At Columbia, I would like to thank my labmates: Rajesh Ranganath, Alp Kucukelbir, Adji Dieng, Maja Rudolph, Jaan Altosaar, Dawen Liang, Stephan Mandt, James McInerney, Fran Ruiz, and Christian Naesseth. I especially thank Rajesh for weathering all my idea pitching, frequent late-night research discussions, and frantic rushes for deadlines. I would like to thank Alp for helping me work on larger projects, iterate on research ideas, and become a better writer and coder. I enjoyed our frequent lunches in the Philosophy Hall. I shared a number of travels with Rajesh and Alp, and it's always been a pleasure discussing ideas with them.

I would also like to thank Andrew Gelman for mentoring me on statistical applications, on inviting me to the Stan team, and in our many conversations. Aki Vehtari, Bob Carpenter, and Daniel Lee have all influenced how I approach applications and software. I especially thank Alp for being inviting as we worked together on variational inference for Stan. I would also like to thank Matt Hoffman for hosting me at Adobe Research (later, Google) to work on Edward.

Finally, I'd like to thank, and dedicate this work to, my parents Van and Mai for their continuing support as well as my two brothers Daniel and Dalton. My parents overcame great hardship as refugees in the United States, and I admire them for their dedication to make their children educated and successful.

Chapter 1: Introduction, Background, & History

Probabilistic modeling is a powerful approach for analyzing empirical information using foundations from probability theory (Tukey, 1962; Newell and Simon, 1976; Box, 1976). Probabilistic models are an essential element of machine learning (Murphy, 2012; Goodfellow et al., 2016) and statistics (Friedman et al., 2001; Gelman et al., 2013), featuring applications across fields such as computational biology (Friedman et al., 2000), computational neuroscience (Dayan and Abbott, 2001), cognitive science (Tenenbaum et al., 2011), information theory (MacKay, 2003), and natural language processing (Manning and Schütze, 1999).

In this thesis, we propose the idea of *deep probabilistic programming*, a synthesis of advances for the design and implementation of systems at the intersection of probabilistic modeling and deep learning. Such systems enable the development of new probabilistic models and inference algorithms that would otherwise be impossible: enabling unprecedented scales to billions of parameters, distributed and mixed precision environments, and AI accelerators; integration and flexibility with neural architectures for modeling massive and high-dimensional datasets; and the use of computation graphs for automatic differentiation and arbitrary manipulation of probabilistic programs for flexible inference algorithms and model criticism strategies.

Below we provide background in the probabilistic approach to machine learning as well as history behind probabilistic systems for their research and eventual deployment. Chapter 2 and Chapter 3 describes the design of deep probabilistic programming systems. Chapter 4 and Chapter 5 discuss applications in novel probabilistic inference strategies as well as novel model classes.

1.1 Probabilistic Machine Learning

The process of data analysis in machine learning and statistics reflects that of the scientific method. Namely, there are core building blocks—interchangeable components which enable rapid iteration

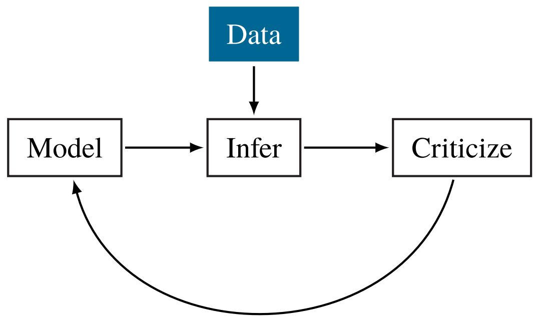


Figure 1.1: Box's loop.

as they build on one another. These building blocks form a cycle that leads to their individual improvements as the cycle is unrolled over time. To formalize this, we follow a philosophy of statistics and machine learning known as Box's loop (Box, 1976; Blei, 2014). Given a phenomena of interest:

1. Build a probabilistic model. The model formalizes a hypothesis about the phenomena using the language of probability.
2. Reason about the phenomena given model and gathered data. This data may come from designing and running an experiment, or it may come from gathering data that already exists (for example, previous experiments, or content such as text or images on the internet).
3. Criticize the model's fit to the data, that is, how well the hypothesis empirically reflects the phenomena. Revise and repeat.

As an illustration, suppose a child flips a coin ten times, with the set of outcomes being

```
[0, 1, 0, 0, 0, 0, 0, 0, 0, 1],
```

where `0` denotes tails and `1` denotes heads. She is interested in the probability that the coin lands heads. To analyze this, she first builds a "model": suppose she assumes the coin flips are independent and land heads with the same probability. Second, she reasons about the phenomenon: she infers the model's hidden structure (the unknown probability value) given data. Finally, she criticizes the model: she analyzes whether her model captures the real-world phenomenon of coin flips. If it doesn't, then she may revise the model and repeat.

Next we more formally describe the three components of probabilistic models, inference, and criticism.

1.1.1 Probabilistic Models

A probabilistic model asserts how observations from a natural phenomenon arise. The model is a *joint distribution* $p(\mathbf{x}, \mathbf{z})$ of observed variables $\mathbf{x}$ corresponding to data, and latent variables $\mathbf{z}$ that provide the hidden structure to generate from $\mathbf{x}$. The joint distribution factorizes into two components.

The *likelihood* $p(\mathbf{x} \mid \mathbf{z})$ is a probability distribution that describes how any data $\mathbf{x}$ depend on the latent variables $\mathbf{z}$. The likelihood posits a data generating process, where the data $\mathbf{x}$ are assumed drawn from the likelihood conditioned on a particular hidden pattern described by $\mathbf{z}$. The *prior* $p(\mathbf{z})$ is a probability distribution that describes the latent variables present in the data. It posits a generating process of the hidden structure.

Ultimately, how the likelihood depends on $\mathbf{z}$ can be incredibly complex in the real world: neural networks are a common class of functions that enable parameterizing the likelihood for high-dimensional distributions, proven empirically to work well across perceptual tasks such as image classification (Krizhevsky et al., 2012). For the purposes of this thesis, we do not provide background on neural networks. Their specific architectures are not central to the thesis; we recommend Goodfellow et al. (2016) as a survey.

1.1.2 Inference of Probabilistic Models

How can we use a model $p(\mathbf{x}, \mathbf{z})$ to analyze gathered data $\mathbf{x}$? In other words, what hidden structure $\mathbf{z}$ explains the data? We seek to infer this hidden structure using the model.

One method of inference leverages Bayes' rule to define the *posterior*

$$p(\mathbf{z} \mid \mathbf{x}) = \frac{p(\mathbf{x}, \mathbf{z})}{\int p(\mathbf{x}, \mathbf{z}) \mathrm{d}\mathbf{z}}.$$

The posterior is the distribution of the latent variables $\mathbf{z}$, conditioned on the observed data $\mathbf{x}$. It is a probabilistic description of the data's hidden representation.

From the perspective of inductivism, as practiced by many Bayesians, the posterior is our updated hypothesis about the latent variables, representing our new subjective belief about the phenomena. From the perspective of hypothetico-deductivism, as practiced by statisticians such as Box, Rubin, and Gelman, the posterior is simply a fitted model to data and thus a falsifiable hypothesis, to be criticized and ultimately revised (Box, 1982; Gelman and Shalizi, 2013).

Inferring the posterior. Now we know what the posterior represents. How do we calculate it? This is the central computational challenge in probabilistic inference. The posterior is difficult to compute because of its normalizing constant, which is the integral in the denominator. This is often a high-dimensional integral that lacks an analytic (closed-form) solution. Thus, calculating the posterior means *approximating* the posterior.

1.1.3 Variational Inference

Variational inference is an umbrella term for algorithms which cast posterior inference as optimization (Hinton and van Camp, 1993; Waterhouse et al., 1996; Jordan et al., 1999a). The core idea involves two steps:

1. posit a family of distributions $q(\mathbf{z}\,;\,\lambda)$ over the latent variables;
2. match $q(\mathbf{z}\,;\,\lambda)$ to the posterior by optimizing over its parameters λ.

This strategy converts the problem of computing the posterior $p(\mathbf{z} \mid \mathbf{x})$ into an optimization problem: minimize a loss function

$$\lambda^* = \arg\min_{\lambda} \operatorname{loss}(p(\mathbf{z} \mid \mathbf{x}), q(\mathbf{z}\,;\,\lambda)).$$

The optimized distribution $q(\mathbf{z}\,;\,\lambda^*)$ is used as a proxy to the posterior $p(\mathbf{z} \mid \mathbf{x})$.

1.1.4 Maximum a Posteriori Estimation

One form of variational inference is known as maximum a posteriori (MAP) estimation. It uses the mode as a point estimate of the posterior distribution,

$$\mathbf{z}_{\text{MAP}} = \arg\max_{\mathbf{z}} p(\mathbf{z} \mid \mathbf{x}) = \arg\max_{\mathbf{z}} \log p(\mathbf{z} \mid \mathbf{x}).$$

Namely, the posited family of variational distributions are simply delta distributions with probability 1 at a point. In practice, we work with logarithms of densities to avoid numerical underflow issues (Murphy, 2012).

The MAP estimate is the most likely configuration of the hidden patterns $\mathbf{z}$ under the model. However, we cannot directly solve this optimization problem because the posterior is typically intractable. To circumvent this, we use Bayes' rule to optimize over the joint density,

$$\mathbf{z}_{\text{MAP}} = \arg\max_{\mathbf{z}} \log p(\mathbf{z} \mid \mathbf{x}) = \arg\max_{\mathbf{z}} \log p(\mathbf{x}, \mathbf{z}).$$

This is valid because

$$\log p(\mathbf{z} \mid \mathbf{x}) = \log p(\mathbf{x}, \mathbf{z}) - \log p(\mathbf{x}) = \log p(\mathbf{x}, \mathbf{z}) - \text{constant in terms of } \mathbf{z}.$$

MAP estimation includes the common scenario of maximum likelihood estimation as a special case,

$$\mathbf{z}_{\text{MAP}} = \arg\max_{\mathbf{z}} p(\mathbf{x}, \mathbf{z}) = \arg\max_{\mathbf{z}} p(\mathbf{x} \mid \mathbf{z}),$$

where the prior $p(\mathbf{z})$ is flat, placing uniform probability over all values $\mathbf{z}$ supports. Placing a nonuniform prior can be thought of as regularizing the estimation, penalizing values away from maximizing the likelihood, which can lead to overfitting. For example, a normal prior or Laplace prior on $\mathbf{z}$ corresponds to ℓ_2 penalization, also known as ridge regression, and ℓ_1 penalization, also known as the LASSO.

Maximum likelihood is also known as cross entropy minimization. For a data set $\mathbf{x} = \{x_n\}$,

$$\mathbf{z}_{\text{MAP}} = \arg\max_{\mathbf{z}} \log p(\mathbf{x} \mid \mathbf{z}) = \arg\max_{\mathbf{z}} \sum_{n=1}^{N} \log p(x_n \mid \mathbf{z}) = \arg\min_{\mathbf{z}} -\frac{1}{N} \sum_{n=1}^{N} \log p(x_n \mid \mathbf{z}).$$

The last expression can be thought of as an approximation to the cross entropy between the true data distribution and $p(\mathbf{x} \mid \mathbf{z})$, using a set of N data points.

Gradient descent. To find the MAP estimate of the latent variables $\mathbf{z}$, we use the gradient of the log joint density $\nabla_{\mathbf{z}} \log p(\mathbf{x}, \mathbf{z})$ and follow it to a (local) optima.

1.1.5 Laplace Approximation

Maximum a posteriori (MAP) estimation approximates the posterior $p(\mathbf{z} \mid \mathbf{x})$ with a point mass (delta function) by simply capturing its mode. MAP is attractive because it is fast and efficient. How can we use MAP to construct a better approximation to the posterior?

The Laplace approximation (Laplace, 1986) is one way of improving a MAP estimate. The idea is to approximate the posterior with a normal distribution centered at the MAP estimate,

$$p(\mathbf{z} \mid \mathbf{x}) \approx \text{Normal}(\mathbf{z} \,;\, \mathbf{z}_{\text{MAP}}, \Lambda^{-1}).$$

This requires computing a precision matrix Λ. Derived from a Taylor expansion, the Laplace approximation uses the Hessian of the negative log joint density at the MAP estimate. It is defined component-wise as

$$\Lambda_{ij} = \frac{\partial^2}{\partial z_i \partial z_j} - \log p(\mathbf{x}, \mathbf{z}).$$

For flat priors (which reduces MAP to maximum likelihood), the precision matrix is known as the observed Fisher information (Fisher, 1925). Edward uses TensorFlow's automatic differentiation, making this distribute.

1.1.6 $\text{KL}(q \| p)$ Minimization

MAP estimation and the Laplace approximation are simple, but make local and Gaussian assumptions in approximating the true posterior distribution. Another popular form of variational inference minimizes the Kullback-Leibler divergence from $q(\mathbf{z} \,;\, \lambda)$ to $p(\mathbf{z} \mid \mathbf{x})$,

$$\begin{aligned} \lambda^* &= \arg\min_{\lambda} \text{KL}(q(\mathbf{z} \,;\, \lambda) \parallel p(\mathbf{z} \mid \mathbf{x})) \\ &= \arg\min_{\lambda} \; \mathbb{E}_{q(\mathbf{z} \,;\, \lambda)} \big[\log q(\mathbf{z} \,;\, \lambda) - \log p(\mathbf{z} \mid \mathbf{x}) \big]. \end{aligned}$$

The KL divergence is a non-symmetric, information theoretic measure of similarity between two probability distributions (Hinton and van Camp, 1993; Waterhouse et al., 1996; Jordan et al., 1999a).

The Evidence Lower Bound. The above optimization problem is intractable because it directly depends on the posterior $p(\mathbf{z} \mid \mathbf{x})$. To tackle this, consider the property

$$\log p(\mathbf{x}) = \mathrm{KL}(q(\mathbf{z}\,;\,\lambda) \parallel p(\mathbf{z} \mid \mathbf{x})) + \mathbb{E}_{q(\mathbf{z}\,;\,\lambda)}\big[\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}\,;\,\lambda)\big]$$

where the left hand side is the logarithm of the marginal likelihood $p(\mathbf{x}) = \int p(\mathbf{x}, \mathbf{z})\mathrm{d}\mathbf{z}$, also known as the model evidence.

The evidence is a constant with respect to the variational parameters λ, so we can minimize $\mathrm{KL}(q\|p)$ by instead maximizing the *evidence lower bound* (ELBO),

$$\mathrm{ELBO}(\lambda) = \mathbb{E}_{q(\mathbf{z}\,;\,\lambda)}\big[\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}\,;\,\lambda)\big].$$

In the ELBO, both $p(\mathbf{x}, \mathbf{z})$ and $q(\mathbf{z}\,;\,\lambda)$ are tractable. The optimization problem reduces to

$$\lambda^* = \arg\max_{\lambda} \mathrm{ELBO}(\lambda).$$

As per its name, the ELBO is a lower bound on the evidence, and optimizing it tries to maximize the probability of observing the data. What does maximizing the ELBO do? Splitting the ELBO reveals a trade-off

$$\mathrm{ELBO}(\lambda) = \mathbb{E}_{q(\mathbf{z}\,;\,\lambda)}[\log p(\mathbf{x}, \mathbf{z})] - \mathbb{E}_{q(\mathbf{z}\,;\,\lambda)}[\log q(\mathbf{z}\,;\,\lambda)],$$

where the first term represents an energy and the second term (including the minus sign) represents the entropy of q. The energy encourages q to focus probability mass where the model puts high probability, $p(\mathbf{x}, \mathbf{z})$. The entropy encourages q to spread probability mass to avoid concentrating to one location.

There are two general strategies to obtain gradients for gradient-based optimization: score function gradient; and reparameterization gradient.

Score function gradient. Gradient descent is a standard approach for optimizing complicated objectives like the ELBO. The idea is to calculate its gradient

$$\nabla_\lambda \text{ ELBO}(\lambda) = \nabla_\lambda \mathbb{E}_{q(\mathbf{z}\,;\,\lambda)}\big[\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}\,;\,\lambda)\big],$$

and update the current set of parameters proportional to the gradient. The score function gradient estimator leverages a property of logarithms to write the gradient as

$$\nabla_\lambda \text{ ELBO}(\lambda) = \mathbb{E}_{q(\mathbf{z}\,;\,\lambda)}\big[\nabla_\lambda \log q(\mathbf{z}\,;\,\lambda)\,\big(\log p(\mathbf{x}, \mathbf{z}) - \log q(\mathbf{z}\,;\,\lambda)\big)\big].$$

The gradient of the ELBO is an expectation over the variational model $q(\mathbf{z}\,;\,\lambda)$; the only new ingredient it requires is the *score function* $\nabla_\lambda \log q(\mathbf{z}\,;\,\lambda)$ (Paisley et al., 2012b; Ranganath et al., 2014).

We can use Monte Carlo integration to obtain noisy estimates of both the ELBO and its gradient. The basic procedure follows these steps:

1. draw S samples $\{\mathbf{z}_s\}_1^S \sim q(\mathbf{z}\,;\,\lambda)$,
2. evaluate the argument of the expectation using $\{\mathbf{z}_s\}_1^S$, and
3. compute the empirical mean of the evaluated quantities.

A Monte Carlo estimate of the gradient is then

$$\nabla_\lambda \text{ ELBO}(\lambda) \approx \frac{1}{S}\sum_{s=1}^{S}\big[\big(\log p(\mathbf{x}, \mathbf{z}_s) - \log q(\mathbf{z}_s\,;\,\lambda)\big)\,\nabla_\lambda \log q(\mathbf{z}_s\,;\,\lambda)\big].$$

This is an unbiased estimate of the actual gradient of the ELBO.

Reparameterization gradient. If the model has differentiable latent variables, then it is generally advantageous to leverage gradient information from the model in order to better traverse the optimization space. One approach to doing this is the reparameterization gradient (Kingma and Welling, 2014a; Rezende et al., 2014).

Some variational distributions $q(\mathbf{z}\,;\,\lambda)$ admit useful reparameterizations. For example, we can reparameterize a normal distribution $\mathbf{z} \sim \text{Normal}(\mu, \Sigma)$ as $\mathbf{z} = \mu + L\epsilon$, where $\epsilon \sim \text{Normal}(0, I)$ and

$\Sigma = LL^\top$. In general, write this as

$$\epsilon \sim q(\epsilon), \qquad \mathbf{z} = \mathbf{z}(\epsilon \; ; \; \lambda),$$

where ϵ is a random variable that does **not** depend on the variational parameters λ. The deterministic function $\mathbf{z}(\cdot ; \lambda)$ encapsulates the variational parameters instead, and following the process is equivalent to directly drawing $\mathbf{z}$ from the original distribution.

The reparameterization gradient leverages this property to write the gradient as

$$\nabla_\lambda \text{ ELBO}(\lambda) = \mathbb{E}_{q(\epsilon)}\big[\nabla_\lambda\big(\log p(\mathbf{x}, \mathbf{z}(\epsilon \; ; \; \lambda)) - \log q(\mathbf{z}(\epsilon \; ; \; \lambda) \; ; \; \lambda)\big)\big].$$

The gradient of the ELBO is an expectation over the base distribution $q(\epsilon)$, and the gradient can be applied directly to the inner expression. We can use Monte Carlo integration to obtain noisy estimates of both the ELBO and its gradient. The basic procedure follows these steps:

1. draw S samples $\{\epsilon_s\}_1^S \sim q(\epsilon)$,
2. evaluate the argument of the expectation using $\{\epsilon_s\}_1^S$, and
3. compute the empirical mean of the evaluated quantities.

A Monte Carlo estimate of the gradient is then

$$\nabla_\lambda \text{ ELBO}(\lambda) \approx \frac{1}{S}\sum_{s=1}^{S}\big[\nabla_\lambda\big(\log p(\mathbf{x}, \mathbf{z}(\epsilon_s \; ; \; \lambda)) - \log q(\mathbf{z}(\epsilon_s \; ; \; \lambda) \; ; \; \lambda)\big)\big].$$

This is an unbiased estimate of the actual gradient of the ELBO. Empirically, it exhibits lower variance than the score function gradient, leading to faster convergence in a large set of problems (Tran et al., 2016b).

1.1.7 Model Criticism

We can never validate whether a model is true. In practice, "all models are wrong" (Box, 1976). However, we can try to uncover where the model goes wrong. Model criticism helps justify the model as an approximation or point to good directions for revising the model.

Model criticism typically analyzes the posterior predictive distribution,

$$p(\mathbf{x}_{\text{new}} \mid \mathbf{x}) = \int p(\mathbf{x}_{\text{new}} \mid \mathbf{z}) p(\mathbf{z} \mid \mathbf{x}) \mathrm{d}\mathbf{z}.$$

The model's posterior predictive can be used to generate new data given past observations and can also make predictions on new data given past observations. It is formed by calculating the likelihood of the new data, averaged over every set of latent variables according to the posterior distribution.

Scoring rules. A scoring rule is a scalar-valued metric for assessing trained models (Winkler, 1994; Gneiting and Raftery, 2007). For example, we can assess models for classification by predicting the label for each observation in the data and comparing it to their true labels. Formally, given two distributions p and q over a space of events x,

$$S(p, q) = \int q(x) S(p, x) dx,$$

where $S(p, x)$ is a real-valued function of a density p over x such as the logarithmic scoring rule ($\log p(x)$). It is common practice to criticize models with data held-out from training. In machine learning, benchmark datasets involve a train and test split.

Posterior predictive checks. Posterior predictive checks (PPCs) analyze the degree to which data generated from the model deviate from data generated from the true distribution. They can be used either numerically to quantify this degree, or graphically to visualize this degree. PPCs can be thought of as a probabilistic generalization of scoring rules, providing a distribution rather than a single value (Box, 1980; Rubin, 1984; Meng, 1994; Gelman et al., 1996).

The simplest PPC works by applying a test statistic on replicated datasets generated from the posterior predictive, such as $T(\mathbf{x}_{\text{new}}) = \max(\mathbf{x}_{\text{new}})$. Applying $T(\mathbf{x}_{\text{new}})$ to new datasets over many data replications induces a distribution. We compare this distribution to the test statistic on the real data $T(\mathbf{x})$.

In Figure 1.2, $T(\mathbf{x})$ falls in a low probability region of this reference distribution: if the model were true, the probability of observing the test statistic is very low. This indicates that the model fits the data poorly according to this check; this suggests an area of improvement for the model.

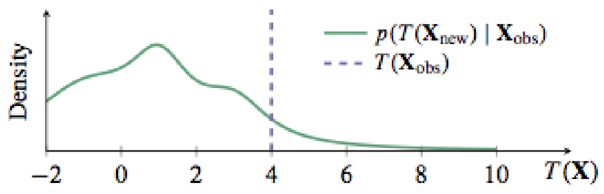


Figure 1.2: Distribution of test statistic replicated over generated datasets, along with the test statistic applied to the observed dataset.

More generally, the test statistic can be a function of the model's latent variables $T(\mathbf{x}, \mathbf{z})$, known as a discrepancy function. Examples of discrepancy functions are the metrics used for scoring rules. We can now interpret the scoring rule as a special case of PPCs: it simply calculates $T(\mathbf{x}, \mathbf{z})$ over the real data and without a reference distribution in mind. A reference distribution allows us to make probabilistic statements about the point, in reference to an overall distribution.

PPCs are an excellent tool for revising models—simplifying or expanding the current model as one examines its fit to data. They are inspired by classical hypothesis testing such as goodness-of-fit-tests; these methods criticize models under the frequentist perspective of large sample assessment.

PPCs can also be applied to tasks such as hypothesis testing, model comparison, model selection, and model averaging. It's important to note that while PPCs can be applied as a form of Bayesian hypothesis testing, hypothesis testing is generally not recommended: binary decision making from a single test is not as common a use case as the broader process of assimilating this information into future model revisions and diagnostics.

1.2 History of probabilistic systems

For examples of probabilistic software systems, we point to two early threads. The first is in artificial intelligence. Expert systems were designed from human expertise, which in turn enabled larger reasoning steps according to existing knowledge (Buchanan et al., 1969; Minsky, 1975). With connectionist models, the design focused on neuron-like processing units, which learn from experience; this drove new applications of artificial intelligence (Hopfield, 1982; Rumelhart et al., 1988).

As a second thread, we point to early work in statistical computing, where interest grew broadly out of efficient computation for problems in statistical analysis. The S language, developed by John Chambers and colleagues at Bell Laboratories (Becker and Chambers, 1984; Chambers and Hastie, 1992), focused on an interactive environment for data analysis, with simple yet rich syntax to quickly turn ideas into software. It is a predecessor to the R language (Ihaka and Gentleman, 1996). More targeted environments such as BUGS (Spiegelhalter et al., 1995), which focuses on Bayesian analysis of statistical models, helped launch the emerging field of probabilistic programming.

We are motivated to build on these early works in probabilistic systems—where in modern applications, new challenges arise in their design and implementation. We highlight two challenges. First, statistics and machine learning have made significant advances in the methodology of probabilistic models and their inference (e.g., Hoffman et al. (2013); Ranganath et al. (2014); Rezende et al. (2014)). For software systems to enable fast experimentation, we require rich abstractions that can capture these advances: it must encompass both a broad class of probabilistic models and a broad class of algorithms for their efficient inference. Second, researchers are increasingly motivated to employ complex probabilistic models and at an unprecedented scale of massive data (Bengio et al., 2013; Ghahramani, 2015; Lake et al., 2016). Thus we require an efficient computing environment that supports distributed training and integration of hardware such as (multiple) GPUs.

A core theme in probabilistic programming is automated inference, used in systems such as BUGS (Spiegelhalter et al., 1995) with a Gibbs sampler to more recent work such as Stan (Carpenter et al., 2015) with the No-U-Turn Sampler (Hoffman and Gelman, 2014) and Automatic Differentiation Variational Inference (Kucukelbir et al., 2017). In this thesis, we examine effectively the opposite theme: flexible, composable inference for research on the algorithms themselves. This use case fits well with machine learning research, where the challenge is to devise better models and algorithms that work on a variety of domains and scale—ultimately toward the goal of more intelligent systems. The theme of automated inference is useful for a separate audience: applied scientists and practioners. These purposes could fit well as a higher-level abstraction built on top of what we describe here in developing more composable and flexible systems.

Chapter 2: Deep Probabilistic Programming

2.1 Introduction

The nature of deep neural networks is compositional. Users can connect layers in creative ways, without having to worry about how to perform testing (forward propagation) or inference (gradient-based optimization, with back propagation and automatic differentiation).

In this chapter, we design compositional representations for probabilistic programming. Probabilistic programming lets users specify generative probabilistic models as programs and then "compile" those models down into inference procedures. Probabilistic models are also compositional in nature, and much work has enabled rich probabilistic programs via compositions of random variables (Goodman et al., 2012; Ghahramani, 2015; Lake et al., 2016).

Less work, however, has considered an analogous compositionality for inference. Rather, many existing probabilistic programming languages treat the inference engine as a black box, abstracted away from the model. These cannot capture probabilistic inferences that reuse the model's representation—a key idea in recent advances in variational inference (Kingma and Welling, 2014b; Rezende and Mohamed, 2015; Tran et al., 2016b), generative adversarial networks (Goodfellow et al., 2014), and also in more classic inferences (Dayan et al., 1995; Gutmann and Hyvärinen, 2010).

We propose Edward[1], a Turing-complete probabilistic programming language which builds on two compositional representations—one for random variables and one for inference. By treating inference as a first class citizen, on a par with modeling, we show that probabilistic programming can be as flexible and computationally efficient as traditional deep learning. For flexibility, we show how Edward makes it easy to fit the same model using a variety of composable inference methods, ranging from point estimation to variational inference to MCMC. For efficiency, we show how to integrate

[1] See Tran et al. (2016a) for details of the API. A companion webpage for this paper is available at `http://edwardlib.org/iclr2017`. It contains more complete examples with runnable code.

Edward into existing computational graph frameworks such as TensorFlow (Abadi et al., 2016). Frameworks like TensorFlow provide computational benefits like distributed training, parallelism, vectorization, and GPU support "for free." For example, we show on a benchmark task that Edward's Hamiltonian Monte Carlo is many times faster than existing software. Further, Edward incurs no runtime overhead: it is as fast as handwritten TensorFlow.

2.2 Compositional Representations for Probabilistic Models

We first develop compositional representations for probabilistic models. We desire two criteria: (a) integration with computational graphs, an efficient framework where nodes represent operations on data and edges represent data communicated between them (Culler, 1986); and (b) invariance of the representation under the graph, that is, the representation can be reused during inference.

Edward defines random variables as the key compositional representation. They are class objects with methods, for example, to compute the log density and to sample. Further, each random variable $\mathbf{x}$ is associated to a tensor (multi-dimensional array) $\mathbf{x}^*$, which represents a single sample $\mathbf{x}^* \sim p(\mathbf{x})$. This association embeds the random variable onto a computational graph on tensors.

The design's simplicity makes it easy to develop probabilistic programs in a computational graph framework. Importantly, all computation is represented on the graph. This enables one to compose random variables with complex deterministic structure such as deep neural networks, a diverse set of math operations, and third party libraries that build on the same framework. The design also enables compositions of random variables to capture complex stochastic structure.

As an illustration, we use a Beta-Bernoulli model, $p(\mathbf{x}, \theta) = \mathrm{Beta}(\theta \,|\, 1, 1) \prod_{n=1}^{50} \mathrm{Bernoulli}(x_n \,|\, \theta)$, where θ is a latent probability shared across the 50 data points $\mathbf{x} \in \{0, 1\}^{50}$. The random variable $\mathbf{x}$ is 50-dimensional, parameterized by the random tensor θ^*. Fetching the object $\mathbf{x}$ runs the graph: it simulates from the generative process and outputs a binary vector of 50 elements.

```
theta = Beta(a=1.0, b=1.0)
x = Bernoulli(p=tf.ones(50) * theta)
```

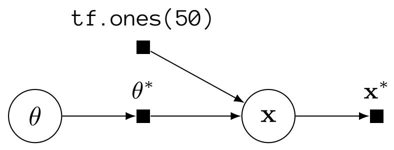
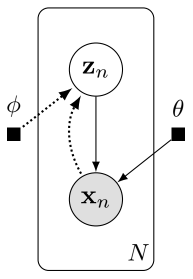


Figure 2.1: Beta-Bernoulli program **(left)** alongside its computational graph **(right)**. Fetching $\mathbf{x}$ from the graph generates a binary vector of 50 elements.



```
# Probabilistic model
z = Normal(mu=tf.zeros([N, d]), sigma=tf.ones([N, d]))
h = Dense(256, activation='relu')(z)
x = Bernoulli(logits=Dense(28 * 28, activation=None)(h))

# Variational model
qx = tf.placeholder(tf.float32, [N, 28 * 28])
qh = Dense(256, activation='relu')(qx)
qz = Normal(mu=Dense(d, activation=None)(qh),
            sigma=Dense(d, activation='softplus')(qh))
```

Figure 2.2: Variational auto-encoder for a data set of 28×28 pixel images: **(left)** graphical model, with dotted lines for the inference model; **(right)** probabilistic program, with 2-layer neural networks.

All computation is registered symbolically on random variables and not over their execution. Symbolic representations do not require reifying the full model, which leads to unreasonable memory consumption for large models (Tristan et al., 2014). Moreover, it enables us to simplify both deterministic and stochastic operations in the graph, before executing any code (Ścibior et al., 2015; Zinkov and Shan, 2016).

With computational graphs, it is also natural to build mutable states within the probabilistic program. As a typical use of computational graphs, such states can define model parameters; in TensorFlow, this is given by a `tf.Variable`. Another use case is for building discriminative models $p(\mathbf{y} \mid \mathbf{x})$, where $\mathbf{x}$ are features that are input as training or test data. The program can be written independent of the data, using a mutable state (`tf.placeholder`) for $\mathbf{x}$ in its graph. During training and testing, we feed the placeholder the appropriate values.

In Appendix A.1, we provide examples of a Bayesian neural network for classification (A.2), latent Dirichlet allocation (A.3), and Gaussian matrix factorization (A.4). We present others below.

2.2.1 Example: Variational Auto-encoder

Figure 3.4 implements a VAE (Kingma and Welling, 2014b; Rezende et al., 2014) in Edward. It comprises a probabilistic model over data and a variational model designed to approximate the former's posterior. Here we use random variables to construct both the probabilistic model and the variational model; they are fit during inference (more details in Section 5.3).

There are N data points $x_n \in \{0, 1\}^{28 \cdot 28}$ each with d latent variables, $z_n \in \mathbb{R}^d$. The program uses Keras (Chollet, 2015) to define neural networks. The probabilistic model is parameterized by a 2-layer neural network, with 256 hidden units (and ReLU activation), and generates 28×28 pixel images. The variational model is parameterized by a 2-layer inference network, with 256 hidden units and outputs parameters of a normal posterior approximation.

The probabilistic program is concise. Core elements of the VAE—such as its distributional assumptions and neural net architectures—are all extensible. With model compositionality, we can embed it into more complicated models (Gregor et al., 2015; Rezende et al., 2016) and for other learning tasks (Kingma et al., 2014). With inference compositionality (which we discuss in Section 5.3), we can embed it into more complicated algorithms, such as with expressive variational approximations (Rezende and Mohamed, 2015; Tran et al., 2016b; Kingma et al., 2016) and alternative objectives (Ranganath et al., 2016a; Li and Turner, 2016; Dieng et al., 2016).

2.2.2 Example: Bayesian Recurrent Neural Network with Variable Length

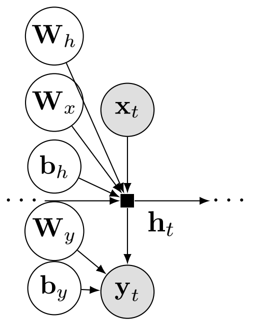
Random variables can also be composed with control flow operations. As an example, Figure 2.3 implements a Bayesian recurrent neural network (RNN) with variable length. The data is a sequence of inputs $\{\mathbf{x}_1, \ldots, \mathbf{x}_T\}$ and outputs $\{y_1, \ldots, y_T\}$ of length T with $\mathbf{x}_t \in \mathbb{R}^D$ and $y_t \in \mathbb{R}$ per time step. For $t = 1, \ldots, T$, a RNN applies the update

$$\mathbf{h}_t = \tanh(\mathbf{W}_h \mathbf{h}_{t-1} + \mathbf{W}_x \mathbf{x}_t + \mathbf{b}_h),$$

where the previous hidden state is $\mathbf{h}_{t-1} \in \mathbb{R}^H$. We feed each hidden state into the output's likelihood, $y_t \sim \mathrm{Normal}(\mathbf{W}_y \mathbf{h}_t + \mathbf{b}_y, 1)$, and we place a standard normal prior over all parameters $\{\mathbf{W}_h \in \mathbb{R}^{H \times H}, \mathbf{W}_x \in \mathbb{R}^{D \times H}, \mathbf{W}_y \in \mathbb{R}^{H \times 1}, \mathbf{b}_h \in \mathbb{R}^H, \mathbf{b}_y \in \mathbb{R}\}$. Our implementation is dynamic: it differs from a RNN with fixed length, which pads and unrolls the computation.

2.2.3 Stochastic Control Flow and Model Parallelism

Random variables can also be placed in the control flow itself, enabling probabilistic programs with stochastic control flow. Stochastic control flow defines dynamic conditional dependencies, known in the literature as contingent or existential dependencies (Mansinghka et al., 2014; Wu et al., 2016). See Figure 2.4, where $\mathbf{x}$ may or may not depend on $\mathbf{a}$ for a given execution. In Appendix A.5, we use stochastic control flow to implement a Dirichlet process mixture model. Tensors with stochastic



```
def rnn_cell(hprev, xt):
  return tf.tanh(tf.dot(hprev, Wh) + tf.dot(xt, Wx) + bh)

Wh = Normal(mu=tf.zeros([H, H]), sigma=tf.ones([H, H]))
Wx = Normal(mu=tf.zeros([D, H]), sigma=tf.ones([D, H]))
Wy = Normal(mu=tf.zeros([H, 1]), sigma=tf.ones([H, 1]))
bh = Normal(mu=tf.zeros(H), sigma=tf.ones(H))
by = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

x = tf.placeholder(tf.float32, [None, D])
h = tf.scan(rnn_cell, x, initializer=tf.zeros(H))
y = Normal(mu=tf.matmul(h, Wy) + by, sigma=1.0)
```

Figure 2.3: Bayesian RNN: **(left)** graphical model; **(right)** probabilistic program. The program has an unspecified number of time steps; it uses a symbolic for loop (`tf.scan`).

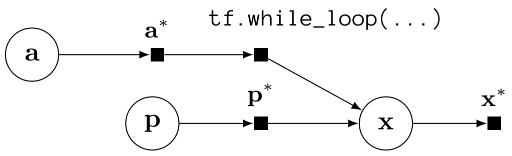


Figure 2.4: Computational graph for a probabilistic program with stochastic control flow.

shape are also possible: for example, `tf.zeros(Poisson(lam=5.0))` defines a vector of zeros with length given by a Poisson draw with rate 5.0.

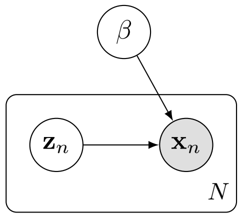
Stochastic control flow produces difficulties for algorithms that use the graph structure because the relationship of conditional dependencies changes across execution traces. The computational graph, however, provides an elegant way of teasing out static conditional dependence structure ($\mathbf{p}$) from dynamic dependence structure ($\mathbf{a}$). We can perform model parallelism (parallel computation across components of the model) over the static structure with GPUs and batch training. We can use more generic computations to handle the dynamic structure.

2.3 Compositional Representations for Inference

We described random variables as a representation for building rich probabilistic programs over computational graphs. We now describe a compositional representation for inference. We desire two criteria: (a) support for many classes of inference, where the form of the inferred posterior depends on the algorithm; and (b) invariance of inference under the computational graph, that is, the posterior can be further composed as part of another model.

To explain our approach, we will use a simple hierarchical model as a running example. Figure 2.5

displays a joint distribution $p(\mathbf{x}, \mathbf{z}, \beta)$ of data $\mathbf{x}$, local variables $\mathbf{z}$, and global variables β. The ideas here extend to more expressive programs.



```
N = 10000  # number of data points
D = 2  # data dimension
K = 5  # number of clusters

beta = Normal(mu=tf.zeros([K, D]), sigma=tf.ones([K, D]))
z = Categorical(logits=tf.zeros([N, K]))
x = Normal(mu=tf.gather(beta, z), sigma=tf.ones([N, D]))
```

Figure 2.5: Hierarchical model: **(left)** graphical model; **(right)** probabilistic program. It is a mixture of Gaussians over D-dimensional data $\{x_n\} \in \mathbb{R}^{N\times D}$. There are K latent cluster means $\beta \in \mathbb{R}^{K\times D}$.

2.3.1 Inference as Stochastic Graph Optimization

The goal of inference is to calculate the posterior distribution $p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}}; \boldsymbol{\theta})$ given data $\mathbf{x}_{\text{train}}$, where $\boldsymbol{\theta}$ are any model parameters that we will compute point estimates for.[2] We formalize this as the following optimization problem:

$$\min_{\boldsymbol{\lambda},\boldsymbol{\theta}} \mathcal{L}(p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}}; \boldsymbol{\theta}),\ q(\mathbf{z}, \beta; \boldsymbol{\lambda})), \tag{2.1}$$

where $q(\mathbf{z}, \beta; \boldsymbol{\lambda})$ is an approximation to the posterior $p(\mathbf{z}, \beta \mid \mathbf{x}_{\text{train}}; \boldsymbol{\theta})$, and $\mathcal{L}$ is a loss function with respect to p and q.

The choice of approximation q, loss $\mathcal{L}$, and rules to update parameters $\{\boldsymbol{\theta}, \boldsymbol{\lambda}\}$ are specified by an inference algorithm. (Note q can be nonparametric, such as a point or a collection of samples.)

In Edward, we write this problem as follows:

```
inference = ed.Inference({beta: qbeta, z: qz}, data={x: x_train})
```

`Inference` is an abstract class which takes two inputs. The first is a collection of latent random variables `beta` and `z`, associated to their "posterior variables" `qbeta` and `qz` respectively. The second is a collection of observed random variables `x`, which is associated to their realizations `x_train`.

The idea is that `Inference` defines and solves the optimization in Equation 2.1. It adjusts parameters

[2]For example, we could replace `x`'s `sigma` argument with `tf.exp(tf.Variable(0.0))*tf.ones([N, D])`. This defines a model parameter initialized at 0 and positive-constrained.

```
qbeta = Normal(
  mu=tf.Variable(tf.zeros([K, D])),
  sigma=tf.exp(tf.Variable(tf.zeros([K, D]))))
qz = Categorical(
  logits=tf.Variable(tf.zeros([N, K])))

inference = ed.VariationalInference(
  {beta: qbeta, z: qz}, data={x: x_train})
```

```
T = 10000  # number of samples
qbeta = Empirical(
  params=tf.Variable(tf.zeros([T, K, D])))
qz = Empirical(
  params=tf.Variable(tf.zeros([T, N])))

inference = ed.MonteCarlo(
  {beta: qbeta, z: qz}, data={x: x_train})
```

Figure 2.6: (left) Variational inference. **(right)** Monte Carlo.

of the distribution of qbeta and qz (and any model parameters) to be close to the posterior.

Class methods are available to finely control the inference. Calling `inference.initialize()` builds a computational graph to update $\{\boldsymbol{\theta}, \boldsymbol{\lambda}\}$. Calling `inference.update()` runs this computation once to update $\{\boldsymbol{\theta}, \boldsymbol{\lambda}\}$; we call the method in a loop until convergence. Importantly, no efficiency is lost in Edward's language: the computational graph is the same as if it were handwritten for a specific model. This means the runtime is the same; also see our experiments in Section 2.4.2.

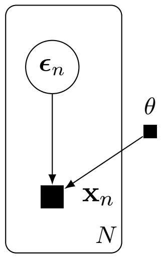
A key concept in Edward is that there is no distinct "model" or "inference" block. A model is simply a collection of random variables, and inference is a way of modifying parameters in that collection subject to another. This reductionism offers significant flexibility. For example, we can infer only parts of a model (e.g., layer-wise training (Hinton et al., 2006)), infer parts used in multiple models (e.g., multi-task learning), or plug in a posterior into a new model (e.g., Bayesian updating).

2.3.2 Classes of Inference

The design of `Inference` is very general. We describe subclasses to represent many algorithms below: variational inference, Monte Carlo, and generative adversarial networks.

Variational inference posits a family of approximating distributions and finds the closest member in the family to the posterior (Jordan et al., 1999a). In Edward, we build the variational family in the graph; see Figure 2.6 (left). For our running example, the family has mutable variables as parameters $\boldsymbol{\lambda} = \{\pi, \mu, \sigma\}$, where $q(\beta; \mu, \sigma) = \text{Normal}(\beta; \mu, \sigma)$ and $q(\mathbf{z}; \pi) = \text{Categorical}(\mathbf{z}; \pi)$.

Specific variational algorithms inherit from the `VariationalInference` class. Each defines its own methods, such as a loss function and gradient. For example, we represent MAP estimation with an approximating family (qbeta and qz) of `PointMass` random variables, i.e., with all probability



```
def generative_network(eps):
  h = Dense(256, activation='relu')(eps)
  return Dense(28 * 28, activation=None)(h)

def discriminative_network(x):
  h = Dense(28 * 28, activation='relu')(x)
  return Dense(h, activation=None)(1)

# Probabilistic model
eps = Normal(mu=tf.zeros([N, d]), sigma=tf.ones([N, d]))
x = generative_network(eps)

inference = ed.GANInference(data={x: x_train},
    discriminator=discriminative_network)
```

Figure 2.7: Generative adversarial networks: **(left)** graphical model; **(right)** probabilistic program. The model (generator) uses a parameterized function (discriminator) for training.

mass concentrated at a point. `MAP` inherits from `VariationalInference` and defines the negative log joint density as the loss function; it uses existing optimizers inside TensorFlow. In Section 2.4.1, we experiment with multiple gradient estimators for black box variational inference (Ranganath et al., 2014). Each estimator implements the same loss (an objective proportional to the divergence $\text{KL}(q \,\|\, p)$) and a different update rule (stochastic gradient).

Monte Carlo approximates the posterior using samples (Robert and Casella, 1999). Monte Carlo is an inference where the approximating family is an empirical distribution, $q(\beta; \{\beta^{(t)}\}) = \frac{1}{T}\sum_{t=1}^{T} \delta(\beta, \beta^{(t)})$ and $q(\mathbf{z}; \{\mathbf{z}^{(t)}\}) = \frac{1}{T}\sum_{t=1}^{T} \delta(\mathbf{z}, \mathbf{z}^{(t)})$. The parameters are $\boldsymbol{\lambda} = \{\beta^{(t)}, \mathbf{z}^{(t)}\}$. See Figure 2.6 (right). Monte Carlo algorithms proceed by updating one sample $\beta^{(t)}, \mathbf{z}^{(t)}$ at a time in the empirical approximation. Specific MC samplers determine the update rules: they can use gradients such as in Hamiltonian Monte Carlo (Neal, 2011) and graph structure such as in sequential Monte Carlo (Doucet et al., 2001).

Edward also supports non-Bayesian methods such as generative adversarial networks (GANs) (Goodfellow et al., 2014). See Figure 2.7. The model posits random noise `eps` over N data points, each with d dimensions; this random noise feeds into a `generative_network` function, a neural network that outputs real-valued data `x`. In addition, there is a `discriminative_network` which takes data as input and outputs the probability that the data is real (in logit parameterization). We build `GANInference`; running it optimizes parameters inside the two neural network functions. This approach extends to many advances in GANs (e.g., Denton et al. (2015); Li et al. (2015)).

```
qbeta = PointMass(params=tf.Variable(tf.zeros([K, D])))
qz = Categorical(logits=tf.Variable(tf.zeros([N, K])))

inference_e = ed.VariationalInference({z: qz}, data={x: x_train, beta: qbeta})
inference_m = ed.MAP({beta: qbeta}, data={x: x_train, z: qz})
...
for _ in range(10000):
  inference_e.update()
  inference_m.update()
```

Figure 2.8: Combining inference algorithms to perform variational EM.

Finally, one can design algorithms that would otherwise require tedious algebraic manipulation. With symbolic algebra on nodes of the computational graph, we can uncover conjugacy relationships between random variables. Users can then integrate out variables to automatically derive classical Gibbs (Gelfand and Smith, 1990), mean-field updates (Bishop, 2006), and exact inference. These algorithms are being currently developed in Edward.

2.3.3 Composing Inferences

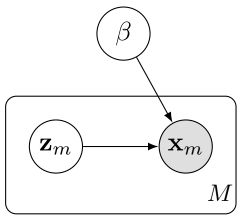
Core to Edward's design is that inference can be written as a collection of separate inference programs. Below we demonstrate variational EM, with an (approximate) E-step over local variables and an M-step over global variables. We instantiate two algorithms, each of which conditions on inferences from the other, and we alternate with one update of each (Neal and Hinton, 1993), This extends to many other cases such as exact EM for exponential families, contrastive divergence (Hinton, 2002), pseudo-marginal methods (Andrieu and Roberts, 2009), and Gibbs sampling within variational inference (Wang and Blei, 2012; Hoffman and Blei, 2015). We can also write message passing algorithms, which solve a collection of local inference problems (Koller and Friedman, 2009). For example, classical message passing uses exact local inference and expectation propagation locally minimizes the Kullback-Leibler divergence, $\text{KL}(p \,\|\, q)$ (Minka, 2001).

2.3.4 Data Subsampling

Stochastic optimization (Bottou, 2010) scales inference to massive data and is key to algorithms such as stochastic gradient Langevin dynamics (Welling and Teh, 2011) and stochastic variational inference (Hoffman et al., 2013). The idea is to cheaply estimate the model's log joint density in an unbiased way. At each step, one subsamples a data set $\{x_m\}$ of size M and then scales densities with respect to local variables,

$$\log p(\mathbf{x}, \mathbf{z}, \beta) = \log p(\beta) + \sum_{n=1}^{N} \Big[\log p(x_n \mid z_n, \beta) + \log p(z_n \mid \beta)\Big]$$
$$\approx \log p(\beta) + \frac{N}{M} \sum_{m=1}^{M} \Big[\log p(x_m \mid z_m, \beta) + \log p(z_m \mid \beta)\Big].$$

To support stochastic optimization, we represent only a subgraph of the full model. This prevents reifying the full model, which can lead to unreasonable memory consumption (Tristan et al., 2014). During initialization, we pass in a dictionary to properly scale the arguments. See Figure 2.9.



```
beta = Normal(mu=tf.zeros([K, D]), sigma=tf.ones([K, D]))
z = Categorical(logits=tf.zeros([M, K]))
x = Normal(mu=tf.gather(beta, z), sigma=tf.ones([M, D]))

qbeta = Normal(mu=tf.Variable(tf.zeros([K, D])),
               sigma=tf.nn.softplus(tf.Variable(tf.zeros([K, D]))))
qz = Categorical(logits=tf.Variable(tf.zeros([M, D])))

inference = ed.VariationalInference({beta: qbeta, z: qz}, data={x: x_batch})
inference.initialize(scale={x: float(N)/M, z: float(N)/M})
```

Figure 2.9: Data subsampling with a hierarchical model. We define a subgraph of the full model, forming a plate of size M rather than N. We then scale all local random variables by N/M.

Conceptually, the scale argument represents scaling for each random variable's plate, as if we had seen that random variable N/M as many times. As an example, Appendix A.6 shows how to implement stochastic variational inference in Edward. The approach extends naturally to streaming data (Doucet et al., 2000; Broderick et al., 2013; McInerney et al., 2015), dynamic batch sizes, and data structures in which working on a subgraph does not immediately apply (Binder et al., 1997; Johnson and Willsky, 2014; Foti et al., 2014).

2.4 Experiments

In this section, we illustrate two main benefits of Edward: flexibility and efficiency. For the former, we show how it is easy to compare different inference algorithms on the same model. For the latter, we show how it is easy to get significant speedups by exploiting computational graphs.

Inference method	Negative log-likelihood
VAE (Kingma and Welling, 2014b)	≤ 88.2
VAE without analytic KL	≤ 89.4
VAE with analytic entropy	≤ 88.1
VAE with score function gradient	≤ 87.9
Normalizing flows (Rezende and Mohamed, 2015)	≤ 85.8
Hierarchical variational model (Ranganath et al., 2016b)	≤ 85.4
Importance-weighted auto-encoders ($K = 50$) (Burda et al., 2016b)	≤ 86.3
HVM with IWAE objective ($K = 5$)	≤ 85.2
Rényi divergence ($\alpha = -1$) (Li and Turner, 2016)	≤ 140.5

Table 2.1: Inference methods for a probabilistic decoder on binarized MNIST. The Edward probabilistic programming language (PPL) is a convenient research platform, making it easy to both develop and experiment with many algorithms.

2.4.1 Recent Methods in Variational Inference

We demonstrate Edward's flexibility for experimenting with complex inference algorithms. We consider the VAE setup from Figure 3.4 and the binarized MNIST data set (Salakhutdinov and Murray, 2008). We use $d = 50$ latent variables per data point and optimize using ADAM. We study different components of the VAE setup using different methods; Appendix A.8 is a complete script. After training we evaluate held-out log likelihoods, which are lower bounds on the true value.

Table 4.1 shows the results. The first method uses the VAE from Figure 3.4. The next three methods use the same VAE but apply different gradient estimators: reparameterization gradient without an analytic KL; reparameterization gradient with an analytic entropy; and the score function gradient (Paisley et al., 2012a; Ranganath et al., 2014). This typically leads to the same optima but at different convergence rates. The score function gradient was slowest. Gradients with an analytic entropy produced difficulties around convergence: we switched to stochastic estimates of the entropy as it approached an optima. We also use hierarchical variational models (HVMS) (Ranganath et al., 2016b) with a normalizing flow prior; it produced similar results as a normalizing flow on the latent variable space (Rezende and Mohamed, 2015), and better than importance-weighted auto-encoders (IWAES) (Burda et al., 2016b).

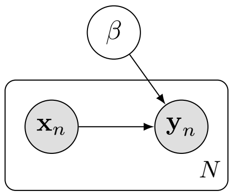
We also study novel combinations, such as HVMS with the IWAE objective, GAN-based optimization on the decoder (with pixel intensity-valued data), and Rényi divergence on the decoder. GAN-based

Probabilistic programming system	Runtime (s)
Handwritten NumPy (1 CPU)	534
PyMC3 (12 CPU) (Salvatier et al., 2015)	30.0
Edward (12 CPU)	**8.2**
Handwritten TensorFlow (GPU)	5.0
Edward (GPU)	**4.9**

Table 2.2: HMC benchmark for large-scale logistic regression. Edward (GPU) is significantly faster than other systems. In addition, Edward has no overhead: it is as fast as handwritten TensorFlow.

optimization does not enable calculation of the log-likelihood; Rényi divergence does not directly optimize for log-likelihood so it does not perform well. The key point is that Edward is a convenient research platform: they are all easy modifications of a given script.

2.4.2 GPU-accelerated Hamiltonian Monte Carlo



```
# Model
x = tf.Variable(x_data, trainable=False)
beta = Normal(mu=tf.zeros(D), sigma=tf.ones(D))
y = Bernoulli(logits=tf.dot(x, beta))

# Inference
qbeta = Empirical(params=tf.Variable(tf.zeros([T, D])))
inference = ed.HMC({beta: qbeta}, data={y: y_data})
inference.run(step_size=0.5 / N, n_steps=10)
```

Figure 2.10: Edward program for Bayesian logistic regression with HMC.

We benchmark runtimes for a fixed number of Hamiltonian Monte Carlo (HMC; Neal, 2011) iterations on modern hardware: a 12-core Intel i7-5930K CPU at 3.50GHz and an NVIDIA Titan X (Maxwell) GPU. We apply logistic regression on the Covertype dataset ($N = 581012$, $D = 54$; responses were binarized) using Edward and PyMC3 (Salvatier et al., 2015). We ran 100 HMC iterations, with 10 leapfrog updates per iteration, a step size of $0.5/N$, and single precision. Figure 2.10 illustrates the program in Edward.

Table 2.2 displays the runtimes.Edward (GPU) features a dramatic and 6x speedup over PyMC3 (12 CPU). This showcases the value of building a PPL on top of computational graphs. The speedup stems from fast matrix multiplication when calculating the model's log-likelihood; GPUs can efficiently parallelize this computation. We expect similar speedups for models whose bottleneck is also matrix multiplication, such as deep neural networks.

There are various reasons for the speedup. For PyMC3, we note Edward's speedup is not a result of

PyMC3's Theano backend compared to Edward's TensorFlow. Rather, PyMC3 does not use Theano for all its computation, so it experiences communication overhead with NumPy. (PyMC3 was actually slower when using the GPU.) We predict that porting Edward's design to Theano would feature similar speedups.

In addition to these speedups, we highlight that Edward has no runtime overhead: it is as fast as handwritten TensorFlow. Following Section 2.3.1, this is because the computational graphs for inference are in fact the same for Edward and the handwritten code.

2.5 Discussion

We described Edward, a Turing-complete PPL with compositional representations for probabilistic models and inference. Edward expands the scope of probabilistic programming to be as flexible and computationally efficient as traditional deep learning. For flexibility, we showed how Edward can use a variety of composable inference methods, capture recent advances in variational inference and generative adversarial networks, and finely control the inference algorithms. For efficiency, we showed how Edward leverages computational graphs to achieve fast, parallelizable computation, scales to massive data, and incurs no runtime overhead over handwritten code.

As with any language design, Edward makes tradeoffs in pursuit of its flexibility and speed for research. For example, an open challenge in Edward is to better facilitate programs with complex control flow and recursion. While possible to represent, it is unknown how to enable their flexible inference strategies. In addition, it is open how to expand Edward's design to dynamic computational graph frameworks—which provide more flexibility in their programming paradigm—but may sacrifice performance. A crucial next step for probabilistic programming is to leverage dynamic computational graphs while maintaining the flexibility and efficiency that Edward offers. We discuss such advances in the next chapter.

Chapter 3: Simple, Distributed, and Accelerated Probabilistic Programming

3.1 Introduction

Many developments in deep learning can be interpreted as blurring the line between model and computation. Some have even gone so far as to declare a new paradigm of "differentiable programming," in which the goal is not merely to train a model but to perform general program synthesis.[1] In this view, attention (Bahdanau et al., 2015) and gating (Hochreiter and Schmidhuber, 1997) describe boolean logic; skip connections (He et al., 2016) and conditional computation (Bengio et al., 2015; Graves, 2016) describe control flow; and external memory (Giles et al., 1990; Graves et al., 2014) accesses elements outside a function's internal scope. Learning algorithms are also increasingly dynamic: for example, learning to learn (Hochreiter et al., 2001), neural architecture search (Zoph and Le, 2017), and optimization within a layer (Amos and Kolter, 2017).

The differentiable programming paradigm encourages modelers to explicitly consider computational expense: one must consider not only a model's statistical properties ("how well does the model capture the true data distribution?"), but its computational, memory, and bandwidth costs ("how efficiently can it train and make predictions?"). This philosophy allows researchers to engineer deep-learning systems that run at the very edge of what modern hardware makes possible.

By contrast, the probabilistic programming community has tended to draw a hard line between model and computation: first, one specifies a probabilistic model as a program; second, one performs an "inference query" to automatically train the model given data (Spiegelhalter et al., 1995; Pfeffer, 2007; Carpenter et al., 2016). This design choice makes it difficult to implement probabilistic models at truly large scales, where training multi-billion parameter models requires splitting model computation across accelerators and scheduling communication (Shazeer et al., 2017). Recent advances such as

[1] Recent advocates of this trend include Tom Dietterich (`https://twitter.com/tdietterich/status/948811925038669824`) and Yann LeCun (`https://www.facebook.com/yann.lecun/posts/10155003011462143`). It is a classic idea in the programming languages field (Baydin et al., 2015).

Edward (Tran et al., 2017) have enabled finer control over inference procedures in deep learning (see also (Mansinghka et al., 2014; Bingham et al., 2018)). However, they all treat inference as a closed system: this makes them difficult to compose with arbitrary computation, and with the broader machine learning ecosystem, such as production platforms (Baylor et al., 2017).

In this paper, we describe a simple approach for embedding probabilistic programming in a deep learning ecosystem; our implementation is in TensorFlow and Python, named Edward2. This lightweight approach offers a low-level modality for flexible modeling—one which deep learners benefit from flexible prototyping with probabilistic primitives, and one which probabilistic modelers benefit from tighter integration with familiar numerical ecosystems.

Contributions. We distill the core of probabilistic programming down to a single abstraction—the random variable. Unlike existing languages, there is no abstraction for learning: algorithms may for example be functions taking a model as input (another function) and returning tensors.

This low-level design has two important implications. First, it enables research flexibility: a researcher has freedom to manipulate model computation for training and testing. Second, it enables bigger models using accelerators such as tensor processing units (TPUs) (Jouppi et al., 2017): TPUs require specialized ops in order to distribute computation and memory across a physical network topology.

We illustrate three applications: a model-parallel variational auto-encoder (VAE) (Kingma and Welling, 2014a) with TPUs; a data-parallel autoregressive model (Image Transformer (Parmar et al., 2018)) with TPUs; and multi-GPU No-U-Turn Sampler (NUTS) (Hoffman and Gelman, 2014). For both a state-of-the-art VAE on 64x64 ImageNet and Image Transformer on 256x256 CelebA-HQ, our approach achieves an optimal linear speedup from 1 to 256 TPUv2 chips. With NUTS, we see a 37x over PyMC3 (Salvatier et al., 2016).

3.2 Random Variables Are All You Need

We outline probabilistic programs in Edward2. They require only one abstraction: a random variable. We then describe how to perform flexible, low-level manipulations using tracing.

3.2.1 Probabilistic Programs, Variational Programs, and Many More

```python
def model():
  p = ed.Beta(1., 1., name="p")
  x = ed.Bernoulli(probs=p,
                   sample_shape=50,
                   name="x")
  return x
```

Figure 3.1: Beta-Bernoulli program. In eager mode, `model()` generates a binary vector of 50 elements. In graph mode, `model()` returns an op to be evaluated in a TensorFlow session.

```python
import neural_net_negative, neural_net_positive

def variational(x):
  eps = ed.Normal(0., 1., sample_shape=2)
  if eps[0] > 0:
    return neural_net_positive(eps[1], x)
  else:
    return neural_net_negative(eps[1], x)
```

Figure 3.2: Variational program (Ranganath et al., 2016a), available in eager mode. Python control flow is applicable to generative processes: given a coin flip, the program generates from one of two neural nets. Their outputs can have differing shape (and structure).

Edward2 reifies any computable probability distribution as a Python function (program). Typically, the function executes the generative process and returns samples.[2] Inputs to the program—along with any scoped Python variables—represent values the distribution conditions on.

To specify random choices in the program, we use `RandomVariables` from Edward (Tran et al., 2016a), which has similarly been built on by Zhusuan (Shi et al., 2017) and Probtorch (Probtorch Developers, 2017). Random variables provide methods such as `log_prob` and `sample`, wrapping TensorFlow Distributions (Dillon et al., 2017). Further, Edward random variables augment a computational graph of TensorFlow operations: each random variable $\mathbf{x}$ is associated to a sampled tensor $\mathbf{x}^* \sim p(\mathbf{x})$ in the graph.

Figure 3.1 illustrates a toy example: a Beta-Bernoulli model, $p(\mathbf{x}, \mathbf{p}) = \text{Beta}(\mathbf{p} \mid 1, 1) \prod_{n=1}^{50} \text{Bernoulli}(x_n \mid \mathbf{p})$, where $\mathbf{p}$ is a latent probability shared across the 50 data points $\mathbf{x} \in \{0, 1\}^{50}$. The random variable x is 50-dimensional, parameterized by the tensor $\mathbf{p}^* \sim p(\mathbf{p})$. As part of TensorFlow, Edward2 supports two execution modes. Eager mode simultaneously places operations onto the computational graph and executes them; here, `model()` calls the generative process and returns a binary vector of 50 elements. Graph mode separately stages graph-building and execution; here, `model()` returns a deferred TensorFlow vector; one may run a TensorFlow session to fetch the vector.

Importantly, all distributions—regardless of downstream use—are written as probabilistic programs. Figure 3.2 illustrates an implicit variational program, i.e., a variational distribution which admits sampling but may not have a tractable density. In general, variational programs (Ranganath et al., 2016a), proposal programs (Cusumano-Towner and Mansinghka, 2018), and discriminators in ad-

[2]Instead of sampling, one can also represent a distribution in terms of its density; see Section 3.3.1.

versarial training (Goodfellow et al., 2014) are computable probability distributions. If we have a mechanism for manipulating these probabilistic programs, we do not need to introduce any additional abstractions to support powerful inference paradigms. Below we demonstrate this flexibility using a model-parallel VAE.

3.2.2 Example: Model-Parallel VAE with TPUs

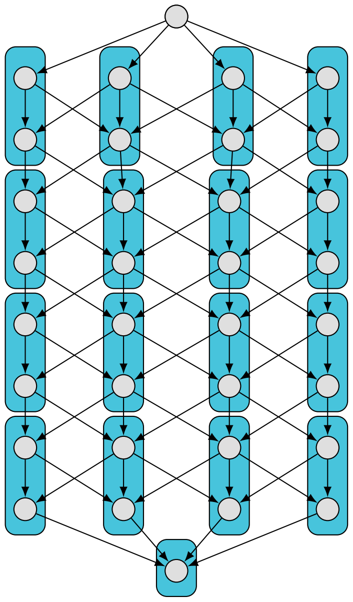
Figure 3.4 implements a model-parallel variational auto-encoder (VAE), which consists of a decoder, prior, and encoder. The decoder generates 16-bit audio (a sequence of T values in $[0, 2^{16} - 1]$ normalized to $[0, 1]$); it employs an autoregressive flow, which for training efficiently parallelizes over sequence length (Papamakarios et al., 2017). The prior posits latents representing a coarse 8-bit resolution over $T/2$ steps; it is learnable with a similar architecture. The encoder compresses each sample into the coarse resolution; it is parameterized by a compressing function.

A TPU cluster arranges cores in a toroidal network, where for example, 512 cores may be arranged as a 16x16x2 torus interconnect. To utilize the cluster, the prior and decoder apply distributed autoregressive flows (Figure 3.3). They split compute across a virtual 4x4 topology in two ways: "across flows", where every 2 flows belong on a different core; and "within a flow", where 4 independent flows apply layers respecting autoregressive ordering (for space, we omit code for splitting within a flow). The encoder splits computation via `compressor`; for space, we also omit it.

The probabilistic programs are concise. They capture recent advances such as autoregressive flows and multi-scale latent variables, and they enable never-before-tried architectures where with 16x16 TPUv2 chips (512 cores), the model can split across 4.1TB memory and utilize up to 10^{16} FLOPS. All elements of the VAE—distributions, architectures, and computation placement—are extensible. For training, we use typical TensorFlow ops; we describe how this works next.

3.2.3 Tracing

We defined probabilistic programs as arbitrary Python functions. To enable flexible training, we apply tracing, a classic technique used across probabilistic programming (e.g., Mansinghka et al., 2014; Tolpin et al., 2016; Ritchie et al., 2016; Ge et al., 2018; Bingham et al., 2018) as well as automatic differentiation (e.g., Maclaurin et al., 2015). A tracer wraps a subset of the language's primitive operations so that the tracer can intercept control just before those operations are executed.



```
import SplitAutoregressiveFlow, masked_network
tfb = tf.contrib.distributions.bijectors

class DistributedAutoregressiveFlow(tfb.Bijector):
  def __init__(flow_size=[4]*8):
    self.flows = []
    for num_splits in flow_size:
      flow = SplitAutoregressiveFlow(masked_network, num_splits)
      self.flows.append(flow)
    self.flows.append(SplitAutoregressiveFlow(masked_network, 1))
    super(DistributedAutoregressiveFlow, self).__init__()

  def _forward(self, x):
    for l, flow in enumerate(self.flows):
      with tf.device(tf.contrib.tpu.core(l//2)):
        x = flow.forward(x)
    return x

  def _inverse_and_log_det_jacobian(self, y):
    ldj = 0.
    for l, flow in enumerate(self.flows[::-1]):
      with tf.device(tf.contrib.tpu.core(l//2)):
        y, new_ldj = flow.inverse_and_log_det_jacobian(y)
        ldj += new_ldj
    return y, ldj
```

Figure 3.3: Distributed autoregressive flows. **(right)** The default length is 8, each with 4 independent flows. Each flow transforms inputs via layers respecting autoregressive ordering. **(left)** Flows are partitioned across a virtual topology of 4x4 cores (rectangles); each core computes 2 flows and is locally connected; a final core aggregates. The virtual topology aligns with the physical TPU topology: for 4x4 TPUs, it is exact; for 16x16 TPUs, it is duplicated for data parallelism.

```
import upsample, compressor

def prior():
  """Uniform noise to 8-bit latent, [u1,...,u(T/2)] -> [z1,...,z(T/2)]"""
  dist = ed.Independent(ed.Uniform(low=tf.zeros([batch_size, T/2])))
  return ed.TransformedDistribution(dist, DistributedAutoregressiveFlow(flow_size))

def decoder(z):
  """Uniform noise + latent to 16-bit audio, [u1,...,uT], [z1,...,z(T/2)] -> [x1,...,xT]"""
  dist = ed.Independent(ed.Uniform(low=tf.zeros([batch_size, T])))
  dist = ed.TransformedDistribution(dist, tfb.Affine(shift=upsample(z)))
  return ed.TransformedDistribution(dist, DistributedAutoregressiveFlow(flow_size))

def encoder(x):
  """16-bit audio to 8-bit latent, [x1,...,xT] -> [z1,...,z(T/2)]"""
  loc, log_scale =  tf.split(compressor(x), 2, axis=-1)
  return ed.Normal(loc=loc, scale=tf.exp(log_scale))
```

Figure 3.4: Model-parallel VAE with TPUs, generating 16-bit audio from 8-bit latents. The prior and decoder split computation according to distributed autoregressive flows. The encoder may split computation according to `compressor`; we omit it for space.

```python
STACK = [lambda f, *a, **k: f(*a, **k)]

@contextmanager
def trace(tracer):
  STACK.append(tracer)
  yield
  STACK.pop()

def traceable(f):
  def fwrapped(*a, **k):
    STACK[-1](f, *a, **k)
  return fwrapped
```

Figure 3.5: Minimal implementation of tracing. `trace` defines a context; any traceable ops executed during it are replaced by calls to `tracer`. `traceable` registers these ops; we register Edward random variables.

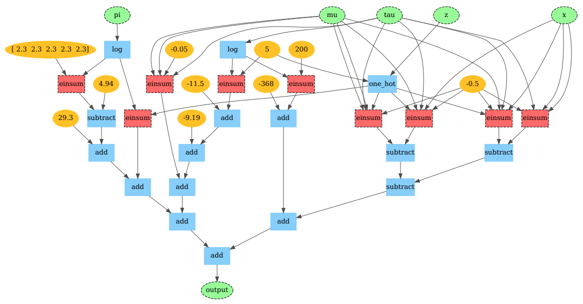


Figure 3.6: A program execution. It is a directed acyclic graph and is traced for various operations such as accumulating log-probabilities or finding conditional independence.

```python
def make_log_joint_fn(model):
  def log_joint_fn(**model_kwargs):
    def tracer(rv_call, *args, **kwargs):
      name = kwargs.get("name")
      kwargs["value"] = model_kwargs.get(name)
      rv = rv_call(*args, **kwargs)
      log_probs.append(tf.sum(rv.log_prob(rv)))
      return rv
    log_probs = []
    with trace(tracer):
      model(**model_kwargs)
    return sum(log_probs)
  return log_joint_fn
```

Figure 3.7: A higher-order function which takes a `model` program as input and returns its log-joint density function.

```python
def mutilate(model, **do_kwargs):
  def mutilated_model(*args, **kwargs):
    def tracer(rv_call, *args, **kwargs):
      name = kwargs.get("name")
      if name in do_kwargs:
        return do_kwargs[name]
      return rv_call(*args, **kwargs)
    with trace(tracer):
      return model(*args, **kwargs)
  return mutilated_model
```

Figure 3.8: A higher-order function which takes a `model` program as input and returns its causally intervened program. Intervention differs from conditioning: it does not change the sampled value but the distribution.

Figure 3.5 displays the core implementation: it is 10 lines of code.[3] `trace` is a context manager which, upon entry, pushes a `tracer` callable to a stack, and upon exit, pops `tracer` from the stack. `traceable` is a decorator: it registers functions so that they may be traced according to the stack. Edward2 registers random variables: for example, `Normal = traceable(edward1.Normal)`. The tracing implementation is also agnostic to the numerical backend. Appendix B.1 applies Figure 3.5 to implement Edward2 on top of SciPy.

3.2.4 Tracing Applications

Tracing is a common tool for probabilistic programming. However, in other languages, tracing primarily serves as an implementation detail to enable inference "meta-programming" procedures. In our approach, we promote it to be a user-level technique for flexible computation. We outline two examples; both are difficult to implement without user access to tracing.

Figure 3.7 illustrates a `make_log_joint` factory function. It takes a `model` program as input and returns its joint density function across a trace. We implement it using a `tracer` which sets random variable `values` to the input and accumulates its log-probability as a side-effect. Section 3.3.3 applies `make_log_joint` in a variational inference algorithm.

Figure 3.8 illustrates causal intervention (Pearl, 2003): it "mutilates" a program by setting random variables indexed by their name to another random variable. Note this effect is propagated to any descendants while leaving non-descendants unaltered: this is possible because Edward2 implicitly traces a dataflow graph over random variables, following a "push" model of evaluation. Other probabilistic operations more naturally follow a "pull" model of evaluation: mean-field variational inference requires evaluating energy terms corresponding to a single factor; we do so by reifying a variational program's trace (e.g., Figure 3.6) and walking backwards from that factor's node in the trace.

3.3 Examples: Learning with Low-Level Functions

We described probabilistic programs and how to manipulate their computation with low-level tracing functions. Unlike existing PPLS, there is no abstraction for learning. Below we provide examples of

[3]Rather than implement tracing, one can also reuse the pre-existing one in an autodiff system. However, our purposes require tracing with user control (*tracer* functions above) in order to manipulate computation. This is not presently available in TensorFlow Eager or Autograd (Maclaurin et al., 2015)—which motivated our implementation.

```
import get_channel_embeddings, add_positional_embedding_nd, local_attention_1d

def image_transformer(inputs, hparams):
  x = get_channel_embeddings(3, inputs, hparams.hidden_size)
  x = tf.reshape(x, [-1, 32*32*3, hparams.hidden_size])
  x = tf.pad(x, [[0, 0], [1, 0], [0, 0]])[:, :-1, :]  # shift pixels right
  x = add_positional_embedding_nd(x, max_length=32*32*3+3)
  x = tf.nn.dropout(x, keep_prob=0.7)
  for _ in range(hparams.num_layers):
    y = local_attention_1d(x, hparams, attention_type="local_mask_right",
                           q_padding="LEFT", kv_padding="LEFT")
    x = tf.contrib.layers.layer_norm(tf.nn.dropout(y, keep_prob=0.7) + x, begin_norm_axis=-1)
    y = tf.layers.dense(x, hparams.filter_size, activation=tf.nn.relu)
    y = tf.layers.dense(y, hparams.hidden_size, activation=None)
    x = tf.contrib.layers.layer_norm(tf.nn.dropout(y, keep_prob=0.7) + x, begin_norm_axis=-1)
  logits = tf.layers.dense(x, 256, activation=None)
  return ed.Categorical(logits=logits).log_prob(inputs)

loss = -tf.reduce_sum(image_transformer(inputs, hparams))  # inputs has shape [batch,32,32,3]
train_op = tf.contrib.tpu.CrossShardOptimizer(tf.train.AdamOptimizer()).minimize(loss)
```
Figure 3.9: Data-parallel Image Transformer with TPUs (Parmar et al., 2018). It is a neural autoregressive model which computes the log-probability of a batch of images with self-attention. Our lightweight design enables representing and training the model as a log-probability function; this is more efficient than the typical representation of programs as a generative process. Embedding and self-attention functions are assumed in the environment; they are available in Tensor2Tensor (Vaswani et al., 2018).

how this works and its implications.

3.3.1 Example: Data-Parallel Image Transformer with TPUs

All PPLs have so far focused on a unifying representation of models, typically as a generative process. However, this can be inefficient in practice for certain models. Because our lightweight approach has no required signature for training, it permits alternative model representations.[4]

For example, Figure 3.9 represents the Image Transformer (Parmar et al., 2018) as a log-probability function. The Image Transformer is a state-of-the-art autoregressive model for image generation, consisting of a Categorical distribution parameterized by a batch of right-shifted images, embeddings, a sequence of alternating self-attention and feedforward layers, and an output layer. The function computes `log_prob` with respect to images and parallelizes over pixel dimensions. Unlike the log-probability, sampling requires programming the autoregressivity in serial, which is inefficient and harder to implement.[5] With the log-probability representation, data parallelism with TPUs is

[4]The Image Transformer provides a performance reason for when density representations may be preferred. Another compelling example are energy-based models $p(x) \propto \exp\{f(x)\}$, where sampling is not even available in closed-form; in contrast, the unnormalized density is.

[5]In principle, one can reify any model in terms of sampling and apply `make_log_joint` to obtain its density. However, `make_log_joint` cannot always be done efficiently in practice, such as in this example. In contrast, the reverse program

```
def nuts(...):
  samples = []
  for _ in range(num_samples):
    state = set_up_trajectory(...)
    depth = 0
    while no_u_turn(state):
      state = extend_trajectory(depth, state)
      depth += 1
    samples.append(state)
  return samples

def extend_trajectory(depth, state):
  if depth == 0:
    state = one_leapfrog_step(state)
  else:
    state = extend_trajectory(depth-1, state)
    if no_u_turn(state):
     state = extend_trajectory(depth-1, state)
  return state
```

Figure 3.10: Core logic in No-U-Turn Sampler (Hoffman and Gelman, 2014). This algorithm has data-dependent non-tail recursion.

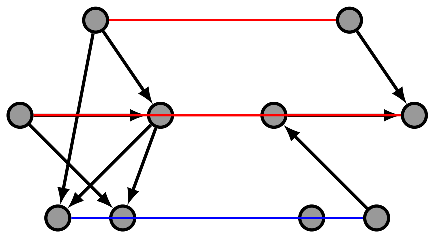


Figure 3.11: Learning often involves matching two execution traces such as a model program's **(left)** and a variational program's **(right)**, or a model program's with data tensors **(bottom)**. Red arrows align prior and variational variables. Blue arrows align observed variables and data; edges from data to variational variables represent amortization.

also immediate by cross-sharding the optimizer. The train op can be wrapped in a TF Estimator, or applied with manual TPU ops in order to aggregate training across cores.

3.3.2 Example: No-U-Turn Sampler

Figure 3.10 demonstrates the core logic behind the No-U-Turn Sampler (NUTS), a Hamiltonian Monte Carlo algorithm which adaptively selects the path length hyperparameter during leapfrog integration. Its implementation uses non-tail recursion, following the pseudo-code in Hoffman and Gelman (2014, Alg 6); both CPUs and GPUs are compatible. See source code for the full implementation; Appendix B.2 also implements a grammar VAE (Kusner et al., 2017) using a data-dependent while loop.

The ability to integrate NUTS requires interoperability with eager mode: NUTS requires Python control flow, as it is difficult to implement recursion natively with TensorFlow ops. (NUTS is not available, e.g., in Edward 1.) However, eager execution has tradeoffs (not unique to our approach). For example, it incurs a non-negligible overhead over graph mode, and it has preliminary support for TPUs. Our lightweight design supports both modes so the user can select either.

transformation from density to sampling can be done efficiently: in this example, sampling can at best compute in serial order; therefore it requires no performance optimization.

3.3.3 Example: Alignment of Probabilistic Programs

Learning algorithms often involve manipulating multiple probabilistic programs. For example, a variational inference algorithm takes two programs as input—the model program and variational program—and computes a loss function for optimization. This requires specifying which variables refer to each other in the two programs.

We apply alignment (Figure 3.11), which is a dictionary of key-value pairs, each from one string (a random variable's `name`) to another (a random variable in the other program). This dictionary provides flexibility over how random variables are aligned, independent of their specifications in each program. For example, this enables ladder VAEs (Sønderby et al., 2016) where prior and variational topological orderings are reversed; and VampPriors (Tomczak and Welling, 2018) where prior and variational parameters are shared.

Figure 3.12 shows variational inference with gradient descent using a fixed preconditioner. It applies `make_log_joint_fn` (Figure 3.7) and assumes `model` applies a random variable with name `'x'` (such as the VAE in Section 3.2.2). Note this extends alignment from Edward 1 to dynamic programs (Tran et al., 2017): instead of aligning nodes in static graphs at construction-time, it aligns nodes in execution traces at runtime. It also has applications for aligning model and proposal programs in Metropolis-Hastings; model and discriminator programs in adversarial training; and even model programs and data infeeding functions ("programs") in input-output pipelines.

3.3.4 Example: Learning to Learn by Variational Inference by Gradient Descent

A lightweight design is not only advantageous for flexible specification of learning algorithms but flexible composability: here, we demonstrate nested inference via learning to learn. Recall Figure 3.12 performs variational inference with gradient descent. Figure 3.13 applies gradient descent on the output of that gradient descent algorithm. It finds the optimal preconditioner (Andrychowicz et al., 2016). This is possible because learning algorithms are simply compositions of numerical operations; the composition is fully differentiable. This differentiability is not possible with Edward, which manipulates `inference` objects: taking gradients of one is not well-defined.[6] See also Appendix B.3 which illustrates Markov chain Monte Carlo within variational inference.

[6]Unlike Edward, Edward2 can also specify distributions over the learning algorithm.

```
import model, variational, align, x

def train(precond):
  def loss_fn(x):
    qz = variational(x)
    log_joint_fn = make_log_joint_fn(model)
    kwargs = {align[rv.name]: rv
              for rv in toposort(qz)}
    energy = log_joint_fn(x=x, **kwargs)
    entropy = sum([tf.reduce_sum(rv.entropy())
                   for rv in toposort(qz)])
    return -energy - entropy

  grad_fn = tfe.implicit_gradients(loss_fn)
  optimizer = tf.train.AdamOptimizer(0.1)
  for _ in range(500):
    grads = tf.tensordot(precond, grad_fn(x), [[1], [0]])
    optimizer.apply_gradients(grads)
  return loss_fn(x)
```

Figure 3.12: Variational inference with preconditioned gradient descent. Edward2 offers writing the probabilistic program and performing arbitrary TensorFlow computation for learning.

```
grad_fn = tfe.gradients_function(train)
optimizer = tf.train.AdamOptimizer(0.1)
for _ in range(100):
  optimizer.apply_gradients(grad_fn())
```

Figure 3.13: Learning-to-learn. It finds the optimal preconditioner for `train` (Figure 3.12) by differentiating the entire learning algorithm with respect to the preconditioner.

3.4 Experiments

We introduced a lightweight approach for embedding probabilistic programming in a deep learning ecosystem. Here, we show that such an approach is particularly advantageous for exploiting modern hardware for multi-TPU VAEs and autoregressive models, and multi-GPU NUTS. CPU experiments use a six-core Intel E5-1650 v4, GPU experiments use 1-8 NVIDIA Tesla V100 GPUs, and TPU experiments use 2nd generation chips under a variety of topology arrangements. The TPUv2 chip comprises two cores: each features roughly 22 teraflops on mixed 16/32-bit precision (it is roughly twice the flops of a NVIDIA Tesla P100 GPU on 32-bit precision). In all distributed experiments, we cross-shard the optimizer for data-parallelism: each shard (core) takes a batch size of 1. All numbers are averaged over 5 runs.

3.4.1 High-Quality Image Generation

We evaluate models with near state-of-the-art results ("bits/dim") for non-autoregressive generation on 64x64 ImageNet (Oord et al., 2016) and autoregressive generation on 256x256 CelebA-HQ (Karras et al., 2018). We evaluate wall clock time of the number of examples (data points) processed per second.

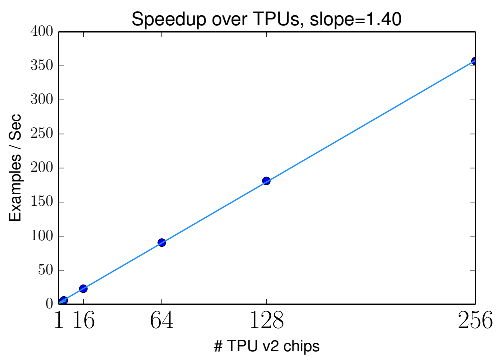


Figure 3.14: Vector-Quantized VAE on 64x64 ImageNet.

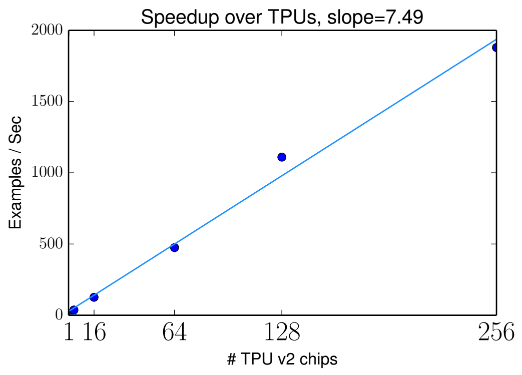


Figure 3.15: Image Transformer on 256x256 CelebA-HQ.

System	Runtime (ms)
PyMC3 (CPU)	74.8
Handwritten TF (CPU)	66.2
Edward2 (CPU)	68.4
Handwritten TF (1 GPU)	**9.5**
Edward2 (1 GPU)	**9.7**
Edward2 (8 GPU)	**2.3**

Table 3.1: Time per leapfrog step for No-U-Turn Sampler in Bayesian logistic regression. Edward2 (GPU) achieves a 37x speedup over PyMC3 (CPU); dynamism is not available in Edward. Edward2 also incurs negligible overhead over handwritten TensorFlow code.

For 64x64 ImageNet, we use a vector-quantized variational auto-encoder trained with soft EM (Roy et al., 2018). It encodes a 64x64x3 pixel image into a 8x8x10 tensor of latents, with a codebook size of 256 and where each code vector has 512 dimensions. The prior is an Image Transformer (Parmar et al., 2018) with 6 layers of local 1D self-attention. The encoder applies 4 convolutional layers with kernel size 5 and stride 2, 2 residual layers, and a dense layer. The decoder applies the reverse of a dense layer, 2 residual layers, and 4 transposed convolutional layers.

For 256x256 CelebA-HQ, we use a relatively small Image Transformer (Parmar et al., 2018) in order to fit the model in memory. It applies 5 layers of local 1D self-attention with block length of 256, hidden sizes of 128, attention key/value channels of 64, and feedforward layers with a hidden size of 256.

Figure 3.14 and Figure 3.15 show that for both models, Edward2 achieves an optimal linear scaling over the number of TPUv2 chips from 1 to 256. In experiments, we also found the larger batch sizes drastically sped up training.

3.4.2 No-U-Turn Sampler

We use the No-U-Turn Sampler (NUTS, (Hoffman and Gelman, 2014)) to illustrate the power of dynamic algorithms on accelerators. NUTS implements a variant of Hamiltonian Monte Carlo in which the fixed trajectory length is replaced by a recursive doubling procedure that adapts the length per iteration.

We compare Bayesian logistic regression using NUTS implemented in PyMC3 (Salvatier et al., 2016) to our eager-mode TensorFlow implementation. The model's log joint density is implemented as "handwritten" TensorFlow code and by a probabilistic program in Edward2; see code in Appendix B.4. We use the Covertype dataset (581,012 data points, 54 features, outcomes are binarized). Since adaptive sampling may lead NUTS iterations to take wildly different numbers of leapfrog steps, we report the average time per leapfrog step, averaged over 5 full NUTS trajectories (in these experiments, that typically amounted to about a thousand leapfrog steps total).

Table 3.1 shows that Edward2 (GPU) has up to a 37x speedup over PyMC3 with multi-threaded CPU. [7] In addition, while Edward2 in principle introduces overhead in eager mode due to its tracing mechanism, the speed differential between Edward2 and handwritten TensorFlow code is neligible (smaller than between-run variation). This demonstrates that the power of the PPL formalism comes with negligible overhead.

3.5 Discussion

We described a simple, low-level approach for embedding probabilistic programming in a deep learning ecosystem. For both a state-of-the-art VAE on 64x64 ImageNet and Image Transformer on 256x256 CelebA-HQ, we achieve an optimal linear speedup from 1 to 256 TPUv2 chips. For NUTS, we see up to 100x speedups over other systems.

As current work, we are pushing on this design as a stage for fundamental research in generative models and Bayesian neural networks (e.g., (Tran and Blei, 2018; Wen et al., 2018; Hafner et al., 2018)). We describe some examples in the next chapters. In addition, our experiments relied on data parallelism to show massive speedups. Recent work has improved distributed programming

[7] PyMC3 is actually slower with GPU than CPU; its code frequently communicates between Theano on the GPU and NumPy on the CPU.

of neural networks for both model parallelism and parallelism over large inputs such as super-high-resolution images (Shazeer et al., 2018). Combined with this work, we hope to push the limits of giant probabilistic models with over 1 trillion parameters and over 4K resolutions (50 million dimensions).

Chapter 4: Applications in Variational Inference

In Chapter 2 and Chapter 3, we described the design of deep probabilistic programming systems. In this chapter and the next, we will delve into the development of new variational inference algorithms and probabilistic models, whose research was explicitly made possible with Edward.

4.1 Introduction

Variational inference is a powerful tool for approximate posterior inference. The idea is to posit a family of distributions over the latent variables and then find the member of that family closest to the posterior. Originally developed in the 1990s (Hinton and van Camp, 1993; Waterhouse et al., 1996; Jordan et al., 1999a), variational inference has enjoyed renewed interest around developing scalable optimization for large datasets (Hoffman et al., 2013), deriving generic strategies for easily fitting many models (Ranganath et al., 2014), and applying neural networks as a flexible parametric family of approximations (Kingma and Welling, 2014b; Rezende et al., 2014). This research has been particularly successful for computing with deep Bayesian models (Neal, 1990; Ranganath et al., 2015a), which require inference of a complex posterior distribution (Hinton et al., 2006).

Classical variational inference typically uses the mean-field family, where each latent variable is independent and governed by its own variational distribution. While convenient, the strong independence limits learning deep representations of data. Newer research aims toward richer families that allow dependencies among the latent variables. One way to introduce dependence is to consider the variational family itself as a model of the latent variables (Lawrence, 2000; Ranganath et al., 2015b). These *variational models* naturally extend to Bayesian hierarchies, which retain the mean-field "likelihood" but introduce dependence through variational latent variables.

In this chapter we develop a powerful new variational model—the variational Gaussian process (VGP). The VGP is a Bayesian nonparametric variational model; its complexity grows efficiently and towards *any* distribution, adapting to the inference problem at hand. We highlight three main contributions of this chapter:

1. We prove a universal approximation theorem: under certain conditions, the VGP can capture any continuous posterior distribution—it is a variational family that can be specified to be as expressive as needed.

2. We derive an efficient stochastic optimization algorithm for variational inference with the VGP. Our algorithm can be used in a wide class of models. Inference with the VGP is a black box variational method (Ranganath et al., 2014).

3. We study the VGP on standard benchmarks for unsupervised learning, applying it to perform inference in deep latent Gaussian models (Rezende et al., 2014) and DRAW (Gregor et al., 2015), a latent attention model. For both models, we report the best results to date.

4.2 Variational Gaussian Process

Variational models introduce latent variables to the variational family, providing a rich construction for posterior approximation (Ranganath et al., 2015b). Here we introduce the variational Gaussian process (VGP), a Bayesian nonparametric variational model that is based on the Gaussian process. The Gaussian process (GP) provides a class of latent variables that lets us capture downstream distributions with varying complexity.

We first review variational models and Gaussian processes. We then outline the mechanics of the VGP and prove that it is a universal approximator.

4.2.1 Variational models

Let $p(\mathbf{z} \mid \mathbf{x})$ denote a posterior distribution over d latent variables $\mathbf{z} = (z_1, \dots, z_d)$ conditioned on a data set $\mathbf{x}$. For a family of distributions $q(\mathbf{z}; \boldsymbol{\lambda})$ parameterized by $\boldsymbol{\lambda}$, variational inference seeks to minimize the divergence $\mathrm{KL}(q(\mathbf{z}; \boldsymbol{\lambda}) \,\|\, p(\mathbf{z} \mid \mathbf{x}))$. This is equivalent to maximizing the ELBO (Wainwright and Jordan, 2008). The ELBO can be written as a sum of the expected log likelihood of the data and the KL divergence between the variational distribution and the prior,

$$\mathcal{L} = \mathbb{E}_{q(\mathbf{z};\boldsymbol{\lambda})}[\log p(\mathbf{x} \mid \mathbf{z})] - \mathrm{KL}(q(\mathbf{z};\boldsymbol{\lambda}) \| p(\mathbf{z})). \tag{4.1}$$

Traditionally, variational inference considers a tractable family of distributions with analytic forms for its density. A common specification is a fully factorized distribution $\prod_i q(z_i; \lambda_i)$, also known as the mean-field family. While mean-field families lead to efficient computation, they limit the expressiveness of the approximation.

The variational family of distributions can be interpreted as a model of the latent variables $\mathbf{z}$, and it can be made richer by introducing new latent variables. Hierarchical variational models consider distributions specified by a variational prior of the mean-field parameters $q(\boldsymbol{\lambda}; \boldsymbol{\theta})$ and a factorized "likelihood" $\prod_i q(z_i \mid \lambda_i)$. This specifies the variational model,

$$q(\mathbf{z}; \boldsymbol{\theta}) = \int \Big[\prod_i q(z_i \mid \lambda_i)\Big] q(\boldsymbol{\lambda}; \boldsymbol{\theta})\, \mathrm{d}\boldsymbol{\lambda}, \tag{4.2}$$

which is governed by prior hyperparameters $\boldsymbol{\theta}$. Hierarchical variational models are richer than classical variational families—their expressiveness is determined by the complexity of the prior $q(\boldsymbol{\lambda})$. Many expressive variational approximations can be viewed under this construct (Saul and Jordan, 1996; Rezende and Mohamed, 2015; Tran et al., 2015).

4.2.2 Gaussian Processes

We now review the Gaussian process (GP) (Rasmussen and Williams, 2006). Consider a data set of m source-target pairs $\mathcal{D} = \{(\mathbf{s}_n, \mathbf{t}_n)\}_{n=1}^m$, where each source $\mathbf{s}_n$ has c covariates paired with a multi-dimensional target $\mathbf{t}_n \in \mathbb{R}^d$. We aim to learn a function over all source-target pairs, $\mathbf{t}_n = f(\mathbf{s}_n)$, where $f : \mathbb{R}^c \to \mathbb{R}^d$ is unknown. Let the function f decouple as $f = (f_1, \dots, f_d)$, where each $f_i : \mathbb{R}^c \to \mathbb{R}$. GP regression estimates the functional form of f by placing a prior,

$$p(f) = \prod_{i=1}^d \mathcal{GP}(f_i; \mathbf{0}, \mathbf{K}_{ss}),$$

where $\mathbf{K}_{ss}$ denotes a covariance function $k(\mathbf{s}, \mathbf{s}')$ evaluated over pairs of inputs $\mathbf{s}, \mathbf{s}' \in \mathbb{R}^c$. In this paper, we consider automatic relevance determination (ARD) kernels

$$k(\mathbf{s}, \mathbf{s}') = \sigma^2_{\text{ARD}} \exp\Big(-\frac{1}{2}\sum_{j=1}^{c} \omega_j (s_j - s'_j)^2 \Big), \tag{4.3}$$

with parameters $\boldsymbol{\theta} = (\sigma^2_{\text{ARD}}, \omega_1, \ldots, \omega_c)$. The weights ω_j tune the importance of each dimension. They can be driven to zero during inference, leading to automatic dimensionality reduction.

Given data $\mathcal{D}$, the conditional distribution of the GP forms a distribution over mappings which interpolate between input-output pairs,

$$p(f \mid \mathcal{D}) = \prod_{i=1}^{d} \mathcal{GP}(f_i; \mathbf{K}_{\boldsymbol{\xi}s}\mathbf{K}_{ss}^{-1}\mathbf{t}_i, \mathbf{K}_{\boldsymbol{\xi}\boldsymbol{\xi}} - \mathbf{K}_{\boldsymbol{\xi}s}\mathbf{K}_{ss}^{-1}\mathbf{K}_{\boldsymbol{\xi}s}^\top). \tag{4.4}$$

Here, $\mathbf{K}_{\boldsymbol{\xi}s}$ denotes the covariance function $k(\boldsymbol{\xi}, \mathbf{s})$ for an input $\boldsymbol{\xi}$ and over all data inputs $\mathbf{s}_n$, and $\mathbf{t}_i$ represents the i^{th} output dimension.

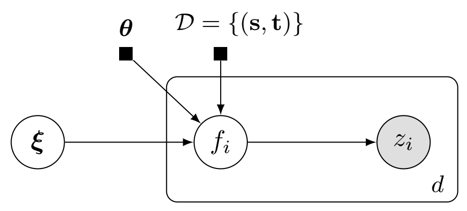
4.2.3 Variational Gaussian Processes

We describe the variational Gaussian process (VGP), a Bayesian nonparametric variational model that admits arbitrary structures to match posterior distributions. The VGP generates $\mathbf{z}$ by generating latent inputs, warping them with random non-linear mappings, and using the warped inputs as parameters to a mean-field distribution. The random mappings are drawn conditional on "variational data," which are variational parameters. We will show that the VGP enables samples from the mean-field to follow arbitrarily complex posteriors.

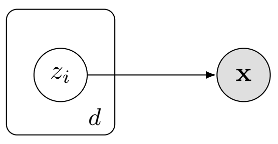
The VGP specifies the following generative process for posterior latent variables $\mathbf{z}$:

1. Draw latent input $\boldsymbol{\xi} \in \mathbb{R}^c$: $\boldsymbol{\xi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$.
2. Draw non-linear mapping $f : \mathbb{R}^c \to \mathbb{R}^d$ conditioned on $\mathcal{D}$: $f \sim \prod_{i=1}^d \mathcal{GP}(\mathbf{0}, \mathbf{K}_{\boldsymbol{\xi}\boldsymbol{\xi}}) \mid \mathcal{D}$.
3. Draw approximate posterior samples $\mathbf{z} \in \text{supp}(p)$: $\mathbf{z} = (z_1, \ldots, z_d) \sim \prod_{i=1}^d q(f_i(\boldsymbol{\xi}))$.

Figure 4.1 displays a graphical model for the VGP. Here, $\mathcal{D} = \{(\mathbf{s}_n, \mathbf{t}_n)\}_{n=1}^m$ represents variational data, comprising input-output pairs that are parameters to the variational distribution. Marginalizing



(a) VARIATIONAL MODEL



(b) GENERATIVE MODEL

Figure 4.1: (a) Graphical model of the variational Gaussian process. The VGP generates samples of latent variables $\mathbf{z}$ by evaluating random non-linear mappings of latent inputs $\boldsymbol{\xi}$, and then drawing mean-field samples parameterized by the mapping. These latent variables aim to follow the posterior distribution for a generative model **(b)**, conditioned on data $\mathbf{x}$.

over all latent inputs and non-linear mappings, the VGP is

$$q_{\text{VGP}}(\mathbf{z}; \boldsymbol{\theta}, \mathcal{D}) = \iint \left[\prod_{i=1}^{d} q(z_i \,|\, f_i(\boldsymbol{\xi}))\right] \left[\prod_{i=1}^{d} \mathcal{GP}(f_i; \mathbf{0}, \mathbf{K}_{\boldsymbol{\xi}\boldsymbol{\xi}}) \,|\, \mathcal{D}\right] \mathcal{N}(\boldsymbol{\xi}; \mathbf{0}, \mathbf{I})\, \mathrm{d}f\, \mathrm{d}\boldsymbol{\xi}. \tag{4.5}$$

The VGP is parameterized by kernel hyperparameters $\boldsymbol{\theta}$ and variational data.

As a variational model, the VGP forms an infinite ensemble of mean-field distributions. A mean-field distribution is given in the first term of the integrand above. It is *conditional* on a fixed function $f(\mathcal{D}ot)$ and input $\boldsymbol{\xi}$; the d outputs $f_i(\boldsymbol{\xi}) = \lambda_i$ are the mean-field's parameters. The VGP is a form of a hierarchical variational model (Equation 4.2) (Ranganath et al., 2015b). It places a continuous Bayesian nonparametric prior over mean-field parameters.

Unlike the mean-field, the VGP can capture correlation between the latent variables. The reason is that it evaluates the d independent GP draws at the same latent input $\boldsymbol{\xi}$. This induces correlation between their outputs, the mean-field parameters, and thus also correlation between the latent variables. Further, the VGP is flexible. The complex non-linear mappings drawn from the GP allow it to capture complex discrete and continuous posteriors.

We emphasize that the VGP needs variational data. Unlike typical GP regression, there are no observed data available to learn a distribution over non-linear mappings of the latent variables $\mathbf{z}$. Thus the "data" are variational parameters that appear in the conditional distribution of f in Equation 4.4. They anchor the random non-linear mappings at certain input-ouput pairs. When optimizing the VGP, the learned variational data enables finds a distribution of the latent variables that closely follows the posterior.

4.2.4 Universal approximation theorem

To understand the capacity of the VGP for representing complex posterior distributions, we analyze the role of the Gaussian process. For simplicity, suppose the latent variables $\mathbf{z}$ are real-valued, and the VGP treats the output of the function draws from the GP as posterior samples. Consider the optimal function f^*, which is the transformation such that when we draw $\boldsymbol{\xi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$ and calculate $\mathbf{z} = f^*(\boldsymbol{\xi})$, the resulting distribution of $\mathbf{z}$ *is* the posterior distribution.

An explicit construction of f^* exists if the dimension of the latent input $\boldsymbol{\xi}$ is equal to the number of latent variables. Let P^{-1} denote the inverse posterior CDF and Φ the standard normal CDF. Using techniques common in copula literature (Nelsen, 2006), the optimal function is

$$f^*(\boldsymbol{\xi}) = P^{-1}(\Phi(\xi_1), \dots, \Phi(\xi_d)).$$

Imagine generating samples $\mathbf{z}$ using this function. For latent input $\boldsymbol{\xi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, the standard normal CDF Φ applies the probability integral transform: it squashes ξ_i such that its output $u_i = \Phi(\xi_i)$ is uniformly distributed on $[0, 1]$. The inverse posterior CDF then transforms the uniform random variables $P^{-1}(u_1, \dots, u_d) = \mathbf{z}$ to follow the posterior. The function produces exact posterior samples.

In the VGP, the random function interpolates the values in the variational data, which are optimized to minimize the KL divergence. Thus, during inference, the distribution of the GP learns to concentrate around this optimal function. This perspective provides intuition behind the following result.

Theorem 1 (Universal approximation)**.** *Let $q(\mathbf{z}; \boldsymbol{\theta}, \mathcal{D})$ denote the variational Gaussian process. Consider a posterior distribution $p(\mathbf{z} \mid \mathbf{x})$ with a finite number of latent variables and continuous quantile function (inverse CDF). There exists a sequence of parameters $(\boldsymbol{\theta}_k, \mathcal{D}_k)$ such that*

$$\lim_{k \to \infty} \mathrm{KL}(q(\mathbf{z}; \boldsymbol{\theta}_k, \mathcal{D}_k) \,\|\, p(\mathbf{z} \mid \mathbf{x})) = 0.$$

See Appendix C.2 for a proof. Theorem 1 states that any posterior distribution with strictly positive density can be represented by a VGP. Thus the VGP is a flexible model for learning posterior distributions.

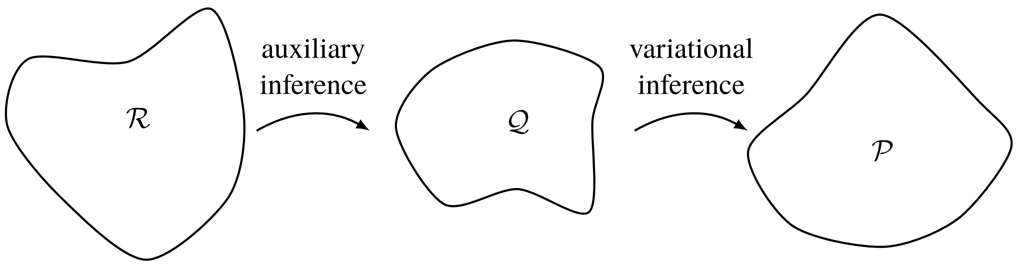


Figure 4.2: Sequence of domain mappings during inference, from variational latent variable space $\mathcal{R}$ to posterior latent variable space $\mathcal{Q}$ to data space $\mathcal{P}$. We perform variational inference in the posterior space and auxiliary inference in the variational space.

4.3 Black box inference

We derive an algorithm for black box inference over a wide class of generative models.

4.3.1 Variational objective

The original ELBO (Equation 4.1) is analytically intractable due to the log density, $\log q_{\text{VGP}}(\mathbf{z})$ (Equation 4.5). To address this, we present a tractable variational objective inspired by auto-encoders (Kingma and Welling, 2014b).

A tractable lower bound to the model evidence $\log p(\mathbf{x})$ can be derived by subtracting an expected KL divergence term from the ELBO,

$$\log p(\mathbf{x}) \geq \mathbb{E}_{q_{\text{VGP}}}[\log p(\mathbf{x} \mid \mathbf{z})] - \text{KL}(q_{\text{VGP}}(\mathbf{z}) \| p(\mathbf{z})) - \mathbb{E}_{q_{\text{VGP}}}\Big[\text{KL}(q(\boldsymbol{\xi}, f \mid \mathbf{z}) \| r(\boldsymbol{\xi}, f \mid \mathbf{z}))\Big],$$

where $r(\boldsymbol{\xi}, f \mid \mathbf{z})$ is an auxiliary model (we describe r in the next subsection). Various versions of this objective have been considered in the literature (Agakov and Barber, 2004), and it has been recently revisited by Salimans et al. (2015) and Ranganath et al. (2015b). We perform variational inference in the posterior latent variable space, minimizing $\text{KL}(q \| p)$ to learn the variational model; for this to occur we perform auxiliary inference in the variational latent variable space, minimizing $\text{KL}(q \| r)$ to learn an auxiliary model. See Figure 4.2.

Unlike previous approaches, we rewrite this variational objective to connect to auto-encoders:

$$\begin{aligned}\widetilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}) = \mathbb{E}_{q_{\text{VGP}}}[\log p(\mathbf{x} \mid \mathbf{z})] - \mathbb{E}_{q_{\text{VGP}}}\Big[\mathrm{KL}(q(\mathbf{z} \mid f(\boldsymbol{\xi})) \| p(\mathbf{z}))\Big] \\ - \mathbb{E}_{q_{\text{VGP}}}\Big[\mathrm{KL}(q(f \mid \boldsymbol{\xi}; \boldsymbol{\theta}) \| r(f \mid \boldsymbol{\xi}, \mathbf{z}; \boldsymbol{\phi})) + \log q(\boldsymbol{\xi}) - \log r(\boldsymbol{\xi} \mid \mathbf{z})\Big],\end{aligned} \tag{4.6}$$

where the KL divergences are now taken over tractable distributions (see Appendix C.3). In auto-encoder parlance, we maximize the expected negative reconstruction error, regularized by two terms: an expected divergence between the variational model and the original model's prior, and an expected divergence between the auxiliary model and the variational model's prior. This is simply a nested instantiation of the variational auto-encoder bound (Kingma and Welling, 2014b): a divergence between the inference model and a prior is taken as regularizers on both the posterior and variational spaces. This interpretation justifies the previously proposed bound for variational models; as we shall see, it also enables lower variance gradients during stochastic optimization.

4.3.2 Auto-encoding variational models

An inference network provide a flexible parameterization of approximating distributions as used in Helmholtz machines (Hinton and Zemel, 1994), deep Boltzmann machines (Salakhutdinov and Larochelle, 2010), and variational auto-encoders (Kingma and Welling, 2014b; Rezende et al., 2014). It replaces local variational parameters with global parameters coming from a neural network. For latent variables $\mathbf{z}_n$ (which correspond to a data point $\mathbf{x}_n$), an inference network specifies a neural network which takes $\mathbf{x}_n$ as input and its local variational parameters $\boldsymbol{\lambda}_n$ as output. This amortizes inference by only defining a set of global parameters.

To auto-encode the VGP we specify inference networks to parameterize both the variational and auxiliary models:

$$\mathbf{x}_n \mapsto q(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\theta}_n), \qquad \mathbf{x}_n, \mathbf{z}_n \mapsto r(\boldsymbol{\xi}_n, f_n \mid \mathbf{x}_n, \mathbf{z}_n; \boldsymbol{\phi}_n).$$

Formally, the output of these mappings are the parameters $\boldsymbol{\theta}_n$ and $\boldsymbol{\phi}_n$ respectively. We write the output as distributions above to emphasize that these mappings are a (global) parameterization of the variational model q and auxiliary model r. The local variational parameters $\boldsymbol{\theta}_n$ for q are the variational data $\mathcal{D}_n$. The auxiliary model r is specified as a fully factorized Gaussian with local variational parameters $\boldsymbol{\phi}_n = (\boldsymbol{\mu}_n \in \mathbb{R}^{c+d}, \boldsymbol{\sigma}_n^2 \in \mathbb{R}^{c+d})$. [1]

4.3.3 Stochastic optimization

We maximize the variational objective $\widetilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi})$ over both $\boldsymbol{\theta}$ and $\boldsymbol{\phi}$, where $\boldsymbol{\theta}$ newly denotes both the kernel hyperparameters and the inference network's parameters for the VGP, and $\boldsymbol{\phi}$ denotes the inference network's parameters for the auxiliary model. Following black box methods, we write the gradient as an expectation and apply stochastic approximations (Robbins and Monro, 1951), sampling from the variational model and evaluating noisy gradients.

First, we reduce variance of the stochastic gradients by analytically deriving any tractable expectations. The KL divergence between $q(\mathbf{z} \mid f(\boldsymbol{\xi}))$ and $p(\mathbf{z})$ is commonly used to reduce variance in traditional variational auto-encoders: it is analytic for deep generative models such as the deep latent Gaussian model (Rezende et al., 2014) and deep recurrent attentive writer (Gregor et al., 2015). The KL divergence between $r(f \mid \boldsymbol{\xi}, \mathbf{z})$ and $q(f \mid \boldsymbol{\xi})$ is analytic as the distributions are both Gaussian. The difference $\log q(\boldsymbol{\xi}) - \log r(\boldsymbol{\xi} \mid \mathbf{z})$ is simply a difference of Gaussian log densities. See Appendix C.3 for more details.

To derive black box gradients, we can first reparameterize the VGP, separating noise generation of samples from the parameters in its generative process (Kingma and Welling, 2014b; Rezende et al., 2014). The GP easily enables reparameterization: for latent inputs $\boldsymbol{\xi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, the transformation $\mathbf{f}(\boldsymbol{\xi}; \boldsymbol{\theta}) = \mathbf{L}\boldsymbol{\xi} + \mathbf{K}_{\boldsymbol{\xi}s}\mathbf{K}_{ss}^{-1}\mathbf{t}_i$ is a location-scale transform, where $\mathbf{L}\mathbf{L}^\top = \mathbf{K}_{\boldsymbol{\xi}\boldsymbol{\xi}} - \mathbf{K}_{\boldsymbol{\xi}s}\mathbf{K}_{ss}^{-1}\mathbf{K}_{\boldsymbol{\xi}s}^\top$. This is equivalent to evaluating $\boldsymbol{\xi}$ with a random mapping from the GP. Suppose the mean-field $q(\mathbf{z} \mid f(\boldsymbol{\xi}))$ is also reparameterizable, and let $\boldsymbol{\epsilon} \sim w$ such that $\mathbf{z}(\boldsymbol{\epsilon}; \mathbf{f})$ is a function of $\boldsymbol{\epsilon}$ whose output $\mathbf{z} \sim q(\mathbf{z} \mid f(\boldsymbol{\xi}))$. This two-level reparameterization is equivalent to the generative process for $\mathbf{z}$ outlined in Section 4.2.3.

[1] We let the kernel hyperparameters of the VGP be fixed across data points. Note also that unique from other auto-encoder approaches, we let r's inference network take both $\mathbf{x}_n$ and $\mathbf{z}_n$ as input: this avoids an explicit specification of the conditional distribution $r(\boldsymbol{\epsilon}, f \mid \mathbf{z})$, which may be difficult to model.

Algorithm 1: Black box inference with a variational Gaussian process

```
Input: Model $p(\mathbf{x}, \mathbf{z})$, Mean-field family $\prod_i q(\mathbf{z}_i \mid f_i(\boldsymbol{\xi}))$.
Output: Variational and auxiliary parameters $(\boldsymbol{\theta}, \boldsymbol{\phi})$.
Initialize $(\boldsymbol{\theta}, \boldsymbol{\phi})$ randomly.
while not converged do
    Draw noise samples $\boldsymbol{\xi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I})$, $\boldsymbol{\epsilon} \sim w$.
    Parameterize variational samples $\mathbf{z} = \mathbf{z}(\boldsymbol{\epsilon}; f(\boldsymbol{\xi}))$, $f(\boldsymbol{\xi}) = \mathbf{f}(\boldsymbol{\xi}; \boldsymbol{\theta})$.
    Update $(\boldsymbol{\theta}, \boldsymbol{\phi})$ with stochastic gradients $\nabla_{\boldsymbol{\theta}} \widetilde{\mathcal{L}}$, $\nabla_{\boldsymbol{\phi}} \widetilde{\mathcal{L}}$.
end
```

We now rewrite the variational objective as

$$\begin{aligned}
\widetilde{\mathcal{L}}(\boldsymbol{\theta}, \boldsymbol{\phi}) &= \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}\Big[\mathbb{E}_{w(\boldsymbol{\epsilon})}\Big[\log p(\mathbf{x} \mid \mathbf{z}(\boldsymbol{\epsilon}; \mathbf{f}))\Big] - \mathrm{KL}(q(\mathbf{z} \,|\, \mathbf{f}) \| p(\mathbf{z}))\Big] \\
&\quad - \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}\Big[\mathbb{E}_{w(\boldsymbol{\epsilon})}\Big[\mathrm{KL}(q(f \,|\, \boldsymbol{\xi}; \boldsymbol{\theta}) \| r(f \,|\, \boldsymbol{\xi}, \mathbf{z}(\boldsymbol{\epsilon}; \mathbf{f}); \boldsymbol{\phi})) + \log q(\boldsymbol{\xi}) - \log r(\boldsymbol{\xi} \,|\, \mathbf{z}(\boldsymbol{\epsilon}; \mathbf{f}))\Big]\Big].
\end{aligned} \tag{4.7}$$

Equation 4.7 enables gradients to move inside the expectations and backpropagate over the nested reparameterization. Thus we can take unbiased stochastic gradients, which exhibit low variance due to both the analytic KL terms and reparameterization. The gradients are derived in Appendix C.4, including the case when the first KL is analytically intractable.

We outline the method in Algorithm 1. For massive data, we apply subsampling on $\mathbf{x}$ (Hoffman et al., 2013). For gradients of the model log-likelihood, we employ convenient differentiation tools such as those in Stan and Theano (Carpenter et al., 2015; Bergstra et al., 2010). For non-differentiable latent variables $\mathbf{z}$, or mean-field distributions without efficient reparameterizations, we apply the black box gradient estimator from Ranganath et al. (2014) to take gradients of the inner expectation.

4.3.4 Computational and storage complexity

The algorithm has $\mathcal{O}(d + m^3 + LH^2)$ complexity, where d is the number of latent variables, m is the size of the variational data, and L is the number of layers of the neural networks with H the average hidden layer size. In particular, the algorithm is linear in the number of latent variables, which is competitive with other variational inference methods. The number of variational and auxiliary parameters has $\mathcal{O}(c + LH)$ complexity; this complexity comes from storing the kernel hyperparameters and the neural network parameters.

Unlike most GP literature, we require no low rank constraints, such as the use of inducing variables for scalable computation (Quiñonero-Candela and Rasmussen, 2005). The variational data serve a similar purpose, but inducing variables reduce the rank of a (fixed) kernel matrix; the variational data directly determine the kernel matrix and thus the kernel matrix is not fixed. Although we haven't found it necessary in practice, see Appendix C.5 for scaling the size of variational data.

4.4 Experiments

Following standard benchmarks for variational inference in deep learning, we learn generative models of images. In particular, we learn the deep latent Gaussian model (DLGM) (Rezende et al., 2014), a layered hierarchy of Gaussian random variables following neural network architecures, and the recently proposed DRAW (Gregor et al., 2015), a latent attention model that iteratively constructs complex images using a recurrent architecture and a sequence of variational auto-encoders (Kingma and Welling, 2014b).

For the learning rate we apply a version of RMSProp (Tieleman and Hinton, 2012), in which we scale the value with a decaying schedule $1/t^{1/2+\epsilon}$ for $\epsilon > 0$. We fix the size of variational data to be 500 across all experiments and set the latent input dimension equal to the number of latent variables.

4.4.1 Binarized MNIST

The binarized MNIST data set (Salakhutdinov and Murray, 2008) consists of 28x28 pixel images with binary-valued outcomes. Training a DLGM, we apply two stochastic layers of 100 random variables and 50 random variables respectively, and in-between each stochastic layer is a deterministic layer with 100 units using tanh nonlinearities. We apply mean-field Gaussian distributions for the stochastic layers and a Bernoulli likelihood. We train the VGP to learn the DLGM for the cases of one stochastic layer and two stochastic layers.

For DRAW (Gregor et al., 2015), we augment the mean-field Gaussian distribution originally used to

Model	$-\log p(\mathbf{x})$	$\leq$
DLGM + VAE [1]		86.76
DLGM + HVI (8 leapfrog steps) [2]	85.51	88.30
DLGM + NF ($k = 80$) [3]		85.10
EoNADE-5 2hl (128 orderings) [4]	84.68	
DBN 2hl [5]	84.55	
DARN 1hl [6]	84.13	
Convolutional VAE + HVI [2]	81.94	83.49
DLGM 2hl + IWAE ($k = 50$) [1]		82.90
DRAW [7]		80.97
DLGM 1hl + VGP		84.79
DLGM 2hl + VGP		81.32
DRAW + VGP		**79.88**

Table 4.1: Negative predictive log-likelihood for binarized MNIST. Previous best results are [1] (Burda et al., 2016a), [2] (Salimans et al., 2015), [3] (Rezende and Mohamed, 2015), [4] (Raiko et al., 2014), [5] (Murray and Salakhutdinov, 2009), [6] (Gregor et al., 2014), [7] (Gregor et al., 2015).

generate the latent samples at each time step with the VGP, as it places a complex variational prior over its parameters. The encoding recurrent neural network now outputs variational data (used for the variational model) as well as mean-field Gaussian parameters (used for the auxiliary model). We use the same architecture hyperparameters as in Gregor et al. (2015).

After training we evaluate test set log likelihood, which are lower bounds on the true value. See Table 4.1 which reports both approximations and lower bounds of $\log p(\mathbf{x})$ for various methods. The VGP achieves the highest known results on log-likelihood using DRAW, reporting a value of **-79.88** compared to the original highest of -80.97. The VGP also achieves the highest known results among the class of non-structure exploiting models using the DLGM, with a value of -81.32 compared to the previous best of -82.90 reported by Burda et al. (2016a).

4.4.2 Sketch

As a demonstration of the VGP's complexity for learning representations, we also examine the Sketch data set (Eitz et al., 2012). It consists of 20,000 human sketches equally distributed over 250 object categories. We partition it into 18,000 training examples and 2,000 test examples. We fix the architecture of DRAW to have a 2x2 read window, 5x5 write attention window, and 64 glimpses—these values were selected using a coarse grid search and choosing the set which lead to the best training

Model	Epochs	$\leq -\log p(\mathbf{x})$
DRAW	100	526.8
	200	479.1
	300	464.5
DRAW + VGP	100	**460.1**
	200	**444.0**
	300	**423.9**

Table 4.2: Negative predictive log-likelihood for Sketch, learned over hundreds of epochs over all 18,000 training examples.

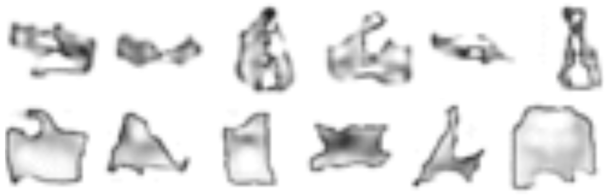


Figure 4.3: Generated images from DRAW with a VGP (top), and DRAW with the original variational auto-encoder (bottom). The VGP learns texture and sharpness, able to sketch more complex shapes.

log likelihood. For inference we use the original auto-encoder version as well as the augmented version with the VGP.

See Table 4.2. DRAW with the VGP achieves a significantly better lower bound, performing better than the original version which has seen state-of-the-art success in many computer vision tasks. (Until the results presented here, the results from the original DRAW were the best reported performance for this data set.). Moreover, the model inferred using the VGP is able to generate more complex images than the original version—it not only performs better but maintains higher visual fidelity.

4.5 Discussion

We present the variational Gaussian process (VGP), a variational model which adapts its shape to match complex posterior distributions. The VGP draws samples from a tractable distribution, and posits a Bayesian nonparametric prior over transformations from the tractable distribution to mean-field parameters. The VGP learns the transformations from the space of all continuous mappings—it is a universal approximator and finds good posterior approximations via optimization.

Chapter 5: Applications in Deep Probabilistic Models

5.1 Introduction

Consider a model of coin tosses. With probabilistic models, one typically posits a latent probability, and supposes each toss is a Bernoulli outcome given this probability (Murphy, 2012; Gelman et al., 2013). After observing a collection of coin tosses, Bayesian analysis lets us describe our inferences about the probability.

However, we know from the laws of physics that the outcome of a coin toss is fully determined by its initial conditions (say, the impulse and angle of flip) (Keller, 1986; Diaconis et al., 2007). Therefore a coin toss' randomness does not originate from a latent probability but in noisy initial parameters. This alternative model incorporates the physical system, better capturing the generative process. Furthermore the model is *implicit*, also known as a simulator: we can sample data from its generative process, but we may not have access to calculate its density (Diggle and Gratton, 1984; Hartig et al., 2011).

Coin tosses are simple, but they serve as a building block for complex implicit models. These models, which capture the laws and theories of real-world physical systems, pervade fields such as population genetics (Pritchard et al., 1999), statistical physics (Anelli et al., 2008), and ecology (Beaumont, 2010); they underlie structural equation models in economics and causality (Pearl, 2003); and they connect deeply to GANS (Goodfellow et al., 2014), which use neural networks to specify a flexible implicit density (Mohamed and Lakshminarayanan, 2016).

Unfortunately, implicit models, including GANS, have seen limited success outside specific domains. There are two reasons. First, it is unknown how to design implicit models for more general applications, exposing rich latent structure such as priors, hierarchies, and sequences. Second, existing methods for inferring latent structure in implicit models do not sufficiently scale to high-dimensional or large data sets. In this paper, we design a new class of implicit models and we develop a new algorithm for accurate and scalable inference.

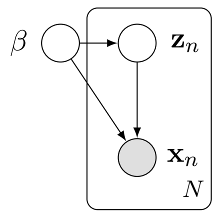
For modeling, Section 5.2 describes *hierarchical implicit models*, a class of Bayesian hierarchical models which only assume a process that generates samples. This class encompasses both simulators in the classical literature and those employed in GANS. For example, we specify a Bayesian GAN, where we place a prior on its parameters. The Bayesian perspective allows GANS to quantify uncertainty and improve data efficiency. We can also apply them to discrete data; this setting is not possible with traditional estimation algorithms for GANS (Kusner and Hernández-Lobato, 2016).

For inference, Section 5.3 develops *likelihood-free variational inference (*LFVI*)*, which combines variational inference with density ratio estimation (Sugiyama et al., 2012; Mohamed and Lakshminarayanan, 2016). Variational inference posits a family of distributions over latent variables and then optimizes to find the member closest to the posterior (Jordan et al., 1999b). Traditional approaches require a likelihood-based model and use crude approximations, employing a simple approximating family for fast computation. LFVI expands variational inference to implicit models and enables accurate variational approximations with implicit variational families: LFVI does not require the variational density to be tractable. Further, unlike previous Bayesian methods for implicit models, LFVI scales to millions of data points with stochastic optimization.

This work has diverse applications. First, we analyze a classical problem from the approximate Bayesian computation (ABC) literature, where the model simulates an ecological system (Beaumont, 2010). We analyze 100,000 time series which is not possible with traditional methods. Second, we analyze a Bayesian GAN, which is a GAN with a prior over its weights. Bayesian GANS outperform corresponding Bayesian neural networks with known likelihoods on several classification tasks. Third, we show how injecting noise into hidden units of recurrent neural networks corresponds to a deep implicit model for flexible sequence generation.

5.2 Hierarchical Implicit Models

Hierarchical models play an important role in sharing statistical strength across examples (Gelman and Hill, 2006). For a broad class of hierarchical Bayesian models, the joint distribution of the hidden



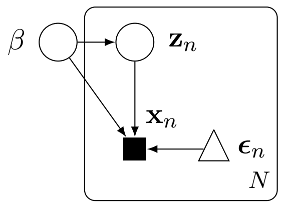


Figure 5.1: (**left**) Hierarchical model, with local variables $\mathbf{z}$ and global variables β. (**right**) **Hierarchical implicit model**. It is a hierarchical model where $\mathbf{x}$ is a deterministic function (denoted with a square) of noise $\boldsymbol{\epsilon}$ (denoted with a triangle).

and observed variables is

$$p(\mathbf{x}, \mathbf{z}, \beta) = p(\beta) \prod_{n=1}^{N} p(\mathbf{x}_n \mid \mathbf{z}_n, \beta) p(\mathbf{z}_n \mid \beta), \tag{5.1}$$

where $\mathbf{x}_n$ is an observation, $\mathbf{z}_n$ are latent variables associated to that observation (local variables), and β are latent variables shared across observations (global variables). See Figure 5.1 (left).

With hierarchical models, local variables can be used for clustering in mixture models, mixed memberships in topic models (Blei et al., 2003), and factors in probabilistic matrix factorization (Salakhutdinov and Mnih, 2008). Global variables can be used to pool information across data points for hierarchical regression (Gelman and Hill, 2006), topic models (Blei et al., 2003), and Bayesian nonparametrics (Teh and Jordan, 2010).

Hierarchical models typically use a tractable likelihood $p(\mathbf{x}_n \mid \mathbf{z}_n, \beta)$. But many likelihoods of interest, such as simulator-based models (Hartig et al., 2011) and generative adversarial networks (Goodfellow et al., 2014), admit high fidelity to the true data generating process and do not admit a tractable likelihood. To overcome this limitation, we develop *hierarchical implicit models* (HIMS).

Hierarchical implicit models have the same joint factorization as Equation 5.1 but only assume that one can sample from the likelihood. Rather than define $p(\mathbf{x}_n \mid \mathbf{z}_n, \beta)$ explicitly, HIMS define a function g that takes in random noise $\boldsymbol{\epsilon}_n \sim s(\cdot)$ and outputs $\mathbf{x}_n$ given $\mathbf{z}_n$ and β,

$$\mathbf{x}_n = g(\boldsymbol{\epsilon}_n \mid \mathbf{z}_n, \beta), \quad \boldsymbol{\epsilon}_n \sim s(\cdot).$$

The induced, implicit likelihood of $\mathbf{x}_n \in A$ given $\mathbf{z}_n$ and β is

$$\mathcal{P}(\mathbf{x}_n \in A \,|\, \mathbf{z}_n, \beta) = \int_{\{g(\boldsymbol{\epsilon}_n \,|\, \mathbf{z}_n, \beta) = \mathbf{x}_n \in A\}} s(\boldsymbol{\epsilon}_n) \, \mathrm{d}\boldsymbol{\epsilon}_n.$$

This integral is typically intractable. It is difficult to find the set to integrate over, and the integration itself may be expensive for arbitrary noise distributions $s(\cdot)$ and functions g.

Figure 5.1 (right) displays the graphical model for HIMS. Noise ($\boldsymbol{\epsilon}_n$) are denoted by triangles; deterministic computation ($\mathbf{x}_n$) are denoted by squares. We illustrate two examples.

Example: Physical Simulators. Given initial conditions, simulators describe a stochastic process that generates data. For example, in population ecology, the Lotka-Volterra model simulates predator-prey populations over time via a stochastic differential equation (Wilkinson, 2011). For prey and predator populations $x_1, x_2 \in \mathbb{R}^+$ respectively, one process is

$$\begin{aligned} \frac{\mathrm{d}x_1}{\mathrm{d}t} &= \beta_1 x_1 - \beta_2 x_1 x_2 + \epsilon_1, & \epsilon_1 &\sim \mathrm{Normal}(0, 10), \\ \frac{\mathrm{d}x_2}{\mathrm{d}t} &= -\beta_2 x_2 + \beta_3 x_1 x_2 + \epsilon_2, & \epsilon_2 &\sim \mathrm{Normal}(0, 10), \end{aligned}$$

where Gaussian noises ϵ_1, ϵ_2 are added at each full time step. The simulator runs for T time steps given initial population sizes for x_1, x_2. Lognormal priors are placed over β. The Lotka-Volterra model is grounded by theory but features an intractable likelihood. We study it in Section 5.4.

Example: Bayesian Generative Adversarial Network. Generative adversarial networks (GANS) define an implicit model and a method for parameter estimation (Goodfellow et al., 2014). They are known to perform well on image generation (Radford et al., 2016). Formally, the implicit model for a GAN is

$$\mathbf{x}_n = g(\boldsymbol{\epsilon}_n; \boldsymbol{\theta}), \quad \boldsymbol{\epsilon}_n \sim s(\cdot), \tag{5.2}$$

where g is a neural network with parameters $\boldsymbol{\theta}$, and s is a standard normal or uniform. The neural network g is typically not invertible; this makes the likelihood intractable.

The parameters $\boldsymbol{\theta}$ in GANS are estimated by divergence minimization between the generated and real data. We make GANs amenable to Bayesian analysis by placing a prior on the parameters $\boldsymbol{\theta}$. We call this a Bayesian GAN. Bayesian GANs enable modeling of parameter uncertainty and are inspired by Bayesian neural networks, which have been shown to improve the uncertainty and data efficiency of standard neural networks (MacKay, 1992; Neal, 1994). We study Bayesian GANs in Section 5.4; Appendix B provides example implementations in the Edward probabilistic programming language (Tran et al., 2016a).

5.3 Likelihood-Free Variational Inference

We described hierarchical implicit models, a rich class of latent variable models with local and global structure alongside an implicit density. Given data, we aim to calculate the model's posterior $p(\mathbf{z}, \beta \mid \mathbf{x}) = p(\mathbf{x}, \mathbf{z}, \beta)/p(\mathbf{x})$. This is difficult as the normalizing constant $p(\mathbf{x})$ is typically intractable. With implicit models, the lack of a likelihood function introduces an additional source of intractability.

We use variational inference (Jordan et al., 1999b). It posits an approximating family $q \in \mathcal{Q}$ and optimizes to find the member closest to $p(\mathbf{z}, \beta \mid \mathbf{x})$. There are many choices of variational objectives that measure closeness (Ranganath et al., 2016a; Li and Turner, 2016; Dieng et al., 2016). To choose an objective, we lay out desiderata for a variational inference algorithm for implicit models:

1. *Scalability*. Machine learning hinges on stochastic optimization to scale to massive data (Bottou, 2010). The variational objective should admit unbiased subsampling with the standard technique,

$$\sum_{n=1}^{N} f(\mathbf{x}_n) \approx \frac{N}{M} \sum_{m=1}^{M} f(\mathbf{x}_m),$$

   where some computation $f(\cdot)$ over the full data is approximated with a mini-batch of data $\{\mathbf{x}_m\}$.

2. *Implicit Local Approximations*. Implicit models specify flexible densities; this induces very complex posterior distributions. Thus we would like a rich approximating family for the per-data point approximations $q(\mathbf{z}_n \mid \mathbf{x}_n, \beta)$. This means the variational objective should only require that one can sample $\mathbf{z}_n \sim q(\mathbf{z}_n \mid \mathbf{x}_n, \beta)$ and not evaluate its density.

One variational objective meeting our desiderata is based on the classical minimization of the KL divergence. (Surprisingly, Appendix C details how the KL is the *only* possible objective among a broad class.)

5.3.1 KL Variational Objective

Classical variational inference minimizes the KL divergence from the variational approximation q to the posterior. This is equivalent to maximizing the ELBO,

$$\mathcal{L} = \mathbb{E}_{q(\beta,\mathbf{z}\,|\,\mathbf{x})}[\log p(\mathbf{x},\mathbf{z},\beta) - \log q(\beta,\mathbf{z}\,|\,\mathbf{x})]. \tag{5.3}$$

Let q factorize in the same way as the posterior,

$$q(\beta,\mathbf{z}\,|\,\mathbf{x}) = q(\beta)\prod_{n=1}^{N} q(\mathbf{z}_n\,|\,\mathbf{x}_n,\beta),$$

where $q(\mathbf{z}_n\,|\,\mathbf{x}_n,\beta)$ is an intractable density and since the data $\mathbf{x}$ is constant during inference, we drop conditioning for the global $q(\beta)$. Substituting p and q's factorization yields

$$\mathcal{L} = \mathbb{E}_{q(\beta)}[\log p(\beta) - \log q(\beta)] + \sum_{n=1}^{N}\mathbb{E}_{q(\beta)q(\mathbf{z}_n\,|\,\mathbf{x}_n,\beta)}[\log p(\mathbf{x}_n,\mathbf{z}_n\,|\,\beta) - \log q(\mathbf{z}_n\,|\,\mathbf{x}_n,\beta)].$$

This objective presents difficulties: the local densities $p(\mathbf{x}_n,\mathbf{z}_n\,|\,\beta)$ and $q(\mathbf{z}_n\,|\,\mathbf{x}_n,\beta)$ are both intractable. To solve this, we consider ratio estimation.

5.3.2 Ratio Estimation for the KL Objective

Let $q(\mathbf{x}_n)$ be the empirical distribution on the observations $\mathbf{x}$ and consider using it in a "variational joint" $q(\mathbf{x}_n,\mathbf{z}_n\,|\,\beta) = q(\mathbf{x}_n)q(\mathbf{z}_n\,|\,\mathbf{x}_n,\beta)$. Now subtract the log empirical $\log q(\mathbf{x}_n)$ from the ELBO above. The ELBO reduces to

$$\mathcal{L} \propto \mathbb{E}_{q(\beta)}[\log p(\beta) - \log q(\beta)] + \sum_{n=1}^{N}\mathbb{E}_{q(\beta)q(\mathbf{z}_n\,|\,\mathbf{x}_n,\beta)}\left[\log\frac{p(\mathbf{x}_n,\mathbf{z}_n\,|\,\beta)}{q(\mathbf{x}_n,\mathbf{z}_n\,|\,\beta)}\right]. \tag{5.4}$$

(Here the proportionality symbol means equality up to additive constants.) Thus the ELBO is a function of the ratio of two intractable densities. If we can form an estimator of this ratio, we can proceed with optimizing the ELBO.

We apply techniques for ratio estimation (Sugiyama et al., 2012). It is a key idea in GANS (Mohamed and Lakshminarayanan, 2016; Uehara et al., 2016), and similar ideas have rearisen in statistics and physics (Gutmann et al., 2014; Cranmer et al., 2015). In particular, we use class probability estimation: given a sample from $p(\cdot)$ or $q(\cdot)$ we aim to estimate the probability that it belongs to $p(\cdot)$. We model this using $\sigma(r(\cdot;\boldsymbol{\theta}))$, where r is a parameterized function (e.g., neural network) taking sample inputs and outputting a real value; σ is the logistic function outputting the probability.

We train $r(\cdot;\boldsymbol{\theta})$ by minimizing a loss function known as a proper scoring rule (Gneiting and Raftery, 2007). For example, in experiments we use the log loss,

$$\mathcal{D}_{\text{log}} = \mathbb{E}_{p(\mathbf{x}_n,\mathbf{z}_n \,|\, \beta)}[-\log\sigma(r(\mathbf{x}_n,\mathbf{z}_n,\beta;\boldsymbol{\theta}))] + \mathbb{E}_{q(\mathbf{x}_n,\mathbf{z}_n \,|\, \beta)}[-\log(1-\sigma(r(\mathbf{x}_n,\mathbf{z}_n,\beta;\boldsymbol{\theta})))]. \quad (5.5)$$

The loss is zero if $\sigma(r(\cdot;\boldsymbol{\theta}))$ returns 1 when a sample is from $p(\cdot)$ and 0 when a sample is from $q(\cdot)$. (We also experiment with the hinge loss; see Section 5.4.) If $r(\cdot;\boldsymbol{\theta})$ is sufficiently expressive, minimizing the loss returns the optimal function (Mohamed and Lakshminarayanan, 2016),

$$r^*(\mathbf{x}_n,\mathbf{z}_n,\beta) = \log p(\mathbf{x}_n,\mathbf{z}_n \,|\, \beta) - \log q(\mathbf{x}_n,\mathbf{z}_n \,|\, \beta).$$

As we minimize Equation 5.5, we use $r(\cdot;\boldsymbol{\theta})$ as a proxy to the log ratio in Equation 5.4. Note r estimates the log ratio; it's of direct interest and more numerically stable than the ratio.

The gradient of $\mathcal{D}_{\text{log}}$ with respect to $\boldsymbol{\theta}$ is

$$\mathbb{E}_{p(\mathbf{x}_n,\mathbf{z}_n \,|\, \beta)}[\nabla_{\boldsymbol{\theta}}\log\sigma(r(\mathbf{x}_n,\mathbf{z}_n,\beta;\boldsymbol{\theta}))] + \mathbb{E}_{q(\mathbf{x}_n,\mathbf{z}_n \,|\, \beta)}[\nabla_{\boldsymbol{\theta}}\log(1-\sigma(r(\mathbf{x}_n,\mathbf{z}_n,\beta;\boldsymbol{\theta})))]. \quad (5.6)$$

We compute unbiased gradients with Monte Carlo.

5.3.3 Stochastic Gradients of the KL Objective

To optimize the ELBO, we use the ratio estimator,

$$\mathcal{L} = \mathbb{E}_{q(\beta \,|\, \mathbf{x})}[\log p(\beta) - \log q(\beta)] + \sum_{n=1}^{N} \mathbb{E}_{q(\beta \,|\, \mathbf{x})q(\mathbf{z}_n \,|\, \mathbf{x}_n,\beta)}[r(\mathbf{x}_n,\mathbf{z}_n,\beta)]. \quad (5.7)$$

All terms are now tractable. We can calculate gradients to optimize the variational family q. Below we assume the priors $p(\beta), p(\mathbf{z}_n \mid \beta)$ are differentiable. (We discuss methods to handle discrete global variables in the next section.)

We focus on reparameterizable variational approximations (Kingma and Welling, 2014b; Rezende et al., 2014). They enable sampling via a differentiable transformation T of random noise, $\delta \sim s(\cdot)$. Due to Equation 5.7, we require the global approximation $q(\beta; \boldsymbol{\lambda})$ to admit a tractable density. With reparameterization, its sample is

$$\beta = T_{\text{global}}(\boldsymbol{\delta}_{\text{global}}; \boldsymbol{\lambda}), \quad \boldsymbol{\delta}_{\text{global}} \sim s(\cdot),$$

for a choice of transformation $T_{\text{global}}(\cdot; \boldsymbol{\lambda})$ and noise $s(\cdot)$. For example, setting $s(\cdot) = \mathcal{N}(0, 1)$ and $T_{\text{global}}(\boldsymbol{\delta}_{\text{global}}) = \mu + \sigma\boldsymbol{\delta}_{\text{global}}$ induces a normal distribution $\mathcal{N}(\mu, \sigma^2)$.

Similarly for the local variables $\mathbf{z}_n$, we specify

$$\mathbf{z}_n = T_{\text{local}}(\boldsymbol{\delta}_n, \mathbf{x}_n, \beta; \boldsymbol{\phi}), \quad \boldsymbol{\delta}_n \sim s(\cdot).$$

Unlike the global approximation, the local variational density $q(\mathbf{z}_n \mid \mathbf{x}_n; \boldsymbol{\phi})$ need not be tractable: the ratio estimator relaxes this requirement. It lets us leverage implicit models not only for data but also for approximate posteriors. In practice, we also amortize computation with inference networks, sharing parameters ϕ across the per-data point approximate posteriors.

The gradient with respect to global parameters $\boldsymbol{\lambda}$ under this approximating family is

$$\nabla_{\boldsymbol{\lambda}}\mathcal{L} = \mathbb{E}_{s(\boldsymbol{\delta}_{\text{global}})}[\nabla_{\boldsymbol{\lambda}}(\log p(\beta) - \log q(\beta))] + \sum_{n=1}^{N} \mathbb{E}_{s(\boldsymbol{\delta}_{\text{global}})s_n(\boldsymbol{\delta}_n)}[\nabla_{\boldsymbol{\lambda}} r(\mathbf{x}_n, \mathbf{z}_n, \beta)]. \tag{5.8}$$

The gradient backpropagates through the local sampling $\mathbf{z}_n = T_{\text{local}}(\boldsymbol{\delta}_n, \mathbf{x}_n, \beta; \boldsymbol{\phi})$ and the global reparameterization $\beta = T_{\text{global}}(\boldsymbol{\delta}_{\text{global}}; \boldsymbol{\lambda})$. We compute unbiased gradients with Monte Carlo. The gradient with respect to local parameters $\boldsymbol{\phi}$ is

$$\nabla_{\boldsymbol{\phi}}\mathcal{L} = \sum_{n=1}^{N} \mathbb{E}_{q(\beta)s(\boldsymbol{\delta}_n)}[\nabla_{\boldsymbol{\phi}} r(\mathbf{x}_n, \mathbf{z}_n, \beta)]. \tag{5.9}$$

Algorithm 2: Likelihood-free variational inference (LFVI)

```
Input  : Model $\mathbf{x}_n, \mathbf{z}_n \sim p(\cdot \mid \beta), p(\beta)$
         Variational approximation $\mathbf{z}_n \sim q(\cdot \mid \mathbf{x}_n, \beta; \boldsymbol{\phi})$, $q(\beta \mid \mathbf{x}; \boldsymbol{\lambda})$,
         Ratio estimator $r(\cdot; \boldsymbol{\theta})$
Output : Variational parameters $\boldsymbol{\lambda}$, $\boldsymbol{\phi}$
Initialize $\boldsymbol{\theta}$, $\boldsymbol{\lambda}$, $\boldsymbol{\phi}$ randomly.
while not converged do
    Compute unbiased estimate of $\nabla_{\boldsymbol{\theta}}\mathcal{D}$ (Equation 5.6), $\nabla_{\boldsymbol{\lambda}}\mathcal{L}$ (Equation 5.8), $\nabla_{\boldsymbol{\phi}}\mathcal{L}$
      (Equation 5.9).
    Update $\boldsymbol{\theta}$, $\boldsymbol{\lambda}$, $\boldsymbol{\phi}$ using stochastic gradient descent.
end
```

where the gradient backpropagates through T_{local}.[1]

5.3.4 Algorithm

Algorithm 2 outlines the procedure. We call it *likelihood-free variational inference (LFVI)*. LFVI is black box: it applies to models in which one can simulate data and local variables, and calculate densities for the global variables. LFVI first updates $\boldsymbol{\theta}$ to improve the ratio estimator r. Then it uses r to update parameters $\{\boldsymbol{\lambda}, \boldsymbol{\phi}\}$ of the variational approximation q. We optimize r and q simultaneously. The algorithm is available in Edward (Tran et al., 2016a).

LFVI is scalable: we can unbiasedly estimate the gradient over the full data set with mini-batches (Hoffman et al., 2013). The algorithm can also handle models of either continuous or discrete data. The requirement for differentiable global variables and reparameterizable global approximations can be relaxed using score function gradients (Ranganath et al., 2014).

Point estimates of the global parameters β suffice for many applications (Goodfellow et al., 2014; Rezende et al., 2014). Algorithm 2 can find point estimates: place a point mass approximation q on the parameters β. This simplifies gradients and corresponds to variational EM.

[1]The ratio r indirectly depends on $\boldsymbol{\phi}$ but its gradient w.r.t. $\boldsymbol{\phi}$ disappears. This is derived via the score function identity and the product rule (see, e.g., Ranganath et al. (2014, Appendix)).

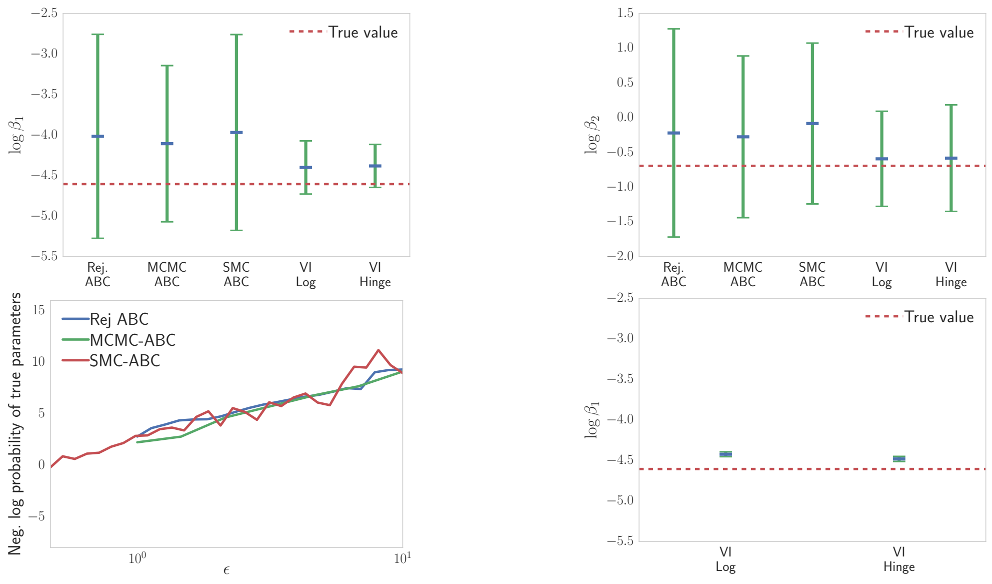


Figure 5.2: (top) Marginal posterior for first two parameters. **(bot. left)** ABC methods over tolerance error. **(bot. right)** Marginal posterior for first parameter on a large-scale data set. Our inference achieves more accurate results and scales to massive data.

5.4 Experiments

We developed new models and inference. For experiments, we study three applications: a large-scale physical simulator for predator-prey populations in ecology; a Bayesian GAN for supervised classification; and a deep implicit model for symbol generation. In addition, Appendix F, provides practical advice on how to address the stability of the ratio estimator by analyzing a toy experiment. We initialize parameters from a standard normal and apply gradient descent with ADAM.

Lotka-Volterra Predator-Prey Simulator. We analyze the Lotka-Volterra simulator of Section 5.2 and follow the same setup and hyperparameters of Papamakarios and Murray (2016). Its global variables β govern rates of change in a simulation of predator-prey populations. To infer them, we posit a mean-field normal approximation (reparameterized to be on the same support) and run Algorithm 2 with both a log loss and hinge loss for the ratio estimation problem; Appendix D details the hinge loss. We compare to rejection ABC, MCMC-ABC, and SMC-ABC (Marin et al., 2012). MCMC-ABC uses a spherical Gaussian proposal; SMC-ABC is manually tuned with a decaying epsilon schedule; all ABC methods are tuned to use the best performing hyperparameters such as the tolerance error.

Figure 5.2 displays results on two data sets. In the top figures and bottom left, we analyze data

	Test Set Error			
Model + Inference	Crabs	Pima	Covertype	MNIST
Bayesian GAN + VI	0.03	**0.232**	**0.154**	**0.0136**
Bayesian GAN + MAP	0.12	0.240	0.185	0.0283
Bayesian NN + VI	**0.02**	0.242	0.164	0.0311
Bayesian NN + MAP	0.05	0.320	0.188	0.0623

Table 5.1: Classification accuracy of Bayesian GAN and Bayesian neural networks across small to medium-size data sets. Bayesian GANs achieve comparable or better performance to their Bayesian neural net counterpart.

consisting of a simulation for $T = 30$ time steps, with recorded values of the populations every 0.2 time units. The bottom left figure calculates the negative log probability of the true parameters over the tolerance error for ABC methods; smaller tolerances result in more accuracy but slower runtime. The top figures compare the marginal posteriors for two parameters using the smallest tolerance for the ABC methods. Rejection ABC, MCMC-ABC, and SMC-ABC all contain the true parameters in their 95% credible interval but are less confident than our methods. Further, they required $100,000$ simulations from the model, with an acceptance rate of 0.004% and 2.990% for rejection ABC and MCMC-ABC respectively.

The bottom right figure analyzes data consisting of $100,000$ time series, each of the same size as the single time series analyzed in the previous figures. This size is not possible with traditional methods. Further, we see that with our methods, the posterior concentrates near the truth. We also experienced little difference in accuracy between using the log loss or the hinge loss for ratio estimation.

Bayesian Generative Adversarial Networks. We analyze Bayesian GANs, described in Section 5.2. Mimicking a use case of Bayesian neural networks (Blundell et al., 2015; Hernández-Lobato et al., 2016), we apply Bayesian GANs for classification on small to medium-size data. The GAN defines a conditional $p(y_n \mid \mathbf{x}_n)$, taking a feature $\mathbf{x}_n \in \mathbb{R}^D$ as input and generating a label $y_n \in \{1, \ldots, K\}$, via the process

$$y_n = g(\mathbf{x}_n, \boldsymbol{\epsilon}_n \mid \boldsymbol{\theta}), \qquad \boldsymbol{\epsilon}_n \sim \mathcal{N}(0, 1), \tag{5.10}$$

where $g(\cdot \mid \boldsymbol{\theta})$ is a 2-layer multilayer perception with ReLU activations, batch normalization, and is parameterized by weights and biases $\boldsymbol{\theta}$. We place normal priors, $\boldsymbol{\theta} \sim \mathcal{N}(0, 1)$.

We analyze two choices of the variational model: one with a mean-field normal approximation for

$q(\boldsymbol{\theta} \mid \mathbf{x})$, and another with a point mass approximation (equivalent to maximum a posteriori). We compare to a Bayesian neural network, which uses the same generative process as Equation 5.10 but draws from a Categorical distribution rather than feeding noise into the neural net. We fit it separately using a mean-field normal approximation and maximum a posteriori. Table 5.1 shows that Bayesian GANS generally outperform their Bayesian neural net counterpart.

Note that Bayesian GANS can analyze discrete data such as in generating a classification label. Traditional GANS for discrete data is an open challenge (Kusner and Hernández-Lobato, 2016). In Appendix E, we compare Bayesian GANS with point estimation to typical GANS. Bayesian GANS are also able to leverage parameter uncertainty for analyzing these small to medium-size data sets.

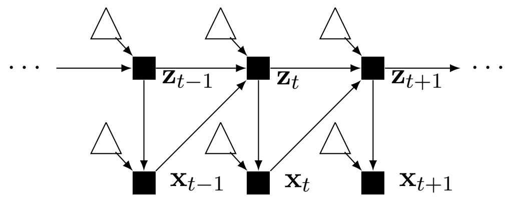
One problem with Bayesian GANS is that they cannot work with very large neural networks: the ratio estimator is a function of global parameters, and thus the input size grows with the size of the neural network. One approach is to make the ratio estimator not a function of the global parameters. Instead of optimizing model parameters via variational EM, we can train the model parameters by backpropagating through the ratio objective instead of the variational objective. An alternative is to use the hidden units as input which is much lower dimensional (Tran and Blei, 2017, Appendix C).

Injecting Noise into Hidden Units. In this section, we show how to build a hierarchical implicit model by simply injecting randomness into hidden units. We model sequences $\mathbf{x} = (\mathbf{x}_1, \dots, \mathbf{x}_T)$ with a recurrent neural network. For $t = 1, \dots, T$,

$$\begin{aligned} \mathbf{z}_t &= g_z(\mathbf{x}_{t-1}, \mathbf{z}_{t-1}, \boldsymbol{\epsilon}_{t,z}), && \boldsymbol{\epsilon}_{t,z} \sim \mathcal{N}(0,1), \\ \mathbf{x}_t &= g_x(\mathbf{z}_t, \boldsymbol{\epsilon}_{t,x}), && \boldsymbol{\epsilon}_{t,x} \sim \mathcal{N}(0,1), \end{aligned}$$

where g_z and g_x are both 1-layer multilayer perceptions with ReLU activation and layer normalization. We place standard normal priors over all weights and biases. See Figure 5.3a.

If the injected noise $\boldsymbol{\epsilon}_{t,z}$ combines linearly with the output of g_z, the induced distribution $p(\mathbf{z}_t \mid \mathbf{x}_{t-1}, \mathbf{z}_{t-1})$ is Gaussian parameterized by that output. This defines a stochastic RNN (Bayer and Osendorfer, 2014; Fraccaro et al., 2016), which generalizes its deterministic connection. With nonlinear combinations, the implicit density is more flexible (and intractable), making previous methods for inference not applicable. In our method, we perform variational inference and specify q to be implicit; we use the



(a) A deep implicit model for sequences. It is a RNN with noise injected into each hidden state. The hidden state is now an implicit latent variable. The same occurs for generating outputs.

```
−x+x/x**x*//x*x+
x/x*x+x*x/x+x+x+
/+x*x+x*x/x/x+x+
/x+*x+x*x/x+x−x+
x/x*x/x*x+x+x+x−
x+x+x/x*x*x+x/x+
```

(b) Generated symbols from the implicit model. Good samples place arithmetic operators between the variable x. The implicit model learned to follow rules from the context free grammar up to some multiple operator repeats.

same architecture as the probability model's implicit priors.

We follow the same setup and hyperparameters as Kusner and Hernández-Lobato (2016) and generate simple one-variable arithmetic sequences following a context free grammar,

$$S \rightarrow x \| S + S \| S - S \| S * S \| S / S,$$

where $\|$ divides possible productions of the grammar. We concatenate the inputs and point estimate the global variables (model parameters) using variational EM. Figure 5.3b displays samples from the inferred model, training on sequences with a maximum of 15 symbols. It achieves sequences which roughly follow the context free grammar.

5.5 Discussion

We developed a class of hierarchical implicit models and likelihood-free variational inference, merging the idea of implicit densities with hierarchical Bayesian modeling and approximate posterior inference. This expands Bayesian analysis with the ability to apply neural samplers, physical simulators, and their combination with rich, interpretable latent structure.

More stable inference with ratio estimation is an open challenge. This is especially important when we analyze large-scale real world applications of implicit models. Recent work for genomics offers a promising solution (Tran and Blei, 2017).

Chapter 5: Bibliography

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y., and Zhang, X. (2016). TensorFlow: A system for large-scale machine learning. *arXiv.org*.

Agakov, F. V. and Barber, D. (2004). An auxiliary variational method. In *Neural Information Processing*, pages 561–566. Springer.

Amos, B. and Kolter, J. Z. (2017). OptNet: Differentiable optimization as a layer in neural networks. In *International Conference on Machine Learning*.

Andrieu, C. and Roberts, G. O. (2009). The pseudo-marginal approach for efficient monte carlo computations. *The Annals of Statistics*, pages 697–725.

Andrychowicz, M., Denil, M., Gomez, S., Hoffman, M. W., Pfau, D., Schaul, T., and de Freitas, N. (2016). Learning to learn by gradient descent by gradient descent. In *Neural Information Processing Systems*.

Anelli, G., Antchev, G., Aspell, P., Avati, V., Bagliesi, M., Berardi, V., Berretti, M., Boccone, V., Bottigli, U., Bozzo, M., et al. (2008). The totem experiment at the CERN large Hadron collider. *Journal of Instrumentation*, 3(08):S08007.

Bahdanau, D., Cho, K., and Bengio, Y. (2015). Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*.

Baydin, A. G., Pearlmutter, B. A., Radul, A. A., and Siskind, J. M. (2015). Automatic differentiation in machine learning: a survey. *arXiv preprint arXiv:1502.05767*.

Bayer, J. and Osendorfer, C. (2014). Learning stochastic recurrent networks. *arXiv preprint arXiv:1411.7610*.

Baylor, D., Breck, E., Cheng, H.-T., Fiedel, N., Foo, C. Y., Haque, Z., Haykal, S., Ispir, M., Jain, V., Koc, L., et al. (2017). TFX: A TensorFlow-based production-scale machine learning platform. In *Knowledge Discovery and Data Mining*.

Beaumont, M. A. (2010). Approximate Bayesian computation in evolution and ecology. *Annual Review of Ecology, Evolution and Systematics*, 41(379-406):1.

Becker, R. A. and Chambers, J. M. (1984). *S: an interactive environment for data analysis and graphics*. CRC Press.

Bengio, E., Bacon, P.-L., Pineau, J., and Precup, D. (2015). Conditional computation in neural networks for faster models. *arXiv preprint arXiv:1511.06297*.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation Learning: A Review and New Perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828.

Bergstra, J., Breuleux, O., Bastien, F., Lamblin, P., Pascanu, R., Desjardins, G., Turian, J., Warde-Farley, D., and Bengio, Y. (2010). Theano: a CPU and GPU math expression compiler. In *Proceedings of the Python for Scientific Computing Conference (SciPy)*.

Binder, J., Murphy, K., and Russell, S. (1997). Space-efficient inference in dynamic probabilistic networks. In *International Joint Conference on Artificial Intelligence*.

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P., Horsfall, P., and Goodman, N. D. (2018). Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538*.

Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer.

Bishop, C. M., Lawrence, N. D., Jordan, M. I., and Jaakkola, T. (1998). Approximating posterior distributions in belief networks using mixtures. In *Neural Information Processing Systems*.

Blei, D. M. (2014). Build, compute, critique, repeat: Data analysis with latent variable models. *Annual Review of Statistics and Its Application*.

Blei, D. M., Ng, A. Y., and Jordan, M. I. (2003). Latent Dirichlet Allocation. *Journal of Machine Learning Research*, 3:993–1022.

Blundell, C., Cornebise, J., Kavukcuoglu, K., and Wierstra, D. (2015). Weight uncertainty in neural networks. In *International Conference on Machine Learning*.

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

Box, G. E. (1980). Sampling and Bayes' inference in scientific modelling and robustness. *Journal of the Royal Statistical Society. Series A (General)*, pages 383–430.

Box, G. E. (1982). An apology for ecumenism in statistics. Technical report, DTIC Document.

Box, G. E. P. (1976). Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799.

Broderick, T., Boyd, N., Wibisono, A., Wilson, A. C., and Jordan, M. I. (2013). Streaming variational Bayes. In *Neural Information Processing Systems*.

Buchanan, B., Sutherland, G., and Feigenbaum, E. A. (1969). *Heuristic DENDRAL: a program for generating explanatory hypotheses in organic chemistry*. American Elsevier.

Burda, Y., Grosse, R., and Salakhutdinov, R. (2016a). Importance weighted autoencoders. In *International Conference on Learning Representations*.

Burda, Y., Grosse, R., and Salakhutdinov, R. (2016b). Importance weighted autoencoders. In *International Conference on Learning Representations*.

Carpenter, B., Gelman, A., Hoffman, M. D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P., and Riddell, A. (2016). Stan: a probabilistic programming language. *Journal of Statistical Software*.

Carpenter, B., Hoffman, M. D., Brubaker, M., Lee, D., Li, P., and Betancourt, M. (2015). The Stan Math Library: Reverse-Mode Automatic Differentiation in C++. *arXiv.org*.

Chambers, J. M. and Hastie, T. J. (1992). *Statistical Models in S*. Chapman & Hall, London.

Chollet, F. (2015). Keras. https://github.com/fchollet/keras.

Cranmer, K., Pavez, J., and Louppe, G. (2015). Approximating likelihood ratios with calibrated discriminative classifiers. *arXiv preprint arXiv:1506.02169*.

Culler, D. E. (1986). Dataflow architectures. Technical report, DTIC Document.

Cunningham, J. P., Shenoy, K. V., and Sahani, M. (2008). Fast Gaussian process methods for point process intensity estimation. In *International Conference on Machine Learning*. ACM.

Cusumano-Towner, M. F. and Mansinghka, V. K. (2018). Using probabilistic programs as proposals. In *POPL Workshop*.

Dayan, P. and Abbott, L. F. (2001). *Theoretical neuroscience*, volume 10. Cambridge, MA: MIT Press.

Dayan, P., Hinton, G. E., Neal, R. M., and Zemel, R. S. (1995). The helmholtz machine. *Neural computation*, 7(5):889–904.

Denton, E. L., Chintala, S., Fergus, R., et al. (2015). Deep generative image models using a Laplacian pyramid of adversarial networks. In *Neural Information Processing Systems*.

Diaconis, P., Holmes, S., and Montgomery, R. (2007). Dynamical bias in the coin toss. *SIAM*, 49(2):211–235.

Dieng, A. B., Tran, D., Ranganath, R., Paisley, J., and Blei, D. M. (2016). χ-divergence for approximate inference. In *arXiv preprint arXiv:1611.00328*.

Diggle, P. J. and Gratton, R. J. (1984). Monte Carlo methods of inference for implicit statistical models. *Journal of the Royal Statistical Society: Series B (Methodological)*, pages 193–227.

Dillon, J., Langmore, I., Tran, D., Brevdo, E., Vasudevan, S., Moore, D., Patton, B., Alemi, A., Hoffman, M., and Saurous, R. (2017). TensorFlow Distributions.

Donahue, J., Krähenbühl, P., and Darrell, T. (2017). Adversarial Feature Learning. In *International Conference on Learning Representations*.

Doucet, A., De Freitas, N., and Gordon, N. (2001). An introduction to sequential monte carlo methods. In *Sequential Monte Carlo methods in practice*, pages 3–14. Springer.

Doucet, A., Godsill, S., and Andrieu, C. (2000). On sequential Monte Carlo sampling methods for Bayesian filtering. *Statistics and Computing*, 10(3):197–208.

Dziugaite, G. K., Roy, D. M., and Ghahramani, Z. (2015). Training generative neural networks via maximum mean discrepancy optimization. In *Uncertainty in Artificial Intelligence*.

Eitz, M., Hays, J., and Alexa, M. (2012). How do humans sketch objects? *ACM Trans. Graph. (Proc. SIGGRAPH)*, 31(4):44:1–44:10.

Fisher, R. A. (1925). Theory of statistical estimation. *Mathematical Proceedings of the Cambridge Philosophical Society*, 22(5).

Foti, N., Xu, J., Laird, D., and Fox, E. (2014). Stochastic variational inference for hidden Markov models. In *Neural Information Processing Systems*.

Fraccaro, M., Sønderby, S. K., Paquet, U., and Winther, O. (2016). Sequential neural models with stochastic layers. In *Neural Information Processing Systems*.

Friedman, J., Hastie, T., and Tibshirani, R. (2001). *The elements of statistical learning*, volume 1. Springer series in statistics Springer, Berlin.

Friedman, N., Linial, M., Nachman, I., and Pe'er, D. (2000). Using bayesian networks to analyze expression data. *Journal of computational biology*, 7(3-4):601–620.

Ge, H., Xu, K., Scibior, A., Ghahramani, Z., et al. (2018). The Turing language for probabilistic programming. In *Artificial Intelligence and Statistics*.

Gelfand, A. E. and Smith, A. F. (1990). Sampling-based approaches to calculating marginal densities. *Journal of the American statistical association*, 85(410):398–409.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013). *Bayesian data analysis*. Texts in Statistical Science Series. CRC Press, Boca Raton, FL, third edition.

Gelman, A. and Hill, J. L. (2006). *Data analysis using regression and multilevel/hierarchical models*. Cambridge University Press.

Gelman, A., Meng, X.-L., and Stern, H. (1996). Posterior predictive assessment of model fitness via realized discrepancies. *Statistica sinica*, pages 733–760.

Gelman, A. and Shalizi, C. R. (2013). Philosophy and the practice of bayesian statistics. *British Journal of Mathematical and Statistical Psychology*, 66(1):8–38.

Ghahramani, Z. (2015). Probabilistic machine learning and artificial intelligence. *Nature*, 521(7553):452–459.

Giles, C. L., Sun, G.-Z., Chen, H.-H., Lee, Y.-C., and Chen, D. (1990). Higher order recurrent networks and grammatical inference. In *Neural Information Processing Systems*.

Gneiting, T. and Raftery, A. E. (2007). Strictly proper scoring rules, prediction, and estimation. *Journal of the American Statistical Association*, 102(477):359–378.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. Book in preparation for MIT Press.

Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. In *Neural Information Processing Systems*.

Goodfellow, I. J. (2014). On distinguishability criteria for estimating generative models. In *ICLR Workshop*.

Goodman, N., Mansinghka, V., Roy, D. M., Bonawitz, K., and Tenenbaum, J. B. (2012). Church: a language for generative models. In *Uncertainty in Artificial Intelligence*.

Graves, A. (2016). Adaptive computation time for recurrent neural networks. *arXiv preprint arXiv:1603.08983*.

Graves, A., Wayne, G., and Danihelka, I. (2014). Neural turing machines. *arXiv preprint arxiv:1410.5401*.

Gregor, K., Danihelka, I., Graves, A., Rezende, D. J., and Wierstra, D. (2015). DRAW: A recurrent neural network for image generation. In *International Conference on Machine Learning*.

Gregor, K., Danihelka, I., Mnih, A., Blundell, C., and Wierstra, D. (2014). Deep autoregressive networks. In *International Conference on Machine Learning*.

Gutmann, M. and Hyvärinen, A. (2010). Noise-contrastive estimation: A new estimation principle for unnormalized statistical models. In *Artificial Intelligence and Statistics*.

Gutmann, M. U., Dutta, R., Kaski, S., and Corander, J. (2014). Statistical Inference of Intractable Generative Models via Classification. *arXiv preprint arXiv:1407.4981*.

Hafner, D., Tran, D., Irpan, A., Lillicrap, T., and Davidson, J. (2018). Reliable uncertainty estimates in deep neural networks using noise contrastive priors. *arXiv preprint*.

Hartig, F., Calabrese, J. M., Reineking, B., Wiegand, T., and Huth, A. (2011). Statistical inference for stochastic simulation models–theory and application. *Ecology Letters*, 14(8):816–827.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Computer Vision and Pattern Recognition*.

Hernández-Lobato, J. M., Li, Y., Rowland, M., Hernández-Lobato, D., Bui, T., and Turner, R. E. (2016). Black-box α-divergence minimization. In *International Conference on Machine Learning*.

Hinton, G. E. (2002). Training products of experts by minimizing contrastive divergence. *Neural Computation*, 14(8):1771–1800.

Hinton, G. E., Osindero, S., and Teh, Y. W. (2006). A Fast Learning Algorithm for Deep Belief Nets. *Neural Computation*, 18:1527–1554.

Hinton, G. E. and van Camp, D. (1993). Keeping the neural networks simple by minimizing the description length of the weights. In *Conference on Learning Theory*, pages 5–13, New York, New York, USA. ACM.

Hinton, G. E. and Zemel, R. S. (1994). Autoencoders, minimum description length, and helmholtz free energy. *Advances in neural information processing systems*, pages 3–3.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Hochreiter, S., Younger, A. S., and Conwell, P. R. (2001). Learning to learn using gradient descent. In *International Conference on Artificial Neural Networks*, pages 87–94.

Hoffman, M. and Blei, D. (2015). Stochastic structured variational inference. In *Artificial Intelligence and Statistics*, pages 361–369.

Hoffman, M. D. (2017). Learning deep latent Gaussian models with Markov chain Monte Carlo. In *International Conference on Machine Learning*.

Hoffman, M. D., Blei, D. M., Wang, C., and Paisley, J. (2013). Stochastic variational inference. *The Journal of Machine Learning Research*, 14(1):1303–1347.

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

Hopfield, J. J. (1982). Neural networks and physical systems with emergent collective computational abilities. *Proceedings of the national academy of sciences*, 79(8):2554–2558.

Ihaka, R. and Gentleman, R. (1996). R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics*, 5(3):299–314.

Johnson, M. and Willsky, A. S. (2014). Stochastic variational inference for Bayesian time series models. In *International Conference on Machine Learning*.

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., and Saul, L. K. (1999a). An Introduction to Variational Methods for Graphical Models. *Machine Learning*, pages 1–51.

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., and Saul, L. K. (1999b). An introduction to variational methods for graphical models. *Machine Learning*.

Jouppi, N. P., Young, C., Patil, N., Patterson, D., Agrawal, G., Bajwa, R., Bates, S., Bhatia, S., Boden, N., Borchers, A., et al. (2017). In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*.

Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2018). Progressive growing of gans for improved quality, stability, and variation. In *International Conference on Learning Representations*.

Keller, J. B. (1986). The probability of heads. *The American Mathematical Monthly*, 93(3):191–197.

Kingma, D. and Welling, M. (2014a). Auto-encoding variational Bayes. In *International Conference on Learning Representations*.

Kingma, D. P., Mohamed, S., Rezende, D. J., and Welling, M. (2014). Semi-supervised learning with deep generative models. In *Neural Information Processing Systems*.

Kingma, D. P., Salimans, T., and Welling, M. (2016). Improving variational inference with inverse autoregressive flow. In *Neural Information Processing Systems*.

Kingma, D. P. and Welling, M. (2014b). Auto-encoding variational Bayes. In *International Conference on Learning Representations*.

Koller, D. and Friedman, N. (2009). *Probabilistic graphical models: principles and techniques*. MIT press.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In *Neural Information Processing Systems*, pages 1097–1105.

Kucukelbir, A., Tran, D., Ranganath, R., Gelman, A., and Blei, D. M. (2017). Automatic differentiation variational inference. *Journal of Machine Learning Research*, 18(14):1–45.

Kusner, M. J. and Hernández-Lobato, J. M. (2016). GANs for sequences of discrete elements with the Gumbel-Softmax distribution. In *NIPS Workshop*.

Kusner, M. J., Paige, B., and Hernández-Lobato, J. M. (2017). Grammar variational autoencoder. In *International Conference on Machine Learning*.

Lake, B. M., Ullman, T. D., Tenenbaum, J. B., and Gershman, S. J. (2016). Building Machines That Learn and Think Like People. *arXiv.org*.

Laplace, P. S. (1986). Memoir on the probability of the causes of events. *Statistical Science*, 1(3):364–378.

Lawrence, N. (2000). *Variational Inference in Probabilistic Models*. PhD thesis.

Lawrence, N. (2005). Probabilistic non-linear principal component analysis with Gaussian process latent variable models. *The Journal of Machine Learning Research*, 6:1783–1816.

Li, Y., Swersky, K., and Zemel, R. (2015). Generative moment matching networks. In *International Conference on Machine Learning*.

Li, Y. and Turner, R. E. (2016). Variational inference with Rényi divergence. In *Neural Information Processing Systems*.

MacKay, D. J. (2003). *Information theory, inference and learning algorithms*. Cambridge university press.

MacKay, D. J. C. (1992). *Bayesian methods for adaptive models*. PhD thesis, California Institute of Technology.

Maclaurin, D., Duvenaud, D., Johnson, M., and Adams, R. P. (2015). Autograd: Reverse-mode differentiation of native Python.

Manning, C. D. and Schütze, H. (1999). *Foundations of statistical natural language processing*, volume 999. MIT Press.

Mansinghka, V., Selsam, D., and Perov, Y. (2014). Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv.org*.

Marin, J.-M., Pudlo, P., Robert, C. P., and Ryder, R. J. (2012). Approximate Bayesian computational methods. *Statistics and Computing*, 22(6):1167–1180.

McInerney, J., Ranganath, R., and Blei, D. M. (2015). The Population Posterior and Bayesian Inference on Streams. In *Neural Information Processing Systems*.

Meng, X.-L. (1994). Posterior predictive p-values. *The Annals of Statistics*, pages 1142–1160.

Minka, T. P. (2001). Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth conference on Uncertainty in artificial intelligence*, pages 362–369. Morgan Kaufmann Publishers Inc.

Minsky, M. (1975). A framework for representing knowledge.

Mohamed, S. and Lakshminarayanan, B. (2016). Learning in Implicit Generative Models. *arXiv preprint arXiv:1610.03483*.

Murphy, K. P. (2012). *Machine learning: a probabilistic perspective*. MIT press.

Murray, I. and Salakhutdinov, R. R. (2009). Evaluating probabilities under high-dimensional latent variable models. In *Advances in neural information processing systems*, pages 1137–1144.

Neal, R. M. (1990). Learning Stochastic Feedforward Networks. Technical report.

Neal, R. M. (1994). *Bayesian Learning for Neural Networks*. PhD thesis, University of Toronto.

Neal, R. M. (2011). MCMC using Hamiltonian dynamics. *Handbook of Markov Chain Monte Carlo*.

Neal, R. M. and Hinton, G. E. (1993). A new view of the em algorithm that justifies incremental and other variants. In *Learning in Graphical Models*, pages 355–368.

Nelsen, R. B. (2006). *An Introduction to Copulas (Springer Series in Statistics)*. Springer-Verlag New York, Inc.

Newell, A. and Simon, H. A. (1976). Computer science as empirical inquiry: Symbols and search. *Communications of the ACM*, 19(3):113–126.

Oord, A. v. d., Kalchbrenner, N., and Kavukcuoglu, K. (2016). Pixel recurrent neural networks. *arXiv preprint arXiv:1601.06759*.

Osborne, M. (2010). *Bayesian Gaussian processes for sequential prediction, optimisation and quadrature*. PhD thesis, Oxford University New College.

Paisley, J., Blei, D. M., and Jordan, M. (2012a). Variational Bayesian inference with stochastic search. In *International Conference on Machine Learning*.

Paisley, J., Blei, D. M., and Jordan, M. I. (2012b). Variational bayesian inference with stochastic search. In *International Conference on Machine Learning*.

Papamakarios, G. and Murray, I. (2016). Fast ϵ-free inference of simulation models with Bayesian conditional density estimation. In *Neural Information Processing Systems*.

Papamakarios, G., Murray, I., and Pavlakou, T. (2017). Masked autoregressive flow for density estimation. In *Advances in Neural Information Processing Systems*, pages 2335–2344.

Parmar, N., Vaswani, A., Uszkoreit, J., Kaiser, L., Shazeer, N., Ku, A., and Tran, D. (2018). Image Transformer. In *International Conference on Machine Learning*.

Pearl, J. (2003). Causality: models, reasoning, and inference. *Econometric Theory*, 19(675-685):46.

Pfeffer, A. (2007). The design and implementation of IBAL: A general-purpose probabilistic language. *Introduction to Statistical Relational Learning*, page 399.

Pritchard, J. K., Seielstad, M. T., Perez-Lezaun, A., and Feldman, M. W. (1999). Population growth of human Y chromosomes: a study of Y chromosome microsatellites. *Molecular Biology and Evolution*, 16(12):1791–1798.

Probtorch Developers (2017). Probtorch. https://github.com/probtorch/probtorch.

Quiñonero-Candela, J. and Rasmussen, C. E. (2005). A unifying view of sparse approximate Gaussian process regression. *Journal of Machine Learning Research*, 6:1939–1959.

Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks. In *International Conference on Learning Representations*.

Raiko, T., Li, Y., Cho, K., and Bengio, Y. (2014). Iterative neural autoregressive distribution estimator nade-k. In *Advances in Neural Information Processing Systems*, pages 325–333.

Ranganath, R., Altosaar, J., Tran, D., and Blei, D. M. (2016a). Operator variational inference. In *Neural Information Processing Systems*.

Ranganath, R., Gerrish, S., and Blei, D. M. (2014). Black Box Variational Inference. In *Artificial Intelligence and Statistics*.

Ranganath, R., Tang, L., Charlin, L., and Blei, D. M. (2015a). Deep exponential families. In *Artificial Intelligence and Statistics*.

Ranganath, R., Tran, D., and Blei, D. M. (2015b). Hierarchical variational models. *arXiv preprint arXiv:1511.02386*.

Ranganath, R., Tran, D., and Blei, D. M. (2016b). Hierarchical variational models. In *International Conference on Machine Learning*.

Rasmussen, C. E. and Williams, C. (2006). *Gaussian Processes for Machine Learning*. MIT Press.

Rezende, D. J. and Mohamed, S. (2015). Variational inference with normalizing flows. In *International Conference on Machine Learning*.

Rezende, D. J., Mohamed, S., Danihelka, I., Gregor, K., and Wierstra, D. (2016). One-shot generalization in deep generative models. In *International Conference on Machine Learning*.

Rezende, D. J., Mohamed, S., and Wierstra, D. (2014). Stochastic Backpropagation and Approximate Inference in Deep Generative Models. In *International Conference on Machine Learning*.

Ritchie, D., Horsfall, P., and Goodman, N. D. (2016). Deep amortized inference for probabilistic programs. *arXiv preprint arXiv:1610.05735*.

Robbins, H. and Monro, S. (1951). A stochastic approximation method. *The Annals of Mathematical Statistics*.

Robert, C. P. and Casella, G. (1999). *Monte Carlo statistical methods*. Springer.

Roy, A., Vaswani, A., Neelakantan, A., and Parmar, N. (2018). Theory and experiments on vector quantized autoencoders. *arXiv preprint arXiv:1805.11063*.

Rubin, D. B. (1984). Bayesianly justifiable and relevant frequency calculations for the applied statistician. *The Annals of Statistics*, 12(4):1151–1172.

Rudolph, M. R., Ruiz, F. J. R., Mandt, S., and Blei, D. M. (2016). Exponential Family Embeddings. In *Neural Information Processing Systems*.

Rumelhart, D. E., McClelland, J. L., Group, P. R., et al. (1988). *Parallel distributed processing*, volume 1. IEEE.

Salakhutdinov, R. and Larochelle, H. (2010). Efficient learning of deep boltzmann machines. In *International Conference on Artificial Intelligence and Statistics*.

Salakhutdinov, R. and Mnih, A. (2008). Bayesian probabilistic matrix factorization using Markov chain Monte Carlo. In *International Conference on Machine Learning*, pages 880–887. ACM.

Salakhutdinov, R. and Murray, I. (2008). On the quantitative analysis of deep belief networks. In *International Conference on Machine Learning*.

Salimans, T., Kingma, D., and Welling, M. (2015). Markov chain Monte Carlo and variational inference: Bridging the gap. In *International Conference on Machine Learning*.

Salvatier, J., Wiecki, T., and Fonnesbeck, C. (2015). Probabilistic Programming in Python using PyMC. *arXiv preprint arXiv:1507.08050*.

Salvatier, J., Wiecki, T. V., and Fonnesbeck, C. (2016). Probabilistic programming in Python using PyMC3. *PeerJ Computer Science*, 2:e55.

Saul, L. K. and Jordan, M. I. (1996). Exploiting tractable substructures in intractable networks. In *Neural Information Processing Systems*.

Ścibior, A., Ghahramani, Z., and Gordon, A. D. (2015). Practical probabilistic programming with monads. In *the 8th ACM SIGPLAN Symposium*, pages 165–176, New York, New York, USA. ACM Press.

Shazeer, N., Cheng, Y., Parmar, N., Tran, D., Vaswani, A., Koanantakool, P., Hawkins, P., Lee, H., Hong, M., Young, C., Sepassi, R., and Hechtman, B. (2018). Mesh-TensorFlow: Deep learning for supercomputers. In *Neural Information Processing Systems*.

Shazeer, N., Mirhoseini, A., Maziarz, K., Davis, A., Le, Q., Hinton, G., and Dean, J. (2017). Outrageously large neural networks: The sparsely-gated mixture-of-experts layer. *arXiv preprint arXiv:1701.06538*.

Shi, J., Chen, J., Zhu, J., Sun, S., Luo, Y., Gu, Y., and Zhou, Y. (2017). Zhusuan: A library for bayesian deep learning. *arXiv preprint arXiv:1709.05870*.

Sønderby, C. K., Raiko, T., Maaløe, L., Sønderby, S. K., and Winther, O. (2016). Ladder variational autoencoders. In *Neural Information Processing Systems*.

Spiegelhalter, D. J., Thomas, A., Best, N. G., and Gilks, W. R. (1995). BUGS: Bayesian inference using Gibbs sampling, version 0.50. *MRC Biostatistics Unit, Cambridge*.

Sugiyama, M., Suzuki, T., and Kanamori, T. (2012). Density-ratio matching under the Bregman divergence: a unified framework of density-ratio estimation. *Annals of the Institute of Statistical . . . .*

Teh, Y. W. and Jordan, M. I. (2010). Hierarchical Bayesian nonparametric models with applications. *Bayesian Nonparametrics*, 1.

Tenenbaum, J. B., Kemp, C., Griffiths, T. L., and Goodman, N. D. (2011). How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022):1279–1285.

Tieleman, T. and Hinton, G. (2012). Lecture 6.5—RmsProp: Divide the gradient by a running average of its recent magnitude. COURSERA: Neural Networks for Machine Learning.

Tipping, M. E. and Bishop, C. M. (1999). Probabilistic principal component analysis. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 61(3):611–622.

Tolpin, D., van de Meent, J.-W., Yang, H., and Wood, F. (2016). Design and implementation of probabilistic programming language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages*, page 6.

Tomczak, J. M. and Welling, M. (2018). Vae with a vampprior. In *Artificial Intelligence and Statistics*.

Tran, D. and Blei, D. M. (2017). Implicit causal models for genome-wide association studies. *arXiv preprint arXiv:1710.10742*.

Tran, D. and Blei, D. M. (2018). Implicit causal models for genome-wide association studies. In *International Conference on Learning Representations*.

Tran, D., Blei, D. M., and Airoldi, E. M. (2015). Copula variational inference. In *Neural Information Processing Systems*.

Tran, D., Hoffman, M. D., Saurous, R. A., Brevdo, E., Murphy, K., and Blei, D. M. (2017). Deep probabilistic programming. In *International Conference on Learning Representations*.

Tran, D., Kucukelbir, A., Dieng, A. B., Rudolph, M., Liang, D., and Blei, D. M. (2016a). Edward: A library for probabilistic modeling, inference, and criticism.

Tran, D., Ranganath, R., and Blei, D. M. (2016b). The variational Gaussian process. In *International Conference on Learning Representations*.

Tristan, J.-B., Huang, D., Tassarotti, J., Pocock, A. C., Green, S., and Steele, G. L. (2014). Augur: Data-parallel probabilistic modeling. In *Neural Information Processing Systems*.

Tukey, J. W. (1962). The Future of Data Analysis. *Annals of Mathematical Statistics*, 33(1):1–67.

Uehara, M., Sato, I., Suzuki, M., Nakayama, K., and Matsuo, Y. (2016). Generative adversarial nets from a density ratio estimation perspective. *arXiv preprint arXiv:1610.02920*.

Van Der Vaart, A. and Van Zanten, H. (2011). Information rates of nonparametric Gaussian process methods. *The Journal of Machine Learning Research*, 12:2095–2119.

Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Kaiser, L., Kalchbrenner, N., Parmar, N., Sepassi, R., Shazeer, N., and Uszkoreit, J. (2018). Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416.

Wainwright, M. J. and Jordan, M. I. (2008). Graphical models, exponential families, and variational inference. *Foundations and Trends® in Machine Learning*, 1(1-2):1–305.

Wang, C. and Blei, D. M. (2012). Truncation-free online variational inference for bayesian nonparametric models. In *Neural Information Processing Systems*, pages 413–421.

Waterhouse, S., MacKay, D., Robinson, T., et al. (1996). Bayesian methods for mixtures of experts. *Advances in neural information processing systems*, pages 351–357.

Welling, M. and Teh, Y. W. (2011). Bayesian learning via stochastic gradient Langevin dynamics. In *International Conference on Machine Learning*.

Wen, Y., Vicol, P., Ba, J., Tran, D., and Grosse, R. (2018). Flipout: Efficient pseudo-independent weight perturbations on mini-batches. In *International Conference on Learning Representations*.

Wilkinson, D. J. (2011). *Stochastic modelling for systems biology*. CRC press.

Winkler, R. L. (1994). Evaluating probabilities: Asymmetric scoring rules. *Management Science*, 40(11):1395–1405.

Wu, Y., Li, L., Russell, S., and Bodik, R. (2016). Swift: Compiled inference for probabilistic programming languages. *arXiv preprint arXiv:1606.09242*.

Zinkov, R. and Shan, C.-c. (2016). Composing inference algorithms as program transformations. *arXiv preprint arXiv:1603.01882*.

Zoph, B. and Le, Q. V. (2017). Neural architecture search with reinforcement learning. In *International Conference on Learning Representations*.

Appendix A: Deep Probabilistic Programming

A.1 Model Examples

There are many examples available at `http://edwardlib.org`, including models, inference methods, and complete scripts. Below we describe several model examples; Appendix A.6 describes an inference example (stochastic variational inference); Appendix A.7 describes complete scripts. All examples in this paper are comprehensive, only leaving out import statements and fixed values. See the companion webpage for this paper (`http://edwardlib.org/iclr2017`) for examples in a machine-readable format with runnable code.

A.2 Bayesian Neural Network for Classification

A Bayesian neural network is a neural network with a prior distribution on its weights.

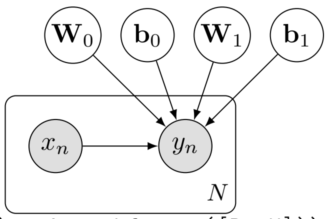
Define the likelihood of an observation $(\mathbf{x}_n, y_n)$ with binary label $y_n \in \{0, 1\}$ as

$$p(y_n \mid \mathbf{W}_0, \mathbf{b}_0, \mathbf{W}_1, \mathbf{b}_1 \ ; \ \mathbf{x}_n) = \text{Bernoulli}(y_n \mid \text{NN}(\mathbf{x}_n \ ; \ \mathbf{W}_0, \mathbf{b}_0, \mathbf{W}_1, \mathbf{b}_1)),$$

where NN is a 2-layer neural network whose weights and biases form the latent variables $\mathbf{W}_0, \mathbf{b}_0, \mathbf{W}_1, \mathbf{b}_1$. Define the prior on the weights and biases to be the standard normal. See Figure A.1. There are N data points, D features, and H hidden units.

A.3 Latent Dirichlet Allocation

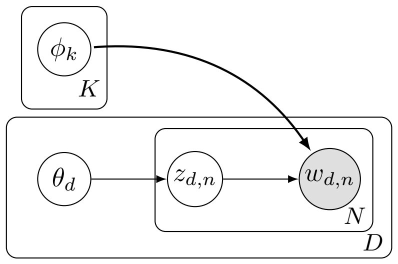
See Figure A.2. Note that the program is written for illustration. We recommend vectorization in practice: instead of storing scalar random variables in lists of lists, one should prefer to represent few random variables, each which have many dimensions.



```
W_0 = Normal(mu=tf.zeros([D, H]), sigma=tf.ones([D, H]))
W_1 = Normal(mu=tf.zeros([H, 1]), sigma=tf.ones([H, 1]))
b_0 = Normal(mu=tf.zeros(H), sigma=tf.ones(H))
b_1 = Normal(mu=tf.zeros(1), sigma=tf.ones(1))

x = tf.placeholder(tf.float32, [N, D])
y = Bernoulli(logits=tf.matmul(tf.nn.tanh(tf.matmul(x, W_0) + b_0), W_1) + b_1)
```

Figure A.1: Bayesian neural network for classification.



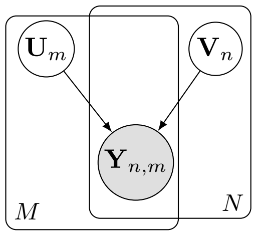
```
D = 4  # number of documents
N = [11502, 213, 1523, 1351]  # words per doc
K = 10  # number of topics
V = 100000  # vocabulary size

theta = Dirichlet(alpha=tf.zeros([D, K]) + 0.1)
phi = Dirichlet(alpha=tf.zeros([K, V]) + 0.05)
z = [[0] * N] * D
w = [[0] * N] * D
for d in range(D):
  for n in range(N[d]):
    z[d][n] = Categorical(pi=theta[d, :])
    w[d][n] = Categorical(pi=phi[z[d][n], :])
```

Figure A.2: Latent Dirichlet allocation (Blei et al., 2003).

A.4 Gaussian Matrix Factorizationn

See Figure A.3.



```
N = 10
M = 10
K = 5  # latent dimension

U = Normal(mu=tf.zeros([M, K]), sigma=tf.ones([M, K]))
V = Normal(mu=tf.zeros([N, K]), sigma=tf.ones([N, K]))
Y = Normal(mu=tf.matmul(U, V, transpose_b=True), sigma=tf.ones([N, M]))
```

Figure A.3: Gaussian matrix factorization.

A.5 Dirichlet Process Mixture Model

See Figure A.4.

A Dirichlet process mixture model is written as follows:

```
mu = DirichletProcess(alpha=0.1, base_cls=Normal, mu=tf.zeros(D), sigma=tf.ones(D), sample_n=N)
x = Normal(mu=mu, sigma=tf.ones([N, D]))
```

where `mu` has shape `(N, D)`. The `DirichletProcess` random variable returns `sample_n=N` draws, each with shape given by the base distribution `Normal(mu, sigma)`. The essential component defining the `DirichletProcess` random variable is a stochastic while loop. We define it below. See Edward's code base for a more involved version with a base distribution.

```
def dirichlet_process(alpha):
  def cond(k, beta_k):
    flip = Bernoulli(p=beta_k)
    return tf.equal(flip, tf.constant(1))

  def body(k, beta_k):
    beta_k = beta_k * Beta(a=1.0, b=alpha)
    return k + 1, beta_k

  k = tf.constant(0)
  beta_k = Beta(a=1.0, b=alpha)
  stick_num, stick_beta = tf.while_loop(cond, body, loop_vars=[k, beta_k])
  return stick_num
```

Figure A.4: Dirichlet process mixture model.

A.6 Inference Example: Stochastic Variational Inference

In the subgraph setting, we do data subsampling while working with a subgraph of the full model. This setting is necessary when the data and model do not fit in memory. It is scalable in that both the algorithm's computational complexity (per iteration) and memory complexity are independent of the data set size.

For the code, we use the running example, a mixture model described in Figure 2.5.

```
N = 10000000  # data set size
D = 2  # data dimension
K = 5  # number of clusters
```

The model is

$$p(\mathbf{x}, \mathbf{z}, \beta) = p(\beta) \prod_{n=1}^{N} p(z_n \mid \beta) p(x_n \mid z_n, \beta).$$

To avoid memory issues, we work on only a subgraph of the model,

$$p(\mathbf{x}, \mathbf{z}, \beta) = p(\beta) \prod_{m=1}^{M} p(z_m \mid \beta) p(x_m \mid z_m, \beta)$$

```
M = 128  # mini-batch size
```

```
beta = Normal(mu=tf.zeros([K, D]), sigma=tf.ones([K, D]))
z = Categorical(logits=tf.zeros([M, K]))
x = Normal(mu=tf.gather(beta, z), sigma=tf.ones([M, D]))
```

Assume the variational model is

$$q(\mathbf{z}, \beta) = q(\beta; \lambda) \prod_{n=1}^{N} q(z_n \mid \beta; \gamma_n),$$

parameterized by $\{\lambda, \{\gamma_n\}\}$. Again, we work on only a subgraph of the model,

$$q(\mathbf{z}, \beta) = q(\beta; \lambda) \prod_{m=1}^{M} q(z_m \mid \beta; \gamma_m).$$

parameterized by $\{\lambda, \{\gamma_m\}\}$. Importantly, only M parameters are stored in memory for $\{\gamma_m\}$ rather than N.

```
qbeta = Normal(mu=tf.Variable(tf.zeros([K, D])),
               sigma=tf.nn.softplus(tf.Variable(tf.zeros[K, D])))
qz_variables = tf.Variable(tf.zeros([M, K]))
qz = Categorical(logits=qz_variables)
```

We use `KLqp`, a variational method that minimizes the divergence measure $\text{KL}(q \,\|\, p)$ (Jordan et al., 1999a). We instantiate two algorithms: a global inference over β given the subset of $\mathbf{z}$ and a local inference over the subset of $\mathbf{z}$ given β. We also pass in a TensorFlow placeholder `x_ph` for the data, so we can change the data at each step.

```
x_ph = tf.placeholder(tf.float32, [M])
inference_global = ed.KLqp({beta: qbeta}, data={x: x_ph, z: qz})
inference_local = ed.KLqp({z: qz}, data={x: x_ph, beta: qbeta})
```

We initialize the algorithms with the `scale` argument, so that computation on `z` and `x` will be scaled appropriately. This enables unbiased estimates for stochastic gradients.

```
inference_global.initialize(scale={x: float(N) / M, z: float(N) / M})
inference_local.initialize(scale={x: float(N) / M, z: float(N) / M})
```

We now run the algorithm, assuming there is a `next_batch` function which provides the next batch of data.

```
qz_init = tf.initialize_variables([qz_variables])
for _ in range(1000):
  x_batch = next_batch(size=M)
  for _ in range(10):  # make local inferences
    inference_local.update(feed_dict={x_ph: x_batch})

  # update global parameters
  inference_global.update(feed_dict={x_ph: x_batch})
  # reinitialize the local factors
  qz_init.run()
```

After each iteration, we also reinitialize the parameters for $q(\mathbf{z} \mid \beta)$; this is because we do inference on a new set of local variational factors for each batch. This demo readily applies to other inference algorithms such as `SGLD` (stochastic gradient Langevin dynamics): simply replace `qbeta` and `qz` with `Empirical` random variables; then call `ed.SGLD` instead of `ed.KLqp`.

Note that if the data and model fit in memory but you'd still like to perform data subsampling for fast inference, we recommend not defining subgraphs. You can reify the full model, and simply index the local variables with a placeholder. The placeholder is fed at runtime to determine which of the local variables to update at a time. (For more details, see the website's API.)

A.7 Complete Examples

A.8 Variational Auto-encoder

See Figure A.5.

A.9 Probabilistic Model for Word Embeddings

See Figure A.6. This example uses data subsampling (Section 2.3.4). The priors and conditional likelihoods are defined only for a minibatch of data. Similarly the variational model only models the embeddings used in a given minibatch. TensorFlow variables contain the embedding vectors for the entire vocabulary. TensorFlow placeholders ensure that the correct embedding vectors are used as variational parameters for a given minibatch.

```python
import edward as ed
import tensorflow as tf

from edward.models import Bernoulli, Normal
from scipy.misc import imsave
from tensorflow.contrib import slim
from tensorflow.examples.tutorials.mnist import input_data

M = 100  # batch size during training
d = 2  # latent variable dimension

# Probability model (subgraph)
z = Normal(mu=tf.zeros([M, d]), sigma=tf.ones([M, d]))
h = Dense(256, activation='relu')(z)
x = Bernoulli(logits=Dense(28 * 28, activation=None)(h))

# Variational model (subgraph)
x_ph = tf.placeholder(tf.float32, [M, 28 * 28])
qh = Dense(256, activation='relu')(x_ph)
qz = Normal(mu=Dense(d, activation=None)(qh),
            sigma=Dense(d, activation='softplus')(qh))

# Bind p(x, z) and q(z | x) to the same TensorFlow placeholder for x.
mnist = input_data.read_data_sets("data/mnist", one_hot=True)
data = {x: x_ph}

inference = ed.KLqp({z: qz}, data)
optimizer = tf.train.RMSPropOptimizer(0.01, epsilon=1.0)
inference.initialize(optimizer=optimizer)

tf.initialize_all_variables().run()

n_epoch = 100
n_iter_per_epoch = 1000
for _ in range(n_epoch):
  for _ in range(n_iter_per_epoch):
    x_train, _ = mnist.train.next_batch(M)
    info_dict = inference.update(feed_dict={x_ph: x_train})

  # Generate images.
  imgs = x.value().eval()
  for m in range(M):
    imsave("img/%d.png" % m, imgs[m].reshape(28, 28))
```

Figure A.5: Complete script for a VAE (Kingma and Welling, 2014b) with batch training. It generates MNIST digits after every 1000 updates.

The Bernoulli variables `y_pos` and `y_neg` are fixed to be 1's and 0's respectively. They model whether a word is indeed the target word for a given context window or has been drawn as a negative sample. Without regularization (via priors), the objective we optimize is identical to negative sampling.

```python
import edward as ed
import tensorflow as tf

from edward.models import Bernoulli, Normal, PointMass

N = 581238  # number of total words
M = 128  # batch size during training
K = 100  # number of factors
ns = 3  # number of negative samples
cs = 4  # context size
L = 50000  # vocabulary size

# Prior over embedding vectors
p_rho = Normal(mu=tf.zeros([M, K]),
               sigma=tf.sqrt(N) * tf.ones([M, K]))
n_rho = Normal(mu=tf.zeros([M, ns, K]),
               sigma=tf.sqrt(N) * tf.ones([M, ns, K]))

# Prior over context vectors
ctx_alphas = Normal(mu=tf.zeros([M, cs, K]),
                    sigma=tf.sqrt(N)*tf.ones([M, cs, K]))

# Conditional likelihoods
ctx_sum = tf.reduce_sum(ctx_alphas, [1])
p_eta = tf.expand_dims(tf.reduce_sum(p_rho * ctx_sum, -1),1)
n_eta = tf.reduce_sum(n_rho * tf.tile(tf.expand_dims(ctx_sum, 1), [1, ns, 1]), -1)
y_pos = Bernoulli(logits=p_eta)
y_neg = Bernoulli(logits=n_eta)

# placeholders for batch training
p_idx = tf.placeholder(tf.int32, [M, 1])
n_idx = tf.placeholder(tf.int32, [M, ns])
ctx_idx = tf.placeholder(tf.int32, [M, cs])

# Variational parameters (embedding vectors)
rho_params = tf.Variable(tf.random_normal([L, K]))
alpha_params = tf.Variable(tf.random_normal([L, K]))

# Variational distribution on embedding vectors
q_p_rho = PointMass(params=tf.squeeze(tf.gather(rho_params, p_idx)))
q_n_rho = PointMass(params=tf.gather(rho_params, n_idx))
q_alpha = PointMass(params=tf.gather(alpha_params, ctx_idx))

inference = ed.MAP(
  {p_rho: q_p_rho, n_rho: q_n_rho, ctx_alphas: q_alpha},
  data={y_pos: tf.ones((M, 1)), y_neg: tf.zeros((M, ns))})

inference.initialize()
tf.initialize_all_variables().run()

for _ in range(inference.n_iter):
  targets, windows, negatives = next_batch(M)  # a function to generate data
  info_dict = inference.update(feed_dict={p_idx: targets, ctx_idx: windows, n_idx: negatives})
  inference.print_progress(info_dict)
```

Figure A.6: Exponential family embedding for binary data (Rudolph et al., 2016). Here, MAP is used to maximize the total sum of conditional log-likelihoods and log-priors.

Appendix B: Simple, Distributed, and Accelerated Probabilistic Programming

B.1 Edward2 on SciPy

We illustrate the broad applicability of our tracing implementation by applying SciPy as a backend.

The implementation wraps `scipy.stats` distributions and registers each `rvs` method as traceable. Variables private from the namescope are explicitly prepended with underscore. Unlike Edward2 on TensorFlow Distributions, generative processes are recorded by calling `rvs` and wrapping Python functions, not Python classes. This is a result of `scipy.stats`'s functional API, which differs from TensorFlow Distributions' object-oriented one.

```python
from scipy import stats

_globals = globals()
for _name in sorted(dir(stats)):
  _candidate = getattr(stats, _name)
  if isinstance(_candidate, (stats._multivariate.multi_rv_generic,
                             stats.rv_continuous,
                             stats.rv_discrete,
                             stats.rv_histogram)):
    _candidate.rvs = traceable(_candidate.rvs)
    _globals[_name] = _candidate
    del _candidate
```

Below is an Edward2 linear regression program on SciPy.

```python
from edward2.scipy import stats as ed  # assuming rvs decorated here

def linear_regression(features):
  coeffs = ed.norm.rvs(loc=0.0, scale=0.1, size=features.shape[1], name="coeffs")
  loc = np.einsum('ij,j->i', features, coeffs)
  labels = ed.norm.rvs(loc=loc, scale=1., size=1, name="labels")
  return labels

log_joint = ed.make_log_joint_fn(linear_regression)

features = np.random.normal(size=[3, 2])
coeffs = np.random.normal(size=[2])
labels = np.random.normal(size=[3])
out = log_joint(features, coeffs=coeffs, labels=labels)
```

See the link to source code for more details.

B.2 Grammar Variational Auto-Encoder

Below implements a grammar VAE (Kusner et al., 2017). It consists of a probabilistic encoder and decoder. It extends probabilistic context-free grammars with neural networks, latent codes, and an encoder for learning representations of discrete structures. The decoder's `logits` is 3-dimensional with shape `[batch_size, max_timesteps, num_production_rules]`.

The encoder takes a string as input and applies `parse_to_one_hot`, a preprocessing step which parses it into a parse tree, extracts production rules from the tree, and converts each production rule into a one-hot vector; it then applies a neural net and outputs a normally-distributed latent code.

The decoder takes a latent code as input and maps it to a sequence of production rules representing the generated string. It applies an RNN followed by a masking step so that the result is a valid sequence of production rules in the grammar. The production rules may then be converted to a string.

```
import parse_to_one_hot

class ProbabilisticGrammarVariational(tf.keras.Model):
  """Amortized variational posterior for a probabilistic grammar."""

  def __init__(self, latent_size):
    """Constructs a variational posterior for a probabilistic grammar."""
    super(ProbabilisticGrammarVariational, self).__init__()
    self.latent_size = latent_size
    self.encoder_net = tf.keras.Sequential([
        tf.keras.layers.Conv1D(64, 3, padding="SAME"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Activation(tf.nn.elu),
        tf.keras.layers.Conv1D(128, 3, padding="SAME"),
        tf.keras.layers.BatchNormalization(),
        tf.keras.layers.Activation(tf.nn.elu),
        tf.keras.layers.Dropout(0.1),
        tf.keras.layers.GlobalAveragePooling1D(),
        tf.keras.layers.Dense(latent_size * 2, activation=None),
    ])

  def call(self, inputs):
    """Runs the model forward to return a stochastic encoding."""
    net = tf.cast(parse_to_one_hot(inputs), dtype=tf.float32)
    net = self.encoder_net(net)
    return ed.MultivariateNormalDiag(
        loc=net[..., :self.latent_size],
        scale_diag=tf.nn.softplus(net[..., self.latent_size:]),
        name="latent_code_posterior")

class ProbabilisticGrammar(tf.keras.Model):
  """Deep generative model over productions which follow a grammar."""

  def __init__(self, grammar, latent_size, num_units):
    """Constructs a probabilistic grammar."""
    super(ProbabilisticGrammar, self).__init__()
    self.grammar = grammar
    self.latent_size = latent_size
    self.lstm = tf.nn.rnn_cell.LSTMCell(num_units)
    self.output_layer = tf.keras.layers.Dense(len(grammar.production_rules))

  def call(self, inputs):
    """Runs the model forward to generate a sequence of productions."""
    del inputs  # unused
    latent_code = ed.MultivariateNormalDiag(loc=tf.zeros(self.latent_size),
                                            sample_shape=1,
                                             name="latent_code")
    state = self.lstm.zero_state(1, dtype=tf.float32)
    t = 0
    productions = []
    stack = [self.grammar.start_symbol]
    while stack:
      symbol = stack.pop()
      net, state = self.lstm(latent_code, state)
      logits = self.output_layer(net) + self.grammar.mask(symbol)
      production = ed.OneHotCategorical(logits=logits,
                                        name="production_" + str(t))
      _, rhs = self.grammar.production_rules[tf.argmax(production, axis=1)]
      for symbol in rhs:
        if symbol in self.grammar.nonterminal_symbols:
          stack.append(symbol)
      productions.append(production)
      t += 1
    return tf.stack(productions, axis=1)
```

See the link to source code for more details.

B.3 Markov chain Monte Carlo within Variational Inference

We demonstrate another level of composability: inference within a probabilistic program. Namely, we apply MCMC to construct a flexible family of distributions for variational inference (Salimans et al., 2015; Hoffman, 2017). We apply a chain of transition kernels specified by NUTS (`nuts`) in Section 3.3.2 and the variational inference algorithm specified by `train` in Figure 3.12.

```
import nuts, train

IMAGE_SHAPE = (32, 32, 3, 256)

def model():
  """Generative model of 32x32x3 8-bit images."""
  decoder_net = tf.keras.Sequential([
      tf.keras.layers.Dense(512, activation=tf.nn.relu),
      tf.keras.layers.Dense(np.prod(IMAGE_SHAPE), activation=None),
      tf.keras.layers.Reshape(IMAGE_SHAPE),
  ])

  z = ed.Normal(loc=tf.zeros([FLAGS.batch_size, FLAGS.latent_size]),
                scale=tf.ones([FLAGS.batch_size, FLAGS.latent_size]),
                name="z")
  x = ed.Categorical(logits=decoder_net(z), name="x")
  return x

def variational(x):
  """Variational model given 32x32x3 8-bit images."""
  encoder_net = tf.keras.Sequential([
      tf.keras.layers.Reshape(np.prod(IMAGE_SHAPE)),
      tf.keras.layers.Dense(512, activation=tf.nn.relu),
      tf.keras.layers.Dense(FLAGS.latent_size * 2, activation=None),
  ])

  net = encoder_net(x)
  qz = ed.Normal(loc=net[..., :FLAGS.latent_size],
                 scale=tf.nn.softplus(net[..., FLAGS.latent_size:]),
                 name="qz")
  for _ in range(FLAGS.mcmc_iterations):
    qz = nuts(current_state=qz,
              target_log_prob_fn=lambda z: ed.make_log_joint(model)(x=x, z=z))
  return qz

align_fn = lambda name: {'z': 'qz'}.get(name)
loss = train(0.1)  # uses model, variational, align_fn, x in scope
```

B.4 No-U-Turn Sampler

We implement an Edward2 program for Bayesian logistic regression with NUTS.

```
import build_dataset

def logistic_regression(features):
  """Bayesian logistic regression for labels given features."""
  coeffs = ed.MultivariateNormalDiag(loc=tf.zeros(features.shape[1]), name="coeffs")
  labels = ed.Bernoulli(logits=tf.tensordot(features, coeffs, [[1], [0]]))
  return labels

def make_target_log_prob_fn():
  """Make target density with log-joint function anchored at data."""
  log_joint_fn = ed.make_log_joint_fn(model)
  def target_log_prob_fn(coeffs):
    return log_joint_fn(features=features, coeffs=coeffs, labels=labels)
  return target_log_prob_fn

features, labels = build_dataset()
coeffs = tf.random_normal(features.shape[1])  # initial state
samples = ed.nuts(current_state=coeffs,
                  target_log_prob_fn=make_target_log_prob_fn())
```

See the link to source code for more details.

Appendix C: Applications in Variational Inference

C.1 Special cases of the variational Gaussian process

We now analyze two special cases of the VGP: by limiting its generative process in various ways, we recover well-known models. This provides intuition behind the VGP's complexity. We show many recently proposed models can also be viewed as special cases of the VGP.

Example 1. *A mixture of mean-field distributions is a* VGP *without a kernel.*

A discrete mixture of mean-field distributions (Bishop et al., 1998; Lawrence, 2000) is a classically studied variational model with dependencies between latent variables. Instead of a mapping which interpolates between inputs of the variational data, suppose the VGP simply performs nearest-neighbors for a latent input $\boldsymbol{\xi}$—selecting the output t_n tied to the nearest variational input s_n. This induces a multinomial distribution of outputs, which samples one of the variational outputs' mean-field parameters.[1] Thus, with a GP prior that interpolates between inputs, the VGP can be seen as a kernel density smoothing of the nearest-neighbor function.

Example 2. *Variational factor analysis is a* VGP *with linear kernel and no variational data.*

Consider factor analysis (Tipping and Bishop, 1999) in the variational space: [2]

$$\boldsymbol{\xi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \qquad \mathbf{z}_i \sim \mathcal{N}(\mathbf{w}^\top \boldsymbol{\xi}, \mathbf{I}).$$

Marginalizing over the latent inputs induces linear dependence in $\mathbf{z}$, $q(\mathbf{z}; \mathbf{w}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \mathbf{w}\mathbf{w}^\top)$.

[1]Formally, given variational input-output pairs $\{(\mathbf{s}_n, \mathbf{t}_n)\}$, the nearest-neighbor function is defined as $f(\boldsymbol{\xi}) = \mathbf{t}_j$, such that $\|\boldsymbol{\xi} - \mathbf{s}_j\| < \|\boldsymbol{\xi} - \mathbf{s}_k\|$ for all k. Then the output's distribution is multinomial with probabilities $P(f(\boldsymbol{\xi}) = \mathbf{t}_j)$, proportional to areas of the partitioned nearest-neighbor space.

[2] For simplicity, we avoid discussion of the VGP's underlying mean-field distribution, i.e., we specify each mean-field factor to be a degenerate point mass at its parameter value.

Consider the dual interpretation

$$\boldsymbol{\xi} \sim \mathcal{N}(\mathbf{0}, \mathbf{I}), \qquad f_i \sim \mathcal{GP}(0, k(\cdot,\cdot)), k(\mathbf{s}, \mathbf{s}') = \mathbf{s}^\top \mathbf{s}', \qquad \mathbf{z}_i = f_i(\boldsymbol{\xi}),$$

with $q(\mathbf{z} \,|\, \boldsymbol{\xi}) = \mathcal{N}(\mathbf{z}; \mathbf{0}, \boldsymbol{\xi}\boldsymbol{\xi}^\top)$. The maximum likelihood estimate of $\mathbf{w}$ in factor analysis is the maximum a posteriori estimate of $\boldsymbol{\xi}$ in the GP formulation. More generally, use of a non-linear kernel induces non-linear dependence in $\mathbf{z}$. Learning the set of kernel hyperparameters $\boldsymbol{\theta}$ thus learns the set capturing the most variation in its latent embedding of $\mathbf{z}$ (Lawrence, 2005).

C.2 Proof of Theorem 1

Theorem 1. *Let $q(\mathbf{z}; \boldsymbol{\theta}, \mathcal{D})$ denote the variational Gaussian process. Consider a posterior distribution $p(\mathbf{z} \,|\, \mathbf{x})$ with a finite number of latent variables and continuous quantile function (inverse CDF). There exists a sequence of parameters $(\boldsymbol{\theta}_k, \mathcal{D}_k)$ such that*

$$\lim_{k \to \infty} \mathrm{KL}(q(\mathbf{z}; \boldsymbol{\theta}_k, \mathcal{D}_k) \,\|\, p(\mathbf{z} \,|\, \mathbf{x})) = 0.$$

Proof. Let the mean-field distribution be given by degenerate delta distributions

$$q(\mathbf{z}_i \,|\, f_i) = \delta_{f_i}(\mathbf{z}_i).$$

Let the size of the latent input be equivalent to the number of latent variables $c = d$ and fix $\sigma^2_{\text{ARD}} = 1$ and $\boldsymbol{\omega}_j = 1$. Furthermore for simplicity, we assume that $\boldsymbol{\xi}$ is drawn uniformly on the d-dimensional hypercube. Then as explained in Section 4.2.4, if we let P^{-1} denote the inverse posterior cumulative distribution function, the optimal f denoted f^* such that

$$\mathrm{KL}(q(\mathbf{z}; \boldsymbol{\theta}) \,\|\, p(\mathbf{z} \,|\, \mathbf{x})) = 0$$

is

$$f^*(\xi) = P^{-1}(\boldsymbol{\xi}_1, ..., \boldsymbol{\xi}_d).$$

Define $\mathcal{O}_k$ to be the set of points $j/2^k$ for $j = 0$ to 2^k, and define $\mathcal{S}_k$ to be the d-dimensional product of $\mathcal{O}_k$. Let $\mathcal{D}_k$ be the set containing the pairs $(s_i, f^*(s_i))$, for each element s_i in $\mathcal{S}_k$. Denote f^k as the GP mapping conditioned on the dataset $\mathcal{D}_k$, this random mapping satisfies $f^k(s_i) = f^*(s_i)$ for all $s_i \in \mathcal{S}_k$ by the noise free prediction property of Gaussian processes (Rasmussen and Williams, 2006). Then by continuity, as $k \to \infty$, f^k converges to f^*. □

A broad condition under which the quantile function of a distribution is continuous is if that distribution has positive density with respect to the Lebesgue measure.

The rate of convergence for finite sizes of the variational data can be studied via posterior contraction rates for GPs under random covariates (Van Der Vaart and Van Zanten, 2011). Only an additional assumption using stronger continuity conditions for the posterior quantile and the use of Matern covariance functions is required for the theory to be applicable in the variational setting.

C.3 Variational objective

We derive the tractable lower bound to the model evidence $\log p(\mathbf{x})$ presented in Equation 4.6. To do this, we first penalize the ELBO with an expected KL term,

$$\begin{aligned}\log p(\mathbf{x}) \geq \mathcal{L} &= \mathbb{E}_{q_{\text{VGP}}}[\log p(\mathbf{x} \mid \mathbf{z})] - \text{KL}(q_{\text{VGP}}(\mathbf{z}) \| p(\mathbf{z})) \\ &\geq \mathbb{E}_{q_{\text{VGP}}}[\log p(\mathbf{x} \mid \mathbf{z})] - \text{KL}(q_{\text{VGP}}(\mathbf{z}) \| p(\mathbf{z})) - \mathbb{E}_{q_{\text{VGP}}}\Big[\text{KL}(q(\boldsymbol{\xi}, f \mid \mathbf{z}) \| r(\boldsymbol{\xi}, f \mid \mathbf{z}))\Big].\end{aligned}$$

We can combine all terms into the expectations as follows:

$$\begin{aligned}\widetilde{\mathcal{L}} &= \mathbb{E}_{q(\mathbf{z}, \boldsymbol{\xi}, f)}\Big[\log p(\mathbf{x} \mid \mathbf{z}) - \log q(\mathbf{z}) + \log p(\mathbf{z}) - \log q(\boldsymbol{\xi}, f \mid \mathbf{z}) + \log r(\boldsymbol{\xi}, f \mid \mathbf{z})\Big] \\ &= \mathbb{E}_{q(\mathbf{z}, \boldsymbol{\xi}, f)}\Big[\log p(\mathbf{x} \mid \mathbf{z}) - \log q(\mathbf{z} \mid f(\boldsymbol{\xi})) + \log p(\mathbf{z}) - \log q(\boldsymbol{\xi}, f) + \log r(\boldsymbol{\xi}, f \mid \mathbf{z})\Big],\end{aligned}$$

where we apply the product rule $q(\mathbf{z})q(\boldsymbol{\xi}, f \mid \mathbf{z}) = q(\mathbf{z} \mid f(\boldsymbol{\xi}))q(\boldsymbol{\xi}, f)$. Recombining terms as KL divergences, and written with parameters $(\boldsymbol{\theta}, \boldsymbol{\phi})$, this recovers the auto-encoded variational objective in Section 4.3:

$$\begin{aligned}\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi}) &= \mathbb{E}_{q_{\text{VGP}}}[\log p(\mathbf{x} \mid \mathbf{z})] - \mathbb{E}_{q_{\text{VGP}}}\Big[\mathrm{KL}(q(\mathbf{z} \mid f(\boldsymbol{\xi}))\|p(\mathbf{z}))\Big] \\ &\quad - \mathbb{E}_{q_{\text{VGP}}}\Big[\mathrm{KL}(q(f \mid \boldsymbol{\xi};\boldsymbol{\theta})\|r(f \mid \boldsymbol{\xi},\mathbf{z};\boldsymbol{\phi})) + \log q(\boldsymbol{\xi}) - \log r(\boldsymbol{\xi} \mid \mathbf{z})\Big].\end{aligned}$$

The KL divergence between the mean-field $q(\mathbf{z} \mid f(\boldsymbol{\xi}))$ and the model prior $p(\mathbf{z})$ is analytically tractable for certain popular models. For example, in the deep latent Gaussian model (Rezende et al., 2014) and DRAW (Gregor et al., 2015), both the mean-field distribution and model prior are Gaussian, leading to an analytic KL term: for Gaussian random variables of dimension d,

$$\begin{aligned}&\mathrm{KL}(\mathcal{N}(\mathbf{x};\mathbf{m}_1,\boldsymbol{\Sigma}_1)\|\mathcal{N}(\mathbf{x};\mathbf{m}_2,\boldsymbol{\Sigma}_2)) = \\ &\quad \frac{1}{2}\Big((\mathbf{m}_1-\mathbf{m}_2)^\top\boldsymbol{\Sigma}_1^{-1}(\mathbf{m}_1-\mathbf{m}_2) + \mathrm{tr}(\boldsymbol{\Sigma}_1^{-1}\boldsymbol{\Sigma}_2 + \log\boldsymbol{\Sigma}_1 - \log\boldsymbol{\Sigma}_2) - d\Big).\end{aligned}$$

In general, when the KL is intractable, we combine the KL term with the reconstruction term, and maximize the variational objective

$$\begin{aligned}\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi}) &= \mathbb{E}_{q_{\text{VGP}}}[\log p(\mathbf{x},\mathbf{z}) - \log q(\mathbf{z} \mid f(\boldsymbol{\xi}))] \\ &\quad - \mathbb{E}_{q_{\text{VGP}}}\Big[\mathrm{KL}(q(f \mid \boldsymbol{\xi};\boldsymbol{\theta})\|r(f \mid \boldsymbol{\xi},\mathbf{z};\boldsymbol{\phi})) + \log q(\boldsymbol{\xi}) - \log r(\boldsymbol{\xi} \mid \mathbf{z})\Big].\end{aligned} \tag{C.1}$$

We expect that this experiences slightly higher variance in the stochastic gradients during optimization.

We now consider the second term. Recall that we specify the auxiliary model to be a fully factorized Gaussian, $r(\boldsymbol{\xi}, f \mid \mathbf{z}) = \mathcal{N}((\boldsymbol{\xi}, f(\boldsymbol{\xi}))^\top \mid \mathbf{z};\mathbf{m},\mathbf{S})$, where $\mathbf{m} \in \mathbb{R}^{c+d}$, $\mathbf{S} \in \mathbb{R}^{c+d}$. Further, the variational priors $q(\boldsymbol{\xi})$ and $q(f \mid \boldsymbol{\xi})$ are both defined to be Gaussian. Therefore it is also a KL divergence between Gaussian distributed random variables. Similarly, $\log q(\boldsymbol{\xi}) - \log r(\boldsymbol{\xi} \mid \mathbf{z})$ is simply a difference of Gaussian log densities. The second expression is simple to compute and backpropagate gradients.

C.4 Gradients of the variational objective

We derive gradients for the variational objective (Equation 4.7). This follows trivially by backpropagation:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}}\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi}) &= \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}[\mathbb{E}_{w(\boldsymbol{\epsilon})}[\nabla_{\boldsymbol{\theta}}\mathbf{f}(\boldsymbol{\xi})\nabla_{\mathbf{f}}\mathbf{z}(\boldsymbol{\epsilon})\nabla_{\mathbf{z}}\log p(\mathbf{x}\,|\,\mathbf{z})]] \\
&\quad - \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}\Big[\mathbb{E}_{w(\boldsymbol{\epsilon})}\Big[\nabla_{\boldsymbol{\theta}}\,\mathrm{KL}(q(\mathbf{z}\,|\,\mathbf{f}(\boldsymbol{\xi};\boldsymbol{\theta}))\|p(\mathbf{z}))\Big]\Big] \\
&\quad - \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}\Big[\mathbb{E}_{w(\boldsymbol{\epsilon})}\Big[\nabla_{\boldsymbol{\theta}}\,\mathrm{KL}(q(f\,|\,\boldsymbol{\xi};\boldsymbol{\theta})\|r(f\,|\,\boldsymbol{\xi},\mathbf{z};\boldsymbol{\phi}))\Big]\Big], \\
\nabla_{\boldsymbol{\phi}}\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi}) &= -\mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}[\mathbb{E}_{w(\boldsymbol{\epsilon})}[\nabla_{\boldsymbol{\phi}}\,\mathrm{KL}(q(f\,|\,\boldsymbol{\xi};\boldsymbol{\theta})\|r(f\,|\,\boldsymbol{\xi},\mathbf{z};\boldsymbol{\phi})) - \nabla_{\boldsymbol{\phi}}\log r(\boldsymbol{\xi}\,|\,\mathbf{z};\boldsymbol{\phi})]],
\end{aligned}
$$

where we assume the KL terms are analytically written from Appendix C.3 and gradients are propagated similarly through their computational graph. In practice, we need only be careful about the expectations, and the gradients of the functions written above are taken care of with automatic differentiation tools.

We also derive gradients for the general variational bound of Equation C.1—it assumes that the first KL term, measuring the divergence between q and the prior for p, is not necessarily tractable. Following the reparameterizations described in Section 4.3.3, this variational objective can be rewritten as

$$
\begin{aligned}
\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi}) &= \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}\Big[\mathbb{E}_{w(\boldsymbol{\epsilon})}\Big[\log p(\mathbf{x},\mathbf{z}(\boldsymbol{\epsilon};\mathbf{f})) - \log q(\mathbf{z}(\boldsymbol{\epsilon};\mathbf{f})\,|\,\mathbf{f})\Big]\Big] \\
&\quad - \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}\Big[\mathbb{E}_{w(\boldsymbol{\epsilon})}\Big[\mathrm{KL}(q(f\,|\,\boldsymbol{\xi};\boldsymbol{\theta})\|r(f\,|\,\boldsymbol{\xi},\mathbf{z}(\boldsymbol{\epsilon};\mathbf{f});\boldsymbol{\phi})) + \log q(\boldsymbol{\xi}) - \log r(\boldsymbol{\xi}\,|\,\mathbf{z}(\boldsymbol{\epsilon};\mathbf{f}))\Big]\Big].
\end{aligned}
$$

We calculate gradients by backpropagating over the nested reparameterizations:

$$
\begin{aligned}
\nabla_{\boldsymbol{\theta}}\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi}) &= \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}[\mathbb{E}_{w(\boldsymbol{\epsilon})}[\nabla_{\boldsymbol{\theta}}\mathbf{f}(\boldsymbol{\xi})\nabla_{\mathbf{f}}\mathbf{z}(\boldsymbol{\epsilon})[\nabla_{\mathbf{z}}\log p(\mathbf{x},\mathbf{z}) - \nabla_{\mathbf{z}}\log q(\mathbf{z}\,|\,\mathbf{f})]]] \\
&\quad - \mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}\Big[\mathbb{E}_{w(\boldsymbol{\epsilon})}\Big[\nabla_{\boldsymbol{\theta}}\,\mathrm{KL}(q(f\,|\,\boldsymbol{\xi};\boldsymbol{\theta})\|r(f\,|\,\boldsymbol{\xi},\mathbf{z};\boldsymbol{\phi}))\Big]\Big] \\
\nabla_{\boldsymbol{\phi}}\widetilde{\mathcal{L}}(\boldsymbol{\theta},\boldsymbol{\phi}) &= -\mathbb{E}_{\mathcal{N}(\boldsymbol{\xi})}[\mathbb{E}_{w(\boldsymbol{\epsilon})}[\nabla_{\boldsymbol{\phi}}\,\mathrm{KL}(q(f\,|\,\boldsymbol{\xi};\boldsymbol{\theta})\|r(f\,|\,\boldsymbol{\xi},\mathbf{z};\boldsymbol{\phi})) - \nabla_{\boldsymbol{\phi}}\log r(\boldsymbol{\xi}\,|\,\mathbf{z};\boldsymbol{\phi})]].
\end{aligned}
$$

C.5 Scaling the size of variational data

If massive sizes of variational data are required, e.g., when its cubic complexity due to inversion of a $m \times m$ matrix becomes the bottleneck during computation, we can scale it further. Consider fixing the variational inputs to lie on a grid. For stationary kernels, this allows us to exploit Toeplitz structure for fast $m \times m$ matrix inversion. In particular, one can embed the Toeplitz matrix into a circulant matrix and apply conjugate gradient combined with fast Fourier transforms in order to compute inverse-matrix vector products in $\mathcal{O}(m \log m)$ computation and $\mathcal{O}(m)$ storage (Cunningham et al., 2008). For product kernels, we can further exploit Kronecker structure to allow fast $m \times m$ matrix inversion in $\mathcal{O}(Pm^{1+1/P})$ operations and $\mathcal{O}(Pm^{2/P})$ storage, where $P > 1$ is the number of kernel products (Osborne, 2010). The ARD kernel specifically leads to $\mathcal{O}(cm^{1+1/c})$ complexity, which is linear in m.

Appendix D: Applications in Probabilistic Models

D.1 Noise versus Latent Variables

Hims have two sources of randomness for each data point: the latent variable $\mathbf{z}_n$ and the noise $\boldsymbol{\epsilon}_n$; these sources of randomness get transformed to produce $\mathbf{x}_n$. Bayesian analysis infers posteriors on latent variables. A natural question is whether one should also infer the posterior of the noise.

The posterior's shape—and ultimately if it is meaningful—is determined by the dimensionality of noise and the transformation. For example, consider the gan model, which has no local latent variable, $\mathbf{x}_n = g(\boldsymbol{\epsilon}_n; \boldsymbol{\theta})$. The conditional $p(\mathbf{x}_n \mid \boldsymbol{\epsilon}_n)$ is a point mass, fully determined by $\boldsymbol{\epsilon}_n$. When $g(\cdot; \boldsymbol{\theta})$ is injective, the posterior $p(\boldsymbol{\epsilon}_n \mid \mathbf{x}_n)$ is also a point mass,

$$p(\boldsymbol{\epsilon}_n \mid \mathbf{x}_n) = \mathbb{I}[\boldsymbol{\epsilon}_n = g^{-1}(\mathbf{x}_n)],$$

where g^{-1} is the left inverse of g.This means for injective functions of the randomness (both noise and latent variables), the "posterior" may be worth analysis as a deterministic hidden representation (Donahue et al., 2017), but it is not random.

The point mass posterior can be found via nonlinear least squares. Nonlinear least squares yields the iterative algorithm

$$\hat{\boldsymbol{\epsilon}}_n = \hat{\boldsymbol{\epsilon}}_n - \rho_t \nabla_{\hat{\boldsymbol{\epsilon}}_n} f(\hat{\boldsymbol{\epsilon}}_n)^\top (f(\hat{\boldsymbol{\epsilon}}_n) - \mathbf{x}_n),$$

for some step size sequence ρ_t. Note the updates will get stuck when the gradient of f is zero. However, the injective property of f allows the iteration to be checked for correctness (simply check if $f(\hat{\boldsymbol{\epsilon}}_n) = \mathbf{x}_n$).

D.2 Implicit Model Examples in Edward

We demonstrate implicit models via example implementations in Edward (Tran et al., 2016a).

Figure D.1 implements a 2-layer deep implicit model. It uses `tf.layers` to define neural networks: `tf.layers.dense(x, 256)` applies a fully connected layer with 256 hidden units and input x; weight and bias parameters are abstracted from the user. The program generates N data points $\mathbf{x}_n \in \mathbb{R}^{10}$ using two layers of implicit latent variables $\mathbf{z}_{n,1}, \mathbf{z}_{n,2} \in \mathbb{R}^d$ and with an implicit likelihood.

Figure D.2 implements a Bayesian GAN for classification. It manually defines a 2-layer neural network, where for each data index, it takes features $\mathbf{x}_n \in \mathbb{R}^{500}$ concatenated with noise $\boldsymbol{\epsilon}_n \in \mathbb{R}$ as input. The output is a label $\mathbf{y}_n \in \{-1, 1\}$, given by the sign of the last layer. We place a standard normal prior over all weights and biases. Running this program while feeding the placeholder $\mathbf{X} \in \mathbb{R}^{N\times 500}$ generates a vector of labels $\mathbf{y} \in \{-1, 1\}^N$.

```python
import tensorflow as tf
from edward.models import Normal

# random noise is Normal(0, 1)
eps2 = Normal(tf.zeros([N, d]), tf.ones([N, d]))
eps1 = Normal(tf.zeros([N, d]), tf.ones([N, d]))
eps0 = Normal(tf.zeros([N, d]), tf.ones([N, d]))

# alternate latent layers z with hidden layers h
z2 = tf.layers.dense(eps2, 128, activation=tf.nn.relu)
h2 = tf.layers.dense(z2, 128, activation=tf.nn.relu)
z1 = tf.layers.dense(tf.concat([eps1, h2], 1), 128, activation=tf.nn.relu)
h1 = tf.layers.dense(z1, 128, activation=tf.nn.relu)
x  = tf.layers.dense(tf.concat([eps0, h1], 1), 10, activation=None)
```

Figure D.1: Two-layer deep implicit model for data points $\mathbf{x}_n \in \mathbb{R}^{10}$. The architecture alternates with stochastic and deterministic layers. To define a stochastic layer, we simply inject noise by concatenating it into the input of a neural net layer.

D.3 KL Uniqueness

An integral probability metric measures distance between two distributions p and q,

$$d(p, q) = \sup_{f\in\mathcal{F}} |\mathbb{E}_p f - \mathbb{E}_q f|.$$

Integral probability metrics have been used for parameter estimation in generative models (Dziugaite et al., 2015) and for variational inference in models with tractable density (Ranganath et al., 2016b).

```python
import tensorflow as tf
from edward.models import Normal

# weights and biases have Normal(0, 1) prior
W1 = Normal(tf.zeros([501, 256]), tf.ones([501, 256]))
W2 = Normal(tf.zeros([256, 1]), tf.ones([256, 1]))
b1 = Normal(tf.zeros(256), tf.ones(256))
b2 = Normal(tf.zeros(1), tf.ones(1))

# set up inputs to neural network
X = tf.placeholder(tf.float32, [N, 500])
eps = Normal(tf.zeros([N, 1]), tf.ones([N, 1]))

# y = neural_network([x, eps])
input = tf.concat([X, eps], 1)
h1 = tf.nn.relu(tf.matmul(input, W1) + b1)
h2 = tf.matmul(h1, W2) + b2
y = tf.reshape(tf.sign(h2), [-1])  # take sign, then flatten
```

Figure D.2: Bayesian GAN for classification, taking $\mathbf{X} \in \mathbb{R}^{N\times 500}$ as input and generating a vector of labels $\mathbf{y} \in \{-1, 1\}^N$. The neural network directly generates the data rather than parameterizing a probability distribution.

In contrast to models with only local latent variables, to infer the posterior, we need an integral probability metric between it and the variational approximation. The direct approach fails because sampling from the posterior is intractable.

An indirect approach requires constructing a sufficiently broad class of functions with posterior expectation zero based on Stein's method (Ranganath et al., 2016b). These constructions require a likelihood function and its gradient. Working around the likelihood would require a form of nonparametric density estimation; unlike ratio estimation, we are unaware of a solution that sufficiently scales to high dimensions.

As another class of divergences, the f divergence is

$$d(p, q) = \mathbb{E}_q\Big[f\left(\frac{p}{q}\right)\Big].$$

Unlike integral probability metrics, f divergences are naturally conducive to ratio estimation, enabling implicit p and implicit q. However, the challenge lies in scalable computation. To subsample data in hierarchical models, we need f to satisfy up to constants $f(ab) = f(a) + f(b)$, so that the expectation becomes a sum over individual data points. For continuous functions, this is a defining property of the $\log$ function. This implies the KL-divergence from q to p is the only f divergence where the subsampling technique in our desiderata is possible.

D.4 Hinge Loss

Let $r(\mathbf{x}_i, \mathbf{z}_i, \beta; \theta)$ output a real value, as with the log loss in Section 4. The hinge loss is

$$\begin{aligned}\mathcal{D}_{\text{hinge}} = \mathbb{E}_{p(\mathbf{x}_n, \mathbf{z}_n \mid \beta)}[\max(0, 1 - r(\mathbf{x}_n, \mathbf{z}_n, \beta; \boldsymbol{\theta}))] + \\ \mathbb{E}_{q(\mathbf{x}_n, \mathbf{z}_n \mid \beta)}[\max(0, 1 + r(\mathbf{x}_n, \mathbf{z}_n, \beta; \boldsymbol{\theta}))].\end{aligned}$$

We minimize this loss function by following unbiased gradients. The gradients are calculated analogously as for the log loss. The optimal r^* is the log ratio.

D.5 Comparing Bayesian GANs with MAP to GANs with MLE

In Section 4, we argued that MAP estimation with a Bayesian GAN enables analysis over discrete data, but GANs—even with a maximum likelihood objective (Goodfellow, 2014)—cannot. This is a surprising result: assuming a flat prior for MAP, the two are ultimately optimizing the same objective. We compare the two below.

For GANs, assume the discriminator outputs a logit probability, so that it's unconstrained instead of on $[0, 1]$. GANs with MLE use the discriminative problem

$$\max_{\boldsymbol{\theta}} \mathbb{E}_{q(\mathbf{x})}[\log \sigma(D(\mathbf{x}; \boldsymbol{\theta}))] + \mathbb{E}_{p(\mathbf{x}; \mathbf{w})}[\log(1 - \sigma(D(\mathbf{x}; \boldsymbol{\theta})))].$$

They use the generative problem

$$\min_{\mathbf{w}} \mathbb{E}_{p(\mathbf{x}; \mathbf{w})}[-\exp(D(\mathbf{x}))].$$

Solving the generative problem with reparameterization gradients requires backpropagating through data generated from the model, $\mathbf{x} \sim p(\mathbf{x}; \mathbf{w})$. This is not possible for discrete $\mathbf{x}$. Further, the exponentiation also makes this objective numerically unstable and thus unusable in practice.

Contrast this with Bayesian GANs with MLE (MAP and a flat prior). This applies a point mass variational approximation $q(\mathbf{w}') = \mathbb{I}[\mathbf{w}' = \mathbf{w}]$. It maximizes the ELBO,

$$\max_{\mathbf{w}} \mathbb{E}_{q(\mathbf{w})}[\log p(\mathbf{w}) - \log q(\mathbf{w})] + \sum_{n=1}^{N} r(\mathbf{x}_n, \mathbf{w}).$$

The first term is zero for a flat prior $p(\mathbf{w}) \propto 1$ and point mass approximation; the problem reduces to

$$\max_{\mathbf{w}} \sum_{n=1}^{N} r(\mathbf{x}_n, \mathbf{w}).$$

Solving this is possible for discrete $\mathbf{x}$: it only requires backpropagating gradients through $r(\mathbf{x}, \mathbf{w})$ with respect to $\mathbf{w}$, all of which is differentiable. Further, the objective does not require a numerically unstable exponentiation.

Ultimately, the difference lies in the role of the ratio estimators. Recall for Bayesian GANS, we use the ratio estimation problem

$$\mathcal{D}_{\text{log}} = \mathbb{E}_{p(\mathbf{x};\mathbf{w})}[-\log \sigma(r(\mathbf{x}, \mathbf{w}; \boldsymbol{\theta}))] + \\ \mathbb{E}_{q(\mathbf{x})}[-\log(1 - \sigma(r(\mathbf{x}, \mathbf{w}; \boldsymbol{\theta})))].$$

The optimal ratio estimator is the log-ratio $r^*(\mathbf{x}, \mathbf{w}) = \log p(\mathbf{x} \mid \mathbf{w}) - \log q(\mathbf{x})$. Optimizing it with respect to $\mathbf{w}$ reduces to optimizing the log-likelihood $\log p(\mathbf{x} \mid \mathbf{w})$. The optimal discriminator for GANS with MLE has the same ratio, $D^*(\mathbf{x}) = \log p(\mathbf{x}; \mathbf{w}) - \log q(\mathbf{x})$; however, it is a constant function with respect to $\mathbf{w}$. Hence one cannot immediately substitute $D^*(\mathbf{x})$ as a proxy to optimizing the likelihood. An alternative is to use importance sampling; the result is the former objective (Goodfellow, 2014).

D.6 Stability of Ratio Estimator

With implicit models, the difference from standard KL variational inference lies in the ratio estimation problem. Thus we would like to assess the accuracy of the ratio estimator. We can check this by comparing to the true ratio under a model with tractable likelihood.

We apply Bayesian linear regression. It features a tractable posterior which we leverage in our analysis.

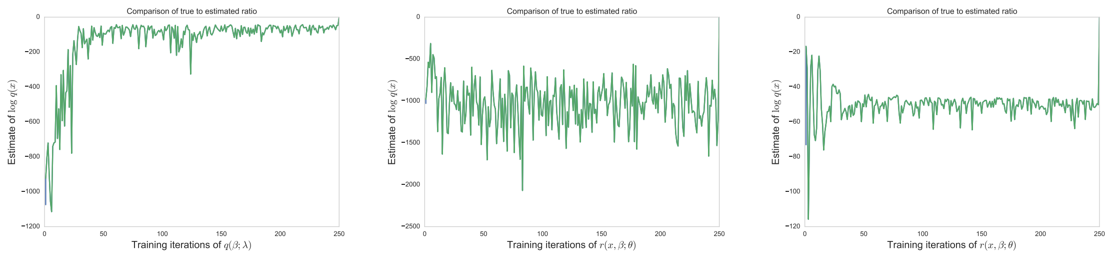


Figure D.3: (left) Difference of ratios over steps of q. Low variance on y-axis means more stable. Interestingly, the ratio estimator is more accurate and stable as q converges to the posterior. **(middle)** Difference of ratios over steps of r; q is fixed at random initialization. The ratio estimator doesn't improve even after many steps. **(right)** Difference of ratios over steps of r; q is fixed at the posterior. The ratio estimator only requires few steps from random initialization to be highly accurate.

We use 50 simulated data points $\{\mathbf{y}_n \in \mathbb{R}^2, \mathbf{x}_n \in \mathbb{R}\}$. The optimal (log) ratio is

$$r^*(\mathbf{x}, \beta) = \log p(\mathbf{x} \mid \beta) - \log q(\mathbf{x}).$$

Note the log-likelihood $\log p(\mathbf{x} \mid \beta)$ minus $r^*(\mathbf{x}, \beta)$ is equal to the empirical distribution $\sum_n \log q(\mathbf{x}_n)$, a constant. Therefore if a ratio estimator r is accurate, its difference with $\log p(\mathbf{x} \mid \beta)$ should be a constant with low variance across values of β.

See Figure D.3. The top graph displays the estimate of $\log q(\mathbf{x})$ over updates of the variational approximation $q(\beta)$; each estimate uses a sample from the current $q(\beta)$. The ratio estimator r is more accurate as q exactly converges to the posterior. This matches our intuition: if data generated from the model is close to the true data, then the ratio is more stable to estimate.

An alternative hypothesis for Figure D.3 is that the ratio estimator has simply accumulated information during training. This turns out to be untrue; see the bottom graphs. On the left, q is fixed at a random initialization; the estimate of $\log q(\mathbf{x})$ is displayed over updates of r. After many updates, r still produces unstable estimates. In contrast, the right shows the same procedure with q fixed at the posterior. r is accurate after few updates.

Several practical insights appear for training. First, it is not helpful to update r multiple times before updating q (at least in initial iterations). Additionally, if the specified model poorly matches the data, training will be difficult across all iterations.

The property that ratio estimation is more accurate as the variational approximation improves is because $q(\mathbf{x}_n)$ is set to be the empirical distribution. (Note we could subtract any density $q(\mathbf{x}_n)$ from the ELBO in Equation 4.) Likelihood-free variational inference finds $q(\beta)$ that makes the observed data likely under $p(\mathbf{x}_n \mid \beta)$, i.e., $p(\mathbf{x}_n \mid \beta)$ gets closer to the empirical distribution at values sampled from $q(\beta)$. Letting $q(\mathbf{x}_n)$ be the empirical distribution means the ratio estimation problem will be less trivially solvable (thus more accurate) as $q(\beta)$ improves.

Note also this motivates why we do not subsume inference of $p(\beta \mid \mathbf{x})$ in the ratio in order to enable implicit global variables and implicit global variational approximations. Namely, estimation requires comparing samples between the prior and the posterior; they rarely overlap for global variables.