Automating Bayesian Computation for Stochastic Simulators with Probabilistic Programming

Tim Reichelt

St Anne's College

University of Oxford

A thesis submitted for the degree of

Doctor of Philosophy

Hilary 2024

Acknowledgements

First and foremost, I would like to thank my supervisors. Tom Rainforth, for dedicating a tremendous amount of time to mentoring me, teaching me valuable lessons about writing papers and rebuttals, showing me how to present my research, and introducing me to the black arts of LaTeX paper formatting. His meticulous attention to detail and unwavering patience in explaining complex concepts have helped me grow into the researcher I am today. Luke Ong, for his guidance and being a seemingly endless source of mathematical knowledge. I would also like to thank Seth Flaxman for stepping in as a ceremonial supervisor at the latter stages of my PhD when Luke left Oxford.

I am grateful to Atılım Güneş Baydin and Brooks Paige for being the assessors in my viva and providing feedback that has lead to an improved thesis.

I would like to thank Adam Goliński for being an amazing co-author and a great guide to the world of machine learning research early on in my PhD. Wendy Poole for making every administrative task a breeze and making the AIMS CDT programme a wonderful social experience.

By far the biggest perk of being a PhD student at Oxford is the people you get to meet along the way. I am grateful for the many friends and colleagues who have made my time in Oxford so enjoyable. There are too many to name here explicitly. If you are reading this, you know who you are.

Finally, I would like to thank Tess who brings the sparkle to everything and helps me remember the important things in life when things get tough. Her parents, Gilli and Martin, for welcoming me to their home during the COVID lockdowns. My parents, Antje and Sebastian, for enabling my dreams and giving me the freedom to develop into the person I am today.

Abstract

Probabilistic programming systems (PPSs) automate the process of running Bayesian inference in stochastic simulator models. These stochastic simulators are ubiquitous in science and engineering: climate researchers build earth system models to predict future climate change; particle physicists build simulators to understand the experimental outcomes of particle colliders; and epidemiologists build models to predict how diseases spread. PPSs give us a principled way to incorporate these simulators into our decision-making process by enabling us to calibrate them to observed data using the tools of Bayesian inference. However to do so, PPS inference algorithms need to deal with all the complexities of modern programming languages. Importantly for this thesis modern PPSs often permit the usage of stochastic control flow, leading to so-called *programs with stochastic support*: programs in which the number and type of latent variables are no longer fixed.

We will make the argument for treating these programs as mixtures over program paths. Using this breakdown we derive a new variational inference algorithm that we term *Support Decomposition Variational Inference (SDVI)*. In contrast to prior work which constructs the variational family on a variable-by-variable basis, SDVI constructs the guide as a mixture over program paths, constructing a separate variational distribution for each path independently. This allows us to bring advances from variational inference from the static support setting to the stochastic support setting.

The breakdown of the program into a mixture over paths does not only help us derive new inference algorithms. We will also use it to investigate the properties of the posterior distribution more generally. Specifically, we show that the weights assigned to individual program paths can often be unstable; a problem that can arise either due to model misspecification or inference approximations. These instabilities make it harder to replicate results and can potentially give the user misleading confidence in their model's inferences. To alleviate these issues, we will propose alternative mechanisms to weight the program paths that instead *optimize the path weights on predictive objectives*.

Many PPSs focus on the goal of automating inference, however, it is important to also consider how the outcomes of inference are used in practice. Many workflows use the outputs of inference engines to estimate downstream expectations. To facilitate this use case, we will introduce the concept of *expectation programming* which allows users to directly define and estimate expectations in a target-aware manner; meaning the backend computation engine specifically tailors the estimation algorithm towards a user-specified expectation.

Contents

1 Introduction

Science and engineering are driven by developing models of the world. Progress is made by continuously comparing the predictions of our models against our observed data and refining our models to better match our observations [Popper, 1959, Box, 1976, 1980, Good, 1983, Rubin, 1984, Mayo, 1996, Jaynes, 2003, Gelman and Shalizi, 2013, Blei, 2014].[1] In a simplified view, this can be portrayed as an infinite loop of: 1) constructing a model of the phenomenon of interest, 2) calibrating the parameters of the model to observed data, 3) testing the calibrated model against new data and going back to step 1 to revise the model. To speed up progress in science, it is important to develop tools and algorithms which speed up each of these steps as much as possible.

Any model that is constructed will necessarily have to make simplifications and approximations when modelling the real-world. For example, when modelling voting patterns of citizens in the UK, we do not go all the way down to the level of trying to model the firing of individual neurons in the brains of each voter; doing this would be computationally prohibitively expensive and would most likely not provide any additional insights into voting patterns. However, by making simplifying assumptions we will inevitably introduce errors into our models. These errors will often exhibit randomness: if our model for predicting voting patterns is solely based on the age of the voter, we will have voters of the same age who vote differently.

[1] This general procedure of building and updating models does not necessarily only apply to science and engineering. In neuroscience, it has also been proposed as a general mechanism for how humans learn from experience and make decisions [Friston, 2010, Tenenbaum et al., 2011, Lake et al., 2017].

The mathematical formalization of probability allows us to construct models which can account for this randomness and enables us to build *probabilistic models*.[2]

Each sufficiently sophisticated model will tend to have a number of free parameters that cannot be determined from pure theory alone but need to be estimated from data. In our voter example, we might extend our model to not just take into account a voter's age but also their income, education, and the postcode they live in. We know that each of these factors will probably influence their vote but we do not know the exact effect each of them has, so we need to *infer* these effects based on observational data.

Dealing with these sort of inference problems is at the heart of the field of statistics [Casella and Berger, 2001, Gelman et al., 2013]. In statistics, we traditionally construct models for the observed data, $\mathbf{y}$, through a parameterized probability distribution $p(\mathbf{y} \mid \theta)$ where the parameters θ live in some parameter space Θ. This thesis will predominantly focus on a Bayesian perspective on statistics which quantifies a modeler's belief about the parameters of the model with probabilities. Hence, to specify a Bayesian model, a modeller first defines their beliefs about the unknown parameter θ *before* observing any data through a *prior distribution* $p(\theta)$. The goal of *inference* is then to compute the *posterior distribution* $p(\theta \mid \mathbf{y})$, the belief about the parameters *after* having observed the data. The relation between the prior ($p(\theta)$), the likelihood ($p(\mathbf{y} \mid \theta)$), and the posterior ($p(\theta \mid \mathbf{y})$) is given by Bayes' rule as a simple application of the rules of conditional probability. However, in practice, computing the posterior distribution is often intractable to compute exactly for complex models and large datasets and we need to resort to *approximate inference* algorithms.

For most applications, computing the posterior distribution is not enough but we want to use our posterior beliefs to make decisions. *Bayesian decision theory* (Robert et al. [2007], Section 2.2) provides a principled framework to make decisions under uncertainty by optimizing a loss which is defined as an *expectation* with respect to the posterior distribution. In that sense, Bayesian decision theory provides us with a model-based mechanism to make decisions because all our decisions depend explicitly

[2]The voter example here describes only one possible source of randomness/uncertainty. Describing and classifying the different sources of randomness and the suitability of probability to model them is a topic of never ending debate in statistics. In machine learning it has been argued for distinguishing between aleatoric, meaning roughly uncertainty that cannot be reduced further, and epistemic uncertainty, meaning roughly the uncertainty that can be reduced by collecting more data or improving the model. However, there is no universally agreed upon precise definition of these terms and they are less established in the wider statistics community which is why we forego their usage here. We will refer the interested reader to Hüllermeier and Waegeman [2021] for a more in-depth discussion of these terms and to Section 1.6 in Gelman et al. [2013] for a general discussion of the different sources of uncertainty from a statistical perspective.

on our probabilistic model. As the quality of our decisions now depends on the quality of our models, we want to make sure that our models are as accurate as possible.

This then leads to a fundamental trade-off in Bayesian statistics: as our models become more sophisticated, the more accurately we can capture the underlying phenomenon but the more difficult it becomes to approximate the posterior distribution with high accuracy. The models used in practice are often the models that sit on the boundary of what is computationally feasible. The task of inference algorithm developers is then to push this boundary ever outwards to allow practitioners to leverage more realistic models.

Over the years, a plethora of different Bayesian inference algorithms have been developed with that goal in mind. However, due to the diversity of models considered in practice, there are multiple different paradigms for constructing inference algorithms. Among the most popular are the Markov chain Monte Carlo (MCMC) [Brooks et al., 2011] and variational inference (VI) [Blei et al., 2017] methods; both of which we will discuss more in Chapter 2.

The inference algorithms employed in practice are often extremely complex making their implementation difficult and error-prone. As a response to this complexity, *probabilistic programming* [Gordon et al., 2014, van de Meent et al., 2018, Barthe et al., 2020] aims to *automate* the process of running inference in user-specified probabilistic models. Because of the diversity in the types of inference algorithms, many *probabilistic programming systems (PPSs)* focus on automating only one particular class of inference algorithm. For example, the BUGS language [Lunn et al., 2000, Spiegelhalter et al., 2007] was developed to allow users to encode graphical models [Koller and Friedman, 2009] and conduct inference using Gibbs sampling [Geman and Geman, 1984, Gelfand and Smith, 1990]. More recently, the Stan language [Carpenter et al., 2017] was developed with the main focus to automate the application of Hamiltonian Monte Carlo (HMC) [Neal, 2011, Betancourt, 2018], a powerful MCMC algorithm.

PPSs significantly lower the barrier of entry to use Bayesian inference algorithms, enabling the application of Bayesian modelling in areas as diverse as epidemiology [Flaxman et al., 2020], player skill rating systems [Herbrich et al., 2006], and election forecasting [Heidemanns et al., 2020]. However, many PPSs restrict the types of models that can be expressed in their input language, as the underlying inference algorithms which they automate can only handle a specific class of models; we will refer to these languages as *restricted PPSs*.

Notwithstanding the success of these restricted PPSs, there are still many probabilistic models that are not easily expressible in these languages. To simplify

inference many PPSs disallow specific language features such as branching on the outcomes of stochastic variables, stochastic loops, and recursion. These constraints inhibit the user to unleash the full modelling flexibility of modern programming languages.

The fact that lifting these restrictions would be useful is demonstrated by the countless scientific models that are already naturally expressed as *stochastic simulators*. For example, the world's largest supercomputers are used to forecast the weather for the next hours and days, simulate the climate for the next decades, and model the behaviour of the smallest particles in the universe. Many of these simulators leverage the aforementioned language features and, hence, cannot be expressed in restricted PPSs. Currently, calibrating these simulators to observed data relies on hand-tuning and a myriad of heuristics. Integrating these models within a PPS, would allow us to automate the calibration process and unlock the full power of Bayesian reasoning. Importantly, we could easily integrate these models in frameworks for Bayesian decision making, giving us a principled mechanism to incorporate the wealth of domain knowledge encoded in these simulators into our decision making process.

So-called *universal* PPSs[3] are designed to place minimal restrictions on the types of models that can be expressed with their input language [Goodman et al., 2008, Wood et al., 2014, Goodman and Stuhlmüller, 2014, Bingham et al., 2019, Cusumano-Towner et al., 2019, Ge et al., 2018]. The development of universal PPSs opens up a whole new avenue of research in Bayesian inference algorithms, as the probabilistic models expressed in universal PPSs can generally not be handled by traditional inference algorithms.

While universal PPSs have had early success in running inference in real-world scientific simulators [Baydin et al., 2019, 2020], they are still in their infancy and more work is needed to understand the inference problems defined in these languages. In general, universal PPSs are still held back by their ability to derive efficient inference algorithms for the type of inference problems that arise in complex simulators. Of particular interest are *programs with stochastic support* which arise if the input PPSs allows for branching on the outcomes of stochastic variables, leading to models in which the number and type of latent variables is no longer fixed.

[3]These languages are called "universal" because they are Turing-complete [Rainforth, 2017]. Another widely used term for these languages is "higher-order" PPS [van de Meent et al., 2018] to describe the fact that the language permits language features such as recursion and higher-order procedures/functions. We will mainly use the term "universal" for brevity and to emphasize the fact that PPS places minimal restrictions on the input program. More careful semantic descriptions of PPSs can be found in van de Meent et al. [2018] or Staton et al. [2016].

Programs with stochastic support are incredibly expressive and therefore give users a lot of flexibility when constructing models. To give a few examples, they allow users to express models for program synthesis based on probabilistic context-free grammars [Saad et al., 2019] or models for the evolution of species based on branching processes [Ronquist et al., 2020]. While there exist specialized inference algorithms for these models, they are often not easily generalizable. Thus, this motivates the need of having automated inference algorithms that can be readily applied to all programs with stochastic support.

Alas, due to its generality the corresponding inference problem poses several obstacles to developing *efficient inference algorithms*. For example, because of the stochastic control flow in the program, the dependency structure between different parameters in the model can vary between program executions making it difficult to efficiently apply standard approaches to inference. We will go into more detail discussing these challenges and existing approaches to inference in programs with stochastic support in Chapter 3.

As a brief aside, we will not consider the problem of how to efficiently run inference in existing simulators which are not already written in a PPS; there already exist approaches to this which rely on hijacking the random number generator of the simulator—via a purposefully designed communication protocol PPX—[Baydin et al., 2019, 2020] or automated code transformation tools [Zhi-Xuan et al., 2021]. The goal of this thesis is rather to isolate the conceptual hurdles that inference algorithms in programs with stochastic support need to overcome to be practical. The empirical validation therefore focuses mainly on models that are already implemented in a PPS. A valuable future research direction would be to make the methods developed in this thesis available as inference backends to arbitrary existing simulators through communication protocols such as PPX.

Overall, having efficient inference algorithms is only one piece in the puzzle to make probabilistic programming systems useful in practice. Most of the time, inference is only one step in a wider Bayesian workflow [Gelman et al., 2020]. Therefore, it is important to consider how our models and inferences are going to be used in practice, in order to inform the design of our systems and algorithms.

One particular prominent issue is that models are almost always *misspecified* due to the need to make simplifying assumptions, as outlined earlier. From a purely Bayesian perspective this is not an issue because our model encodes our subjective beliefs about the world but from a practical perspective it is important to be aware of the consequences of using a model that is misspecified [Box, 1976, Gelman and Shalizi, 2013].

Another aspect that tends to be overlooked is the fact that the results of inference are often used in downstream analyses that manifest as the need to *estimate expectations*. Even though it is completely valid to decouple the computational tasks of inference and expectation estimation, we are potentially missing out on computational efficiencies that can be gained by considering these tasks jointly [Gelman and Meng, 1998, Lacoste-Julien et al., 2011, Owen, 2013, Goliński et al., 2019, Rainforth et al., 2020].

This thesis will touch on all of these aspects: inference, model misspecification, and expectation estimation. The contributions specific to this thesis are outlined next.

1.1 Contributions of the Thesis

In Chapter 4, we will introduce a new variational inference algorithm for programs with stochastic support. While variational inference has had wide success in models with static support, they have faced significant complications when faced with models with stochastic support. We will demonstrate how existing variational algorithms for programs with stochastic support [Paige, 2016, van de Meent et al., 2018, Wingate and Weber, 2013] often struggle. To improve them, we present a new mechanism to construct the variational family based on decomposing the input program into sub-programs called straight-line programs (SLPs).

In Chapter 5 we will look more closely at the form of the posterior distributions in programs with stochastic support. Because the posterior distribution in models with stochastic support decomposes as a weighted sum over different SLPs, we can draw explicit connections to the literature on Bayesian model averaging (BMA) [Hoeting et al., 1999]. Crucially, when faced with model misspecification the path weights in the posterior can become unstable, leading to nonsensical inferences. As a remedy we will propose to instead optimise the path weights by optimising a predictive objective which leads to weights that are more stable in the face of the misspecification.

Finally in Chapter 6, we further develop the idea of having alternative quantities of interest as the computational goal in probabilistic programming systems. We introduce the concept of *expectation programming* which, instead of focusing on the calculation of posterior distributions, focuses on the *estimation of expectations*. Traditionally, in PPSs the user first runs an inference algorithm and then uses the outputs of inference to run any downstream analysis which can often be formulated as expectations under the posterior. However, if the user knows the target expectation ahead of running the inference, we can leverage more efficient

computational pipelines for the estimation. We provide a concrete implementation of our expectation programming concept as an extension of the Turing PPS [Ge et al., 2018] which we call *Expectation Programming in Turing (EPT)*.

2

Fundamentals of Bayesian Statistics

In this section we will give a brief introduction to the fundamentals of Bayesian statistics. This is by no means an extensive introduction to the topic, more comprehensive treatments can be found in Bishop [2006], Robert et al. [2007], Gelman et al. [2013], McElreath [2018], or Murphy [2022]. It would be disingenuous to present the Bayesian approach to statistics as the only approach without mentioning its criticisms and alternatives which we will do in Section 2.4.

2.1 Probabilistic Models

As mentioned in the introduction, before we can make any statistical inferences we first have to specify a model. Bayesian statistics requires us to specify a *joint probability distribution* over the (fixed) observed data $\mathbf{y}$ and the parameters of the model θ, usually in the form of a probability density $p(\mathbf{y}, \theta)$.[1] The parameter space of the model $\theta \in \Theta$ is not necessarily finite dimensional; later chapters will discuss more explicitly how to deal with infinite dimensional parameter spaces. This joint density factorises into a *likelihood* function $p(\mathbf{y} \mid \theta)$ and a *prior* $p(\theta)$ over the parameters.[2] The prior specifies our beliefs about the parameters before observing any data.

[1] We will use the term 'density' somewhat informally. Our parameters and data are not necessarily continuous but could also be a mix of discrete and continuous variables. Formally, then we are talking about the Radon-Nikodym derivative of a probability measure with respect to a suitable reference measure. A full formalization of this would require going into a measure theoretic treatment of probability which we will eschew in favour of a simplified notation.

[2] While the likelihood is denoted as a conditional probability distribution, it actually is a function of the parameters θ and not the data $\mathbf{y}$ because the data is fixed. For this reason the likelihood is sometimes also denoted as $L(\theta \mid \mathbf{y}) := p(\mathbf{y} \mid \theta)$.

In practice, we often have access to additional inputs $\mathbf{x}$ (also sometimes referred to as covariates, explanatory variables, or independent variables) which are taken to be fixed and not explicitly modelled as random. In this case our joint probability model can be denoted as $p(\mathbf{y}, \theta \mid \mathbf{x})$. For notational clarity, we will mostly omit the inputs from here but the extension to the case with inputs is straightforward.

To update our beliefs after having observed the data, we can apply Bayes' rule to obtain the *posterior*

$$p(\theta \mid \mathbf{y}) = \frac{p(\mathbf{y} \mid \theta)p(\theta)}{p(\mathbf{y})} = \frac{p(\mathbf{y} \mid \theta)p(\theta)}{\int_\Theta p(\mathbf{y} \mid \theta)p(\theta)d\theta}. \tag{2.1}$$

The posterior is a natural consequence of applying the rules of conditional probability, however, in practice computing it is associated with major computational complexity. The main obstacle to computing the posterior is the normalising constant, $p(\mathbf{y})$, which is also known as the *marginal likelihood* or *evidence*. As the normalising constant is an integral over the (potentially high-dimensional) parameter space, it is generally intractable to compute, making the posterior intractable to compute as well.

A notable class of models in which the posterior can be computed exactly are *conjugate models* [Raiffa and Schlaifer, 2000]. In these models the prior and likelihood are chosen such that the posterior is of the same form as the prior and the prior-posterior update can be computed using known algebraic rules. Computing the posterior in conjugate models is then simply a matter of plugging in the data into a known formula.

However, in general, there is no analytic formula available and we will have to resort to approximate methods to compute the posterior. Luckily, in the past two decades significant progress has been made in developing ever more powerful inference algorithms, in turn allowing the use of ever more complex and realistic models in practice. We will discuss these inference algorithms in Section 2.3.

Example: Logistic regression. Let us assume we are a bank and want to predict whether a given transaction is fraudulent or not. We have collected a dataset of N transactions with data about the amount of the transaction $\mathbf{x} \in \mathbb{R}^N$ and a corresponding label of whether the transaction was fraudulent or not $\mathbf{y} \in \{0, 1\}^N$. We can specify a simple logistic regression model to model the probability of a transaction being fraudulent as a function of the amount of the transaction

$$p(\mathbf{y} \mid \alpha, \beta, \mathbf{x}) = \prod_{n=1}^{N} \sigma(\alpha + \beta x_n)^{y_n}(1 - \sigma(\alpha + \beta x_n))^{(1-y_n)} \tag{2.2}$$

where the intercept, α, and slope, β, are the parameters of our model, $\theta = \{\alpha, \beta\}$ and the sigmoid function is given by $\sigma(x) = 1/(1 + \exp(-x))$. Specifying a prior on our parameters $p(\alpha, \beta) = \mathcal{N}(\alpha; 0, 1)\,\mathcal{N}(\beta; 0, 1)$ completes the specification of our Bayesian model. Even for this naively simple model it is not possible to compute the posterior distribution in closed-form.

2.2 Decision Theory

The previous section introduced the concept of a probabilistic model and how to update our beliefs about the parameters of the model after observing data. However, computing the posterior distribution is not necessarily end goal of an analysis in an applied setting. Generally, we want to use our models to inform our decision making. Otherwise, what is the point of going through all the hard work of constructing a realistic model if at the end of the day it does not influence our decision making?

We can formalise the idea of decision making through the framework of decision theory [Robert et al., 2007] in which we consider a set of possible decisions $\delta \in \mathcal{D}$ and a *loss function* $\mathsf{L}(\theta, \delta) : \Theta \times \mathcal{D} \to \mathbb{R}$ which measures the cost of making decision δ when the true state of the world is θ. Since our uncertainty about the state of the world is measured by the posterior distribution, we can compute the *posterior expected loss*

$$\varrho(\delta) := \mathbb{E}_{p(\theta|\mathbf{y})}[\mathsf{L}(\theta, \delta)] = \int_{\Theta} \mathsf{L}(\theta, \delta) p(\theta \mid \mathbf{y}) d\theta. \tag{2.3}$$

to inform our decision making. Hence, in the classical Bayesian framework the optimal decision minimises the posterior expected loss

$$\delta^* = \underset{\delta \in \mathcal{D}}{\operatorname{argmin}}\, \varrho(\delta). \tag{2.4}$$

Example (continued): Logistic regression. Continuing with our fraud prediction example from above, our end goal is not learn about the parameters of the logistic regression model but to make decisions about whether to approve or decline a transaction. The decision is then to *predict* for a new transaction x^* whether it is fraudulent or not, i.e. assigning a label $y^* \in \{0, 1\}$ to x^*. For binary classification the log-loss is a commonly choice of loss function leading the posterior expected loss that can be expressed as

$$\varrho(y^*) = \mathbb{E}_{p(\alpha,\beta|\mathbf{y},\mathbf{x})}\left[-\left(y^* \log(\sigma(\alpha + \beta x^*)) + (1 - y^*) \log(1 - \sigma(\alpha + \beta x^*))\right)\right]. \tag{2.5}$$

Intuitively, optimizing the log-loss leads to picking the label that is assigned the highest posterior expected log probability.

There is a lot more to be said about Bayesian decision theory, we will refer the reader to discussions in Robert et al. [2007], Bishop [2006], Gelman et al. [2013] for a more comprehensive treatment of the subject. For now, the crucial point is that making decisions requires us to be able to compute the posterior expected loss and solve the optimisation problem. As we noted in Section 2.3, computing the posterior distribution is generally intractable and, hence, computing the posterior expected loss is as well. We therefore have to resort to estimation methods to approximate ϱ. This need to estimate expectations guides the development of many common approximate inference algorithms which we will discuss in the next section.

2.3 Inference Algorithms

We have now established that the key mechanism with which we interact with the posterior distribution is through evaluating expectations. For a general target function $f(\theta)$, the expectation under the posterior distribution is given by

$$\mu := \mathbb{E}_{p(\theta|\mathbf{y})}\left[f(\theta)\right] = \int_{\Theta} f(\theta)p(\theta \mid \mathbf{y})d\theta. \tag{2.6}$$

For most interesting models that are used in practice, there is no analytic solution to compute the integral defined by the expectation. Instead, we need to find ways to compute *estimates* of our *estimand*, μ. Arguably, the most fundamental estimator is the *simple Monte Carlo estimator* [Owen, 2013] which is constructed by sampling from the distribution of interest and averaging the target function over the samples

$$\hat{\mu}_S := \frac{1}{S}\sum_{s=1}^{S} f(\theta^{(s)}) \quad \text{where} \quad \theta^{(s)} \sim p(\cdot \mid \mathbf{y}). \tag{2.7}$$

Having constructed our first estimator the natural next question to ask is: What makes a good estimator? Generally, there are three properties we care about: *consistency*, *bias*, and *variance*. An estimator is consistent if in the limit of large sample sizes it converges to the true estimand (making the standard assumption that μ exists and is finite). Based on the weak and strong law of large numbers, there are also two types consistency. An estimator is *weakly consistent* if it converges in probability to the true estimand, i.e.

$$\lim_{S\to\infty} \mathbb{P}\left(|\hat{\mu}_S - \mu| > \epsilon\right) = 0 \quad \text{for all} \quad \epsilon > 0. \tag{2.8}$$

On the other hand an estimator is *strongly consistent* if it converges almost surely to the true estimand, i.e.

$$\mathbb{P}\left(\lim_{S\to\infty} |\hat{\mu}_S - \mu| = 0\right) = 1. \tag{2.9}$$

Consistency ensures that having infinite time and compute we will eventually converge to the true estimand. However, in the real world we are restricted with finite time and resources so it is important to consider what happens for a finite number of samples S. An estimator is *unbiased* if its expectation is equal to the true estimand, i.e. $\mathbb{E}\left[\hat{\mu}_S\right] = \mu$. For the simple Monte Carlo estimator we can easily show that it is unbiased using the linearity of the expectation operator

$$\mathbb{E}\left[\hat{\mu}_S\right] = \frac{1}{S}\sum_{s=1}^{S}\mathbb{E}\left[f(\theta^{(s)})\right] = \frac{1}{S}\sum_{s=1}^{S}\mu = \mu. \tag{2.10}$$

If an estimator is unbiased, we know that for many different independent estimates, the average will converge to the true estimand. Finally, the *variance* of an estimator gives an indication of how much the estimate will vary around the true estimand. Assuming the variance of $p(\theta \mid \mathbf{y})$ is given by $\sigma^2 < \infty$, the variance of the simple Monte Carlo estimator is given by

$$\mathbb{E}\left[(\hat{\mu}_S - \mu)^2\right] = \frac{\sigma^2}{S}. \tag{2.11}$$

In summary, we generally want our estimator to be consistent, unbiased, and have low variance.

The decision theoretic framework in combination with Monte Carlo estimators provides the foundational motivation for many Bayesian inference algorithms. To inform decision making, we have to compute expectations under the posterior distribution and we can estimate these expectations by generating samples. This is why so many inference algorithms are focused on generating samples. Sections 2.3.1 and 2.3.2 will discuss two of the most fundamental sampling algorithms: importance sampling and Markov Chain Monte Carlo (MCMC).

2.3.1 Importance Sampling

The key obstacle to applying the simple Monte Carlo estimator to estimate posterior expectations is the inability to sample from the posterior. Importance sampling assumes that we have an alternative *proposal distribution* $q(\theta)$ which we can sample from instead. Some basic algebraic manipulations allow us to reformulate the expectation under the posterior

$$\mathbb{E}_{p(\theta|\mathbf{y})}\left[f(\theta)\right] = \int_{\Theta} f(\theta)p(\theta \mid \mathbf{y})d\theta \tag{2.12}$$

as an expectation under the proposal distribution

$$= \int_{\Theta} f(\theta)\frac{p(\theta \mid \mathbf{y})}{q(\theta)}q(\theta)d\theta = \mathbb{E}_{q(\theta)}\left[f(\theta)\frac{p(\theta \mid \mathbf{y})}{q(\theta)}\right]. \tag{2.13}$$

Algorithm 1 Self-Normalised Importance Sampling

Require: Joint distribution p, proposal q, number of samples S

1: **for** $s = 1, \dots, S$ **do**
2: Sample $\theta^{(s)} \sim q(\theta)$
3: Compute importance weight $w^{(s)} = \frac{p(\mathbf{y}, \theta^{(s)})}{q(\theta^{(s)})}$
4: **end for**
5: Compute normalised importance weights $w^{(s)} \leftarrow \frac{w^{(s)}}{\sum_{s=1}^{S} w^{(s)}}$
6: Return $\{(w^{(1)}, \theta^{(1)}), \dots, (w^{(S)}, \theta^{(S)})\}$

This then leads to the *importance sampling (IS) estimator*

$$\hat{\mu}_S^{\text{IS}} = \frac{1}{S} \sum_{s=1}^{S} w^{(s)} f(\theta^{(s)}) \quad \text{where} \quad w^{(s)} = \frac{p(\theta^{(s)} \mid \mathbf{y})}{q(\theta^{(s)})}, \quad \theta^{(s)} \sim q(\cdot). \tag{2.14}$$

The *importance weights* $w^{(s)}$ are the key mechanism with which we adjust our estimate to account for the difference between the proposal and the target distribution. To have a valid proposal we require that $q(\theta) > 0$ whenever $p(\theta \mid \mathbf{y}) \neq 0$ so that we avoid division by zero. However, in practice we cannot compute the importance weight because we do not know the normalising constant of the posterior distribution. We can instead use the ratio $p(\mathbf{y}, \theta)/q(\theta)$ to compute unnormalised weights and then normalise the weights to obtain the *self-normalised importance sampling (SIS) estimator* [Owen, 2013]

$$\hat{\mu}_S^{\text{SIS}} := \frac{\sum_{i=1}^{N} w^{(s)} f(\theta^{(s)})}{\sum_{s=1}^{S} w^{(s)}} \quad \text{where} \quad w^{(s)} = \frac{p(\mathbf{y}, \theta^{(s)})}{q(\theta^{(s)})}, \theta^{(s)} \sim q(\cdot). \tag{2.15}$$

Note that the weights themselves do not depend on the target function f, hence we can generate a set of samples and compute their corresponding weights once and then re-use the weighted samples for different target functions. This then provides us with our first inference algorithm that is summarised in Algorithm 1. All we need to do is sample from the proposal distribution, compute the importance weights, and then normalise the weights to obtain the weighted samples. Crucially, the sampling from the proposal can be done in parallel which can be done very efficiently on modern hardware.

It is important to note that the SIS estimator is generally biased and there is a fundamental lower limit on the variance that can be achieved by the estimator, even with an optimal proposal that has access to the true value of μ, we will refer to Owen [2013] for a more detailed discussion of this topic. In Chapter 6 we will discuss a mechanism that overcomes this fundamental limit.

Algorithm 2 Metropolis-Hastings

Require: Joint distribution p, number of samples N, initial point θ_0

for $s = 1, \ldots, S$ **do**
  New proposal $\theta' \sim \kappa(\cdot \mid \theta^{(s-1)})$
  $\alpha = \min\left(1, \frac{p(\theta', \mathbf{y})\kappa(\theta^{(s-1)} \mid \theta')}{p(\theta^{(s-1)}, \mathbf{y})\kappa(\theta' \mid \theta^{(s-1)})}\right)$
  Sample $u \sim \text{Uniform}(0, 1)$
  if $u < \alpha$ **then**
    $\theta^{(s)} \leftarrow \theta'$
  else
    $\theta^{(s)} \leftarrow \theta^{(s-1)}$
  end if
end for
Return $\{\theta^{(1)}, \ldots, \theta^{(S)}\}$

2.3.2 Markov Chain Monte Carlo

Importance sampling requires the construction of a proposal distribution $q(\theta)$ that we can easily sample from. Additionally, as the dimensionality of the parameter space increases it becomes significantly more difficult to construct a suitable proposal distribution. The performance of importance sampling tends to degrade exponentially with the dimensionality of the parameter space, a phenomenon known as the *curse of dimensionality*. In high-dimensional spaces, probability distributions often behave in unintuitive ways. For example, most of the probability mass of a high-dimensional Gaussian distribution is concentrated in a thin shell away from the mode in an area that is known as the *typical set* [Betancourt, 2018].

Markov chain Monte Carlo (MCMC) methods [Robert and Casella, 2004, Owen, 2013, Brooks et al., 2011] aim to overcome this challenge by instead making *local moves* that try to find and then explore the typical set of the target distribution. Formally, MCMC methods are a general class of algorithms that aim to generate samples from a target distribution by constructing a Markov chain which has the target distribution as its stationary distribution. More precisely, a Markov chain is a sequence of random variables $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(S)}$ which satisfy the *Markov property*, meaning the distribution of $\theta^{(s)}$ only depends on $\theta^{(s-1)}$ and not on any previous values

$$p(\theta^{(s)} \mid \theta^{(1)}, \ldots, \theta^{(s-1)}) = p(\theta^{(s)} \mid \theta^{(s-1)}). \tag{2.16}$$

In the context of MCMC, the distribution $p(\theta^{(s)} \mid \theta^{(s-1)})$ is conventionally referred to as the *transition kernel* and we will denote it as $\kappa(\theta^{(s)} \mid \theta^{(s-1)})$ from here on. Hence, to specify a (time-homogenous) Markov chain we only need to specify an initial distribution $p(\theta^{(1)})$ and a transition kernel $\kappa(\theta^{(s)} \mid \theta^{(s-1)})$. Overall, our goal

is to construct Markov chains so that $\theta^{(s)}$ is distributed according to the posterior $p(\theta \mid \mathbf{y})$ as $s \to \infty$. This is to say, we want the unique *stationary distribution* of the Markov chain to be the posterior distribution. Two requirements need to be satisfied for this to hold. The first is that the posterior distribution is invariant under the application of the transition kernel which is formalised as

$$p(\theta \mid \mathbf{y}) = \int_{\Theta} p(\theta' \mid \mathbf{y}) \kappa(\theta \mid \theta') d\theta. \tag{2.17}$$

The second requirement is that the Markov chain is *ergodic* which means that it is *irreducible* and *aperiodic*. A Markov chain is irreducible if from an arbitrary starting state we can reach any other state and it is aperiodic if there is no state which can only be reached at certain times.

Now that we have outlined the basic requirements that we want our Markov chain to satisfy, we can think about how we can construct one in practice. A common mechanism to satisfy the invariance property is to ensure that the transition kernel adheres to *detailed balance*, i.e.

$$p(\theta' \mid \mathbf{y}) \kappa(\theta \mid \theta') = p(\theta \mid \mathbf{y}) \kappa(\theta' \mid \theta). \tag{2.18}$$

Detailed balance is a sufficient but not necessary condition to satisfy the invariance property. MCMC algorithms that do not satisfy detailed balance are known as *non-reversible* [Bierkens et al., 2016, Bouchard-Côté et al., 2018, Andrieu and Livingstone, 2019, Syed et al., 2021, Bierkens et al., 2020].

The Metropolis-Hastings (MH) [3] algorithm [Metropolis et al., 1953, Hastings, 1970] provides the foundation for the development for a wide range of MCMC algorithms. The high-level idea is to propose a new state θ' from the current state θ, compute the *acceptance probability*

$$\alpha = \min \left(1, \frac{p(\theta', \mathbf{y}) \kappa(\theta \mid \theta')}{p(\theta, \mathbf{y}) \kappa(\theta' \mid \theta)} \right), \tag{2.19}$$

then sample a random number $u \sim \text{Uniform}(0, 1)$ and, finally, accept the proposal if $u < \alpha$. Algorithm 2 provides an outline of the MH algorithm. It is possible to show that the transition kernel of the MH algorithm satisfies detailed balance and, hence, the invariance property (see e.g. Section 11.4 Owen [2013]).

[3] An early version of the algorithm assuming symmetric proposals was presented in Metropolis et al. [1953] and a more general version was presented in Hastings [1970]. The current conventional naming scheme of the algorithm omits the other co-authors on the original 1953 paper, namely Arianna Rosenbluth, Marshall Rosenbluth, Augusta Teller, and Edward Teller. Some significant discussion about who should receive credit for the original work can be found in Gubernatis [2005].

The sequence of samples $\theta^{(1)}, \theta^{(2)}, \ldots, \theta^{(S)}$ generated by an MCMC algorithm can be used to estimate expectations with a Monte Carlo estimator. However, the samples are only distributed according to the target distribution in the infinite limit. In practice, we are always restricted to a finite number of samples, so the MCMC samples will only be approximately distributed according to the target distribution and the initial samples tend to be a poor representation of the target. Additionally, consecutive samples are correlated as they are generated based on local moves. These issues can introduce biases into our estimates and measures to overcome these complications lead to the usage of *burn-in* periods, where a fixed number of initial samples of the Markov chain are discarded from the estimator, and *thinning*, where only every k-th sample is retained to reduce the correlation between samples.

There is much more to say about MCMC methods, including the development of more advanced algorithms, such as Hamiltonian Monte Carlo (HMC), and the more detailed theoretical properties of these algorithms. However, in the interest of brevity and conciseness we will refer the interested reader to the existing literature [Robert and Casella, 2004, Owen, 2013, Brooks et al., 2011].

2.3.3 Variational Inference

MCMC methods are robust and reliable inference algorithms for many models encountered in practice. However, they can struggle to converge quickly to the target distribution, especially in problems with large datasets. An alternative approach to inference is to instead recast the inference problem as an optimization problem, this is known as *variational inference (VI)* [Jordan et al., 1999, Wainwright et al., 2008, Blei et al., 2017]. In VI, we postulate a *variational family* $q(\theta; \phi)$ with *variational parameters* ϕ and then aim to optimize the variational parameters such that the variational distribution is close to the true posterior distribution. The key idea in VI is that the variational family consists of distributions that we are able to sample from and can evaluate the density of.

A common measure to quantify the "closeness" between the two distributions is the KL divergence. VI algorithms aim to minimize the *reverse* or *exclusive* KL[4]

$$\mathrm{KL}\left(q(\theta;\phi) \parallel p(\theta \mid \mathbf{y})\right) = \mathbb{E}_{q(\theta;\phi)}\left[\log \frac{q(\theta;\phi)}{p(\theta \mid \mathbf{y})}\right]. \tag{2.20}$$

[4]Inference algorithms which aim to minimize the forward or inclusive KL, $\mathrm{KL}(p \parallel q)$ are generally not referred to as VI algorithms but are known under other names. Prominent examples include expectation propagation [Minka, 2001], reweighted wake-sleep [Bornschein and Bengio, 2015], and Markovian score climbing [Naesseth et al., 2020].

however, as the posterior is not tractable, we cannot use this form of the KL divergence to optimize the variational parameters. Instead, we can apply Bayes rule to rewrite the KL divergence as

$$= \mathbb{E}_{q(\theta;\phi)}\left[\log \frac{q(\theta;\phi)p(\mathbf{y})}{p(\mathbf{y},\theta)}\right] \tag{2.21}$$

$$= \mathbb{E}_{q(\theta;\phi)}\left[\log \frac{q(\theta;\phi)}{p(\mathbf{y},\theta)}\right] + \log p(\mathbf{y}). \tag{2.22}$$

Note that the first term in Eq. (2.22) only needs access to the joint probability density $p(\mathbf{y}, \theta)$ which is tractable, while the second term does not depend on the variational parameters at all. Hence, we can see that minimizing the KL divergence is equivalent to maximizing the negative of the first term in Eq. (2.22) which is known as the *evidence lower bound (ELBO)* The name stems from the fact that if we re-arrange Eq. (2.22) and use the fact that the KL divergence is non-negative, we obtain

$$\log p(\mathbf{y}) \geqslant -\mathbb{E}_{q(\theta;\phi)}\left[\log \frac{q(\theta;\phi)}{p(\mathbf{y},\theta)}\right] = \mathbb{E}_{q(\theta;\phi)}\left[\log \frac{p(\mathbf{y},\theta)}{q(\theta;\phi)}\right] =: \mathcal{L}(\phi). \tag{2.23}$$

It is important to pause for a second and reason about the properties of the ELBO. The inequality becomes tight if the variational distribution is equal to the true posterior, i.e. $\mathrm{KL}\left(q(\theta;\phi) \parallel p(\theta \mid \mathbf{y})\right) = 0$. Additionally, if the posterior is multi-modal and the variational family is unimodal then the variational distribution will tend to collapse onto a single mode instead of trying to cover all modes. This is known as the *mode-seeking* behaviour of the ELBO [Blei et al., 2017] and is caused by the fact that the ELBO heavily penalises the variational distribution for placing mass in regions where the true posterior has low density. Finally, maximizing the ELBO is, apart from a few special cases, a *non-convex* optimization problem which means that there potentially exist multiple local optima.

Having established an objective for optimization, the next step lies in constructing a suitable variational family for our parameter space Θ which can be potentially high-dimensional. Constructing flexible, but still tractable, variational families in high-dimensional parameter spaces is an ongoing research challenge [Agrawal et al., 2020]. A common simplifying assumption is to construct a *mean-field* variational family which assumes that the parameters are independent, leading to the factorization[5]

$$q(\theta;\phi) = \prod_{i=1}^{D} q_i(\theta_i;\phi_i). \tag{2.24}$$

The mean-field variational family is practical because it allows us to derive efficient algorithms for optimizing the ELBO. An early example of this is coordinate ascent

[5]Here θ_i denotes the i-th component of the D-dimensional vector θ.

variational inference (CAVI) which updates each variational parameter in turn while keeping the others fixed [Bishop, 2006]. However, CAVI requires access to model dependent conditional distributions which can be difficult to derive for complex models. In general, for long the adoption of VI was limited by the need to develop custom variational families and optimization algorithms for each model and variational family combination which often involved tedious mathematical derivations [Jaakkola and Jordan, 1997, Ghahramani and Beal, 2000, Jordan et al., 1999, Blei and Lafferty, 2006, Braun and McAuliffe, 2010, Knowles and Minka, 2011, Blei et al., 2017].

Significant progress came from utilizing techniques from stochastic optimization [Robbins and Monro, 1951, Asmussen and Glynn, 2007] which leverages gradient estimates of the ELBO to optimize the VI parameters [Paisley et al., 2012, Nott et al., 2012, Wingate and Weber, 2013, Salimans and Knowles, 2013, Kingma and Welling, 2014, Rezende et al., 2014, Ranganath et al., 2014, Blei et al., 2017]. Given the objective $\mathcal{L}(\phi)$ and estimates $h_t(\phi)$ of the gradient $\nabla_\phi \mathcal{L}(\phi)$, we can iteratively update the variational parameters using the update rule

$$\phi_{t+1} = \phi_t + \rho_t h_t(\phi_t) \tag{2.25}$$

under certain regularity conditions on $\mathcal{L}(\phi)$ this update rule will converge to a local optimum of the ELBO if the learning rates ρ_t satisfy the Robbins-Monro conditions [Robbins and Monro, 1951]

$$\sum_{t=1}^{\infty} \rho_t = \infty, \quad \sum_{t=1}^{\infty} \rho_t^2 < \infty. \tag{2.26}$$

The usage of a Robbins-Monro learning rate schedule is motivated by its theoretical properties; however, in practice, other optimization algorithms, such as Adam [Kingma and Ba, 2015], are often used instead due to their empirical success. Stochastic optimization requires estimates of the gradient of the ELBO with respect to the variational parameters

$$\nabla_\phi \mathcal{L}(\phi) = \nabla_\phi \mathbb{E}_{q(\theta;\phi)} \left[\log \frac{p(\mathbf{y}, \theta)}{q(\theta; \phi)} \right]. \tag{2.27}$$

This is complicated by the fact that the expectation is itself defined with respect to the variational distribution; naively moving the differentiation operator inside the expectation and applying a Monte Carlo estimator would lead to an invalid estimator. Instead, we need specialised estimators for this problem. In fact, finding efficient estimators for the gradient of the ELBO which are applicable to wide

range of models and variational families is a key challenge in the development of VI algorithms. The two most common classes of estimators are the *score-function estimator* and the *pathwise gradient estimator* (see Mohamed et al. [2020] for an extensive discussion on these).

The *score-function estimator* [Kleijnen and Rubinstein, 1996], also sometimes referred to as the REINFORCE estimator [Williams, 1992], makes use of the identity $\nabla_\phi q(\theta;\phi) = q(\theta;\phi)(\nabla_\phi \log q(\theta;\phi))$ to rewrite the gradient of the ELBO as

$$\nabla_\phi \mathcal{L}(\phi) = \mathbb{E}_{q(\theta;\phi)}\left[(\nabla_\phi \log q(\theta;\phi))\,(\log p(\mathbf{y},\theta) - \log q(\theta;\phi))\right], \tag{2.28}$$

which permits the Monte Carlo estimate

$$\approx \frac{1}{S}\sum_{s=1}^{S}\left(\nabla_\phi \log q(\theta^{(s)};\phi)\right)\left(\log p(\mathbf{y},\theta^{(s)}) - \log q(\theta^{(s)};\phi)\right), \tag{2.29}$$

where $\theta^{(s)} \sim q(\theta;\phi)$. The term *black-box variational inference (BBVI)* was popularized by Ranganath et al. [2014] to describe the use of the score-function estimator in combination with stochastic optimization. A significant issue in practice is that the estimator in Eq. (2.29) has high variance which can make optimization difficult and a variety of variance reduction techniques usually need to be applied to the standard score-function estimator to make it practical [Paisley et al., 2012, Mnih and Gregor, 2014, Ranganath et al., 2014, Titsias and Lázaro-Gredilla, 2015, Mohamed et al., 2020].

As an alternative, the *pathwise gradient estimator*, also often referred to as the "reparameterization trick" [Rezende et al., 2014, Kingma and Welling, 2014], relies on reparameterizing the variational family, $q(\theta;\phi)$, such that we can sample from it by applying a deterministic transformation, $S_\phi(\epsilon)$, to an independently sampled noise variable, $\epsilon \sim p(\epsilon)$, to rewrite the gradient of the ELBO as

$$\nabla_\phi \mathcal{L}(\phi) = \mathbb{E}_{p(\epsilon)}\left[\nabla_\phi \log \frac{p(\mathbf{y}, S_\phi(\epsilon))}{q(S_\phi(\epsilon);\phi)}\right]. \tag{2.30}$$

Notably, the pathwise gradient estimator generally exhibits significantly lower variance than the score-function estimator. Theoretical results show that the variance of the pathwise gradient estimator is bounded by the Lipschitz factor of $f(\theta) = \log(p(\mathbf{y},\theta)) - \log(q(\theta;\phi))$ but does not depend on the dimensionality of the parameter space [Glasserman, 2004], explaining its success in models with high-dimensional parameter spaces [Rezende et al., 2014, Kingma and Welling, 2014]. Additionally, for a specific variational parameter ϕ_i, by differentiating the model density, $p(\mathbf{y},\theta)$, the pathwise gradient estimator will exclude all the terms

which are not influenced by ϕ_i [Mohamed et al., 2020] and will reduce the order of the terms in $f(\theta)$ [Xu et al., 2019].

As a result, the pathwise gradient estimator is then often the preferred choice in practice, especially since modern automatic differentiation (AD) tools [Baydin et al., 2018] make it easy to compute the required gradients. However, the pathwise gradient estimator requires additional assumptions about the model and variational family compared to the score-function estimator. Most notably, we require access to the gradients of the joint model density with respect to the latents, $\nabla_\theta \log p(\mathbf{y}, \theta)$; that the joint density is differentiable in the whole support of θ; and that we can reparameterize the variational family in the first place (see Mohamed et al. [2020] for a more precise statement on the assumptions required for the pathwise gradient estimator). The score function estimator is then mostly used as a fallback when the pathwise gradient estimator is not applicable, and a significant amount of research has been dedicated to broaden the class of models for which the pathwise gradient estimator is applicable [Lee et al., 2018, Figurnov et al., 2018, Jankowiak and Obermeyer, 2018].

A particular instantiation of the general framework of using the pathwise gradient estimator to train a variational approximation is given by the *automatic differentiation variational inference (ADVI)* algorithm [Kucukelbir et al., 2017]. ADVI considers a restricted class of models known as *differentiable probability models* which are assumed to have continuous parameters $\Theta \subseteq \mathbb{R}^D$ and a differentiable joint density $\nabla_\theta \log p(\mathbf{y}, \theta)$ for all $\theta \in \Theta$.

To fully automate the construction of the variational family, ADVI transforms the constrained parameters into an unconstrained space using a differentiable transformation $T : \Theta \to \mathbb{R}^D$. The construction of the transformation can be fully automated from the model specification and does not require any user input. It is constructed by considering the support of individual parameters in the parameter space. For example, if parameter θ_i is defined with a Gamma prior, it is constrained to be positive, and to transform it to an unconstrained space we can apply the transformation $T_i(\theta_i) = \log(\theta_i)$. A pre-specified library of transformations indicates how to map parameters with restricted support to an unconstrained space. The transformations from individual variables are then combined to form the full transformation $T(\theta) = (T_1(\theta_1), \ldots, T_D(\theta_D))$.[6] Applying a change of variables

[6]The full procedure also easily generalizes to the setting where a subset of the parameters is assumed to have multivariate priors, e.g. a Dirichlet distribution.

using this transformation defines a new probability density in the unconstrained parameter space, $\zeta \in \mathbb{R}^D$, given by

$$p(\mathbf{y}, \zeta) = p(\mathbf{y}, T^{-1}(\zeta))|\det\left(J_{T^{-1}}(\zeta)\right)|. \tag{2.31}$$

where $J_{T^{-1}}(\zeta)$ is the Jacobian of the transformation T^{-1}.

This unconstrained model density now allows us to define a variational family in the unconstrained space, $\zeta \in \mathbb{R}^D$, which is significantly easier than working in the constrained space Θ. Notably, it is now easier to induce correlations into the variational family as it is possible to use a multivariate normal distribution as the variational family. The multivariate normal distribution also allows for the application of the type of reparameterization that is necessary to apply the pathwise gradient estimator. Specifically, if our variational family is given by $q(\zeta; \phi) = \mathcal{N}(\zeta; \mu, \Sigma)$, where the variational parameters are given by $\phi = (\mu, \Sigma)$, we can sample from it by first sampling from a standard normal distribution $\epsilon \sim p(\epsilon) = \mathcal{N}(0, \mathbf{I})$ and then apply the transformation $S_\phi(\epsilon) := \mu + L\epsilon$ where L is the Cholesky decomposition of the covariance matrix $\Sigma = LL^\top$. Notably, because the transformation T is potentially non-linear, the variational family in the true parameter space Θ can be non-gaussian. Notably, the transformation T does not contain any additional trainable parameters that need to be considered during the optimization process. The only parameters that are being optimized are the parameters of the variational distribution in the unconstrained space (i.e. $\phi = (\mu, \Sigma)$).[7]

Using the two transformations $T^{-1}(\zeta)$ and $S_\phi(\epsilon)$, along with the law of the unconscious statistician, we can then rewrite the ELBO as an expectation under the noise distribution

$$\mathcal{L}(\phi) = \mathbb{E}_{p(\epsilon)}\left[f(\epsilon, \phi)\right]. \tag{2.32}$$

where

$$f(\epsilon, \phi) := \log p(\mathbf{y}, T^{-1}(S_\phi(\epsilon))) + \log\left|\det\left(J_{T^{-1}}(S_\phi(\epsilon))\right)\right| - \log q(S_\phi(\epsilon); \phi). \tag{2.33}$$

Crucially, the noise distribution does not depend on the variational parameters, allowing us to interchange the gradient and expectation operators. We can then estimate the gradient of the ELBO using the pathwise gradient estimator as

$$\nabla_\phi \mathcal{L}(\phi) = \mathbb{E}_{p(\epsilon)}\left[\nabla_\phi f(\epsilon, \phi)\right], \tag{2.34}$$

$$\approx \frac{1}{S}\sum_{s=1}^{S} \nabla_\phi f(\epsilon^{(s)}, \phi) \quad \text{where} \quad \epsilon^{(s)} \sim p(\epsilon). \tag{2.35}$$

[7] There are other approaches to constructing variational families that do train the transformation parameters such as normalizing flows [Papamakarios et al., 2021] but we will not discuss them further here.

Algorithm 3 Automatic Differentiation Variational Inference (ADVI)

Require: Joint distribution p, variational family q, number of iterations I, initial parameters $\phi^{(0)}$, Robbins-Monro learning rate schedule ρ_i

1: Construct transformation T
2: **for** $i = 1, \ldots, I$ **do**
3: Draw S samples $\epsilon^{(s)} \sim p(\epsilon)$
4: Compute Monte Carlo estimate of $\nabla_\phi \mathcal{L}(\phi^{(i-1)})$ using Eq. (2.35)
5: $\phi^{(i)} \leftarrow \phi^{(i-1)} + \rho_i \nabla_\phi \mathcal{L}(\phi^{(i-1)})$
6: **end for**
7: Return $\phi^{(I)}$

This then provides us with all the necessary ingredients for the ADVI algorithm which is stated in Algorithm 3. The algorithm iteratively optimizes the variational parameters using the pathwise gradient estimator and a Robbins-Monro learning rate schedule (although other choices for the optimizer are also often used in practice). While the version presented here runs optimizations for a fixed number of steps it is also possible to run optimization until a desirable convergence criteria is reached.

A major obstacle for scaling Bayesian inference to large datasets is the cost of evaluating the model likelihood, $p(\mathbf{y} \mid \theta)$, which is also required for the ELBO. The variational inference approach can more easily scale to large datasets through data subsampling [Hoffman et al., 2013] by noting that the data only appears in the ELBO through the likelihood. Then assuming the likelihood factorizes as $\log p(\mathbf{y} \mid \theta) = \sum_{i=1}^{N} \log p(y_i \mid \theta)$, it can be approximated by sampling a subset of the data of size B and then scaling the log-likelihood by $\frac{N}{B}$ s.t. $\log p(\mathbf{y} \mid \theta) \approx \frac{N}{B} \sum_{i \in \mathcal{I}_B} \log p(y_i \mid \theta)$. This leads to an optimization that has been termed "doubly" stochastic [Titsias and Lázaro-Gredilla, 2014] as there are two sources of stochasticity: the Monte Carlo estimate of the ELBO and the data subsampling.

This section only provided a brief overview of the key ideas behind VI. There are still many areas of open research, such as how to construct flexible variational families that are still tractable while being able to capture complex posterior distributions [Rezende and Mohamed, 2015, Papamakarios et al., 2021, Morningstar et al., 2021, Ambrogioni et al., 2021, Ko et al., 2024, Klein et al., 2024]; deriving more efficient gradient estimators [Mnih and Gregor, 2014, Schulman et al., 2015, Maddison et al., 2016, Jang et al., 2016, Oates et al., 2017, Roeder et al., 2017, Lee et al., 2018, Shi et al., 2022]; or considering alternative divergence measures [Li and Turner, 2016, Bamler et al., 2017, Lopez et al., 2020, Geffner and Domke, 2020, Welandawe et al., 2022, Yi and Liu, 2023, Giordano et al., 2024]. For recent reviews of the advances in the field we refer the reader to Zhang et al. [2018], Agrawal et al. [2020], and Dhaka et al. [2021].

2.4 Justifications for the Bayesian Approach

So far we have focused on the Bayesian approach to statistics and decision making. In this section we will briefly discuss the Frequentist approach which is the most prominent alternative to the Bayesian viewpoint. The debate about which approach is preferable has been going on for decades and is still ongoing. Here, we only scratch the surface the debate but it should be noted that by now it is not uncommon for practitioners to mix ideas from Frequentists and Bayesian statistics in their work.

In our introductions to probabilistic modelling and decision making in Sections 2.1 and 2.2 we used probabilities to quantify our beliefs about the unknown model parameters. However, the Frequentist approach to statistics disagrees with this usage of probability. Instead, it argues that probabilities should only be used to quantify the frequency of events [Von Mises, 1981]. Hence, we cannot use probabilities to quantify our beliefs about unknown model parameters, as these are non-observables and therefore their frequentist properties cannot be measured directly. In a pure Frequentist framework, uncertainty stems from the fact that we only get to observe one realisation of the data and uncertainty then measures how our inferences would change if the data would change.

The Frequentist approach is then often characterized by evaluating estimators based on the *Frequentist risk*

$$R(\theta, \delta) = \mathbb{E}_{p(\mathbf{y}|\theta)} \left[\mathsf{L}(\theta, \delta(\mathbf{y})) \right]. \tag{2.36}$$

Crucially, the Frequentist philosophy of quantifying uncertainty by variation in the observed data is signified by now taking the expectation over the data $\mathbf{y}$, instead of the parameters θ. However, to evaluate the risk we somehow need to set the parameters θ. The Frequentists approach does not prescribe a unique way to select the parameter, instead a variety of methods are employed in practice. The most common approaches include plugging in a specific value θ^*, finding the parameter setting which maximizes the risk R (leading to minimax estimators), or averaging over possible parameter settings using a prior distribution $p(\theta)$ (leading to the Bayes risk). These techniques then provide mechanisms to derive and evaluate estimators for the parameters θ and measure the uncertainty in the estimates by constructing confidence intervals (CIs) on the estimates [Casella and Berger, 2001].

Compared to the Bayesian approach, the Frequentist approach does not prescribe a unique mechanism for using the estimated parameters to make decisions. Instead, different approaches to decision making are often used depending on the specific problem setting. Within the context of supervised learning, arguably the dominant

approach is based on *empirical risk minimization (ERM)*. In supervised learning we have a set of features $\mathbf{x}$ and a set of labels $\mathbf{y}$, and we aim to learn a decision function $f(x)$ which maps the features to the labels. In the ERM setting, we define the *population risk* [Murphy, 2022] as

$$R(f) = \mathbb{E}_{p_{\text{true}}(y,x)} \left[\mathsf{L}(y, f(x)) \right], \tag{2.37}$$

where the expectation is taken with respect to *the true data generating distribution*, p_{true}. In practice, this distribution is unknown and we instead estimate the population risk using the *empirical risk*

$$R(f) = \frac{1}{N} \sum_{n=1}^{N} \mathsf{L}(y_n, f(x_n)). \tag{2.38}$$

The decision function f is then optimized based on this empirical risk (hence, the name empirical risk minimization)

$$\hat{f} = \underset{f \in \mathcal{H}}{\operatorname{argmin}} \, R(f), \tag{2.39}$$

where $\mathcal{H}$ is the hypothesis class containing all the possible functions. Notably, this is a fundamentally different approach to the decision making task then the one we presented in Section 2.2. ERM does not build an explicit model of the data generating process and instead directly optimizes the decision function based on the collected data. In contrast, Bayesian approaches generally try to model the data generating process explicitly and then use the model to optimize our decisions. However, it is important to note that Bayesian models might not specify a model for all the variables in the dataset. For example, in a supervised setting the features $\mathbf{x}$ are often not modelled as random but rather are treated as a fixed input to the model.[8]

In ERM, we tend to split the data into training and validation set. The training set is used to optimize the decision function. Consequently, the empirical risk evaluated on the training set is no longer a good estimate for the population risk because the decision function has been optimized to minimize it. Hence, the validation set is then used to provide an estimate of the population risk. If our overall dataset is small, it might not be feasible to have a validation set that is

[8] This also explains why the Frequentist vs Bayesian distinction is not the same as the generative vs discriminative distinction in machine learning. In their purist instantiations, the Frequentist and Bayesian approaches disagree in how to estimate model parameters θ. For example, an autoregressive language model [Vaswani et al., 2017] is a generative model, however, the parameters of the autoregressive model are still optimized based on a loss defined on a training data set. A Bayesian neural network [MacKay, 1995] for a supervised classification task could be viewed as a discriminative model, however, crucially the parameters of the neural network are computed by approximating the posterior distribution.

large enough to provide a reliable estimate of the population risk. The solution of *cross-validation* is then to have multiple training and validation data splits and then average the empirical risk over these splits.

In a purely Bayesian framework there is no need for cross-validation as our model fully encapsulates our beliefs and uncertainty.[9] There is also no concept of a "wrong" model. Our prior and likelihood encode our subjective belief about the state of the world and our beliefs get updated through the rules of Bayesian inference as new data arrives.

However, this view clearly has some practical limitations. In general, it is quite challenging for humans to be able to build mathematical probability models which accurately represent their beliefs and be aware of all the consequences of their modelling choices. Furthermore, the inferences implied by the laws of probability can itself sometimes lead to counter-intuitive results [Gelman and Yao, 2020]. Hence, a more pragmatic approach to Bayesian modelling acknowledges the need to validate a models predictions and inferences on held-out data [Gelman et al., 2013]. In the "hypothetico-deductive" view of Bayesian modelling [Gelman and Shalizi, 2013], the decision theoretic framework from Section. 2.2 is not just used to make final decision but also to check the adequacy of models and their assumptions. Crucially, all parts of a model should be checked, including the prior distribution, the likelihood function, and the decision function.

Given all the potential pitfalls with building probabilistic modelling, one might ask: if we can just directly optimize decisions using observed data with the ERM framework, why bother building models at all and why should we use *probabilistic* models?

There are many answers to the second questions, which are concisely summarised in Bruinsma et al. [2021]. Notably, there are multiple desirable properties we can list that a given modelling framework should have—e.g. inference should be *coherent/consistent* meaning that inferences should not depend on the order the data arrives in—and we can show that the laws of probability satisfies these properties [Cox, 1946, Savage, 1972, Jaynes, 2003]. Probabilistic models also have desirable frequentist properties if we assume that our postulated model matches the true data generating distribution: the posterior will asymptotically concentrate on the true parameter if we assume that data was generated from the model [Van der Vaart, 2000] and we can make optimal predictions made using the posterior predictive distributions [Aitchison, 1975].

[9] Interestingly, there are deep connections between cross-validation and the marginal likelihood in a Bayesian model, see e.g. Fong and Holmes [2019].

We should build models because they can help us understand the world. In science and engineering, our understanding of the world is mainly driven by developing idealised, but ever more elaborate, models of the world. The mathematical formalization of probability then provides a convenient mechanism to introduce uncertainties into our models and the Bayesian framework allows us to easily incorporate this uncertainty into our decision making process. Furthermore, by basing decisions based on probabilistic models we can easily trace back our decisions to the assumptions we made in our models. This in turn helps us to improve our models and our understanding of the world.

The Bayesian approach to statistics can then be a useful formalism to build and use probabilistic models but we should not be dogmatic about it. A view that was succinctly summarised in Rubin [1984]: "The applied statistician should be Bayesian in principle and calibrated to the real world in practice. [They] should attempt to use specifications that lead to approximately calibrated procedures under reasonable deviations from [their assumptions]. [They] should avoid models that are contradicted by observed data in relevant ways — frequency calculations for hypothetical replications can model a model's adequacy and help to suggest more appropriate models."

3

Probabilistic Programming Systems

As we showed in the previous chapter, Bayesian statistics hinges on the ability to build expressive models and to run inference in them. Probabilistic programmings systems (PPSs) [Gordon et al., 2014, van de Meent et al., 2018, Barthe et al., 2020] facilitate this process by providing programming abstractions that enable users to express probabilistic models with programming languages. Fundamentally, there are multiple reasons of why we might want to use programming languages to express probabilistic models. A non-exhaustive list of motivations includes:

1. In most models used in practice, we need to resort to approximate inference algorithms of the sort presented in Section 2.3. However, implementing these algorithms is often quite complex and can be error-prone. Applied modellers are often not experts in implementing inference algorithms, so we want to provide them with mechanisms to *automate inference* without them having to implement their own inference algorithm.

2. Programming languages provide a *flexible abstraction layer to express models*. Language constructs such as loops, recursion, and higher-order functions allow us to concisely define models that would be cumbersome to express in more classical modelling frameworks.

3. Many *models are already expressed as stochastic simulators*. Fields as diverse as climate science [Eyring et al., 2016], particle physics [Baydin et al., 2020], and drug discovery [Schymkowitz et al., 2005] develop sophisticated simulators which encode decades of domain knowledge. We want to be able to use these simulators as probabilistic models in a Bayesian framework to have access to more expressive models when guiding our decision making.

4. By using programming languages to express models, we open up the possibility of using techniques from programming languages research to derive new inference algorithms.

Of course, these motivations are not mutually exclusive, however, they are important to understand the different design trade-offs that are made in the process of developing PPSs. In general, PPSs provide the user with a *modelling language* to denote probabilistic models and a library of functions to run inference in these models.

The separation of modelling and inference is an important aspect of most PPSs. In principle, in a Bayesian framework it does not matter what inference algorithm we use to make decisions, as long as our inferences are accurate. In practice, however, the choice of algorithm can often have an influence on our decision making because approximate inference algorithms might fail to give good approximations to the posterior. Further, for computational reasons it can sometimes be beneficial to break the abstraction barrier between modelling and inference and optimize the definition of a model for a specific inference algorithm. Overall, however, the separation of modelling and inference still provides a useful distinction as it allows modeller to focus on building good models and inference algorithm developers to focus on developing better inference algorithms.

Early PPSs were generally focused on automating a specific inference algorithm which then guided the design of the modelling language. Crucially, many inference algorithms developed in the statistics community tend to make assumptions on the form of the model. For example, ADVI (Section 2.3.3) assumes that the model is differentiable and that we can compute the gradients of the log density. Hence, early PPSs tended to design the modelling language with restrictions in place to ensure that the specified model satisfies the assumptions of the inference algorithm. After Zhou [2020], we will refer to these sorts of PPSs as *restricted PPSs*.

In contrasts, *universal PPSs* interpret arbitrary stochastic simulator models expressed in general-purpose programming languages as probabilistic models and lift the constraints of restricted PPSs. Crucially, universal PPSs give rise to a whole new class of probabilistic models which can invalidate a lot of the assumptions made by conventional inference algorithms. One important example of this is the ability to define models with stochastic control flow which leads to *models with stochastic support*; a class of models we will describe in more detail in Section 3.1.3.

It is important to note that this chapter is not meant as an extensive review of all PPSs and their different implementation approaches. Instead, we will focus on the essential ideas behind probabilistic programming that are necessary for

the remaining chapters. Relatedly, this thesis is concerned with statistical PPSs. There is also a growing body of work on logic-based PPSs [De Raedt et al., 2007, De Raedt and Kersting, 2008, Fierens et al., 2015, Manhaeve et al., 2018] but they will not be discussed here. While they are also based on Bayesian reasoning, they often have quite considerable differences in their underlying theory and algorithms, leading to different design decisions in practice.

3.1 From Probabilistic Programs to Probabilistic Models

Just as there are many different design choices in the development of standard programming languages, there are also many different design choices in the development of PPSs. A crucial decision to make for every PPS is whether to develop a custom domain-specific language (DSL) for expressing models or to embed the PPS into an existing programming language.

Many PPSs that develop a DSL are designed to easily compile a model to an *internal representation* for inference. Notably, having a purpose-built DSL and compiler allows the PPS to optimize the internal representation for the given inference algorithm. Furthermore, the DSL allows the PPS to easily enforce restrictions on the class of models users can define.

As a specific example of a PPS with a DSL, Stan [Carpenter et al., 2017] allows users to specify differentiable continuous probability models (as defined in Section 2.3). Models in Stan are defined using an imperative DSL that gets compiled to a C++ program that is able to compute the log joint probability density of the model and evaluate its gradients through the use of reverse-mode automatic differentiation [Carpenter et al., 2017]. The focus on differentiable probability models means that Stan does not support models with discrete latent parameters, unless they are explicitly marginalized out. The Stan program for the logistic regression model from Section 2.1 is shown in Figure 3.1.

While using a custom DSL to define models can be beneficial for enforcing model constraints, it can also be a barrier of entry for new users because they will have to learn an entirely new programming language. Another option is to embed the PPS into an existing programming language, either through a library or language extension. This is the approach taken by languages such as Turing [Ge et al., 2018], Gen [Cusumano-Towner et al., 2019], Anglican [Wood et al., 2014], Pyro [Bingham

```
data {
  int<lower=0> N;
  vector[N] x;
  array[N] int<lower=0, upper=1> y;
}
parameters {
  real alpha;
  real beta;
}
model {
  y ~ bernoulli_logit(alpha + beta * x);
}
```

Figure 3.1: Logistic regression in Stan.

```
def model(x, y):
  alpha = pyro.sample("alpha", dist.Normal(0, 1))
  beta = pyro.sample("beta", dist.Normal(0, 1))
  logits = torch.sigmoid(alpha + beta * x)
  with pyro.plate("data", x.shape[0]):
    pyro.sample("y", Bernoulli(logits=logits), obs=y)
```

Figure 3.2: Logistic regression in Pyro.

et al., 2019], PyMC [Salvatier et al., 2016], Edward [Tran et al., 2016], PyProb [Baydin et al., 2020], and ProbTorch [Siddharth et al., 2017].[1]

Crucially, by embedding the PPS into an existing programming language, the PPS can leverage the ecosystem of libraries and tools from that language. For example, Pyro is built on top of PyTorch [Paszke et al., 2019] and can utilize its automatic differentiation capabilities to compute gradients. Furthermore, the user can easily use custom functions within a model to define more complex models. As we see in Figure 3.2, the logistic regression model in Pyro is defined using standard Python constructs and PyTorch functions, hence Pyro models are simply standard Python functions which internally make calls to the Pyro library.

3.1.1 Evaluation-Based Probabilistic Programming Systems

This thesis mainly focuses on *evaluation-based* PPSs which extend standard programming languages through two constructs: `sample` and `observe`.[2] In general,

[1]Note, that the distinction between a PPS with a DSL and a PPS embedded into an existing programming language is orthogonal from the distinction between restricted and universal PPSs we introduced earlier. For example, PyMC is a PPS embedded into Python that restricts the type of models users can define, whereas Venture is a universal PPS with a custom DSL.

[2]The reason we focus on the evaluation-based approach in more detail here, as opposed to e.g. the compilation-based approach taken by Stan, is because it is arguably the dominant approach

the `sample(addr, dist)` statement takes as input a lexical *address*, `addr`, and a *distribution object*, `dist`; allowing users to define random variables in the program. Similarly, the `observe(addr, dist, value)` statement takes as input a lexical address, a distribution object and additionally a *value*; allowing users to *condition* the program on observed data.

Note, that not every language uses these exact names and syntax but we use them here to be consistent with prior work [Rainforth, 2017, van de Meent et al., 2018]. For example, as can be seen in Figure 3.2, Pyro, instead of providing an explicit separate `observe` primitive, overloads the `sample` primitive to take an additional `obs` argument which is used for conditioning. Overall, the ability to condition the program on observed data is the key property that distinguishes probabilistic programs from 'usual' programs [Gordon et al., 2014].

The exact behaviour of `sample` and `observe` can change based on the evaluation context in which they are used. In standard "forward execution", i.e. the program is executed without the goal of running inference, `sample` generates a random draw from the corresponding distribution object using a programming language's random number generator. On the other hand, the `observe` is either a no-op or simply returns the `value` passed to it as an argument. As we will see in the next section, in the context of running an inference algorithm, the behaviour of these primitives can change.

Importantly, in PPSs we assume that the `sample` statements *are the only source of randomness in the program.* If we fix the outcome of the sample statements then the behaviour of the program is assumed to be completely deterministic.

3.1.2 Programs with Static Support

We will now describe how probabilistic programs in evaluation-based PPSs implicitly define an unnormalized density function. In this section, we first focus on describing the densities defined by programs with *static support*, which we define as programs that do not branch on the outcomes of `sample` statements. On a high-level, a probabilistic program implicitly defines an unnormalized density function through its use of `sample` and `observe`. As the program gets executed, each encounter of a `sample` or `observe` statement adds another term to the unnormalized density function.

To formalize this a bit further, let a_i be the address of the ith `sample` statement encountered during execution of the program, θ_i be the random draw generated

for universal PPSs and, notably, the approach taken by Pyro and Turing which we use in later chapters. Furthermore, while not every PPS uses the `sample` and `observe` constructs, they apply to a wide range of PPSs and are a useful abstraction to understand the core concepts of PPSs.

from the corresponding distribution object $g_a(\cdot \mid \eta_i)$, where η_i are the arguments passed to the distribution object. Similarly, let b_j denote the jth `observe` statement, y_j the value passed to it, and $h_{b_j}(\cdot \mid \phi_j)$ the corresponding distribution object. The execution of a program then generates a *trace* $\theta_{1:n_\theta}$ and an *address path* $a_{1:n_\theta}$. We will also sometimes use the term *raw random draws* to refer to the values $\theta_{1:n_\theta}$. This allows us to define the *unnormalized density function*

$$\gamma(\theta_{1:n_\theta}) := \prod_{i=1}^{n_\theta} g_{a_i}(\theta_i \mid \eta_i) \prod_{j=1}^{n_y} h_{b_j}(y_j \mid \phi_j). \tag{3.1}$$

Note, that we use the term 'density function' in a loose sense here. In general, probabilistic programs denote measures (or kernels if there are free variables).[3] When we talk about the density function of a probabilistic program, formally we are referring to the Radon-Nikodym derivative of the measure denoted by this program with respect to an appropriate reference measure, where this reference measure is itself implicitly defined by the program.[4] We will not delve deeper into the measure-theoretic foundations of more formal semantic treatments of PPSs but instead refer the interested reader to Kozen [1979], Borgström et al. [2016], Staton et al. [2016], Lew et al. [2019], Barthe et al. [2020], and Mak et al. [2021a].

The property of static support manifests in the unnormalized density γ by having n_θ, g_{a_i}, n_y, and h_{b_j} fixed between different program executions. However, η_i and ϕ_j are random variables but are defined as a pushforward from the raw random draws. In this restricted setting, we can view `sample` statements as defining the prior on latent variables and `observe` statements as defining the likelihood terms.

Notably, some PPSs actually provide inference algorithms which assume that the model has static support without explicitly enforcing the constraint. For example, Pyro allows users to run HMC on user-defined models without validating that the model indeed has static support, leading to incorrect results if that assumption is violated [Mak et al., 2021b]. In Chapter 4, we show how this can also be problematic for Pyro's variational inference algorithms.

3.1.3 Programs with Stochastic Support

Higher-order programming languages are clearly a useful tool to specify models which is proven by the fact that there already exist many scientific models that

[3] For conciseness we avoid explicitly denoting the free variables of a program.

[4] In cases in which we want to explicitly denote the reference measure, we will denote it with $\mu(\theta_{1:n_\theta})$ and integration with respect to it as $\int \gamma(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta})$. However, for the sake of conciseness, we will mostly omit denoting it explicitly.

```
def model_with_stochastic_support():
  z0 = sample("z0", Normal(0, 2))
  if z0 < 0:
    z1 = sample("z1", Normal(5, 2))
    sample("z2", Normal(z1, 1), obs=[9])
  else:
    z1 = sample("z1", Normal(5, 2))
    z2 = sample("z2", Normal(z1, 2))
    sample("z3", Normal(z2, 1), obs=[9])
```

Figure 3.3: A simple Pyro program with stochastic support.

are expressed as stochastic simulators. Ideally, we would want to be able to use any program that is defined as a stochastic simulator as a probabilistic model in a Bayesian framework. This requires us to be able to not just deal with the simple programs with static support but also allow programs that utilize programming language features such as stochastic branching, loops, and recursion.

Universal PPSs aim to fully unleash this potential of using higher-order programming languages to specify probabilistic models. Examples of universal PPSs include Church [Goodman et al., 2008], WebPPL [Goodman and Stuhlmüller, 2014], Anglican [Wood et al., 2014, Tolpin et al., 2016], Venture [Mansinghka et al., 2014], Gen [Cusumano-Towner et al., 2019], Turing [Ge et al., 2018], Birch [Murray and Schön, 2018], PyProb [Baydin et al., 2020], ProbTorch [Siddharth et al., 2017], and Pyro [Bingham et al., 2019]. The development of universal PPSs has gone hand in hand with the development of novel inference algorithms which can handle all the intricacies of programs defined in universal PPSs.

One important class of problems that we will focus on are *programs with stochastic support*. These programs arise when languages permit branching based on the outcomes of `sample` statements. The branching introduces discontinuities into the density function defined by the program which complicates inference. Even more, different branches potentially contain different number of `sample` and `observe` statements. Hence, the number and type of latent variables and observed variables can change between different executions of the program.[5]

Figure 3.3 shows a simple example of a program with stochastic support. The program has two branches and which branch is taken depends on the outcome

[5]We chose the term *programs with stochastic support* partly to be consistent with prior work in the area [Zhou et al., 2020] and to emphasize that we are concerned with programs that have stochastic branching. While these programs are often written in universal PPSs, they can also be written in certain restricted PPSs—e.g. Zhou et al. [2019] present a first-order PPSs that allows for stochastic branching—which is why we use the qualification "with stochastic support" to describe this class of programs rather than "program in an universal PPS".

of the first `sample` statement. Crucially, the *traces of this program can have different lengths*; both $t_1 = [-0.3, 4.3]$ and $t_2 = [1.4, 8.2, 7.3]$ are valid traces of the program. It is instructive to go through the program step-by-step to see how the different traces are generated.

For t_1: the `sample` statement at line 2 generates a random draw -0.3 with address `"z0"`, the conditional statement at line 3 evaluates to true, the `sample` statement at line 4 generates a random draw 4.3 with address `"z1"`, and the program terminates without encountering any further `sample` statements.[6]

For t_2: the `sample` statement at line 2 generates a random draw 1.4 with address `"z0"`, the conditional statement at line 3 evaluates to false, the `sample` statement at line 7 generates a random draw 8.2 with address `"z1"`, the `sample` statement at line 8 generates a random draw 7.3 with address `"z2"`, and the program terminates without encountering any further `sample` statements.

However, note that not every trace of length 2 or 3 is valid. For example, $t_3 = [1.2, 3.3]$ is not a valid trace. Assuming the `sample` statement on line 2 generates a random draw 1.2 with address `"z0"`, the conditional statement at line 3 evaluates to false, the `sample` statement at line 7 generates a random draw 3.3 with address `"z1"`, but then there will be another random draw with address `"z2"` before the program terminates. Hence, the trace t_3 is incomplete and not a valid trace.

Naturally, a program's density should only be defined on *valid traces*, and we denote the domain of valid traces as Θ. Even though the length of the traces n_θ is now a random variable, we can still define the unnormalized density function, γ, of a program on the traces

$$\gamma(\theta_{1:n_\theta}) := \mathbb{I}\left[\theta_{1:n_\theta} \in \Theta\right] \prod_{i=1}^{n_\theta} g_{a_i}(\theta_i \mid \eta_i) \prod_{j=1}^{n_y} h_{b_j}(y_j \mid \phi_j). \tag{3.2}$$

This definition makes sense for a large class of programs, permitting stochastic branching, higher-order functions, and recursion [Rainforth, 2017, §4.3]. However, for this function to correspond to a valid unnormalized probability density we need to assume that a) the program halts with probability 1 and b) that the integral over the entire domain of γ with respect to the implicitly defined reference measure is finite, i.e. $0 < \int_\Theta \gamma(\theta_{1:n_\theta}) d\theta_{1:n_\theta} < \infty$.

While the definition of the density for programs with stochastic support (Eq. (3.2)) is seemingly similar to the one for programs with static support (Eq. (3.1)), their differences have a significant impact on the difficulty of the task of inference. All of

[6] To stay consistent with terminology in the rest of the thesis we will refer to Pyro's `sample` function with an observation passed in as an `observe` statement.

```
def model(y):
  num_clusters = pyro.sample(
    "num_clusters", dist.Poisson(self.poisson_rate)
  )
  num_clusters += 1 # avoid num_clusters = 0
  cluster_means = [
    pyro.sample(f"mean_{k}", dist.Normal(0, 10))
    for k in range(num_clusters)
  ]
  cluster_means = torch.stack(cluster_means)
  with pyro.plate("data", y.shape[0]):
    mix = dist.Categorical(logits=torch.ones(num_clusters))
    comp = dist.Normal(cluster_means, 0.1)
    pyro.sample("obs", dist.MixtureSameFamily(mix, comp), obs=y)
```

Figure 3.4: Mixture model with a Poisson prior on the number of components.

n_θ, n_y, a_i, b_j, g_{a_i}, and h_{b_j} are now random variables and no longer fixed constants.[7] Additionally, the indicator function $\mathbb{I}[\theta_{1:n_\theta} \in \Theta]$ constraining the density to valid traces introduces discontinuities into the density function.

Furthermore, programs with stochastic support are no longer restricted to having a finite upper bound on the length of the traces, leading to models with potentially infinite dimensional support. One of the simplest practical models with infinite dimensional support is a Gaussian mixture model (GMM) with a Poisson prior on the number of components. The Pyro code for such a model is shown in Figure 3.4. In the program, we place a Poisson prior on the number of components in the model and then sample the component means from a normal distribution.

Using models with infinite support is a powerful mechanism to specify models whose complexity increases with the amount of data. While not all models with stochastic support have infinite support, any inference algorithm for programs with stochastic support should consider how to deal with this use case to allow users maximum modelling flexibility. However, it also poses a significant challenge for the design of new inference algorithms, as we can no longer assume that the number of latent variables our inference algorithm needs to consider is fixed. It also makes it difficult to compile the program into an intermediate representation that the inference algorithm can operate on.

This is why many inference algorithms for universal PPSs take the evaluation-based approach we introduced earlier. The exact implementation details for how different PPSs control the behaviour of `sample` and `observe` statements varies

[7] Though, we still assume that all the randomness is generated through the `sample` statements and therefore n_θ, n_y, a_i, b_j, g_{a_i}, and h_{b_j} can all be computed as a pushforward from the raw random draws $\theta_{1:n_\theta}$ [Rainforth, 2017, §4.3].

between PPSs. To highlight a couple of approaches: Anglican [Wood et al., 2014, Tolpin et al., 2016] uses the continuation-passing-style (CPS) transform to be able to interrupt, fork, and resume the input program during execution; Pyro [Bingham et al., 2019] uses effect handlers [Plotkin and Power, 2001, Plotkin and Pretnar, 2009] which can be composed with each other; Turing [Ge et al., 2018] uses Julia's multiple dispatch system [Bezanson et al., 2017] to change the behaviour based on the type of inference algorithm used.

3.1.3.1 Differences to Bayesian Nonparametric Models

Readers well-versed in Bayesian statistics might raise the fact that Bayesian nonparametric (BNP) models already allow practitioners to define models with infinite dimensional parameter spaces and wonder how the BNP models differ from programs with stochastic support.

To construct models on infinite dimensional parameter spaces, BNPs expresses models as *stochastic processes*. When constructing a new BNP model, significant care needs to be taken in ensuring that the constructed model actually satisfies all the requirements to be a valid stochastic process and indeed defines a valid probability measure. Due to the significant effort that this requires, there are a limited number of well-studied, fundamental stochastic processes which most BNP models are based on. Among these are the *Dirichlet process* (DP) [Hjort et al., 2010], a stochastic process (amongst others) whose sample paths are discrete distributions, and the *Gaussian process* (GP) [Rasmussen and Williams, 2006], which defines a distribution over continuous functions.

The modelling flexibility of BNP models also comes with a number of challenges. One of these is that BNP models can be *non-consistent* [Diaconis and Freedman, 1986, Freedman, 1999, Grünwald and Langford, 2007], e.g. the posterior of a Dirichlet process mixture model will not concentrate on the true number of clusters in the data if the observed data is actually generated from a finite mixture model [Miller and Harrison, 2013]. Not all BNP models exhibit these non-consistencies but it shows that care needs to be taken when applying these models in practice. Additionally, most BNP models also require the derivation of custom inference algorithms, as the non-standard probability spaces that they are defined on make it difficult to apply standard inference algorithms.

At the same time, due to the flexibility of these models it is often not necessary for practitioners to develop custom BNP models for every new application. For example, GPs are widely used for regression and classification tasks and are often the

go-to model for modelling non-linear functions in the Bayesian statistics literature [Gelman et al., 2013].

So how do programs with stochastic support differ from BNP models then? Fundamentally, they represent two different mechanisms for expressing probabilistic models. As just described, BNP models are generally expressed by defining valid *stochastic processes* which tends to require a fair bit of sophisticated mathematical machinery. In contrast, programs with stochastic support are defined through specifying a stochastic simulator which in turn implicitly defines an *unnormalized joint density*.

This seemingly innocuous difference does indeed have practical implications. While many BNP models can be expressed as stochastic simulators, this is not true for all BNP models. Interestingly, BNP models need not be defined through a joint density and can instead be represented as a collection of conditional distributions; some BNP models might not even have a computable joint density [Orbanz and Teh, 2010]. On the flipside, if a model has a computable joint density, this does not necessarily imply that the model also has computable conditional distributions which has implications for probabilistic programming [Ackerman et al., 2019]. Furthermore, BNP is generally concerned with models that have infinite dimensional latent parameter spaces (hence the name 'nonparametric') while programs with stochastic support also encompass models with finite dimensional latent parameter spaces (which occur when the number of possible program paths in a program is finite).

Thus, neither are BNP models a subset of programs with stochastic support nor are programs with stochastic support a subset of BNP models. Both modelling frameworks have their respective strength and can be complementary to each other. Indeed, the intersection of these two fields has been a fruitful area of research (e.g. [Goodman et al., 2008, Roy et al., 2008, Roy, 2011, Saad et al., 2019]). Probabilistic programming is appealing because many practitioners are already specifying models as simulators, demonstrating that writing programs is a natural mechanism for practitioners to express models. Deriving inference algorithms for programs in universal PPSs then gives practitioners the ability to specify models in a familiar way without requiring them to learn new ways of constructing models.

3.2 Inference Algorithms for Probabilistic Programs with Stochastic Support

Eq. (3.2) specified the form of the unnormalized density defined by programs with stochastic support. The goal of inference is now to find a representation of the

distribution specified by the *normalized density*

$$\pi(\theta_{1:n_\theta}) := \frac{\gamma(\theta_{1:n_\theta})}{Z} \quad \text{where} \quad Z := \int_\Theta \gamma(\theta_{1:n_\theta})\, d\theta_{1:n_\theta}. \tag{3.3}$$

where we re-iterate again that the integration is done with respect to an implicitly defined reference measure, and that we assume that a valid probabilistic programg has a normalization constant that exists and is finite, i.e. $0 < Z < \infty$.

The high-level approaches to inference we introduced in Section 2.3 will still be relevant for programs with stochastic support, although, the algorithms need to be adapted to deal with the new problem setting.

3.2.1 Importance Sampling

As we explained in Section 2.3, importance sampling (IS) is one of the most elementary inference algorithms. As a reminder, IS fundamentally requires two steps: generating a set of samples from a proposal distribution q and assigning a weight to each sample based on the target and proposal densities.

A key obstacle to applying IS to programs with stochastic support is then to construct a valid proposal distribution due to the fact that the validity of a trace is difficult to establish without running the program. Therefore, a common technique is to *use the program itself as a proposal distribution* which automatically ensures that the proposal distribution is valid.

Let $\theta'_{1:n_\theta}$ be a trace sampled from the program as a proposal. The unnormalized importance weight is then given by the ratio $\gamma(\theta'_{1:n_\theta})/q(\theta'_{1:n_\theta})$. In the case of using the input program itself to generate traces, the proposal density is given by the product of all the conditional distributions $g_{a_i}(\theta_i \mid \eta_i)$ corresponding to the `sample` statements encountered during the execution of the program. However, the exact terms will appear in our (unnormalized) target density γ, resulting in a cancellation of the terms. The only remaining terms in the importance weights are the terms $h_{b_j}(y_j \mid \phi_j)$ which are added while encountering the `observe` statements.

To generate valid weighted samples that can be used for inference, we therefore only need to execute the program S times, record the traces and the corresponding `observe` terms $h_{b_j}(y_j \mid \phi_j)$. Because the unnormalized weights only depend on the `observe` terms, this technique is also referred to as *likelihood weighting*.

While this method is a valid inference algorithm that is simple to implement, it has similar drawbacks to the standard IS algorithm. Notably, it will struggle to scale to higher dimensions because it is more unlikely that the program will be a good proposal and will generally suffer from the curse of dimensionality.

3.2.2 Markov Chain Monte Carlo Approaches

Stochastic support substantially complicates the application of MCMC methods. Consider the problem of deriving a single-site Metropolis-Hastings algorithm for a program with stochastic support. Assuming we have a trace $\theta_{1:n_\theta}$, we can update a single latent variable θ_i with address a_i by proposing a new value θ_i' from the prior $g_{a_i}(\cdot \mid \eta_i)$. However, there is no guarantee that the new trace $\theta'_{1:n_\theta}$ (with θ_i replaced with θ_i') will be valid.

Again, it is important to remember that the program is deterministic for a fixed trace $\theta_{1:n_\theta}$. Hence, if we only change the ith value of the trace, then the prefix $\theta_{1:i-1}$ will always lead to the `sample` statement with address a_i with distribution $g_{a_i}(\cdot \mid \eta_i)$. However, a change in the value at address a_i might lead to the program taking a different branch downstream which can change the validity of the trace. A way around this is to regenerate the trace from the `sample` statement with address a_i onwards.

This approach to inference is known as *single-site/lightweight MH* or the random database algorithm [Wingate et al., 2011]. While the high-level idea of this approach is clear, it can be difficult to correctly implement the approach in practice [Kiselyov, 2016, Hur et al., 2015]. Naively regenerating all values downstream can be wasteful and inefficient which has lead to methods exploring the ability to re-use values from the existing trace $\theta_{1:n_\theta}$ [Yang et al., 2014, Mansinghka et al., 2014, Ritchie et al., 2016a, Cusumano-Towner et al., 2019]. Additionally, random-walk MH [Le, 2016] provides further improvements by constructing a local proposal instead of using the prior.

Single-site MH is easy to implement because it only requires changing the behaviour of the `sample` and `observe` statements and does not require any additional program analysis. However, this also limits its efficiency and scalability. In the setting of static support Hamiltonian Monte Carlo (HMC) [Neal, 2011] has been tremendously successful in practice due to its ability to leverage the gradients of the log density. This has inspired recent developments of nonparametric HMC [Mak et al., 2021b, 2022] algorithms which can be applied to programs with stochastic support and can leverage gradient information but they still struggle with the problem of efficiently transitioning between different program paths.

The aforementioned approaches all provide an *automated* mechanism to apply MCMC inference to a program with stochastic support. However, it is also possible to manually construct MCMC algorithms through the framework of reversible-jump MCMC (RJMCMC) [Green, 1995, Roberts et al., 2019] and its generalization

involutive MCMC [Neklyudov et al., 2020]. Especially the PPS Gen [Cusumano-Towner et al., 2019] has done extensive work on providing a framework for users to specify manual transition kernels [Cusumano-Towner et al., 2020]. While this provides a powerful mechanism to construct efficient MCMC kernels for programs with stochastic support, it requires a lot of expertise and therefore constitutes a higher barrier of entry for users.

3.2.3 Divide, Conquer, and Combine

Overall, transitioning between different branches of a program is the source of many difficulties when generating samples from a program's posteriors. As we saw in the previous section, changing a single value might invalidate the trace, meaning we often have to regenerate large parts of the trace from scratch and even if a lot of the samples in the trace can be re-used, there is actually no guarantee that the resulting trace will be in a region with high posterior density.

To alleviate these complications of transitioning between different branches, Zhou et al. [2020] introduce the novel Divide, Conquer, and Combine (DCC) framework for inference in programs with stochastic support. The main idea behind DCC is to decompose the program into its constitution program paths, referred to as straight-line programs (SLPs), then run inference in each SLP separately, and in a final step combine the inferences from all SLPs. Crucially, inference within a single SLP reduces back to the static support setting. This makes it easier to efficiently generate samples from a single SLP compared to the whole program at once.

To formalize this a bit further, Zhou et al. [2020] showed that decomposing a program into its constituent SLP partitions the domain of the traces, Θ, into a (countable) number of disjoint subsets $\{\Theta_k\}$ indexed through some index set $\mathcal{K}$, i.e. $\Theta = \bigcup_{k\in\mathcal{K}} \Theta_k$ and $\Theta_i \cap \Theta_j = \varnothing$ for all $i \neq j$ with $i, j \in \mathcal{K}$. The *local unnormalized density* for the kth SLP is then given by

$$\gamma_k(\theta_{1:n_\theta}) := \mathbb{I}\left[\theta_{1:n_\theta} \in \Theta_k\right] \gamma(\theta_{1:n_\theta}). \tag{3.4}$$

Now, this density has *static support* and can be used as a target for inference to approximate the *local normalized density* $\pi_k(\theta) = \gamma(\theta)/Z_k$ where $Z_k = \int_{\Theta_k} \gamma(\theta) d\theta$ denotes the *local normalization constant*. The normalized density for the whole program then becomes a weighted sum of the local normalized densities

$$\pi(\theta) = \sum_{k\in\mathcal{K}} \frac{\gamma_k(\theta)}{Z_k} = \sum_{k\in\mathcal{K}} \frac{Z_k \pi_k(\theta)}{\sum_{k'\in\mathcal{K}} Z_{k'}}. \tag{3.5}$$

Hence, in the DCC setup any local inference algorithm also needs a mechanism to estimate the normalization constant of the kth SLP. In their implementation of DCC, Zhou et al. [2020] use a version of the single-site MH algorithm implemented in Anglican as the local inference algorithm with the normalization constant estimated using an adaptive importance sampler [Martino et al., 2017].

To implement the DCC approach in practice, we also need a mechanism to actually discover the constituent SLPs of a program. While this could in theory be done using program analysis techniques [Sankaranarayanan et al., 2013, Chaganty et al., 2013, Beutner et al., 2022], this would make it difficult to deal with the case of an unbounded (but countable) number of SLPs. Instead, Zhou et al. [2020] discover SLPs dynamically by simply recording the address traces of the program during execution, each address traces then defines its own SLP. An initial set of SLPs can be discovered at the start to initialize the inference process but the set of SLPs can be expanded during inference by occasionally running global MCMC steps on the program and keeping track of the proposed SLPs. For all newly proposed SLPs, DCC then runs a few local MCMC steps, estimates the local normalization constant, and decides whether to further run inference on that SLP based on the estimated normalization constant. Crucially, in contrast to other sampling based inference algorithms like RJMCMC or lightweight MH, whether to run inference on a proposed SLP is then no longer based on a single proposed sample but on a larger set of samples.

In general, most SLPs will have a relatively small normalization constant, hence collecting the same number of samples from each discovered SLP is not an efficient use of computational resources. Zhou et al. [2020] derive an efficient *resource allocation* mechanism to determine how many samples to collect from each SLP. Their resource allocation scheme models the problem of selecting SLPs as a multi-armed bandit problem with the need to balance exploration and exploitation, using a version of the upper confidence bound algorithm from the bandits literature adapted to the inference setting [Rainforth et al., 2018].

3.2.4 Variational Inference

Constructing a suitable variational inference (VI) algorithm for programs with stochastic support runs into similar issues as the IS approach as we need to construct a suitable variational family $q(\cdot;\phi)$ (also sometimes referred to as a *guide* in the probabilistic programming literature).

The BBVI algorithm [Ranganath et al., 2014, Wingate and Weber, 2013] can be extended to programs with stochastic support by constructing a guide on a

variable-by-variable basis [Paige, 2016, van de Meent et al., 2016]. The approach presented by Paige [2016], which is implemented in Anglican, lazily constructs the guide during inference. Specifically, we can construct a guide for the input program by executing the program forward and every time we encounter a `sample` statement with a new address a_i we initialize a new variational distribution $q_{a_i}(\cdot \mid \phi_{a_i})$. For a specific trace $\theta_{1:n_\theta}$ this then leads to the mean-field variational family of the form

$$q(\theta_{1:n_\theta}; \phi) = \prod_{i=1}^{n_\theta} q_{a_i}(\theta_i \mid \phi_{a_i}). \tag{3.6}$$

Notably, any `sample` statement with the same address will get assigned the same guide. Essentially, the same gradient estimators as in BBVI [Ranganath et al., 2014, Wingate and Weber, 2013] can be used to train the variational parameters. We can generate S samples from the guide by executing the program and instead of sampling from the distribution object, $g_{a_i}(\cdot \mid \eta_i)$, that is passed to a given `sample` statement we instead sample from the corresponding variational family, $q_{a_i}(\cdot \mid \phi_{a_i})$, and record the difference in log densities $\log g_{a_i}(\theta_i \mid \eta_i) - \log q_{a_i}(\theta_i \mid \phi_{a_i})$ and the gradient $\nabla_{\phi_{a_i}} \log q_{a_i}(\theta_i^{(s)} \mid \phi_{a_i})$. For the `observe` statements we simply record the log density $\log h_{b_j}(y_j \mid \phi_j)$.

If we execute the program S times in this manner, all the recorded quantities together allow us to compute the score-function/REINFORCE gradient estimator for the variational parameters at address a_j

$$\nabla_{\phi_{a_j}} \mathcal{L}(\phi) \approx \frac{1}{S_{a_j}} \sum_{s=1}^{S_{a_j}} \log \frac{\gamma(\theta^{(s)})}{q(\theta^{(s)}; \phi)} \left(\nabla_{\phi_{a_j}} \log q_{a_j}(\theta_j^{(s)} \mid \phi_{a_j}) \right), \tag{3.7}$$

where S_{a_i} is the number of times the address a_i was sampled (due to the property of stochastic support a given address might not appear in every execution trace). The estimator in Eq. (3.7) can struggle to produce low-variance estimates without further variance reduction methods. A standard approach is to use control variates [Paisley et al., 2012] to reduce the variance but the use of more sophisticated variance reduction methods, like Rao-Blackwellization, is challenging due to the properties of stochastic support [Paige, 2016].

The derivation of a BBVI algorithm for universal PPSs actually has some non-trivial applications. van de Meent et al. [2016] exploited connections between inference and control [Attias, 2003, Levine, 2018] to represent policies in a reinforcement learning setting as programs. Running BBVI inference in a suitable setup then allows the user to optimize the policy.

Due to the challenges of fully automating the construction of guides, some PPSs such as Pyro or Gen allow the user to manually define the guide. These languages then provide mechanisms to automate the optimization of the variational parameters, including the automated use of gradient estimators and variance reduction methods [Bingham et al., 2019]. However, in the setting of stochastic support even just checking whether a given guide is valid for a program is difficult to establish [Lee et al., 2019, Lew et al., 2019].

Overall, for programs with stochastic support it is still an open question with how to best construct suitable guides. In Chapter 4, we will try and make some progress on this research question.

3.2.5 Amortized Inference and Inference Compilation

So far we have assumed that the data in a model or program is fixed and that we only want to run inference once. *Amortized inference* [Gershman and Goodman, 2014] considers the problem of running inference in the same model multiple times on different data sets. In the context of universal PPSs, Le et al. [2017] introduced *inference compilation* which trains an autoregressive neural network to generate suitable importance sampling proposal distribution for a fixed program.

In inference compilation, the proposal has a similar form to the guide in the BBVI setting with the added complexity that it now takes in the observed data $\mathbf{y}$ as well

$$q(\theta_{1:n_\theta} \mid \mathbf{y}; \phi) = \prod_{i=1}^{n_\theta} q_{a_i}(\theta_i \mid \eta(\theta_{1:i-1}, \mathbf{y}; \phi)). \tag{3.8}$$

The regressor η is parameterized by ϕ and maps the previously encountered sampled variables along with the observed data to the parameters of the proposal distribution q_{a_i}. It is usually parameterized as a recurrent neural network [Hochreiter and Schmidhuber, 1997, Le et al., 2017], potentially extended with an attention mechanism [Vaswani et al., 2017, Harvey et al., 2019].

Notably, the inference compilation proposal is trained with a different objective than the VI guides. Minimizing the reverse KL through maximizing the ELBO leads to "mode-seeking" behaviour of the guide. This is undesirable if we want to use the trained approximation as a proposal in an importance sampling scheme because we generally want the proposal to be more "diffuse" than the target density. Hence, inference compilation targets the expected forward KL divergence which leads to more mode-covering behaviour of the proposal, similar to previous methods for learning IS proposals [Paige and Wood, 2016]. Targeting the expected forward KL divergence instead of just the forward KL divergence avoids the need to generate

samples from the posterior distribution during training. Instead, the model itself can be used to generate the training data for the proposal by changing the behaviour of the `observe` statements. It is possible to sample alternative datasets by ignoring the value used for conditioning that is passed to the `observe` statement and instead sample a new value from the distribution object. Let $\pi(\theta, \mathbf{y})$ denote the distribution defined by this sampling process, inference compilation then minimizes the objective

$$\mathcal{L}(\phi) = \mathbb{E}_{\pi(\theta,\mathbf{y})} \left[-\log q(\theta \mid \mathbf{y}; \phi) \right]. \tag{3.9}$$

This objective is essentially equivalent to doing density estimation on samples generated from the program and it can be minimized using stochastic gradient descent. This inference compilation approach has been successfully applied on complex, real-world simulators from particle physics [Baydin et al., 2019, 2020]. Baydin et al. [2019] were able to apply inference compilation to a simulator that is not written in a PPS through the use of a communication protocol called PPX[8]. PPX can be used more generally to adapt existing simulator code that is not written in a PPS and, by only changing the calls to random number generators in the code, makes it possible to run inference in the simulator.

More generally, neural networks have been extremely successful in providing flexible mechanisms to construct guides [Ritchie et al., 2016b]. They are especially powerful in setting in which the model also has learnable parameters that are optimized jointly with the guide according to a variational objective [Kingma and Welling, 2014]. However, a lot of these techniques require a lot of data to train the neural networks and also usually require manual effort to construct the neural network architectures.

[8] `https://github.com/pyprob/ppx`

4

Rethinking Variational Inference for Programs with Stochastic Support

4.1 Introduction

As outlined in the previous chapter, probabilistic programming systems (PPSs) enable users to express probabilistic models with computer programs and provide tools for inference. Restricted PPS, such as Stan [Carpenter et al., 2017] or PyMC3 [Salvatier et al., 2016], limit the expressiveness of their language to ensure that existing inference algorithms are applicable to the models encoded with the PPSs. In contrast, *universal PPSs* [Tolpin et al., 2016, Goodman et al., 2008, Bingham et al., 2019, Ge et al., 2018, Cusumano-Towner et al., 2019, Mansinghka et al., 2014, Narayanan et al., 2016, Goodman and Stuhlmüller, 2014, Murray and Schön, 2018] can encode non-standard models by allowing stochastic branching. This leads to challenging inference problems because it allows for programs where the sequence of variables—not just the variable values—changes between executions, leading to *models with stochastic support*. These models have applications in numerous fields, such as natural language processing [Manning and Schütze, 1999], Bayesian Nonparametrics [Richardson and Green, 1997], and statistical phylogenetics [Ronquist et al., 2020]. A wide range of simulator-based models similarly require such stochastic control flow [Baydin et al., 2019, Le et al., 2017, Gram-Hansen et al., 2019].

The effectiveness of PPSs is heavily reliant on the underlying inference schemes they support. Variational inference (VI) is one of the most popular such schemes, both in PPSs and more generally [Blei et al., 2017, Kucukelbir et al., 2015, Agrawal et al., 2020]. This popularity is due to its ability to use derivatives to scale to large

datasets and high-dimensional models [Zhang et al., 2018, Hoffman et al., 2013, Rezende and Mohamed, 2015, Kingma and Welling, 2014], often providing much faster and more scalable inferences compared to Monte Carlo approaches [Brooks et al., 2011]. To provide the required derivatives, a number of modern universal PPSs—such as Pyro [Bingham et al., 2019], ProbTorch [Siddharth et al., 2017], PyProb [Baydin et al., 2019], Gen [Cusumano-Towner et al., 2019], and Turing [Ge et al., 2018]—have introduced automatic differentiation [Baydin et al., 2018] capabilities for programs with stochastic control flow. One of the core aims behind these developments was to support VI schemes in such settings [Bingham et al., 2019].

However, constructing appropriate variational families, typically known as guides in PPSs, can be very challenging for problems with stochastic support, even for expert users. This is because the stochasticity of the control flow induces discontinuities and complex dependency structures that are difficult to remain faithful to and design parameterized approximations for. Furthermore, while there are a plethora of different automatic guide construction schemes for static support problems [Kucukelbir et al., 2015, Blei et al., 2017, Agrawal et al., 2020], there is a lack of suitable schemes applicable to models with stochastic support. Consequently, existing methods tend to give unreliable results in such settings, as we demonstrate in Figure 4.1.

We argue that a significant factor of this shortfall is that standard practice—for both manual and automated methods—is to construct the guide on a variable-by-variable basis [Wingate and Weber, 2013, Paige, 2016, van de Meent et al., 2016, 2018]. Namely, existing approaches generally use a single global guide that mirrors the control flow of the input program, then introduce a variational approximation for each unique variable. This is problematic because control flows inherently introduce discontinuities into the program's density function, such that the conditional distribution of each variable will typically change significantly whenever the program path—that is the sequence of random variables—changes. Thus it is extremely challenging to learn a single approximation for each variable that is appropriate across all paths. Further, as the set of variables that exist can itself be stochastic, it is difficult for such guides to appropriately condition on previously sampled variables. Existing automated approaches, therefore, typically rely on mean-field assumptions [Paige, 2016], thereby forgoing any conditioning on the program path itself, consequently leading to poor approximations for most problems.

To overcome these difficulties, we propose *Support Decomposition Variational Inference* (SDVI), a new VI approach based around a novel way of constructing the variational guide. SDVI "rethinks" the guide construction by breaking it down

```
def model():
  x = sample("x", Normal(0, 1))
  if x < 0:
    z = sample("z1", Normal(-3, 1))
  else:
    z = sample("z2", Normal(3, 1))
  sample("y", Normal(z, 2), obs=2.0)
```

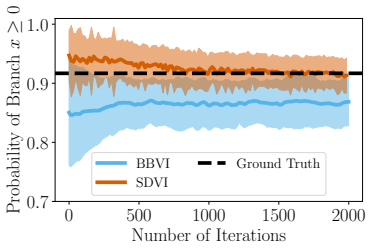


Figure 4.1: Pyro program with stochastic control flow [Left]. Existing procedures for automatically constructing the guide mirror the control flow of the input program [BBVI, Blue]. However, this produces an inherently limited variational family, leading to unsatisfactory performance despite the problem's simplicity. By breaking down the guide over paths, SDVI [Red] is able to provide accurate inference. Results computed over 10^2 replications, plotted are mean and standard deviation.

over *paths*, instead of building it on a variable-by-variable basis. Specifically, by utilizing the fact that any program can be reformulated as a mixture of *straight-line programs* (SLPs) [Chaganty et al., 2013, Sankaranarayanan et al., 2013, Zhou et al., 2020]—each defined by a unique realization of the path—SDVI constructs the guide as a mixture of sub-guides with static support. We show that optimizing the variational objective with this guide structure leads to a natural decomposition of the overall optimization problem into independent sub-problems, each taking the form of a VI with static support. The sub-guides can thus be effectively constructed and trained using more standard VI techniques, before being recombined to form our overall variational approximation. To make SDVI accessible to a wide audience, we have implemented it in Pyro [Bingham et al., 2019]. We evaluate it on a set of example problems with synthetic and real-world data, finding that it provides substantial improvements over existing techniques.

4.2 Background

For clarity, we here briefly restate the key definitions for programs with stochastic support and variational inference. We refer the interested reader for more extensive introductions to these topics to Sections 3.1.3 and 2.3.3, respectively.

4.2.1 Probabilistic Programs with Stochastic Support

PPSs allow users to express probabilistic models and condition on observed data [Gordon et al., 2014, van de Meent et al., 2018]. A common mechanism to achieve this is to extend standard programming languages with two new primitives: `sample` and `observe`.[1] `sample(addr, dist)` draws samples from the distribution object

[1] As an alternative interface, Pyro instead overloads the `sample` primitive: `sample(addr,dist,obs=data)` ≡ `observe(addr,data,dist)`.

`dist`, where `addr` is a unique lexical identifier. `observe(addr, data, dist)` enables conditioning on an observed outcome `data`, where `dist` and `addr` are as before. For problems admitting a Bayesian formulation, the `sample` and `observe` terms can informally be thought of as prior and likelihood factors respectively.

Universal PPSs allow users to write complex models whose support can vary from one execution to the next, e.g. stochastic branching can mean certain variables only sometimes exist. This can substantially complicate the process of performing inference.

A probabilistic program in a universal PPS defines an unnormalized *density function* $\gamma(\theta_{1:n_\theta})$ over the *raw random draws* $\theta_{1:n_\theta} \in \Theta$—defined as the (sequences of) direct outputs of `sample` statements—where $n_\theta \in \mathbb{N}^+$ is itself potentially random. Though each outcome of $\theta_{1:n_\theta}$ uniquely defines a program execution, it is notationally convenient to further associate an *address* a_i to each draw θ_i that indicates the position in the program the draw was made. This address can be uniquely defined as the tuple formed by the `addr` of the `sample` and the number of times that `sample` has previously been called. For a given execution of the program, the addresses now form an *address path* $A = a_{1:n_\theta}$.

Each `sample` statement encountered during execution contributes the factor $g_{a_i}(\theta_i \mid \eta_i)$ to the program density, where a_i is the address of the `sample` statement, g_{a_i} is a parameterized density function, and η_i are its associated parameters. Similarly, each encountered `observe` statement contributes the factor $h_{b_j}(y_j \mid \phi_j)$, with b_j denoting an address, y_j the observed value, h_{b_j} a parameterized density function, and ϕ_j its parameters. Following [Rainforth, 2017, §4.3.2], we write the program density function as

$$\gamma(\theta_{1:n_\theta}) := \prod_{i=1}^{n_\theta} g_{a_i}(\theta_i \mid \eta_i) \prod_{j=1}^{n_y} h_{b_j}(y_j \mid \phi_j). \tag{4.1}$$

All of $n_\theta, n_y, a_{1:n_\theta}, \eta_{1:n_\theta}, y_{1:n_y}, b_{1:n_y}$, and $\phi_{1:n_y}$ are potentially random variables. The goal of *inference* is to approximate the conditional distribution of the program, which has normalized density $\pi(\theta_{1:n_\theta}) = \gamma(\theta_{1:n_\theta})/Z$ with marginal likelihood $Z = \int_\Theta \gamma(\theta_{1:n_\theta}) d\theta_{1:n_\theta}$ and the integral is computed with respect to a reference measure that is implicitly defined by the `sample` statements in the program.

4.2.2 Variational Inference

Variational Inference (VI) [Wainwright et al., 2008, Blei et al., 2017] solves the inference problem by transforming it to an optimization problem. Specifically, given an unnormalized joint distribution $\gamma(\theta)$ and a parameterized distribution

$q(\theta;\phi)$, VI computes the variational parameters ϕ such that $q(\theta;\phi)$ most closely approximates $\pi(\theta) = \gamma(\theta)/Z$. This is most commonly done by maximizing the *Evidence Lower Bound* (ELBO) $\mathcal{L}(\phi) := \mathbb{E}_{q(\theta;\phi)}\left[\log \gamma(\theta) - \log q(\theta;\phi)\right]$ via stochastic gradient ascent using Monte Carlo estimates of $\nabla_\phi \mathcal{L}(\phi)$ [Mohamed et al., 2020]. Two popular estimators are the score function estimator [Kleijnen and Rubinstein, 1996, Williams, 1992], and the reparameterized gradient estimator [Rezende et al., 2014, Kingma and Welling, 2014, Titsias and Lázaro-Gredilla, 2014]. The latter provides lower variance gradient estimates but requires that the distribution $q(\theta;\phi)$ can be reparameterized and that $\gamma(\theta)$ is differentiable everywhere.

4.3 Difficulties for Variational Inference in Universal PPSs

The starting point for any VI scheme is to construct an appropriate variational family, also known as a guide. To automate inference, we desire to (at least partially) automate the process of constructing this guide. Existing methods for this all generate the guide on a variable-by-variable basis [Wingate and Weber, 2013, Paige, 2016, van de Meent et al., 2018]: they introduce a single variational distribution $q_{a_i}(\theta;\phi_{a_i})$ for each unique sampling address a_i, then form the guide by replacing all the original random draws, $\theta_i \sim g_{a_i}(\cdot \mid \eta_i)$, with draws from the corresponding variational distribution instead. This forms a global guide that maintains the stochastic dependency structure of the original program, such that the guide itself has stochastic support.

Our motivating insight is that this high-level approach has some fundamental limitations. Consider the simple example from Figure 4.1. Here the variable `x` influences the program's control flow. This causes discontinuities that mean it is difficult to approximate its conditional density with a single variational approximation, especially if that variational approximation is restricted to a simple distribution class. Here the different possible paths are essentially working against each other, as what is helpful for the approximation of `x` on one path, is generally detrimental for the other.

A further complication occurs when the stochastic control flow of a program influences whether a variable exists at all. Here it can become extremely challenging to set up guides which are faithful to the dependency structure of previously sampled variables [Webb et al., 2018], as the set of variables we condition on is itself stochastic. Because of this, existing approaches often rely on mean-field assumptions [Wingate and Weber, 2013, Paige, 2016, van de Meent et al., 2016].

However, this assumption is rarely reasonable given that the path typically strongly influences the distribution of individual variables.

Finally, creating a single unique variational approximation for each address also leads to challenging optimization problems: the same address can be present in multiple program paths, and the number of variables and their dependencies can vary between different paths.

4.4 Support Decomposition Variational Inference

We now introduce a novel VI approach to overcome the challenges mentioned in Section 4.3. We call our method *Support Decomposition Variational Inference* (SDVI), because the key design decision is the choice (and automatic construction) of a guide that takes the form of a mixture distribution over the set of possible paths a program can take. That is, rather than constructing a single guide with the same stochastic control flow as the original program and a separate variational approximation for each *unique address*, we instead construct separate sub-guides with deterministic control flows for each *unique path*. These are then combined into an overall guide using the mixture distribution which maximizes the overall ELBO. As we will see, this alternative approach substantially simplifies the process of constructing effective guides and allows the full weight of the well-developed techniques for VI in the static support setting to be brought to bear on problems with stochastic support.

4.4.1 Decomposing Probabilistic Programs into Straight-Line Programs

As noted by, e.g., [Chaganty et al., 2013, Sankaranarayanan et al., 2013, Zhou et al., 2020], all probabilistic programs can be reformulated as mixture distributions over *straight-line programs* (SLPs), which are sub-programs without any stochastic control flow. Building on our earlier notation, the constituent SLPs of a program correspond directly to the unique instances of the program path, A. Given the path, the set of variables making up the raw random draws in the program is fixed, along with their form and reference distribution; that is each SLP represents a probabilistic model with fixed variable typing and support.

Following the notation of Zhou et al. [2020], we can apply an arbitrary fixed ordering on the set of SLPs in a program, such that we can uniquely define and index them using the set of possible addresses A_k for $k \in \mathcal{K}$, where $\mathcal{K}$ is a countable (but potentially infinite) indexing set. Each SLP A_k now corresponds to a particular

sub-region, Θ_k, of the raw random draw sample space, Θ. These sub-regions are disjoint and their union is the full sample space. Unlike Θ, each element in any given Θ_k has the same length n_k and is measurable with respect to the same reference measure. The unnormalized density for the kth SLP is now given by

$$\gamma_k(\theta_{1:n_k}) = \mathbb{I}[\theta_{1:n_k} \in \Theta_k]\, \gamma(\theta_{1:n_k}) \tag{4.2}$$

$$= \mathbb{I}[\theta_{1:n_k} \in \Theta_k] \prod_{i=1}^{n_k} g_{A_k[i]}(\theta_i \mid \eta_i) \prod_{j=1}^{n_y} h_{b_j}(y_j \mid \phi_j), \tag{4.3}$$

and the unnormalized density function for the original program can be written as a simple sum of the individual SLP densities: $\gamma(\theta_{1:n_x}) = \sum_{k \in \mathcal{K}} \gamma_k(\theta_{1:n_x})$. The corresponding normalized conditional density can then be written as mixture distribution over the conditional distributions of the individual SLPs, with mixture weights given by their (normalized) local partition functions:

$$\pi(\theta) = \sum_{k \in \mathcal{K}} \pi(\theta \mid k)\, \pi(k) \tag{4.4}$$

where

$$\pi(\theta \mid k) = \frac{\gamma_k(\theta)}{Z_k},\ \pi(k) = \frac{Z_k}{\sum_{\ell \in \mathcal{K}} Z_\ell},\ Z_k = \int_{\Theta_k} \gamma_k(\theta) d\theta. \tag{4.5}$$

4.4.2 Decomposing the Variational Family into Straight-Line Programs

The key idea behind SDVI is now to construct the guide using a factorization that is analogous to that of the SLP decomposition above. Precisely, we aim to learn variational approximations of the form

$$q(\theta; \phi, \lambda) = \sum_{k=1}^{K} q_k(\theta; \phi_k) q(k; \lambda) \tag{4.6}$$

where $q(k; \lambda)$ defines a categorical distribution over the indices of the SLPs, with support $k \in \{1, \ldots, K\}$; and $q_k(\theta; \phi_k)$ is the local guide of the kth SLP, with support $\theta \in \Theta_k$. Critically, as each Θ_k represents a fixed support, the local variational families q_k can be automatically constructed using standard techniques for static problems, as we discuss in Section 4.4.5. Note that it is valid for the guide $q(\theta; \phi, \lambda)$ to not cover all SLPs, i.e. it is possible that $K < |\mathcal{K}|$.

Writing $\phi = \{\phi_k\}_{k=1}^K$, the KL divergence we wish to minimize for standard VI is now

$$\mathrm{KL}(q(\theta; \phi, \lambda) \parallel \pi(\theta)) = \mathbb{E}_{q(\theta;\phi,\lambda)} \left[\log q(\theta; \phi, \lambda) - \log \pi(\theta)\right], \tag{4.7}$$

which we call the *global KL divergence*. By standard reasoning, minimizing this is equivalent to maximizing the *global ELBO*

$$\mathcal{L}(\phi, \lambda) = \mathbb{E}_{q(\theta;\phi,\lambda)}\left[\log \frac{\gamma(\theta)}{q(\theta;\phi,\lambda)}\right], \tag{4.8}$$

$$= \int_{\Theta} q(\theta;\phi,\lambda) \log \frac{\gamma(\theta)}{q(\theta;\phi,\lambda)} d\theta, \tag{4.9}$$

using the fact that the subsets Θ_k provide a partition of Θ we can write the integral as a sum of integrals over the subsets

$$= \sum_{k=1}^{K} \int_{\Theta_k} q(\theta;\phi,\lambda) \log \frac{\gamma(\theta)}{q(\theta;\phi,\lambda)} d\theta, \tag{4.10}$$

using the factorization of the guide, $q(\theta;\phi,\lambda)$, in Eq. (4.6) and the fact that for $\theta \in \Theta_k$ the program density is given by the density for the kth SLP, i.e. $\gamma(\theta) = \gamma_k(\theta)$, we get

$$= \sum_{k=1}^{K} \int_{\Theta_k} q_k(\theta;\phi_k) q(k;\lambda) \log \frac{\gamma_k(\theta)}{q_k(\theta;\phi_k) q(k;\lambda)} d\theta. \tag{4.11}$$

We can then move the guide weight for the kth SLP, $q(k;\lambda)$, outside of the integral as it does not depend on the latents

$$= \sum_{k=1}^{K} q(k;\lambda) \int_{\Theta_k} q_k(\theta;\phi_k) \log \frac{\gamma_k(\theta)}{q_k(\theta;\phi_k)} d\theta - \log q(k;\lambda), \tag{4.12}$$

which we can write concisely as

$$= \mathbb{E}_{q(k;\lambda)}\left[\mathcal{L}_k(\phi_k) - \log q(k;\lambda)\right], \tag{4.13}$$

where we define the *local ELBO* for the kth SLP as

$$\mathcal{L}_k(\phi_k) := \mathbb{E}_{q_k(\theta;\phi_k)}\left[\log \frac{\gamma_k(\theta)}{q_k(\theta;\phi_k)}\right]. \tag{4.14}$$

Notice that each local ELBO, $\mathcal{L}_k(\phi_k)$, depends only on the local variational parameter, ϕ_k, and the local SLP density, γ_k; it is completely independent of $q(k;\lambda)$, the other SLPs, and $\phi_{k'}$ for $k' \neq k$. Thus, it follows from Eq. (4.13) that the inference problem for the whole program can be decomposed into independent 'local' inference problems for the component SLPs, along with establishing the mixture probabilities $q(k;\lambda)$. Furthermore, it turns out that the optimal $q(k;\lambda)$ is simply the softmax of $\mathcal{L}_1, \ldots, \mathcal{L}_K$, as shown by the following result.

Proposition 1. *Let $L = \{\mathcal{L}_1, \ldots, \mathcal{L}_K\}$ be the set of local ELBOs, defined as per (4.14), where L is countable but potentially not finite. If $0 < \sum_{k=1}^{K} \exp(\mathcal{L}_k) < \infty$, then the optimal corresponding $q(k;\lambda^*)$ in terms of the global ELBO from Eq. (4.8) is given by*

$$q(k;\lambda^*) = \frac{\exp(\mathcal{L}_k)}{\sum_{\ell=1}^{K} \exp(\mathcal{L}_\ell)}. \tag{4.15}$$

Proof. By the assumption that $0 < \sum_{k=1}^{K} \exp(\mathcal{L}_k) < \infty$, we have that

$$p(k) = \frac{\exp(\mathcal{L}_k)}{\sum_{\ell=1}^{K} \exp(\mathcal{L}_\ell)} \tag{4.16}$$

forms a valid probability mass function over $k \in \{1, \ldots, K\}$. We can therefore rewrite the global ELBO from Eq. (4.13) as

$$\mathcal{L}(\phi, \lambda) = \mathbb{E}_{q(k;\lambda)} \left[\log \frac{\exp(\mathcal{L}_k)}{\sum_{\ell=1}^{K} \exp(\mathcal{L}_\ell)} - \log q(k;\lambda) \right] + \log \sum_{\ell=1}^{K} \exp(\mathcal{L}_\ell) \tag{4.17}$$

$$= -\mathrm{KL}\left(q(k;\lambda) \parallel \frac{\exp(\mathcal{L}_k)}{\sum_{\ell=1}^{K} \exp(\mathcal{L}_\ell)} \right) + \log \sum_{\ell=1}^{K} \exp(\mathcal{L}_\ell) \tag{4.18}$$

Now as the second term in the above equation is constant in $q(k;\lambda)$ and a KL divergence is minimized when the two distributions are the same, we can immediately conclude the desired result that the optimal $q(k;\lambda^*)$ is

$$q(k;\lambda^*) = \frac{\exp(\mathcal{L}_k)}{\sum_{\ell=1}^{K} \exp(\mathcal{L}_\ell)}. \tag{4.19}$$

□

Additionally, for the optimal setting of the mixture distribution, $q(k;\lambda^*)$, the KL divergence in Eq. (4.18) goes to zero and the global ELBO then reduces to

$$\mathcal{L}(\phi, \lambda^*) = \log \sum_{k=1}^{K} \exp(\mathcal{L}_k).$$

Though each of the $\mathcal{L}_k$ terms here is itself intractable, they can be estimated efficiently and accurately by simple Monte Carlo. We can thus straightforwardly construct our weighting over SLPs, $q(k;\lambda)$, once we have learned our local variational approximations. Conveniently, these two processes are perfectly separable, $q(k;\lambda)$ is not needed until *after* the individual local guides, q_k, are trained.

4.4.3 Finding SLPs

We have just shown how we can solve the VI problem of a probabilistic program with stochastic support by reducing it to a set of *independent* and *simpler* VI problems, each concerning an SLP, a program with static support. However, we still need a mechanism to 'discover' the SLPs, i.e. extract the possible address paths from a program.

Here we first note that we only need to consider an SLP if it has a non-zero probability of being identified under forward simulation of the program, while

Algorithm 4 Support Decomposition Variational Inference

Require: Target program γ, iteration budget T, minimum no. of SH candidates m

Extract SLPs $\{\gamma_k\}_{k=1}^K$ from γ and set $\mathcal{C} = \{1, \ldots, K\}$ ▷ Sec. 4.4.3
Formulate guide q_k for each SLP and initialize parameters ϕ_k ▷ Sec. 4.4.5
for $l = 1, \ldots, L = \lceil \log_2(K) - \log_2(m) + 1 \rceil$ **do**
    for $k \in \mathcal{C}$ **do**
        Run $\left\lfloor \frac{T}{L\lvert\mathcal{C}\rvert} \right\rfloor$ optimization steps of ϕ_k targeting $\mathcal{L}_{\text{surr},k}(\phi_k)$ ▷ Sec. 4.4.5
    end for
    Remove $\min(\lfloor \frac{\lvert\mathcal{C}\rvert}{2} \rfloor, \lvert\mathcal{C}\rvert - m)$ SLPs from $\mathcal{C}$ with the lowest $\mathcal{L}_k(\phi_k)$ ▷ Sec. 4.4.4
end for
Truncate q_k outside of SLP support, Θ_k, using Eq. (4.24)
Estimate each $\mathcal{L}_k(\phi_k)$ using Monte Carlo estimate of Eq. (4.13)
Calculate $q(k; \lambda)$ according to Eq. (4.15)
return $q(\theta; \phi, \lambda)$ as per Eq. (4.6)

ignoring conditioning statements. This hints at a cheap and simple discovery mechanism whereby we draw samples by forward simulation and take note of the unique paths that have been generated. We can either do this upfront, or in an online manner whereby we seek new SLPs as our budget increases and we have scope to deal with them (see Appendix A.1 for details). Although this is a stochastic procedure that is not guaranteed to find all the SLPs for finite budgets, for the problems considered in our experiments, it was always able to reliably identify all SLPs with non-negligible posterior mass. Nonetheless, this approach may not be sufficient for all problems, such as when the likelihood concentrates in an area of very low prior mass. Here one should instead look to employ more sophisticated discovery methods instead, such as those based on MCMC sampling [Zhou et al., 2020] or static analysis of the program code [Chaganty et al., 2013, Nori et al., 2015, Beutner et al., 2022].

4.4.4 Allocating Resources

Using the same amount of computational budget on each SLP is potentially wasteful, particularly if there is a large number of SLPs with insignificant marginal likelihoods. Therefore, we seek a scheme that allocates more computational resources to promising SLPs, making sure to exploit the fact that the different inference problems are trivially parallelizable.

To formalize this resource allocation problem, let T represent some fixed resource budget. Further, let t_k be the amount of this budget we spend on optimizing the kth SLP, such that $\sum_k t_k = T$ at the end of our training. Our ultimate aim is produce the maximum possible final global ELBO, which will be a function of $\phi_1(t_1), \ldots, \phi_K(t_K)$, where $\phi_k(t_k)$ denotes the value of ϕ_k achieved after allocating

t_k resources to that SLP. By plugging the optimal mixture distribution $q(k;\lambda)$ from Eq. (4.15) into Eq. (4.8), we see that, after some rearranging, our resource allocation can be formulated as trying to maximize

$$\mathcal{L}(\phi, \lambda^*) = \log \sum_{k=1}^{K} \exp(\mathcal{L}_k(\phi_k(t_k))) \quad \text{s.t.} \quad \sum_{k=1}^{K} t_k = T. \tag{4.20}$$

In practice, this is not a suitable objective for controlling our resource allocation directly, as it is still itself a random variable given $t_1, \ldots, t_K$, because the optimization procedure is stochastic. Moreover, we cannot consider its expectation, since the distribution of the $\phi_k(t_k)$ is unknown. However, it does provide insight into how we ideally would like to allocate resources: we want to allocate more resources to SLPs whose exponentiated ELBOs are significant. In particular, we can think of the 'reward' for allocating ϵ more resources to SLP k as $\exp(\mathcal{L}_k(\phi_k(t_k + \epsilon))) - \exp(\mathcal{L}_k(\phi_k(t_k)))$.

One could now, in principle, formulate the problem as a sequential decision making problem [Lattimore and Szepesvári, 2020]. However, the diminishing nature of the rewards and the fact that they are highly unlikely to be sub-Gaussian, along with the need to allow choosing multiple arms at once for parallelization, mean that setting up such an approach that is effective in practice is likely to be quite challenging.

Instead, we propose a simple heuristic, based on the *Successive Halving* algorithm (SH) [Karnin et al., 2013] (see Appendix A.1 for a description), an approach commonly used for resource allocation in hyperparameter optimization [Li et al., 2018, Falkner et al., 2018]. In standard SH, the final objective is to identify and train the single best candidate, whereas ours is to maximize the *sum* of all the local ELBOs. Despite this difference, the use of SH can still be justified by the fact that the distribution over $\exp(\mathcal{L}_k(\phi_k(\infty)))$ will typically be heavily concentrated to a small number of SLPs, often only a single one. Nonetheless, we make a small adaptation to the approach to stop over-focusing on a single SLP: we stop the halving process when a chosen number, $1 \leqslant m \leqslant K$, of the candidates are left, with $m = 1$ corresponding to standard SH. The minimum proportion of the budget allocated to any given candidate by this scheme is $1/(K\lceil \log_2 K - \log_2 m + 1 \rceil)$, so we can use m as a hyperparameter to control how evenly resources are allocated, with $m = K$ corresponding to uniform allocation. This approach is also helpful for parallelization, as we can set m equal to the number of available cores.

Putting everything together, a summary of our SDVI algorithm is given in Algorithm 4. In Appendix A.1, we further show how this can be extended to an

online variant of the approach, wherein we repeatedly run SH using the objective $\exp(\alpha\mathcal{L}_k(\phi_k(t_k)))/t_k$, where $0 < \alpha \leqslant 1$ is a hyperparameter, with smaller values of α encouraging more exploration.

4.4.5 Formulating and Training the Local Guides

In Algorithm 4 we assume a mechanism to construct the *local guide* $q_k(\theta;\phi_k)$ for each SLP specified by path A_k. In many situations—notably when the program path is uniquely determined by the sampled values from discrete distributions—it is possible to construct guides q_k that are guaranteed to place support within the sub-region Θ_k, which, in turn, allows us to use the reparameterized gradient estimator for the gradients of $\mathcal{L}_k(\phi_k)$. Many models encountered in practice, e.g. mixture models [Richardson and Green, 1997], have this property. In this case it is possible to eliminate all the variables which influence the control flow by conditioning, effectively setting them to constants. We will discuss this further in Section 4.4.5.1.

In situations where we cannot easily construct a q_k which places support only within Θ_k, we need to take care when training our guide. Recall that for path A_k the number of variables n_k and their type sequence is fixed, which allows us to construct an initial guide $\tilde{q}_k$ with correct dimensionality and variable typing. Let the support of this guide be denoted by $\Theta'_k = \mathrm{supp}(\tilde{q}_k)$. In general, we will have $\Theta_k \subset \Theta'_k$, because the control flow in the program imposes additional constraints on each individual variable. Having constructed a guide with $\mathrm{supp}(\tilde{q}_k) = \Theta'_k$, one might be tempted to optimize $\mathrm{KL}(\tilde{q}_k(\theta;\phi_k) \parallel \pi(\theta \mid k))$, but we cannot guarantee the absolute continuity condition (i.e. $\tilde{q}_k(\theta;\phi_k) = 0$ if $\pi(\theta \mid k) = 0$), and so, the KL divergence may not be well-defined, giving an ELBO of $-\infty$. To alleviate this issue we temporarily create a new surrogate target density defined as

$$\tilde{\gamma}_k(\theta_{1:n_k}) := \gamma_k(\theta_{1:n_k}) + c\,\mathbb{I}[\theta_{1:n_k} \notin \Theta_k], \tag{4.21}$$

for a small positive, finite constant c. This surrogate density is used solely for optimizing $\tilde{q}_k(\theta;\phi_k)$. We train ϕ_k to optimize the corresponding surrogate ELBO

$$\mathcal{L}_{\mathrm{surr},k}(\phi_k) := \mathbb{E}_{\tilde{q}_k(\theta;\phi_k)}\left[\log \tilde{\gamma}_k(\theta) - \log \tilde{q}_k(\theta;\phi_k)\right]. \tag{4.22}$$

We need to be careful to choose an appropriate c that is sufficiently small compared to the values of $\gamma_k(\theta)$ for $\theta \in \Theta_k$, which we ensure by setting c adaptively. During the SLP discovery phase (Line 1 in Algorithm 4), we keep track of the smallest density value encountered so far, and call that d_{min}. We then set $c = 0.01 d_{min}$ to ensure that the density values for $\tilde{\gamma}_k(\theta)$ outside of Θ_k are significantly below the

values of $\tilde{\gamma}_k(\theta)$ for $\theta \in \Theta_k$. Hence, optimizing Eq. (4.22) faithfully optimizes q_k to be a good approximation to $\gamma_k(\theta)$ while avoiding the issues of infinite ELBO values. While $\tilde{\gamma}_k$ is not a proper unnormalized density (it will in general not integrate to a finite value) this is not an issue in practice due to the mode-seeking behaviour of optimizing the ELBO.

Unfortunately, the bounds on the support of the SLP inevitably create a discontinuity in the objective. Thus, for fully unbiased gradients we need to use the score function estimator or some extension thereof. Note that $\mathcal{L}_{\text{surr},k}(\phi_k)$ retains the desirable property of the standard ELBO that, if the observations are conditionally independent given the latent variables, we can get unbiased estimates of the ELBO using minibatches of the full dataset [Hoffman et al., 2013, Titsias and Lázaro-Gredilla, 2014, Kucukelbir et al., 2015].

Further we need to be careful when initializing $\tilde{q}_k$ as we require it to place sufficient probability mass within Θ_k to provide a suitable training signal. To ensure this, we initialize ϕ_k by minimizing the *forward* KL divergence between the prior density of the kth SLP and $\tilde{q}_k(x; \phi_k)$

$$\mathrm{KL}(\pi_{prior,k}(\theta) \parallel \tilde{q}_k(\theta; \phi_k)) \propto \mathbb{E}_{\pi_{prior}(\theta)} \left[-\mathbb{I}[\theta \in \Theta_k] \log \tilde{q}_k(\theta; \phi_k) \right] \tag{4.23}$$

where $\pi_{prior}(\theta_{1:n_\theta}) := \prod_{i=1}^{n_\theta} g_{a_i}(\theta_i \mid \eta_i)$ is the product of all the `sample` statements encountered during execution. This objective can be optimized via stochastic gradient descent (cf. Appendix A.2). Note that, for the purpose of initialization, we are targeting the prior, and thus we do not have to resort to expensive schemes to estimate the gradients which are necessary if one aims to minimize the forward KL targeting the posterior [Naesseth et al., 2020].

So far we have outlined how to train $\tilde{q}_k$ but to evaluate the local ELBOs, $\mathcal{L}_k$, we need to construct a distribution q_k which satisfies the hard constraint $\mathrm{supp}(q_k) = \Theta_k$. Our solution for this is *truncating* $\tilde{q}_k$ by checking whether specific raw random draws $\theta'_{1:n_k}$ are valid for the path A_k, i.e. whether $\mathbb{I}\left[\theta'_{1:n_k} \in \Theta_k\right]$. We can do this by simply executing the program with fixed draws set to $\theta'_{1:n_k}$ and then noting that the program terminates and follows the address path A_k if, and only if, $\theta'_{1:n_k} \in \Theta_k$. Thus, we truncate $\tilde{q}_k$ using

$$q_k(\theta; \phi_k) = \frac{\tilde{q}_k(\theta; \phi_k)\mathbb{I}\left[\theta \in \Theta_k\right]}{\tilde{Z}_k(\phi_k)}, \tag{4.24}$$

$$\text{where} \quad \tilde{Z}_k(\phi_k) = \int_{\Theta'_k} \tilde{q}_k(\theta; \phi_k)\mathbb{I}\left[\theta \in \Theta_k\right] d\theta. \tag{4.25}$$

Hence, q_k is implicitly defined as the output of a rejection sampler with $\tilde{q}_k$ as a proposal. Note, that as we use the surrogate ELBO in Eq. (4.22) when training

ϕ_k, we never need to take gradients through q_k or $\tilde{Z}_k(\phi_k)$, thereby avoiding the significant practical issues this would cause (see Appendix A.6). Thus, the local guide q_k (Eq. (4.24)) is only used for estimating the local ELBOs (Eq. (4.13)). This is done by first drawing N samples $\{x^{(i)}\}_{i=1}^N$ from $\tilde{q}_k$, then rejecting samples which do not fall into the SLP and estimate $\tilde{Z}_k$ as the acceptance rate of this sampler (i.e. N_A/N where N_A is the number of samples accepted). Using A to denote the set of indices of accepted samples, we form our ELBO estimate as

$$\hat{\mathcal{L}}_k := \frac{1}{N_A} \sum_{i \in A} \log(N_A \, \gamma_k(\theta^{(i)})) - \log(N \, \tilde{q}_k(\theta^{(i)}; \phi_k)). \tag{4.26}$$

Note here that A and N_A are random variables that both implicitly depend on ϕ_k, which is why we can use this for estimation, but not training.

4.4.5.1 Exploiting Program Structure: Discrete Branching Optimization

In practice, many user-defined programs have structural properties which can be exploited to construct a valid local guide directly and deterministically (without resorting to the stochastic mechanism described in Section 4.4.5). Specifically, consider the class of programs whose program paths are determined solely by variables sampled from discrete distributions. For these programs, we can assume that for each SLP (kth, say) there is an (ordered) set of indices $I_{\text{branch}} \subset \{1, \ldots, n_k\} = I$ and a set of constants $r_{k,1}, \ldots, r_{k,|I_{\text{branch}}|} \in \mathbb{Z}$ such that the local unnormalized densities are expressible as

$$\gamma_k(\theta_{1:n_k}) = \gamma(\theta_{1:n_k}) \prod_{l=1}^{|I_{\text{branch}}|} \mathbb{I}\left[\theta_{I_{\text{branch}}[l]} = r_{k,l}\right]$$

where $I_{\text{branch}}[j]$ means the jth element in I_{branch}. It follows that we can construct densities for the kth SLP on a subset of variables in $\theta_{1:n_k}$ by eliminating all the variables given by indices I_{branch} (by instantiating them to constants). This is effectively equivalent to replacing the `sample` statements corresponding to the variables which influence the control flow with `observe` statements which induces a new program density that has the form

$$\tilde{\gamma}_k(\theta_{1:n'_k}) = \prod_{i=1}^{n'_k} g_{A_k[I'[i]]}(\theta_i|\eta_i) \prod_{l=1}^{|I_{\text{branch}}|} g_{A_k[I_{\text{branch}}[l]]}(r_{k,l} \mid \eta_l) \prod_{j=1}^{n_y} h_{b_j}(y_j \mid \phi_j) \tag{4.27}$$

where $I' := [1, \ldots, n_k] \backslash I_{\text{branch}}$, and $n'_k := |I'|$. Furthermore, if all the remaining r.v. are continuous distributions with support in $\mathbb{R}$ (i.e. $\text{supp}(f_{A_k[i]}) = \mathbb{R}$ for $i \in I'$) then $\tilde{\gamma}_k(\theta_{1:n'_k})$ itself has support in $\mathbb{R}^{n'_k}$. It is then straightforward to construct

a guide q_k with support in $\mathbb{R}^{n'_k}$ using existing methods, and we can get gradient estimates using the reparameterization gradient estimator (assuming there are no more discontinuities in $\tilde{\gamma}_k$).

To realize the discrete branching optimization in our Pyro implementation we allow users to annotate the `sample` statements which influence the branching. While it is in principle possible to automatically identify programs with discrete branching using program analysis, formalizing and implementing such a program analysis tool to work with arbitrary Pyro program would be a significant contribution in itself which is out of scope for this paper as we are focused on the statistical evaluation of SDVI. Specifically, the relevant `sample` statements within a Pyro program can be annotated as follows: `pyro.sample("x", dist.Poisson(7), infer={"branching": True})`. Our implementation of SDVI is then able to use these annotations to create the density $\tilde{\gamma}_k$ in Eq. (4.27).

4.5 Related Work

The vast majority of prior work on deriving automated VI algorithms focuses on the setting of static support [Ranganath et al., 2014, Kucukelbir et al., 2017, Rezende and Mohamed, 2015, Ambrogioni et al., 2021, Dhaka et al., 2021, Agrawal et al., 2020, Lee et al., 2018]. More generally, there have been models with stochastic support for which bespoke guides where developed which do not follow the control-flow structure of the input program [Eslami et al., 2016]. However, these custom guides do not leverage the breakdown of the input program into SLPs.

The Divide-Conquer-Combine (DCC) algorithm [Zhou et al., 2020] also exploits the breakdown of the program density into individual SLPs. However, Zhou et al. [2020] mainly focused on local inference algorithms that are sampling based, especially MCMC. As we showed in Section 4.4 unique challenges and opportunities arise when we consider the breakdown from a variational perspective. Further, our work shows that using a variational family based on SLPs naturally leads to divide-and-conquer style algorithm, due to the resulting separability of the ELBO. One of the most practical differences is that SDVI only requires (exponentiated) ELBOs to be estimated for each SLP, rather than marginal likelihoods. The former can typically be estimated substantially more accurately for a given budget, allowing SDVI to scale better to high dimensional problems (see Section 4.6.2). Chaganty et al. [2013] and Sankaranarayanan et al. [2013] both also use the general idea of breaking down programs into SLPs, but both papers consider starkly different problem settings. Neither have any direct link to variational inference.

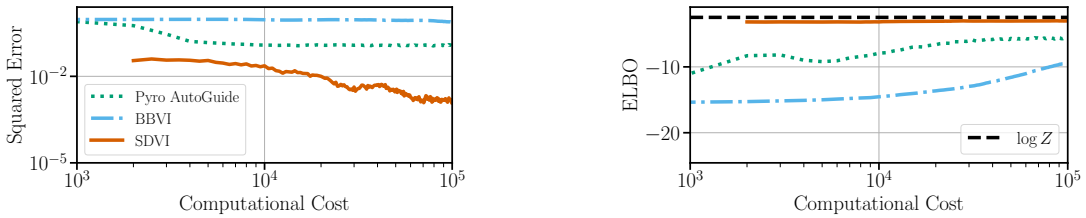
Our work is situated in the larger context of automated inference for universal PPSs. Other popular approaches include particle-based methods [Paige and Wood, 2014, Ge et al., 2018, Wood et al., 2014, Rainforth et al., 2016, Murray and Schön, 2018] and MCMC approaches with automated proposals [Wingate et al., 2011, Le, 2016, Mak et al., 2021b] (see Section 3.2 for more details).

Of particular relevance for variational inference is the related technique of *amortized* inference [Stuhlmüller et al., 2013, Paige and Wood, 2016, Le et al., 2017, Webb et al., 2018, Harvey et al., 2019] that we briefly discussed in Section 3.2.5. Constructing a guide in these settings faces additional challenges as we also need to define a mapping from the observed data to the variational parameters. This mapping often comes in the form of a complex neural network with many parameters that need to be trained on a large dataset (at the same time this makes it possible to scale the technique to complex simulators [Baydin et al., 2019]). Amortized techniques sometimes also target alternative objectives, such as the forward KL divergence, because the goal is no longer to learn a guide that directly approximates the posterior but instead to learn a guide that can be used as a proposal in an importance sampling setting [Le et al., 2017, Harvey et al., 2019]. In contrast, SDVI is able construct guides with fewer parameters[2] and it targets the conventional ELBO objective. Extending SDVI to the setting of amortized inference and comparing it against existing approaches for constructing amortized guides is an exciting direction for future research.

4.6 Experiments

To make SDVI easily accessible to practitioners we have implemented it in Pyro with code available at `github.com/treigerm/sdvi_neurips`. The first baseline we consider, **Pyro AutoGuide**, uses the `AutoNormalMessenger` class to automatically generate a guide, and trains it with Pyro's built-in tools for VI (`http://pyro.ai/examples/svi_part_i.html`). As an additional VI baseline, we also implement a custom guide for each model which uses the variable-by-variable scheme outlined in Section 4.3, in combination with the score function gradient estimator; we refer to this baseline as **BBVI**. For SDVI, we run SH until there are 10 active SLPs left (i.e. $m = 10$ in Algorithm 4) and parallelize the computation across 10 cores. We further construct each local guide distribution q_k as a mean-field normal. The specific configurations for each method for each experiment are provided in Appendix A.3.

[2] At the same time SDVI is able leverage advances from static support setting to construct flexible guides through e.g. normalizing flows. Whether to use more simple local guides with interpretable parameters or maximally flexible ones is a decision that is somewhat problem dependent. Crucially, SDVI gives the user the opportunity to make this decision.



(a) Squared error. **(b)** ELBOs.

Figure 4.2: Results for the model in § 4.6.1. Computational cost is measured in the number of likelihood evaluations. For each metric we show the mean and standard deviation over 10 runs. a) Squared error between the true SLP weights and the estimated SLP weights. b) Evidence Lower Bounds (ELBOs) for the variational algorithms, dashed line indicates the analytic log marginal likelihood.

4.6.1 Program with Normal Distributions

We use our first experiment to further clarify the failure modes of existing VI approaches. We consider an extension of the model from Figure 4.1 to contain more SLPs. The full model is

$$u \sim \mathcal{N}(0, 5^2),$$
$$x \sim \mathcal{N}(z, 1), \quad \text{where} \quad z = \begin{cases} 0, & \text{if } u \in (-\infty, -4] \\ K, & \text{if } u \in (-5+K, -4+K] \text{ for } K = 1, \ldots, 8 \\ 9, & \text{if } u \in (4, \infty) \end{cases}$$
$$y \sim \mathcal{N}(x, 1).$$

We assume we have observed $y = 2$. The results in Figure 4.2 demonstrate that SDVI is able to overcome the limitations of the other variational approaches. BBVI and Pyro AutoGuide both use the same guide in this model; BBVI uses the score function gradient estimator for training, whereas Pyro AutoGuide uses the reparameterized gradient estimator. This difference results in different posterior approximations for the different baselines. The BBVI guide tends to place all its mass on a single SLP and then provides a suitable approximation for only that one SLP, ignoring all the others. This explains the large standard deviations for the ELBO values in Figure 4.2b as the ELBOs in different SLPs will converge to drastically different values. For Pyro AutoGuide the biased gradient estimates will train the variational approximation for variable u to be close to the prior $\mathcal{N}(0, 5^2)$. SDVI is able to avoid the shortcomings of the baselines as it provides an overall better posterior approximation leading to larger ELBO values, i.e. lower KL divergences to the true posterior, and a more accurate weighting of the different SLPs (Figure 4.2a).

4.6.2 Infinite Gaussian Mixture Model

Our next model is a Gaussian Mixture Model (GMM) with a Poisson prior on the number of clusters:

$$\begin{aligned} K &\sim \text{Poisson}(9) + 1; \\ u_k &\sim \mathcal{N}(\mathbf{0}, 10\,\mathrm{I}) \text{ for } \; k = 1, \ldots, K; \\ y &\sim \frac{1}{K} \sum_{k=1}^{K} \mathcal{N}(\mu_k, 0.1\,\mathrm{I}), \end{aligned}$$

where I is the $D \times D$ identity matrix and $\mathbf{0}$ is a D dimensional vector of zeros (we set $D = 100$). A similar model was considered in Zhou et al. [2020] but with $D = 1$ instead of $D = 100$. We generate a dataset of 1250 observations with $K = 5$. To compare and evaluate the different algorithms, we hold out 250 data points as a test dataset to compute the log posterior predictive density (LPPD).

The Pyro AutoGuide baseline from the previous experiment is not applicable here since it assumes all latent variables are continuous. In BBVI, for practical reasons, we had to cap the maximum number of clusters in the guide at 25 (cf. Appendix A.3). To provide a further baseline, we have also implemented **DCC** [Zhou et al., 2020] in Pyro with Random-walk lightweight Metropolis-Hastings (RMH) [Le, 2016] as a local inference algorithm. We chose DCC in particular because it also exploits the same breakdown into SLPs, so comparing against DCC is an opportunity to highlight the benefits of using a VI method.

In this model, the observations are assumed to be conditionally independent given the latent variables, thus enabling SDVI to work on subsets of the whole dataset [Hoffman et al., 2013, Kucukelbir et al., 2015]. Specifically, we run SDVI on a model which samples a random minibatch of size $B = 100$ at each iteration and then scales the likelihood by the factor N/B, where $N = 1000$ is the size of the full dataset; we refer to this setup as Stochastic SDVI (S-SDVI). Furthermore, for this model SDVI is able to directly construct valid local guides q_k (using the mechanism for models branching on discrete variables outlined in Section 4.4.5) and therefore (S-)SDVI can use the reparameterized gradient estimator.

Table 4.1 shows that SDVI and S-SDVI significantly outperform the baselines, yielding a several orders of magnitude larger posterior predictive density and providing the only reasonable predictions for the numbers of clusters. In the few instances were (S-)SDVI returns a suboptimal MAP estimate of $K = 6$, this was because the local guide for the SLP with 5 components had fallen into a local model that fails to correctly identify all the clusters in the data, in turn returning a suboptimal local ELBO. BBVI and DCC struggle with this model due to the

Table 4.1: Log posterior predictive densitiy (LPPD), ELBO, and maximum a posteriori (MAP) estimate for K for GMM model. Mean and standard deviation for LPPD and ELBO computed over 5 runs.

Method	LPPD ($\uparrow, \times 10^3$)	ELBO ($\uparrow, \times 10^3$)	MAP K
DCC	-9842.90 ± 3904.57	N/A	14, 11, 16, 14, 15
BBVI	-2217.07 ± 146.31	-8770.55 ± 544.95	25, 25, 25, 25, 25
SDVI	$\mathbf{32.84 \pm 0.02}$	$\mathbf{128.76 \pm 0.17}$	5, 5, 6, 6, 5
S-SDVI	$\mathbf{32.80 \pm 0.02}$	$\mathbf{128.63 \pm 0.22}$	5, 5, 6, 5, 6

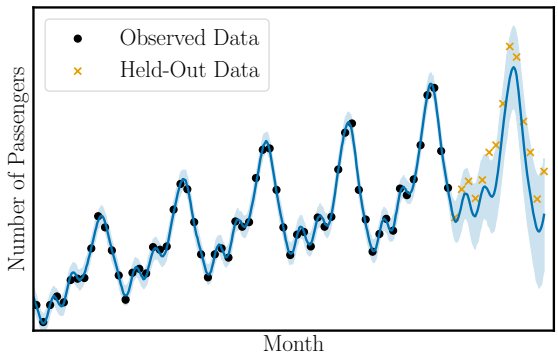


Figure 4.3: Posterior predictions of the GP for SDVI, shaded regions indicate 2 standard deviations that are computed from 100 posterior samples.

high-dimensional parameter space. DCC's local inference algorithm, RMH, only updates one variable at a time which results in slow mixing times. Note, DCC does not provide any ELBO values; its marginal likelihood estimator PI-MAIS [Martino et al., 2017] constructs an importance sampling (IS) proposal distribution based on the outputs of MCMC chains which could theoretically be used to estimate an ELBO value. However, as IS requires over-dispersed proposals, the ELBO scores for this approach are trivially $-\infty$, preventing a sensible comparison.

4.6.3 Inferring Gaussian Process Kernels

For our final experiment, we consider the problem of inferring the kernel structure of a Gaussian Process (GP). Following [Duvenaud et al., 2013, Janz et al., 2016], we place a prior over kernel functions using a probabilistic context-free grammar (PCFG) . We consider the squared exponential (SE), rational quadratic (RQ), periodic (PER), and linear (LIN) base kernels, and use the production rules

$$\mathcal{K} \rightarrow \text{SE} \mid \text{RQ} \mid \text{PER} \mid \text{LIN} \mid \mathcal{K} \times \mathcal{K} \mid \mathcal{K} + \mathcal{K}.$$

Table 4.2: Final log posterior predictive densitiy (LPPD) and ELBO for GP model. Shown are mean and standard deviation computed over 5 runs.

Method	LPPD (↑)	ELBO (↑)
DCC	-58.92 ± 32.47	N/A
BBVI	-18.82 ± 1.20	-48.48 ± 0.33
SDVI	$\mathbf{2.05 \pm 3.30}$	$\mathbf{34.53 \pm 21.42}$

Sampling from the PCFG is implemented with a recursive probabilistic program that uses samples from a categorical distribution to decide which production rule in the PCFG should be applied. In addition to the kernel structure, we also perform inference over the kernel hyperparameters for each base kernel and the observation noise; we place an inverse-gamma prior on each base kernel hyperparameter and a half-normal prior on the observation noise. We further assume a normal likelihood function and marginalize out the latent GP. Additional model details are in Appendix A.3. We apply this model to a dataset of monthly counts of international airline passengers [Box et al., 2015], withholding the last 10 % of all observations as a test dataset.

For SDVI we can construct local valid proposals q_k using the mechanism for models with discrete branching outlined in Section 4.4.5. Hence, in each SLP the local guide q_k provides a posterior approximation over the kernel hyperparameters and the observation noise; the posterior distribution over kernel structures is implicitly defined through the mixture distribution over program paths. Table 4.2 shows that SDVI provides higher LPPD values, and is also able to achieve a higher final ELBO value compared to BBVI. Figure 4.3 shows the posterior predictions for the SDVI run with the median LPPD score. SDVI is able to provide qualitatively reasonable predictions, as the predictions follow the periodic trend in the observed data.

4.7 Discussion

We believe that SDVI provides a number of significant contributions towards the goal of effective (automated) inference for probabilistic programs with stochastic support, nonetheless it still naturally has some limitations. Perhaps the most obvious is that it, if there is a very large number of SLPs that cannot be easily discounted from having significant posterior mass, it can be challenging to learn effective variational approximations for all of them, such that SDVI is likely to perform poorly if the number becomes too large. Here, customized conventional

VI or reversible jump MCMC approaches might be preferable, as they can be set up to focus on the transitions between SLPs, rather than trying to carefully characterize individual SLPs.

Another limitation is that our current focus on automation means that there are still open questions about how best to construct more customized guides within the SDVI framework. Here the breakdown into individual SLPs and use of resource allocation strategies will still often be useful, but changes to our implementation would be required to allow more user control and customization. For example, the discovery of individual SLPs using the prior is a potential current failure mode, and it would be useful to support the use of more sophisticated program analysis techniques (e.g. [Beutner et al., 2022]).

A more subtle limitation is that the local inferences of each SLP can sometimes still be quite challenging themselves. If the boundaries constraining the support for an SLP are very complex, it will be challenging to learn a good guide, meaning we might need advanced local variational families (e.g. normalizing flows [Papamakarios et al., 2021]) and/or gradient estimators [Lee et al., 2018, 2023, Lew et al., 2023, Wagner et al., 2024]. In general, it might also be possible to extend the space of programs for which we can analytically construct valid guides for individual SLPs, as we have done for programs with discrete branching. Such problems also occur in static support settings and are usually much more manageable than the original stochastic support problem, but further work is needed to fully automate dealing with them.

Finally, variational methods are often used not only for inference, but as a basis for model learning as well. In principle, SDVI could also be used in such settings, but as described in Appendix A.5, there are still some hurdles that need to be overcome to do this in practice.

4.8 Conclusion

We have presented SDVI and shown that it is able to overcome the limitations of existing VI approaches for programs with stochastic support by using a novel guide structure that breaks the program down into SLPs with fixed support, rather than matching the original stochastic control flow. The structure of the variational family separates the ELBO into multiple independent inference problems which naturally motivates a divide-and-conquer style training procedure with explicit resource allocation. Experimentally we found that these innovations meant that SDVI was able to provide significant performance improvements over the previous state-of-the-art approaches.

5

Beyond Bayesian Model Averaging over Paths in Probabilistic Programs with Stochastic Support

5.1 Introduction

As we saw in previous chapters, universal probabilistic programming systems (PPSs) provide flexible frameworks for expressing powerful probabilistic models, along with tools to aid performing inference in them. By permitting branching on the outcomes of sampling statements, they allow users to express programs with *stochastic support*, wherein the number of latent variables varies between program executions, leading to challenging inference problems.

We showed in Chapter 3 that such programs can be thought of as a combination of independent sub-programs, each with static support, known as straight-line programs (SLP) [Chaganty et al., 2013, Sankaranarayanan et al., 2013, Luo et al., 2021]. The overall posterior is then given by the weighted sum of individual SLP posteriors, with weights corresponding to the local normalization constants of the SLPs; a breakdown we exploited in Chapter 4 to derive an improved variational inference algorithm for programs with stochastic support.

In this chapter, we show that this decomposition also reveals that the posterior of *any* program with stochastic support is a Bayesian Model Averaging (BMA) [Hoeting et al., 1999] over the constituent SLPs of the program. Thus, all PPS inference engines are implicitly estimating a BMA when the program has stochastic support, whether they explicitly account for this or not.

However, it is widely acknowledged in the Bayesian statistics literature that BMA can be a problematic mechanism for combining the posteriors of individual models [Minka, 2000, Yao et al., 2018], with alternatives often preferred in practice, especially when our aim is to make good predictions. In particular, BMA often performs poorly under *model misspecification* [Gelman and Yao, 2020, Oelrich et al., 2020], wherein it tends to produce *overconfident* posterior model weights that collapse towards a single model [Huggins and Miller, 2021, Yang and Zhu, 2018]. Given that models will rarely be perfect when working with real data [Box, 1976, Key et al., 1999, Vehtari and Ojanen, 2012], this is a serious practical concern that has been observed to cause notable issues in many applied fields [Yang and Zhu, 2018, Smets and Wouters, 2007, Leff et al., 2008].

We argue that PPSs need to account for these shortfalls and provide access to more robust weighting schemes. To provide such alternatives, we suggest optimizing the SLP weights for *predictive performance*. Specifically, we introduce weighting schemes based on *stacking of predictive distributions* [Wolpert, 1992, Breiman, 1996, LeBlanc and Tibshirani, 1996, Yao et al., 2018] and *PAC-Bayes objectives* [Masegosa, 2020, Masiha et al., 2021, Morningstar et al., 2022, Alquier, 2023]. We show how to run them as a cheap post-processing step on the outputs of any sample-based inference scheme, and demonstrate that they provide more robust weights with better predictive performance.

In summary, our contributions are: (a) By interpreting the posterior in programs with stochastic support as a BMA, we show that the weights assigned to SLPs can be unstable, e.g. due to model misspecificiation. (b) Providing a general scheme to adapt PPS inference algorithms to utilize alternative weighting schemes, with an implementation in Pyro [Bingham et al., 2019]. (c) Investigating their behaviour for a variety of different programs and showing its benefits on synthetic and real-world data.

5.2 Background

5.2.1 Bayesian Model Averaging

An important question in Bayesian statistics is how to best combine the inferences of different possible models. In a pure Bayesian framework, this is done by weighting the model posteriors according to their *posterior model probability*, leading to a framework called Bayesian model averaging (BMA) [Hoeting et al., 1999].

To be more precise, assume we have a countable set of Bayesian models indexed by k, each with corresponding latent parameters $\theta_k \in \Theta_k$, prior $p_k(\theta_k)$, and likelihood

$p_k(y|\theta_k)$, where y is data we want to condition on.[1] In BMA, we set a prior over which model generated the data, $p(M = k)$, from which we can derive the posterior model probability

$$p(M{=}k \mid y) \propto p(y \mid M{=}k)\, p(M{=}k) \tag{5.1}$$

where $p(y \mid M{=}k) = \int p_k(y|\theta_k)\, p_k(\theta_k) d\theta_k$ is the *model evidence*, or marginal likelihood, for the kth model.

Predictions and expectations can now be calculated by combining those from individual models using $p(M{=}k \mid y)$ as weights. In particular, the posterior predictive distribution for new hypothetical data, $\tilde{y}$, is given by

$$p(\tilde{y} \mid y) = \sum_k p(M{=}k \mid y)\, \mathbb{E}_{p_k(\theta_k|y)} \left[p_k(\tilde{y}|\theta_k) \right], \tag{5.2}$$

where $p_k(\theta_k|y) \propto p_k(\theta_k) p_k(y|\theta_k)$ and $p_k(\tilde{y}|\theta_k)$ are the local posterior and local parameterized predictive distribution, respectively.

5.2.1.1 Criticisms of BMA

In practice, our models will never be able to capture the full complexities of the real world as, in the words of George Box, "all models are wrong, some are useful" [Box, 1976]. It is therefore important to investigate the behaviour of frameworks when our models are misspecified, that is when there exist no θ_k and k s.t. $p_k(y|\theta_k) = p_{\text{true}}(y)$ for all possible y, where $p_{\text{true}}(y)$ is the (unknown) true data generating distribution.

Crucially, BMA implicitly assumes that the data was sampled from exactly one of the constituent models. This is often referred to as the $\mathcal{M}$-closed assumption [Bernardo and Smith, 2009, Clyde and Iversen, 2013, Key et al., 1999]. As a result, as the amount of data increases the BMA weights will always (except for a few special edge cases) collapse on a single model [Clyde and Iversen, 2013]; the approach reverts to just performing model selection. Consequently, BMA predictions are often inferior compared to other model combination techniques [Minka, 2000, Yao et al., 2018].

Viewed another way, model misspecification tends to lead to posterior model probabilities that are *overconfident*: both empirical and theoretical results have shown that they too readily collapse on a single model [Huggins and Miller, 2021, Yang and Zhu, 2018], even when there are multiple plausible models with similar predictive performance. Moreover, the exact model onto which the posterior collapses can change drastically when regenerating the data from $p_{\text{true}}(y)$. In

[1] Note our formulations apply equally when there are also inputs the model is conditioned on, i.e. we have $p_k(y|\theta_k, x)$, but we negate this from our notation to avoid clutter.

general, we expect there to be some variance in the BMA weights due to the fact that we need to *estimate* the model evidence for many real-world models. However, previous work has demonstrated that overconfidence is an issue even with *analytic* BMA weights and not an artifact of using approximate inference algorithms [Huggins and Miller, 2021, Oelrich et al., 2020].

5.2.2 Introduction to Scoring Rules

Scoring rules are functions which take as input a *probabilistic forecast* and a realized event.[2] The goal of scoring rules is then to evaluate the quality of the probabilistic forecast. In a Bayesian context, these scores are often instead referred to as utilities and Bayesian decision theory aims to maximize the predicted utility of a given action (see Chapter 2).

More formally, assume we have a random variable on the sample space $(\Omega, \mathcal{A})$ and $\mathcal{P}$ is a convex class of probability measures on $(\Omega, \mathcal{A})$. Then, any member $P \in \mathcal{P}$ is referred to as a probabilistic forecast and a scoring rule is a function $S : \mathcal{P} \times \Omega \to [-\infty, \infty]$ s.t. $S(P, \cdot)$ is $\mathcal{P}$-quasi-integrable for all forecasts $P \in \mathcal{P}$. So for a probabilistic forecast P and observed event $y \in \Omega$, $S(P, y)$ is the score which indicates the quality of our forecast. For notational convenience, if P and Q are both probabilistic forecasts, we define $S(P, Q) = \int S(P, y) dQ(y)$. Then a scoring rule is *proper* if $S(Q, Q) \geqslant S(P, Q)$ holds for all $P \in \mathcal{P}$, and *strictly proper* if the equality holds only when $P = Q$ almost surely.

Some common examples of scoring rules include: the *quadratic score* $S(\rho, y) = 2\rho(y) - \|\rho\|_2^2$, where ρ is a predictive density; the *logarithmic score* $S(\rho, y) = \log \rho(y)$; and the *continuous-ranked probability score* $S(F, y) = -\int (F(y') - \mathbb{I}[y' \geqslant y]) dy'$, where F is the cumulative distribution function of the forecast. Under regularity conditions, Bernardo [1979] showed that the logarithmic scoring rule is the only proper *local* scoring rule where a local scoring rule is a rule that depends on the predictive density ρ only through the actual observed event y.

We refer the reader to Gneiting and Raftery [2007] for more extensive details on scoring rules.

[2] This introduction mainly follows from Gneiting and Raftery [2007] and Yao et al. [2018].

5.2.3 Programs with Stochastic Support

As outlined in Chapter 5, a probabilistic program can be interpreted as defining an *unnormalized density function* $\gamma : \Theta \to \mathbb{R}^{\geqslant 0}$, where Θ denotes the sample space of the latent variables in the program [Borgström et al., 2016, Staton et al., 2016]. The goal of inference is then to find a representation of the normalized program density $\pi(\theta) = \gamma(\theta)/\int \gamma(\theta) d\theta$, where $d\theta$ is an implicitly defined reference measure [Gordon et al., 2014, Rainforth, 2017, van de Meent et al., 2018]. One can informally think of $\pi(\theta)$ as a posterior distribution, $p(\theta|y)$.

Universal PPS allow users to branch on the outcomes of random sampling statements leading to programs with *stochastic support*. As we saw in Chapter 4, an important property of such programs is that they can be decomposed into (a countable number of) straight-line programs (SLPs), sub-programs without any control flow [Chaganty et al., 2013, Sankaranarayanan et al., 2013, Zhou et al., 2020, Luo et al., 2021, Reichelt et al., 2022a]. These SLPs are effectively the different possible control-flow paths that exist in the program and they are defined by their *address path*, i.e. the sequence of the lexical addresses encountered during the program's execution.

Each SLP corresponds to a disjoint sub-region, Θ_k, of the overall sample space (such that $\Theta = \bigcup_k \Theta_k$) and has the *local unnormalized density* $\gamma_k(\theta) := \mathbb{I}\left[\theta \in \Theta_k\right] \gamma(\theta)$. The unnormalized density for the whole program can thus be written as $\gamma(\theta) = \sum_k \gamma_k(\theta)$. Similarly, the normalized program density can be rewritten as

$$\pi(\theta) = \sum_k \frac{Z_k}{\sum_\ell Z_\ell} \pi_k(\theta), \tag{5.3}$$

where $Z_k = \int \gamma_k(\theta) d\theta$ and $\pi_k(\theta) = \gamma_k(\theta)/Z_k$ are the *local normalization constant* and *local posterior* respectively. We refer to $\pi(\theta)$ as the *full Bayes posterior*. Note that the disjoint supports of the SLPs means that there exists exactly one $k : \pi_k(\theta) > 0$ for any given θ.

5.3 Full Inference in Programs with Stochastic Support is BMA

Examining Eq. (5.3), we immediately see that $\pi(\theta)$ is a weighted sum of localized posteriors. This decomposition also reveals that using the full Bayes posterior to calculate predictions or expectations is implicitly performing a BMA over

the individual SLPs. To see this, consider calculating the expectation of some parameterized predictive density $p(\tilde{y}|\theta)$:

$$\mathbb{E}_{\pi(\theta)}\left[p(\tilde{y}|\theta)\right] = \sum_k \frac{Z_k}{\sum_\ell Z_\ell} \mathbb{E}_{\theta_k \sim \pi_k}\left[p_k(\tilde{y}|\theta_k)\right], \tag{5.4}$$

where p_k is any conditional density function such that $p_k(\tilde{y}|\theta) = p(\tilde{y}|\theta)\,\forall \tilde{y}, \theta \in \theta_k$, and we have defined new random variables θ_k drawn from the local posterior of the kth SLP. We thus have that the downstream posterior predictive on $\tilde{y}$ is a weighted sum of the posterior predictives that would result from using the kth SLP instead of our full program.

There is now a clear analog between Eq. (5.4) and Eq. (5.2). To show that the former corresponds to a BMA, all that remains is to show that the weights can be interpreted as posterior model probabilities. At a high-level, this follows simply from the fact that the Z_k are analogous to (unnormalized) posterior model probabilities (note, though, they are *not* analogous to the model evidences).

To be more precise, consider the factorization $\gamma(\theta) = g(\theta)h(\theta)$ where $g(\theta)$ corresponds to all terms from the sampling and $h(\theta)$ all terms from conditioning statements, such that we can think of them as prior and likelihood components respectively. The prior probability of choosing the kth SLP is now given by $P_k := \int g(\theta)\mathbb{I}\left[\theta \in \Theta_k\right] d\theta$, while the "model evidence" is

$$E_k := \int \frac{g(\theta)\mathbb{I}\left[\theta \in \Theta_k\right]}{P_k} h(\theta) d\theta. \tag{5.5}$$

The posterior model probability is then equal to

$$\frac{P_k E_k}{\sum_\ell P_\ell E_\ell} = \frac{\int g(\theta)\mathbb{I}\left[\theta \in \Theta_k\right] h(\theta) d\theta}{\sum_\ell \int g(\theta)\mathbb{I}\left[\theta \in \Theta_\ell\right] h(\theta) d\theta} = \frac{Z_k}{\sum_\ell Z_\ell}. \tag{5.6}$$

Thus, Eq. (5.4) is a BMA with model prior $p(M = k) = P_k$, local posteriors $p_k(\theta_k|y) = \pi_k(\theta_k)$, model evidences $p(y|M = k) = E_k$, and identical local parameterized predictive distributions $p_k(\tilde{y}|\theta_k) = p(\tilde{y}|\theta = \theta_k)$.

Having realized that using the full Bayes posterior leads to BMA, we can instead define a generalized model averaging scheme over the SLPs that is explicitly parameterized by a learnable set of weights, w:

$$\tilde{\pi}(\theta; w) := \sum_k w_k\, \pi_k(\theta) \tag{5.7}$$

with $\sum_k w_k = 1, w_k \geqslant 0$. This opens the door for considering alternative approaches that avoid the shortfalls of BMA. We refer to $w_k \propto Z_k$ as the *BMA weights*; the choice made by all current inference engines.

At this point, a critical reader might argue that all Bayesian inference in general, and not just the weights in a BMA, is sensitive to model misspecification. However, averaging over a finite discrete set of models has been highlighted as a special case in which Bayesian inference can give counter-intuitive results and is especially susceptible to misspecification [Yao et al., 2018, Gelman and Yao, 2020, Oelrich et al., 2020]. Additionally, in most realistic models used in practice we need to estimate the posterior and the local normalization constants. As we will show in our experiments in Section 5.6 this can be an additional source of variance leading to sub-optimal predictions. This thus motivates treating the SLP weights as explicit parameters that we may wish to set in a non-Bayesian manner, while leaving the local posteriors unchanged.

5.4 Weighting Program Paths Using Predictive Objectives

There are different ways we can choose the SLP weights, w_k, in Eq. (5.7), with BMA only one possible choice. A simple, albeit crude, alternative would be to just set them equally. While we find that this can sometimes empirically outperform the BMA weights (see Section 5.6), it is clearly not an appropriate general-purpose solution and there are cases where it can perform very poorly.

To provide more principled alternatives, we now show how the weights can be optimized to maximize predictive performance. For the purpose of exposition, we will introduce the main ideas through the eyes of *stacking* [Yao et al., 2018, Wolpert, 1992, Breiman, 1996, LeBlanc and Tibshirani, 1996] but we will show in Section 5.4.4 how we can also use PAC-Bayes objectives [Morningstar et al., 2022] to fit the SLP weights.

5.4.1 Stacking Objective for PPSs

The goal of stacking is to improve predictions by optimizing the model weights, w. To achieve this, we need to define a method for making predictions for a hypothetical new observation, which we denote as $\tilde{y} \in \mathcal{Y}$. Just as we previously defined a generalized version of the posterior in Eq. (5.7), we now need to establish a generalized version of the posterior predictive.

For simplicity, we will assume for now that an explicit predictive density, $p(\cdot \mid \cdot) : \mathcal{Y} \times \Theta \to \mathbb{R}^{\geqslant 0}$, has been provided, before showing how this can instead be

derived from the program itself in Section 5.4.2. This ensures for each SLP we have a *local posterior predictive density*

$$\rho_k(\tilde{y}) := \mathbb{E}_{\pi_k(\theta)}\left[p(\tilde{y} \mid \theta)\right]. \tag{5.8}$$

With this, we can define the *stacked predictive density*

$$\tilde{\rho}(\tilde{y}; w) := \mathbb{E}_{\tilde{\pi}(\theta;w)}\left[p(\tilde{y} \mid \theta)\right] = \sum_k w_k\, \rho_k(\tilde{y}) \tag{5.9}$$

In its most general form, stacking defines an objective with a user-defined scoring rule $S(\tilde{\rho}, \tilde{y})$ which takes as input a predictive distribution and a data point (Gneiting and Raftery [2007]; see Section 5.2.2). It then optimizes the weights, w, to maximize the expected score

$$R(w; S) := \mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[S(\tilde{\rho}(\cdot \mid w), \tilde{y})\right]. \tag{5.10}$$

We will focus on using the logarithmic score rule, as it is by far the most popular one used in practice, yielding the objective

$$R(w) := \mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log\left(\sum_k w_k\, \rho_k(\tilde{y})\right)\right]. \tag{5.11}$$

Maximizing the stacking objective is equivalent to minimizing the KL divergence between the true data generating distribution and the stacked predictive density [Gneiting and Raftery, 2007, Yao et al., 2018], due to the relationship

$$\begin{aligned} \mathrm{KL}(p_{\text{true}}(\tilde{y}) \parallel \tilde{\rho}(\tilde{y}; w)) &= \mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log p_{\text{true}}(\tilde{y})\right] - \mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log \tilde{\rho}(\tilde{y}; w)\right], & (5.12)\\ &= \mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log p_{\text{true}}(\tilde{y})\right] - R(w), & (5.13) \end{aligned}$$

where the first term, $\mathbb{E}_{p_{\text{true}}(\tilde{y})}\left[\log p_{\text{true}}(\tilde{y})\right]$, is a constant that does not depend on the stacking weights, w.

We now need a mechanism to estimate the expectation w.r.t. $p_{\text{true}}(\tilde{y})$ in Eq. (5.11). Multiple strategies for this exist [Vehtari and Ojanen, 2012]. We will first show how to do so using an explicit validation set, $\{\tilde{y}_\ell\}_{\ell=1}^{L}$, before describing how to avoid the use of a validation set for a broad class of models in Section 5.4.3.

As an aside, the stacking weights should be interpreted differently from BMA weights. In BMA, the weight of the kth SLP represents the posterior probability that the data was generated from the kth SLP. In contrast, stacking generates a mixture of the local posterior predictive distributions and optimizes the mixture weights on held-out data. Hence, the stacking weight for the kth SLP estimates the probability that a new data point is drawn from the kth SLP.

Algorithm 5 Stacking as Post-Processing (Section 5.4.2)

Require: Program γ, Weighted samples from base inference procedure $\{(v_s, \theta_s)\}_{s=1}^{S}$
1: For each θ_s, record address path and return values $\{g(\tilde{y}_\ell \mid \theta_s)\}_{\ell=1}^{L}$ ▷ Sec. 5.4.2
2: Partition indices $1, \ldots, S$ into subsets $\{I_k\}_{k=1}^{K}$ using address paths ▷ Sec. 5.4.2
3: Compute predictive densities $\hat{\rho}_k(\tilde{y}_\ell)$ ▷ Eq. (5.18)
4: Compute $w^* = \operatorname{argmax} \hat{R}(w)$ ▷ Eq. (5.19)
5: Compute new sample weights ω_s ▷ Eq. (5.20)
6: **return** $\{(\omega_s, \theta_s)\}_{s=1}^{S}$

5.4.2 Stacking as Post-Processing

Given (normalised) weighted samples $\{(v_s, \theta_s)\}_{s=1}^{S}$ generated from an arbitrary inference algorithm, the posterior of the program is approximated by the empirical measure

$$\hat{\pi}(\theta) = \sum_{s=1}^{S} v_s \, \delta_{\theta_s}(\theta), \tag{5.14}$$

unweighted sampling schemes correspond to the special case $v_s = 1/S$. The local posteriors of the kth SLP are consequently approximated by all the samples which fall into the kth SLP, i.e.

$$\hat{\pi}_k(\theta) := \sum_{s \in I_k} \frac{v_s}{V_k} \, \delta_{\theta_s}(\theta) \tag{5.15}$$

where $I_k := \{s \in \{1, \ldots, S\} \mid \theta_s \in \Theta_k\}$ are the indices of the samples from the kth SLP and $V_k := \sum_{s \in I_k} v_s$ is the sum of all the associated sample weights. Recall from Section 5.2.3 that the SLP of a sample θ_s is determined by its address path. Thus, we can generate the index sets I_k by grouping all samples with the same path.

The full Bayes posterior implicitly uses the approximation $\hat{\pi}(\theta) = \sum_k V_k \, \hat{\pi}_k(\theta)$, assigning the weight V_k to each SLP. We instead replace these with the learnable SLP weights w_k:

$$\tilde{\pi}(\theta; w) \approx \sum_k w_k \, \hat{\pi}_k(\theta) = \sum_k \sum_{s \in I_k} \frac{w_k \, v_s}{V_k} \, \delta_{\theta_s}(\theta). \tag{5.16}$$

We can then use this approximation to get an estimate of the stacked predictive density defined in Eq. (5.9)

$$\tilde{\rho}(\tilde{y}_\ell; w) \approx \sum\nolimits_k w_k \, \hat{\rho}_k(\tilde{y}_\ell) \tag{5.17}$$

$$\text{where} \quad \hat{\rho}_k(\tilde{y}_\ell) := \sum\nolimits_{s \in I_k} (v_s / V_k) \, p(\tilde{y}_\ell \mid \theta_s) \tag{5.18}$$

are the local posterior approximations. In our implementation, the user implicitly defines the $p(\tilde{y}_\ell \mid \cdot)$ through the program return values. We can now approximate $R(w)$ using

$$\hat{R}(w) := \frac{1}{L} \sum_{\ell=1}^{L} \log \left(\sum_k w_k \, \hat{\rho}_k(\tilde{y}_\ell) \right). \tag{5.19}$$

Note that, as the $\hat{\rho}_k(\tilde{y}_\ell)$ do not depend on the SLP weights, we can precompute these estimates before optimizing w in a separate, typically cheap, procedure.

After having obtained the optimized weights, $w^* = \text{argmax}_w \hat{R}(w)$, we want to be able to obtain estimates w.r.t. the reweighted normalized density $\tilde{\pi}(\theta; w^*)$. This can be done easily by reweighting the individual posterior samples θ_s. Letting $k(\theta_s)$ denote the SLP index of sample θ_s we can rewrite Eq. (5.16) as

$$\tilde{\pi}(\theta; w) \approx \sum_{s=1}^{S} \omega_s \, \delta_{\theta_s}(\theta), \quad \text{with} \quad \omega_s := \frac{w_{k(\theta_s)} \, v_s}{V_{k(\theta_s)}}. \tag{5.20}$$

Alg. 5 summarizes the high-level steps of our post-processing stacking procedure. Given an input program and corresponding (weighted) posterior samples, we first use the program to extract the address path and return values for each sample θ_s (e.g. using the `Trace` data structure in Pyro). Then, we compute the index subsets $I_1, \ldots, I_K$ by grouping unique address paths together. We can then compute the estimate of the local posterior predictive densities $\hat{\rho}_k(\tilde{y}_\ell)$ and store the estimates in a $K \times L$ matrix. Finally, this matrix can be used to evaluate our stacking objective $\hat{R}(w)$ and thus optimize the weights. This optimization can be done cheaply relative to the cost of inference as it is a convex optimization of a small number of parameters and does not require further inferences; we use the L-BFGS-B algorithm [Byrd et al., 1995, Zhu et al., 1997] for this. In Appendix B.3 we provide further details on our full implementation in Pyro.

5.4.3 Stacking Without Validation Sets

So far, we have assumed the existence of an explicit predictive density p and held-out data $\{\tilde{y}_\ell\}_{\ell=1}^{L}$, but neither is often actually needed to utilize stacking. Specifically, if we assume that the local unnormalized SLP densities model the observed data $y_i \in \mathcal{Y}$ as conditionally independent given parameters θ, then for each SLP we can write the local unnormalized density as

$$\gamma_k(\theta, y_{1:N}) = \mathbb{I}\left[\theta \in \Theta_k\right] g_k(\theta) \prod_{i=1}^{N} h_k(y_i \mid \theta),$$

where $g_k : \Theta \to \mathbb{R}^{\geqslant 0}$ represents a prior density on θ. We then use this to derive the following leave-one out (LOO) cross-validation estimator for Eq. (5.11),

$$R_{\mathrm{LOO}}(w) = \frac{1}{N}\sum_{i=1}^{N} \log \sum_k w_k \, \rho_k(y_i \mid y_{-i}),$$

where $\rho_k(y_i \mid y_{-i})$ denotes the local posterior predictive density corresponding to $\gamma_k(\theta, y_{1:N} \backslash y_i)$, i.e. that results from removing the i-th observation term from γ_k.

Naively computing each of the predictive distributions $\rho_k(y_i \mid y_{-i})$ would require running inference N times to evaluate the stacking objective. However, as shown in Yao et al. [2018], this can be avoided using Pareto smoothed importance sampling to estimate the LOO densities (PSIS-LOO) [Vehtari et al., 2017]. Computing the PSIS-LOO approximations to the densities $\rho_k(y_i \mid y_{-i})$ requires access the individual likelihood terms $h_k(y_i \mid \theta_s)$ for each posterior sample θ_s (c.f. Appendix B.2). Luckily, these can be extracted automatically for many common PPSs, e.g. using the `loo` function of the ArviZ library [Kumar et al., 2019] for Pyro models.

5.4.4 Regularized Stacking and PAC-Bayes

Stacking directly optimizes an estimate of the expected predictive density on held-out data (Eq. (5.11)). Such estimates are fundamentally based on a finite amount of data and optimizing them directly can, at least in principle, lead to overfitting. Our proposed remedy for this is to add an additional KL regularization term inspired by PAC-Bayes objectives. Namely, we consider

$$\begin{aligned} R_\beta(w) := \frac{1}{L}\sum_{\ell=1}^{L} \log \left(\sum_k w_k \rho_k(\tilde{y}_\ell) \right) \\ - (1/\beta L)\, \mathrm{KL}(\mathrm{Categorical}(w_1, \ldots, w_K) \parallel r(k)), \end{aligned}$$

where $r(k)$ is a reference weighting we want to regularize towards. Since we want to discourage the SLP weights from collapsing towards a single SLP, we will generally take $r(k)$ to be the uniform distribution. The hyperparameter β controls the amount of regularization: $\beta \to \infty$ recovers the standard stacking objective, and $\beta \to 0$ leads to the weights following $r(k)$.

This regularized objective corresponds to a particular instantiation of a PAC-Bayes bound that was proposed by Morningstar et al. [2022]. In the PAC-Bayes literature, $-R(w)$ (Eq. (5.11)) is sometimes referred to as the *true predictive risk*. Hence, optimizing $R_\beta(w)$ can be viewed as optimizing a stochastic bound on that true predictive risk, see Appendix B.5 for details.

5.5 Related Work

Alternatives to BMA. Bayesian model combination (BMC) [Minka, 2000, Monteith et al., 2011, Kim and Ghahramani, 2012] aims to break the BMA assumption that the data was generated from exactly one of the candidate models. This is done by specifying a new extended model that explicitly combines the predictions of all candidate models. However, fitting the new extended model is significantly more expensive, as the inference task no longer breaks down into independent sub-problems. Further, how to best combine predictions from different models is highly problem dependent and a modelling decision in its own right and, hence, not suitable for automation.

Yao et al. [2022] introduced Bayesian hierarchical stacking, which infers different weights for different regions in the covariate space, similar to the frequentist mixture of experts [Gormley and Frühwirth-Schnatter, 2019]. This is less suitable for the fully automated PPS setting because it requires knowledge of the covariate space and assumes the existence of covariates in the first place. So-called Pseudo-BMA weights [Geisser and Eddy, 1979] use LOO predictive densities to replace marginal likelihoods but have been shown to work less well than stacking [Yao et al., 2018]. BayesBag [Huggins and Miller, 2021] weights models by generating bootstrapped datasets, and averaging the normalization constant over datasets. This is computationally very intensive as it requires running inference separately in each bootstrapped dataset.

PAC-Bayes. Warrell and Gerstein [2022] extend PAC-Bayes bounds to work with hierarchical models inspired by deep probabilistic programs [Tran et al., 2017] and use this extension to derive bounds for multi-task settings such as transfer and meta-learning. However, they do not consider programs with stochastic support, the impact of model misspecification, nor integrate their method with a PPS. PAC-Bayes style arguments have also been used to reason about the hardness of posterior inference in PPS [Freer et al., 2010].

Programs with stochastic support as BMA. Existing inference algorithms for programs with stochastic support [Wingate et al., 2011, Yang et al., 2014, Wood et al., 2014, Rainforth et al., 2016, Le et al., 2017, Mak et al., 2021b, 2022] all implicitly generate a weighting of individual SLPs through the proportion of (weighted) samples generated from each SLP. Some inference algorithms are adaptions and extensions of the reversible-jump MCMC methods that were originally developed for the BMA setting [Green, 1995, Roberts et al., 2019, Cusumano-Towner et al., 2020]. However, previous work does not discuss the inherent issues with targeting the BMA model weights and the consequences of this for

making predictions; existing algorithms which explicitly assign weights to SLPs only target the default BMA model weights [Zhou et al., 2020, Luo et al., 2021, Reichelt et al., 2022a].

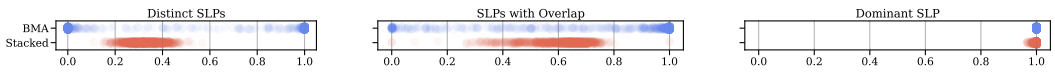
5.6 Experiments

We will now compare different weighting schemes on a range of models and datasets. Our quantitative measure for comparison will be the average log posterior predictive density (LPPD) on held-out data $\tilde{y}_{1:T}$, where LPPD $:= \frac{1}{T}\sum_{t=1}^{T} \log \tilde{\rho}(\tilde{y}_t; w)$. In particular, we focus on the difference in LPPD from other methods to stacking, $\text{LPPD}_{\text{Diff}} = \text{LPPD}_{\text{Other}} - \text{LPPD}_{\text{Stacking}}$.

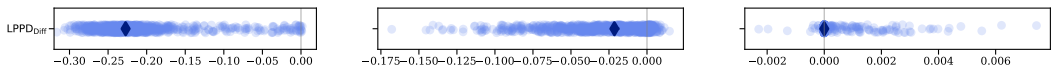
Additionally, we will investigate the behaviour of the SLP weights w_k; a major criticism of the BMA weights is that they are too sensitive to minor changes in the data. To ensure replicable analysis, we desire the SLP weights to be *robust* and *consistent*, i.e. they should be similar across different possible generated datasets.

Except when otherwise indicated, we use a variant of the divide, conquer, and combine (DCC) inference algorithm [Zhou et al., 2020] for the base inference algorithm. Our DCC implementation uses HMC [Neal, 2011, Hoffman and Gelman, 2014, Betancourt, 2018] for the local inference algorithm of each SLP and allocates the computational budget uniformly between SLPs. Additionally, we also consider the reversible-jump MCMC (RJMCMC) algorithm implemented in Gen [Cusumano-Towner et al., 2019]. Our implementation is available at `https://github.com/treigerm/beyond_bma_in_probprog`.

5.6.1 When is Stacking Helpful?



(a) Computed weight for the first SLP; each dot is the weight computed in one of the datasets



(b) $\text{LPPD}_{\text{Diff}}$ on held-out data. Negative values imply stacking is performing better. Diamonds show medians.

Figure 5.1: Behaviour of the BMA and stacked weights in the models as described in Section 5.6.1.

First, to develop an understanding of the scenarios in which stacking is beneficial, we consider three simple examples in which we limit ourselves to input programs

with two SLPs. Unless otherwise stated, BMA weights are computed analytically and the stacking weights are based on PSIS-LOO. For each problem, we generate 10^3 datasets with 200 data points each and generate another 10^3 data points to evaluate the held-out LPPD.

Distinct SLPs. For the first setting, we assume the data is generated from a standard normal, $y_i \sim \mathcal{N}(0,1)$. We consider a program with two misspecified SLPs where the unnormalized density for the kth SLP is given by

$$\gamma_k(\theta_1, \theta_2) = \mathbb{I}\left[\theta_2 = k\right] \frac{1}{2} \prod_{i=1}^{N} \mathcal{N}(y_i; \theta_1, \sigma_k^2)\, \mathcal{N}(\theta_1; 0, 1) \tag{5.21}$$

where we set $\sigma_1^2 = 0.62177$ and $\sigma_2^2 = 2$ (cf. Appendix B.4 for the corresponding Pyro program). This example is adapted from Yang and Zhu [2018].

SLPs with overlap. Next, we generate a dataset using the linear regression model $y_i = \sum_{d=1}^{4} \beta_d\, x_{i,d} + \epsilon_i$ where $\epsilon_i \sim \mathcal{N}(0,1)$, $x_{i,d} \sim \mathcal{N}(0,1)$ and $\beta = [1.5, 1.5, 0.3, 0.1]$. For inference, we consider a program with two SLPs with unnormalized densities

$$\gamma_k(\theta_1, \theta_2, \theta_3) = \mathbb{I}\left[\theta_3 = k\right] \frac{1}{2} \prod_{i=1}^{N} \mathcal{N}(y_i; f_k(\theta_1, \theta_2, x_i), 1) \prod_{j=1}^{2} \mathcal{N}(\theta_j; 0, 1), \tag{5.22}$$

where for the first SLP $f_1(\theta_1, \theta_2, x_i) = \theta_1 x_{i,1} + \theta_2 x_{i,3}$ and for the second $f_2(\theta_1, \theta_2, x_i) = \theta_1 x_{i,1} + \theta_2 x_{i,4}$. Note, that the covariates $x_{i,d}$ are modelled as fixed by the program, so here the two different SLPs essentially fit the slope coefficients of a linear regression model. Both SLPs are misspecified because they do not have access to all the covariates. However, the two sub-models share complexity/expressiveness, as they both have access to the first covariate.

Dominant SLP. Lastly, we generate data from $y_i = f(x_i) + \epsilon_i$ with $f(x) = 2\, x_i + \sin(5\, x_i)$ and $\epsilon_i, x_i \sim \mathcal{N}(0,1)$. Our program for inference has two SLPs which are defined as

$$\gamma_k(\theta_1, \theta_2, \theta_3) = \mathbb{I}\left[\theta_3 = k\right] \frac{1}{2} \prod_{i=1}^{N} \mathcal{N}(y_i; f_k(\theta_1, x_i), \theta_2^2)\, \mathcal{N}(\theta_1; 0, 1)\, \Gamma(\theta_2; 1, 1) \tag{5.23}$$

where $f_1(\theta, x) = \theta\, x$, $f_2(\theta, x) = \sin(\theta\, x)$, and $\Gamma(\theta; \alpha, \beta)$ is a Gamma distribution parameterized by shape α, and rate β. The first SLP will provide a significantly better fit to the data as it is able to recover the dominant linear trend. We use importance sampling to estimate the BMA weights (see Appendix B.4).

Results. The results are presented in Figure 5.1. For the first two models the stacked weights lead to better predictive performance and more robust weights.

While there is some variation in the stacked weights, their overall distribution is *unimodal* whereas the distribution of the BMA weights are *bimodal* instead. The tendency of the BMA weights to collapse on either 0 or 1 is exactly the *overconfident* behaviour we described in Section 5.3. In the setting with one dominant SLP, both the BMA and stacking weights lead to similar predictive performance (with BMA doing slightly better) and consistently collapse onto the dominant SLP; here this is desirable behaviour as the first SLP is clearly superior.

5.6.2 Subset Regression

To further outline the issues of the default BMA weights, we consider a regression problem with data generated from a linear model of the form $y_i = \epsilon_i + \sum_{d=1}^{15} \beta_d \, x_{i,d}$ where $\epsilon_i \sim \mathcal{N}(0, 1)$ and all the covariates $x_{n,d}$ are drawn independently from $\mathcal{N}(5, 1)$. Note, the covariates are sampled to generate the synthetic data set, but they are modelled as fixed by the program. The ground-truth values for the regression coefficients β_d are set following a scheme used in Breiman [1996] and Yao et al. [2018] which ensures all covariates are relevant for the prediction of y_n (c.f. Appendix B.4).

We compare multiple methods: 1. **Stacked**, using PSIS-LOO to compute weights, w (Section 5.4.3); 2. **Stacked (Val)**, which uses an explicit validation set instead (Section 5.4.2); 2. **BMA**, using the PI-MAIS algorithm [Martino et al., 2017] to compute local normalization constants (this is the default weighting in DCC); 3. **BMA (Analytic)**, using analytic solutions for the local normalization constants; 4. **RJMCMC**, implemented in Gen [Cusumano-Towner et al., 2019];[3] 5. **Equal**, weights each SLP equally. We use warm colors to present the results which use one of our proposed alternative objectives to set SLP weights and cooler colours for methods which target the BMA weights. Note **BMA**, **BMA (Analytic)**, and **RJMCMC** all target the posterior distribution in Eq. (5.3); in practice, differences between these methods will arise due to differences in the quality of the posterior approximation.

Our input program has 15 SLPs and each SLP only gets access to one of the covariates so our overall model is misspecified. However, since every covariate influences the targets y_i, each SLP is relevant for making good predictions. We generate 50 different datasets from the true data generating distribution, with 200 data points used to run our inferences and 10^3 data points to evaluate the held-out log posterior predictive density. For the **Stacked (Val)** we use half

[3] We also have conducted experiments that run stacking on top RJMCMC and found that this also led to improvements in predictive performance (c.f. Appendix B.4). Here, we only present the stacking results with samples generated from DCC as this gave the best base inference procedure.

Table 5.1: $\text{LPPD}_{\text{Diff}}$ (↑ better) for models in Section 5.6.2, 5.6.3, and 5.6.4, results computed over 10 replications. Bold indicates no significant difference to Stacked under a Wilcoxon signed-rank test.

Model	Stacked	Stacked (Val)	BMA	BMA (Analytic)	RJMCMC	Equal
Subset	**0.0**	-0.01 ± 0.01	-0.11 ± 0.05	-0.11 ± 0.05	-0.11 ± 0.04	-0.02 ± 0.01
Fun. Ind. (misspecified)	**0.0**	$\mathbf{-1.73e{-3} \pm 2.98e{-3}}$	$\mathbf{-9.10e{-4} \pm 2.42e{-3}}$	N/A	-0.08 ± 0.03	-0.31 ± 0.01
Fun. Ind. (well-specified)	**0.0**	$\mathbf{-5.61e{-3} \pm 7.14e{-3}}$	$-3.76e{-1} \pm 2.51e{-1}$	N/A	-2.44 ± 0.32	-2.31 ± 0.09
California	**0.0**	$\mathbf{-1.55e{-3} \pm 2.89e{-3}}$	$-2.10e{-2} \pm 6.19e{-3}$	$-2.10e{-2} \pm 6.01e{-3}$	$-2.82e{-1} \pm 1.19e{-1}$	$-1.96e{-1} \pm 3.11e{-3}$
Diabetes	**0.0**	$\mathbf{-8.22e{-3} \pm 1.44e{-2}}$	$\mathbf{-1.01e{-2} \pm 1.64e{-2}}$	N/A	$-3.83e{-2} \pm 2.19e{-2}$	$-3.66e{-2} \pm 9.97e{-3}$
Stroke	**0.0**	$\mathbf{-7.79e{-4} \pm 1.57e{-3}}$	$-6.22e{-3} \pm 3.81e{-3}$	N/A	$-2.25e{-1} \pm 9.94e{-2}$	$-1.31e{-1} \pm 5.68e{-3}$

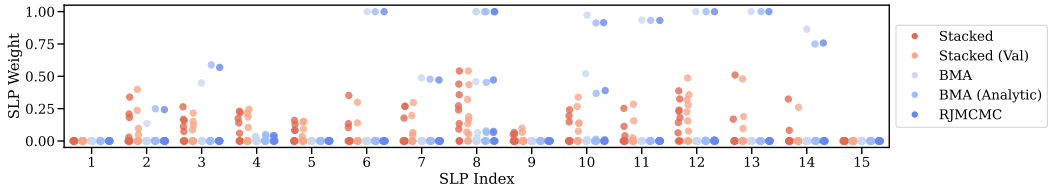


Figure 5.2: SLP weights for problem in Section 5.6.2. Each dot represents the weight of the corresponding SLP in the model. Results are computed over 10 generated datasets.

of the 200 data points for inference and use the rest to estimate the stacking objective (c.f. Eq. (5.19)).

Tab. 5.1 shows that stacking outperforms all other methods in terms of predictive performance. The BMA weights here actually provide even worse predictions than weighting each SLP equally. Figure 5.2 shows the behaviour of the weights for the SLPs over different randomly generated datasets. Both BMA and BMA (Analytic) exhibit clear signs of overconfidence as described in Section 5.2.1: the weights often collapse onto a single SLP, but the exact SLP changes between datasets, leading to a bimodal sampling distribution for the SLP weights. As expected, RJMCMC produces qualitatively and quantitivaly similar results to the other Bayesian weighting mechanisms. This is in contrast with the Stacked weights which are more evenly spread. The Stacked (Val) weights behave qualitatively similarly to using PSIS-LOO, but with slightly worse predictive performance. This is likely due to the corresponding reduction in training set size.

5.6.3 Function Induction

Our next example investigates how well stacking scales to a larger number of SLPs. We generate observations from the relation $y_i = -x_i + 2\sin(2\,x_i^2) + \epsilon_i$ with $\epsilon_i \sim \mathcal{N}(0, 0.1^2)$ and the inputs x_i uniformly sampled between -5 and 5. We generate 400 data points used for inference and 10^3 data points for evaluation. Following Zhou et al. [2020], we use a probabilistic context-free grammar (PCFG) to posit a model over functions. We consider two PCFGs, the *misspecified* PCFG

has production rules $e \to \{x \mid \sin(a * e) \mid a * e + b * e\}$, where x is a terminal symbol denoting an input, and a, b are coefficients to be inferred. The *well-specified* PCFG additionally includes the terminal symbol x^2. Therefore, the well-specified PCFG can express the data generating function whereas the misspecified one cannot. The program recursively samples from the PCFG using samples from categorical distributions to select production rules and defines latent variables for all the coefficients in a given expression (c.f. Appendix B.4). Note analytic BMA weights cannot be calculated here.

Tab. 5.1 shows that **Stacked** provides better predictions compared to all other methods, even when the model is well-specified! Notably, inference in this model is particularly challenging due to the fact that distinct SLPs can have similar or even identical posterior predictive distributions due to symmetries in the PCFG, e.g. $x + \sin(x)$ and $\sin(x) + x$ correspond to two separate SLPs. The fact that stacking outperforms the methods targeting the BMA weights in the well-specified case is an indicator that the inference algorithms are struggling in this model. Indeed, we found **RJMCMC** tends to get stuck in a single SLP which is a well-known issue with MCMC methods for programs with stochastic support (all weights are shown in Appendix B.4). Even though **Stacked** weights give better predictions, we found that the weights themselves exhibit relatively high-variance. This is due to the finite sample size of the dataset and the usage of approximate inference algorithms which introduce variance in estimating the stacking objective. However, the fact that stacking is able to produce superior predictions shows that it can be a useful mechanism for improving predictive performance even when inference algorithms struggle to produce accurate posteriors approximations.

5.6.4 Variable Selection

Next, we apply stacking to real-world classification and regression tasks. Here, we have a matrix of covariates $X \in \mathbb{R}^{N \times D}$ and targets $y_{1:N}$, and we want to do variable selection, i.e. select a subset of the features $\mathcal{D} \subseteq \{1, \ldots, D\}$ to make predictions. This problem of variable selection can be encoded as a probabilistic program with stochastic support in which each SLP corresponds to one of the potential subsets of the features $\mathcal{D}$. We consider three different datasets: *California* (regression) [Pace and Barry, 1997], *Diabetes* (classification) [Smith et al., 1988], and *Stroke* (classification) [Kaggle, 2020]. For the regression, our model is a linear regression with conjugate priors, permitting an analytic solution to the BMA weights. For the classification tasks, we use a logistic regression model which does not permit an analytic solution. As the true data generating process in this

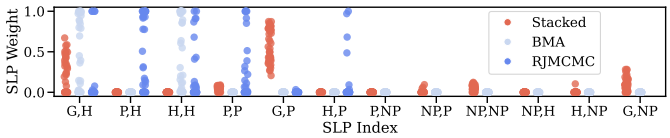


Figure 5.3: SLP weights for Section 5.6.5. X-tick labels indicate the different modelling choices for α and β; the pattern is "α model choice, β model choice" with P = pooling, NP = no pooling, H = hierarchical, and G = group-level predictor.

Table 5.2: $\text{LPPD}_{\text{Diff}}$ for Radon model.

Method	$\text{LPPD}_{\text{Diff}}$ (↑)
Stacked	**0.0**
BMA	$-8.24\text{e}{-3} \pm 1.20\text{e}{-2}$
RJMCMC	$-5.99\text{e}{-2} \pm 1.86\text{e}{-2}$
Equal	$-1.88\text{e}{-2} \pm 1.18\text{e}{-2}$

setting is unknown, we run each method on different train-test splits to estimate the variation in the weights and predictions.

Tab. 5.1 shows the LPPD values of the weighting schemes on different datasets. The **Stacked** and **Stacked (Val)** weighting schemes generally give better predictive performance compared to the alternatives. Overall, these results show that stacking can be beneficial for predictive performance even on real-world data.

5.6.5 Radon Contamination

Our final example considers the analysis of data about Radon contamination for houses in different US counties [Gelman and Hill, 2006]. We here only give a high-level description of the dataset and model, full details in Appendix B.4. For each house recorded in the dataset, we have radon measurements, y_i, as well as, a covariate, $x_i \in \{0, 1\}$, which indicates whether the measurement was made in the basement ($x_i = 0$) or first floor of the house ($x_i = 1$). Our program for this dataset has at its core the regression relation $y_i = \alpha + \beta\, x_i + \epsilon_i$ and the different SLPs in the program make different assumptions for how to model the coefficients α and β. For both, we can either: 1) Fit the same coefficient across all counties; 2) Fit a separate coefficient α_c for each county c; or 3) Have separate coefficients for each county but assume they come from the same underlying population distribution. For the intercept term we also consider a fourth option: using county-wide level uranium measurements as a group-level predictor. The program considers all combinations of modelling choices for α and β, i.e. it has $4 \cdot 3 = 12$ SLPs.

Each SLP in this program can have potentially hundreds of latent variables and exhibit complex posterior geometries, making this a good testing ground to compare the different weighting schemes. In Tab. 5.2 we find that the **Stacked** weights give better performance compared to the BMA weights (we do not consider using a validation set here because some counties contain only a handful of observations). Figure 5.3 shows that the BMA weights tend to concentrate on SLPs with modelling

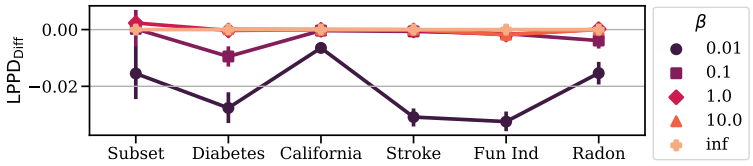


Figure 5.4: Impact of regularization parameter β on predictive performance in the different models (higher is better). Plotted are mean and standard deviation.

choices "H,H" or "G,H" and have a bimodal sampling distribution. The **Stacked** weights are more robust, giving more consistent results between different train-test splits and more conservative weights. Notably, **RJMCMC** here collapses onto different SLPs than the BMA weights. This is due to the fact that the RJMCMC struggles with SLPs which have a large number of latent variables. In the limit of infinite samples, the behaviour of RJMCMC and BMA will be identical but Figure 5.3 illustrates nicely that approximate inference algorithms might collapse onto different SLPs based on the quality of their posterior approximation.

5.6.6 Impact of Regularization: PAC-Bayes

As we have shown in Section 5.4.4, the PAC-Bayes bound $R_\beta(w)$ offers an alternative objective to fit the weights and can be interpreted as the stacking loss with an added regularization term where the hyperparameter β controls the amount of regularization. Smaller values of β push the stacking weights closer to the uniform distribution over SLPs. In Figure 5.4 we plot the effect of varying β on the predictive performance on the different models. In our experiments, values of β below 1 tend to lead to worse predictive performance and the stacking objective with no regularization ($\beta = \infty$ in Figure 5.4) is not outperformed by any form of regularization. However, depending on the application setting some level of regularization might still be desirable.

5.7 Discussion

We have demonstrated that in programs with stochastic support, the conventional posterior path probabilities can be unstable (e.g. due to model misspecification or inference approximations) and that this in turn can lead to sub-optimal predictions.

In practice, one of the key sources of instability is model misspecification. When dealing with misspecification, the general advice is to revise or expand the model [Gelman et al., 2020]. However, when using real-world data it is often not

Table 5.3: Timings for running inference and stacking, averaged over 5 runs. Inference is conducted using DCC.

	Subset	Var. Select.	Fun. Ind.	Radon
Inference	29 s	297 s	285 m	700 s
Stacking	0.09 s	43 s	11 s	0.2 s

obvious how to further expand a model and mitigate against misspecification. The radon experiment (Section 5.6.5) is a good example here as it is already the result of multiple model iterations and expansions, with no clear strategy for how to extend it further. Additionally, revising and fitting a new expanded model is often prohibitively expensive and therefore not a viable alternative. As our timings in Tab. 5.3 demonstrate, stacking is a very cheap procedure compared to the cost of inference. With the automated post-processing techniques presented in this paper, stacking can therefore be conveniently applied at the end of an analysis after the user has gone through multiple iterations of model building. Thus, rather than viewing stacking as a replacement for model expansion, we view it as a useful tool to safeguard against the instabilities of the default BMA weighting scheme.

While we have demonstrated that the instability in the BMA weights can appear in realistic models and datasets, for any given problem there is no guarantee that the BMA weights will indeed be unstable. For example, as we saw in the initial experiments in Section 5.6.1, stacking and BMA will produce similar weights if there is one SLP which clearly dominates the others. However, finding clear criteria that determine when BMA will lead to unstable weights is still an area of open research [Yang and Zhu, 2018, Oelrich et al., 2020, Huggins and Miller, 2021], so for practitioners it is difficult to know a priori whether a given model will produce unstable SLP weights or not.

Overall, this means there are few reasons not to use stacking: it is cheap, easy-to-use, provides generally more robust weights, and leads to improved predictions.

6

Expectation Programming: Target-Aware Expectation Estimation

6.1 Motivation

Estimating *expectations* is at the center of many scientific workflows. For example, the decision theoretic foundations of most statistical paradigms that we presented in Chapter 2 are rooted in calculating the expectation of a loss function [Robert and Casella, 2004]. Carrying out this estimation often requires *approximate inference* to be performed: we may not be able to directly draw samples of the random variable we wish to calculate the expectation of, or a simple Monte Carlo estimate might produce problematically high variance.

Probabilistic programming systems (PPSs) provide a powerful basis for encoding such inference problems and then assisting with, or even fully automating, the approximation of their solution [Gordon et al., 2014, van de Meent et al., 2018]. As we saw in the previous chapter, programs are typically specified (often indirectly) through an unnormalized density $\gamma(\theta)$. Assuming analytic solutions are not available, the role of the system's inference engine is now to construct an approximation, $\hat{\pi}(\theta)$, for the distribution specified by the normalized density $\pi(\theta) = \gamma(\theta)/Z$, where Z is an unknown normalizing constant and $\pi(\theta)$ typically represents a conditional distribution, such as the posterior in a Bayesian modelling setting. This approximation can then be used in turn for downstream tasks, such as approximating one or more expectations.

Though ostensibly very general, our key insight is that this standard PPS computational pipeline—which is implicitly followed by all contemporary PPSs that

conduct inference approximately (e.g. Bingham et al. [2019], Carpenter et al. [2017], Cusumano-Towner et al. [2019], Ge et al. [2018], Salvatier et al. [2016], Tran et al. [2016], Wood et al. [2014], Mansinghka et al. [2014], Goodman and Stuhlmüller [2014], Murray and Schön [2018], Minka et al. [2018])—can be highly suboptimal when our ultimate aim is to estimate a particular expectation, $\mathbb{E}_{\pi(\theta)}[f(\theta)]$. This is because such a pipeline fails to perform estimation in a *target-aware* fashion: it does not allow information about f to be exploited by the inference engine, thereby forgoing the substantial empirical gains that using information about f can yield [Torrie and Valleau, 1977, Hesterberg, 1988, Wolpert, 1991, Oh and Berger, 1992, Evans et al., 1995, Meng and Wong, 1996, Chen et al., 1997, Gelman and Meng, 1998, Lacoste-Julien et al., 2011, Owen, 2013, Golinski et al., 2019, Rainforth et al., 2020]. Note here that it is not generally possible to incorporate the required information about f by adjusting the model definition; fundamental changes to the computational pipeline itself are required.

To address this, we introduce, and formalize, the concept of *expectation programming*. Here an expectation program is analogous to a probabilistic program, but its target quantity of interest is the expected value of the program's return values, rather than their conditional distribution. This subtle distinction leads to changes in the requirements for the program to be valid, and, critically, the estimation that must be performed by the backend inference engine. This, in turn, allows us to construct computational pipelines which are target-aware, utilizing information in the program itself to estimate expectations substantially more efficiently than can be achieved by existing PPSs.

We realize our expectation programming concept through a specific system we call **EPT** (Expectation Programming in Turing), built upon the Turing PPS [Ge et al., 2018]. EPT takes as input a Turing-style program and uses a combination of program transformations and existing inference strategies to construct target-aware estimators via the TABI approach of [Rainforth et al., 2020].

We formally demonstrate the statistical soundness of EPT, proving that it produces consistent estimates under nominal assumptions. We further show empirically that it can be used to express and run effective inference for a number of problems, finding that it produces estimates that are significantly more accurate than conventional usage of Turing. As part of this, we also implement a new annealed importance sampling (AnIS) [Neal, 1998] inference engine for Turing, finding that this allows for effective marginal likelihood estimation in a much wider array of problems than Turing's previously supported inference strategies.

To summarize, our key contributions are: a) identifying the shortfall of existing PPSs when estimating expectations and introducing the concept of expectation programming to address this; b) developing EPT as a particular realization of the expectation programming concept; c) formalizing the notion of an expectation program and demonstrating the statistical correctness of EPT; d) introducing a new AnIS inference engine to Turing; and e) showing that EPT can provide substantial empirical benefits over conventional use of Turing on real problems.

6.2 Background

6.2.1 Turing Programs as Densities

To provide a basis for introducing expectation programming, we consider the PPS Turing (Ge et al. [2018], `https://turing.ml/dev/docs/using-turing/`), but note that the concepts introduced apply to PPSs in general. We provide a brief introduction to Turing here, along with our own new formalism for the densities Turing programs define that by building on the approach from Chapter 3. This is necessitated by some technical intricacies of the expectation programming approach. To assist with this, we will use the following simple Turing program as a running example:

```
@model function model(y)
    x ~ Normal(0, 1)
    @addlogprob!(0.1)
    y ~ Normal(x, 1)
end
```

A Turing program is defined similarly to a normal Julia function [Bezanson et al., 2017]: the `@model` macro indicates the definition of a Turing model, with tilde statements inside the body, e.g. `x ~ Normal(0, 1)`, to denote probabilistic model components. Observed data can be passed in as a formal argument to the function. If the variable name on the left-hand side of the tilde statement is not part of the arguments of the function then it is interpreted as a random variable.

Let $\theta_{1:n_\theta}$ denote the set of direct outputs from sampling statements and $y_{1:n_y}$ the observed data. We can view Turing programs as defining an unnormalized density $\gamma(\theta_{1:n_\theta})$ (with an implicit appropriate reference measure). To compute the density for a given $\theta_{1:n_\theta}$ the program executes like a normal Julia program, while keeping track of the density of the current execution. Specifically, when Turing reaches a tilde statement corresponding to a random variable, it samples a value for θ_i, evaluates the density of this draw, and factors this into the overall execution density.

We denote the density of the draw as $g_i(\theta_i|\eta_i)$, where η_i denotes the form of the sampling statement and η_i its parameters. For the tilde statements corresponding to the observed data, it evaluates the density function $h_j(y_j|\phi_j)$—where h_j and ϕ_j are analogous to g_i and η_i respectively—and factors the overall density accordingly.

Sometimes a user might want to add additional factors to the density without using a tilde statement. For this, Turing provides the `@addlogprob!(log_p)` primitive which multiplies the density of the current execution by an arbitrary value `exp(log_p)`. We use $\psi_1, \ldots, \psi_K$ to denote all the terms that are added to the density using `@addlogprob!`.

Putting these together, the unnormalized density defined by any valid program trace can be written as

$$\gamma(\theta_{1:n_\theta}) = \prod_{i=1}^{n_\theta} g_i(\theta_i|\eta_i) \prod_{j=1}^{m} h_j(y_j|\phi_j) \prod_{k=1}^{K} \exp(\psi_k). \tag{6.1}$$

Our example program thus defines the density $\gamma(\theta) = \exp(0.1)\mathcal{N}(\theta; 0, 1)\mathcal{N}(y; \theta, 1)$, with a fixed input y. Note here that again everything (i.e. n_θ, $\theta_{1:n_\theta}$, $\eta_{1:n_\theta}$, $g_{1:n_\theta}$, n_y, $y_{1:n_y}$, $\phi_{1:n_y}$, $h_{1:n_y}$, K, $\psi_{1:K}$) can be a random variable because of potential stochasticity in the program path. However, using the program itself, everything is deterministically calculable from $\theta_{1:n_\theta}$, which can thus be thought of as the 'raw' random draws that dictate all the randomness of the program; everything else is a pushforward of these.

6.2.2 Annealed Importance Sampling

Annealed importance sampling (AnIS) [Neal, 1998] is an inference algorithm which was developed with the goal of efficiently estimating the normalization constant Z of an unnormalized density $\gamma(\theta)$. Similar to standard importance sampling (see Section 2.3.1) AnIS generates a set of weighted samples by sampling from a proposal distribution. However, AnIS uses an annealing scheme to implicitly define this proposal. It works by defining a sequence of annealing distributions $\pi_0(\theta), \ldots, \pi_n(\theta)$ which interpolate between a simple base distribution $\pi_0(\theta)$ (typically the prior for a Bayesian model) and the complex target density $\pi_n(\theta) = \gamma(\theta)$. The most common scheme is to take

$$\pi_i(\theta) \propto \lambda_i(\theta) = \pi_0(\theta)^{1-\beta_n} \gamma(\theta)^{\beta_n}, \tag{6.2}$$

with $0 = \beta_0 < \ldots < \beta_n = 1$. The algorithm further requires the definition of Markov chain transition kernels $\kappa_1(\theta' \mid \theta), \ldots, \kappa_{n-1}(\theta' \mid \theta)$ and proceeds to generate the j^{th}

weighted sample as follows: First, sample initial particle $\theta_j^{(1)} \sim \pi_0(\theta)$, then for $i = 1, \ldots, (n-1)$, generate $\theta_j^{(i+1)} \sim \kappa_i(\cdot \mid \theta_j^{(i)})$ and, finally, return sample $\theta_j^{(n)}$ with weight

$$w_j = \frac{\lambda_1(\theta_j^{(1)})\lambda_2(\theta_j^{(2)}) \ldots \lambda_n(\theta_j^{(n)})}{\pi_0(\theta_j^{(1)})\lambda_1(\theta_j^{(2)}) \ldots \lambda_{n-1}(\theta_j^{(n)})} \tag{6.3}$$

We can estimate expectations with the weights and samples just as in importance sampling. Thus we can estimate the expectation and the normalization constant as

$$\mathbb{E}_{\pi(\theta)}[f(\theta)] \approx \frac{\sum_{j=1}^N w_j f(\theta_j^{(n)})}{\sum_{j=1}^N w_j} \quad \text{and} \quad Z \approx \frac{1}{N}\sum_{j=1}^N w_j.$$

6.2.3 Target-Aware Inference

Consider the problem of estimating an expectation of the form $\mathbb{E}_{\pi(\theta)}[f(\theta)]$ where $f(\theta)$ is known, but $\pi(\theta)$ cannot be directly evaluated or sampled from. Namely, $\pi(\theta) = \gamma(\theta)/Z$ where $\gamma(\theta)$ is a known unnormalized density, but Z is an unknown normalization constant (sometimes referred to as the marginal likelihood).

The inference engines in PPSs like Turing are setup to approximate $\pi(\theta)$ of this form. As such, the standard pipeline to approximate an expectation using a PPS is to first approximate $\pi(\theta)$ (e.g. with samples) and then use this to approximate the expectation in turn.

Unfortunately, this ignores information about f and is therefore suboptimal if f is known [Golinski et al., 2019]. While one might initially expect that information about f can be easily incorporated through simple model adjustments, this is unfortunately not the case in practice: any adjustments we make will mean we need to estimate an additional corrective factor on top of performing inference for the new model. Indeed, naive approaches to incorporating information about f, like adding $|f(\theta)|$ as a density factor to the model, have been found to typically worsen, rather than improve, the final estimates [Rainforth et al., 2020].

Rainforth et al. [2020] recently showed that this issue stems from fundamental limitations of the efficacy of using a *single* Monte Carlo estimator for such expectations. Namely, through their Target-Aware Bayesian Inference (TABI) framework, they show that by breaking down the expectation into three parts:

$$\mu := \mathbb{E}_{\pi(\theta)}[f(\theta)] = \frac{Z_1^+ - Z_1^-}{Z_2}, \tag{6.4}$$

where

$$Z_1^+ = \int \gamma(\theta) f^+(\theta) d\theta, \quad Z_1^- = \int \gamma(\theta) f^-(\theta) d\theta, \quad Z_2 = \int \gamma(\theta) d\theta,$$
$$f^+(\theta) = \max(f(\theta), 0), \quad f^-(\theta) = -\min(f(\theta), 0)$$

and then estimating each term separately, one can often achieve a substantially improved overall estimator,

$$\mathbb{E}_{\pi(\theta)}[f(\theta)] \approx \hat{\mu} := \frac{\hat{Z}_1^+ - \hat{Z}_1^-}{\hat{Z}_2}. \tag{6.5}$$

The intuition here is that each individual term can often be estimated more accurately in isolation than the original expectation. To see this, first note that the three subcomponents can be seen as the respective normalization constants of the three densities

$$\begin{aligned} \gamma_1^+(\theta) &\propto \gamma(\theta) f^+(\theta), \\ \gamma_1^-(\theta) &\propto \gamma(\theta) f^-(\theta), \\ \gamma_2(\theta) &= \gamma(\theta). \end{aligned} \tag{6.6}$$

The TABI framework now allows one to define a separate estimator *tailored* to each of these problems. In general, it allows one to repurpose any algorithm which provides estimates of the normalization constant into a target-aware inference algorithm by separately applying it to each of $\gamma_1^+(\theta)$, $\gamma_1^-(\theta)$, and $\gamma_2(\theta)$.

As the following proposition shows, TABI can theoretically achieve an arbitrarily low error for any fixed sample budget ($\geqslant 3$), unlike standard approaches such as self-normalized importance sampling whose expected error is lower bounded even when using an optimal proposal/sampler.

Proposition 2 (Rainforth et al. [2020])**.** *If $Z_1^+, Z_1^-, Z_2 < \infty$ and we use importance sampling estimators $\hat{Z}_1^+, \hat{Z}_1^-, \hat{Z}_2$ with the corresponding optimal set of proposals $q_1^+(\theta) \propto \gamma(\theta) f^+(\theta)$, $q_1^-(\theta) \propto \gamma(\theta) f^-(\theta)$, and $q_2(\theta) \propto \gamma_2(\theta)$, then the TABI estimator in (6.5) satisfies*

$$\mathbb{E}[\hat{\mu}] = \mu, \qquad \mathrm{Var}[\hat{\mu}] = 0. \tag{6.7}$$

See Section 3.2 of Rainforth et al. [2020] for a formal proof; it relies on applying the standard importance sampling estimator results to each of the three components.

Naturally, sampling exactly from the optimal proposals is not always possible. To investigate the behaviour that one might expect to see in practice it is instructive to consider the asymptotic mean squared error (MSE) of the TABI estimator. Until the end of this sub-section we will assume that we have $f(\theta) > 0$ and the TABI estimator simplifies to $\hat{\mu} = \hat{Z}_1/\hat{Z}_2$.[1] The following results (first presented and proven in Rainforth et al. [2020]) characterise the asymptotic mean squared error of the TABI estimator.

[1] The results can be extended to the general case, we only consider the simplified case here to avoid clutter in the notation.

Proposition 3 (Rainforth et al. [2020]). *Let* $\xi_1 := (\hat{Z}_1 - Z_1)/\sigma_1$, $\xi_2 := (\hat{Z}_2 - Z_2)/\sigma_2$, $\sigma_1^2 := \mathrm{Var}[\hat{Z}_1]$, *and* $\sigma_2^2 := \mathrm{Var}[\hat{Z}_2]$. *Assuming that* $\hat{Z}_1$ *and* $\hat{Z}_2$ *are independent estimators, the asymptotic mean squared error of the TABI estimator is given by*

$$\mathbb{E}[(\hat{\mu} - \mu)^2] = \frac{\sigma_2^2 \mu^2}{Z_2^2} \left(\kappa^2 + 1\right) + \mathcal{O}(\epsilon) \tag{6.8}$$

where $\kappa := \sigma_1/(\mu\sigma_2)$ *and* $\mathcal{O}(\epsilon)$ *represents terms that are dominated asymptotically.*

The term κ measures the relative effectiveness of the two estimators $\hat{Z}_1$ and $\hat{Z}_2$. Conventional estimators like self-normalized importance sampling or MCMC can be viewed as using the same proposal for both $\hat{Z}_1$ and $\hat{Z}_2$. Then, if we have a mismatch between $\pi(\theta)$ and $\pi(\theta)f(\theta)$, it can become difficult to keep the term κ small; any decrease in σ_1 will likely lead to an increase in σ_2, and vice versa. By separately targeting $\hat{Z}_1$ and $\hat{Z}_2$, TABI estimators are able to break this deadlock and achieve a lower MSE than conventional estimators. Hence, overall the achievable gains increase, both theoretically and empirically, with the degree of mismatch between $\pi(\theta)$ and $\pi(\theta)f(\theta)$. For a more detailed discussion and further empirical validation of the theoretical results of the TABI estimator we refer to Rainforth et al. [2020].

6.3 Expectation Programming

At a high level, *expectation programming* adapts probabilistic programming systems to automate the estimation of expectations in a *target-aware* manner. As we now explain, an *expectation program* is analogous to a probabilistic program, but where the quantity of interest is the expectation of its return values under the program's conditional distribution, rather than the conditional distribution itself.

6.3.1 Formalization

To formalize the concept of an expectation program, we first statistically formalize probabilistic programs as follows.

Definition 1. *A probabilistic program* $\mathcal{P}$ *in a probabilistic programming language defines an unnormalized density* $\gamma(\theta_{1:n_\theta})$ *over the raw random draws* $\theta_{1:n_\theta} \in \Theta$ *of the program, which collectively we refer to as the program trace, along with an implicitly defined reference measure* μ.

We let $\pi(\theta_{1:n_\theta}) = \gamma(\theta_{1:n_\theta}) / Z$ denote the normalized density with the normalization constant $Z = \int_\Theta \gamma(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta})$. Here $\pi(\theta_{1:n_\theta})$ and μ combined implicitly define the conditional probability distribution specified by $\mathcal{P}$, which we denote $\mathbb{P}(X) = \int_X \pi(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta})$.

To ensure that the induced probability measure of a program is well-defined, we require that $\gamma(\theta_{1:n_\theta})$ corresponds to a valid unnormalized density. This guarantees that there is a valid probability distribution the inference algorithm of the particular PPS can converge to. We use this to formalize the concept of a *valid* probabilistic program as follows.

Definition 2. *A probabilistic program, $\mathcal{P}$, is* valid *(and defines a valid unnormalized probabilistic program density $\gamma(\theta_{1:n_\theta})$) if and only if both of the following hold: $\gamma(\theta_{1:n_\theta}) \geqslant 0, \forall \theta_{1:n_\theta} \in \Theta$; and $0 < \int_\Theta \gamma(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}) < \infty$.*

For Turing we have described how programs specify $\gamma(\theta_{1:n_\theta})$ in Section 6.2.1, but Definitions 1 and 2 apply more generally and only require that we can derive an unnormalized density function for a given program; a requirement that is satisfied by most existing popular PPSs.

We can now formalize the concept of an expectation program by associating return values to our program:

Definition 3. *An expectation program, $\mathcal{E}$, is a probabilistic program (as per Definition 1) with an associated set of return values $F \in \mathcal{F} \subseteq \mathbb{R}^d$ that are a deterministic mapping of the trace $\theta_{1:n_\theta}$.*

From this definition we see that expectation programs are largely equivalent to probabilistic programs, indeed programs in any PPS that allows return values will also be expectation programs provided their outputs are numeric and fixed dimensional. However, as their underlying quantity of interest is the expectation of their return values, $\mathbb{E}[F]$, they require a slightly different set of assumptions to ensure validity as follows.

Definition 4. *An expectation program $\mathcal{E}$ is valid if and only if it is a valid probabilistic program and F is integrable.*

Here the additional requirement of the expectation program's outputs being integrable essentially equates to requiring that the expectation $\mathbb{E}[F]$ exists and $\mathbb{E}[|F_i|] < \infty$ for each dimension F_i of F. This is generally a very weak requirement, and strictly weaker than an assumption typically implicitly made by existing PPSs when confirming the validity of their inference engines. To ensure correctness most

PPSs assume that a particular inference algorithm will converge to the distribution of F (i.e. the distribution over return values). A standard PPS Monte Carlo inference engine will now produce a sequence of samples F_n, $n = 1, 2, \ldots$ and consistency requires that F_n converges in distribution to F as $n \to \infty$. This is equivalent to requiring that for *any* integrable function h, $\mathbb{E}[h(F_n)] \to \mathbb{E}[h(F)]$; and it presupposes that the distribution of F is a finite measure, i.e., $\mathbb{E}[F]$ is finite. We thus see our assumption is strictly weaker than that of standard PPSs that allow return values from programs: we only need convergence in the case where h is the identity mapping, not all integrable functions.

To link expectation programs back into our early expectation notation, we now note that the requirement for the return values to be a deterministic mapping of the trace means that we can write $F = f(\theta_{1:n_\theta})$, such that $\mathbb{E}[F] = \mathbb{E}_{\pi(\theta_{1:n_\theta})}[f(\theta_{1:n_\theta})]$. Thus, the formal definition of the function we are taking the expectation of is that it is the full mapping from the raw random draws to the returned values, rather than what is lexically written in any `return` statement(s). This is why, for instance, it is still valid to have multiple different `return` statements in a program; provided each `return` statement defines the same number of return values. In practice, this is not something we need to worry about when writing either models or inference engines as the law of the unconscious statistician relieves us from explicitly delineating the random variable defined by our function (the expectation of this random variable does not vary if we change the parameterization of our model). However, the distinction is important for ensuring validity and to identify the precise target function we wish to extract information about when making the inference target-aware.

6.3.2 Target-Aware Inference Engines

The key idea of our expectation programming paradigm is to use the formalisms from the previous section to set up inference engines that exploit information from f to perform target-aware estimation. As explained in Section 6.2.3, this can lead to estimators that provide substantial performance improvements over the standard PPS approach of simply approximating $\pi(\theta_{1:n_\theta})$, ignoring $f(\theta_{1:n_\theta})$ completely.

Note that the approximate computation we are performing here is fundamentally different to that of conventional inference engines: we are estimating an expectation, rather than approximating a conditional distribution. This means the form of the outputs from our engine will change, while we will have to exploit additional information about the program. As such, we will generally need to make changes to how the program itself is processed, rather than just implementing a new inference

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)              # x ~ 𝒩(x; 0, 1)
    y ~ Normal(x, 1)              # y ~ 𝒩(y; x, 1)
    return x^3                    # f(θ) = x^3
end
expct_estimate, diagnostics =
  estimate_expectation(expt_prog(2),
    TABI(marginal_likelihood_estimator =
      TuringAlgorithm(AnIS(),num_samples=100)))
```

Figure 6.1: An example of estimating an expectation with EPT. Here `estimate_expectation` is our "do estimation" call which takes in expectation program `expt_prog` (with input $y = 2$) and an estimation method to apply (here a TABI estimator using annealed importance sampling), and returns an estimate for the expected return value of `expt_prog`.

engine in the existing PPS structure. Thankfully though, it will still usually be possible to repurpose existing inference engines as part of an overall target-aware estimation scheme, as we now show.

6.3.3 Expectation Programming in Turing

We now introduce a particular realization of the expectation programming concept which we call *Expectation Programming in Turing* (EPT). EPT builds on the PPS Turing to provide a highly effective, and surprisingly simple, mechanism to perform expectation programming. It allows users to specify $\gamma(\theta)$ analogously to how they would using Turing's `@model` macro, and uses Turing's `return` semantics to define F and thus $f(\theta)$.

The key component of the EPT framework is splitting up the estimation of the desired expectation as per the TABI framework of Section 6.2.3. To do so we use source-code transformations to generate three different Turing programs, one for each of the densities $\gamma_1^+(\theta)$, $\gamma_1^-(\theta)$, and $\gamma_2(\theta)$ (as per Equation (6.6)). We then estimate the expectation by individually estimating the normalization constant of each of these densities and then combining them as per Equation (6.4). Generating valid Turing programs allows us to leverage any inference algorithm in Turing that provides marginal likelihood estimates to estimate the quantities Z_1^+, Z_1^-, and Z_2. This modularity means that we do not have to implement custom inference algorithms that would only work with EPT.

Estimating expectations with EPT is done in two stages. First, users define an expectation program with the `@expectation` macro, which is a drop-in replacement for `@model`, and an example for which is shown in Figure 6.1. Using code

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end
```

```
@model function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    tmp = x^3
    @addlogprob!(log(max(tmp, 0)))
    return tmp
end
```

Figure 6.2: The results of one of the three program transformations applied to the EPT `@expectation` program from Figure 6.1 [left]. Presented is the transformation into a valid Turing `@model` program [right] corresponding to the density $\gamma_1^+(\theta) \propto \gamma(\theta) f^+(\theta)$. The transformed code fragment is highlighted. The full transformation is slightly more complex due to Turing's internals. Appendix C.2 shows the full source code transformation.

transformations, `@expectation` automatically generates the three Turing programs representing the densities $\gamma_1^+(\theta)$, $\gamma_1^-(\theta)$, and $\gamma_2(\theta)$. This happens behind the scenes and the user does not need to deal with the transformed programs directly.

To estimate the expectation, the user calls `estimate_expectation(expt_prog, method)`, where `method` specifies the estimation approach to be used. At present, the only supported class of methods is `TABI`, which implements the previously explained TABI estimators, but the syntax is designed to allow for easy addition of hypothetical alternative approaches.

EPT then estimates the normalization constants Z_1^+, Z_1^-, and Z_2 by running a Turing inference algorithm on each Turing program generated by `@expectation` and combining the normalization constant estimates to form an estimate of the expectation. In the example in Figure 6.1, we use `TABI` with annealed importance sampling `AnIS`, which is a new Turing inference algorithm that we have added to the system for the purposes of this paper. `TuringAlgorithm` is a thin-wrapper object storing the necessary information that allows `TABI` to use a Turing inference method. `AnIS` can be substituted with any other Turing inference algorithm that returns a marginal likelihood estimate. Here `AnIS()` implies the use of some arbitrary default AnIS parameters regarding the Markov chain transition kernel, and the number and spacing of intermediate potentials used.

6.3.4 Program Transformations

We now consider how to generate the Turing programs corresponding to each of the TABI densities. Note that expectation programs in EPT are also valid Turing models, i.e., replacing `@expectation` with `@model` yields a valid Turing program. Such a program corresponds to the unnormalized density $\gamma_2(\theta) = \gamma(\theta)$ without requiring any transformation of the source-code.

To create a Turing program corresponding to $\gamma_1^+(\theta)$, we need to multiply the unnormalized density of the unaltered Turing program $\gamma(\theta)$ by $\max(f(\theta), 0)$. This is achieved using Turing's aforementioned `@addlogprob!` primitive, such that we can think of it as adding a new factor $\max(f(\theta_{1:n_\theta}), 0)$ to the program density definition in (6.1). Our transformations are pattern matching procedures that find all the `return expr` statements in the function body and then a) create a new local variable `tmp = expr` (where `tmp` is a unique identifier generated using `gensym()`), b) insert a statement `@addlogprob!(log(max(tmp, 0)))` before the `return`, and c) change the return statement itself to `return tmp`. A concrete example of the transformation is presented in Figure 6.2. The transformation for $\gamma_1^-(\theta)$ is analogous but inserts a statement `@addlogprob!(log(-min(tmp, 0)))` instead.

Users can define multiple expectations by specifying multiple return values, while each individual return value needs to almost surely be a numerical scalar. This ensures that each target expectation is well defined and individually identified. For each return expression, we apply our program transformation separately and derive a corresponding TABI estimator for each. For example, if we have `return expr1, expr2, expr3`, the program transformation for $\{\gamma_1^+(\theta)\}_2$ would add the statement `@addlogprob!(log(max(expr2, 0)))`. Appendix C.7 shows a full example of this.

6.3.5 Validity of EPT

We now formalize and demonstrate the statistical correctness of the EPT approach. For simplicity, we will assume throughout that programs almost surely return a single scalar value (i.e. the probability that the return value fails to be a well-defined scalar is 0). Generalization to programs with multiple return values is straightforward (provided the number of return values is fixed) by considering each return value separately in isolation (as EPT does itself).

Proposition 4. *Let $\mathcal{E}$ be a valid expectation program in EPT with unnormalized density $\gamma(\theta_{1:n_\theta})$, defined on possible traces $\theta_{1:n_\theta} \in \Theta$, with return value $F = f(\theta_{1:n_\theta})$. Then $\gamma_1^+(\theta_{1:n_\theta}) := \gamma(\theta_{1:n_\theta}) \max(0, f(\theta_{1:n_\theta}))$, $\gamma_1^-(\theta_{1:n_\theta}) := -\gamma(\theta_{1:n_\theta}) \min(0, f(\theta_{1:n_\theta}))$, and $\gamma_2(\theta_{1:n_\theta}) := \gamma(\theta_{1:n_\theta})$ are all valid unnormalized probabilistic program densities. Further, if $\{\hat{Z}_1^+\}_m$, $\{\hat{Z}_1^-\}_m$, $\{\hat{Z}_2\}_m$ are sequences of estimators for $m \in \mathbb{N}^+$ such that*

$$\{\hat{Z}_1^\pm\}_m \xrightarrow{p} \int_\Theta \gamma_1^\pm(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}), \tag{6.9}$$

$$\{\hat{Z}_2\}_m \xrightarrow{p} \int_\Theta \gamma_2(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}) \tag{6.10}$$

where $\xrightarrow{p}$ *means convergence in probability as* $m \to \infty$*, then*

$$\frac{\{\hat{Z}_1^+\}_m - \{\hat{Z}_1^-\}_m}{\{\hat{Z}_2\}_m} \xrightarrow{p} \mathbb{E}[F]. \tag{6.11}$$

Proof. We start by noting that as $\gamma_2(\theta_{1:n})$ is identical to $\gamma(\theta_{1:n})$, it is by assumption a valid unnormalized program density. Meanwhile, by construction, $\gamma(\theta_{1:n})_1^+, \gamma(\theta_{1:n})_1^- \geqslant 0, \forall \theta_{1:n} \in \Theta$. Further, each can be written in the form of (6.1) by taking the correspond definition of $\gamma(\theta_{1:n})$ and adding in factors $\exp(\psi_{K+1}) = \max(0, f(\theta_{1:n}))$ and $\exp(\psi_{K+1}) = -\min(0, f(\theta_{1:n}))$ for $\gamma(\theta_{1:n})_1^+$ and $\gamma(\theta_{1:n})_1^-$ respectively. To finish the proof that $\gamma^{\pm}(\theta_{1:n})$ are valid densities, we show that $0 < Z_1^{\pm} < \infty$.

Starting with the standard definition of an expectation for arbitrary random variables, we can express $\mathbb{E}[F]$ as

$$\int_\Theta f(\theta_{1:n_\theta}) d\mathbb{P}(\theta_{1:n_\theta}) \tag{6.12}$$

$$= \int_\Theta f^+(\theta_{1:n_\theta}) d\mathbb{P}(\theta_{1:n_\theta}) - \int_\Theta f^-(\theta_{1:n_\theta}) d\mathbb{P}(\theta_{1:n_\theta}). \tag{6.13}$$

Noting that if F is integrable then by the definition of the Lebesgue integral both $\int_\Theta f^+(\theta_{1:n_\theta}) d\mathbb{P}(\theta_{1:n_\theta}) < \infty$ and $\int_\Theta f^-(\theta_{1:n_\theta}) d\mathbb{P}(\theta_{1:n_\theta}) < \infty$. Now inserting the distribution the program defines over $\theta_{1:n_\theta}$,

$$= \int_\Theta f^+(\theta_{1:n_\theta}) \pi(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}) - \int_\Theta f^-(\theta_{1:n_\theta}) \pi(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}) \tag{6.14}$$

and noting that, by assumption, the density satisfies $\gamma(\theta_{1:n_\theta}) \geqslant 0$ for all $\theta_{1:n_\theta} \in \Theta$ and the program's normalization constant is finite, $0 < \int_\Theta \gamma(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}) < \infty$,

$$= \frac{\int_\Theta f^+(\theta_{1:n_\theta}) \gamma(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}) - \int_\Theta f^-(\theta_{1:n_\theta}) \gamma(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta})}{\int_\Theta \gamma(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta})}, . \tag{6.15}$$

Using the definitions of γ_1^+ and γ_1^- gives

$$= \frac{\int_\Theta \gamma_1^+(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta}) - \int_\Theta \gamma_1^-(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta})}{\int_\Theta \gamma_2(\theta_{1:n_\theta}) d\mu(\theta_{1:n_\theta})} \tag{6.16}$$

$$=: \frac{Z_1^+ - Z_1^-}{Z_2}. \tag{6.17}$$

In our theorem statement we have assumed that $\{\hat{Z}_1^+\}_m \xrightarrow{p} Z_1^+$, $\{\hat{Z}_1^-\}_m \xrightarrow{p} Z_1^-$, and $\{\hat{Z}_2\}_m \xrightarrow{p} Z_2$, from which it now follows by Slutsky's Theorem that

$$\frac{\{\hat{Z}_1^+\}_m - \{\hat{Z}_1^-\}_m}{\{\hat{Z}_2\}_m} \xrightarrow{p} \frac{Z_1^+ - Z_1^-}{Z_2} = \mathbb{E}[F] \tag{6.18}$$

as required. □

Proposition 4 shows that if we have programs with the desired densities and we use consistent marginal likelihood estimators for each, then our resulting expectation estimates will themselves be consistent. The latter is covered by the consistency of Turing's own inference engines. The former requires that our transformed programs are valid Turing programs with the intended densities. We now show that this is indeed the case.

Given an input EPT program $\mathcal{E}$, EPT applies transformations to get the three Turing programs $\mathcal{P}_1^+$, $\mathcal{P}_1^-$, and $\mathcal{P}_2$ with $\gamma_1^+(\theta_{1:n_\theta})$, $\gamma_1^-(\theta_{1:n_\theta})$, and $\gamma_2(\theta_{1:n_\theta})$ as their respective densities. To ensure that the transformations for $\gamma_1^+(\theta_{1:n_\theta})$ and $\gamma_1^-(\theta_{1:n_\theta})$ are correct, we need to ensure that a) the inserted code in our transformations is itself valid, b) the transformation does not have any unintended side effects, and c) the new density terms add valid factors to the program density. The first is true as the operation of the transformed sections of code are identical to the originals except for the new `@addlogprob!` terms, which themselves produce no outputs and, by construction, use only the variables that are in scope. The second is guaranteed by ensuring that the `tmp` variables are given unique identifiers that cannot clash with each other or any other variables in the program. The third follows from the restriction that each return value must almost surely be a numerical scalar, coupled with the fact that the added density factors (namely `max(tmp, 0)` and `-min(tmp, 0)`) are non-negative by construction.

Thus, we have shown that EPT will produce a consistent estimation of program expectations, under the assumptions of Definition 4 and the consistency of the base inference algorithms implemented in Turing. In addition to this, EPT inherits all the desirable estimation properties of the TABI estimator that were discussed in Section 6.2.3. Altogether, this provides a theoretical motivation for the use of EPT which, as will show in Section 6.5, is underlined by empirical results.

6.4 Related Work

Our focus is explicitly on the case of *estimating* expectations. Though a few papers [Gordon et al., 2014, Zinkov and Shan, 2017] have provided alternative formalizations for the expectation defined by a probabilistic program, none do this from the perspective of directly targeting this expectation as the quantity to estimate. Relatedly, a few languages provide primitives to compute expectations *analytically* in the rare situation where this is possible, such as Hakaru [Zinkov and Shan, 2017] or λPSI [Gehr et al., 2020]. Unlike in our setting, these do not require notable changes to the backend computation from the standard inference

setting because the underlying problem remains the same: calculate an integral analytically. The contributions of these works are thus somewhat tangential to our own, with our key message being that *estimating* expectations *efficiently* requires a distinct computational pipeline to that of modern PPSs.

Some PPSs also provide syntactic sugars for forming expectation estimates from the samples produced by inference, but these do not adjust the inference itself to exploit target function information. For example, in Stan [Carpenter et al., 2017] users can apply target functions to posterior samples using the `generated_quantities` block. Similarly, in Pyro [Bingham et al., 2019] the return values are stored along with MCMC posterior samples, thus allowing expectations to be estimated by taking empirical averages. PyMC3 [Salvatier et al., 2016] allows users to track deterministic transformations of the latent variables. Turing itself also provides a `generated_quantities` function, similar to Stan (see Appendix C.1 for an example).

6.5 Experiments

We demonstrate the effectiveness of the EPT target-aware inference methods on three problems: a synthetic numerical example, an SIR epidemiology model, and a Bayesian hierarchical model of Radon concentration. Our EPT implementation and the code for all experiments is contained in the supplementary material.

The performance of EPT depends on the performance of the chosen marginal likelihood estimator. At the time of writing, Turing provides implementations of Sequential Monte Carlo [Del Moral et al., 2006] and Importance Sampling (IS) as inference algorithms that provide marginal likelihood estimates, but only allows using the prior as the proposal which can never be target-aware. To address this issue, we implemented a new Turing inference engine that uses Annealed Importance Sampling (AnIS) [Neal, 1998]. We chose AnIS because it is effective at estimating the normalization constant even for high-dimensional models [Wallach et al., 2009, Salakhutdinov and Larochelle, 2010, Wu et al., 2017].

AnIS requires setting two hyperparameters: an annealing schedule and a transition kernel. Currently, users can choose between two transition kernels: Metropolis-Hastings (MH) implemented in `AdvancedMH.jl` [Turing Development Team, 2020] and Hamiltonian Monte Carlo (HMC) [Neal, 2011, Hoffman and Gelman, 2014, Betancourt, 2018] in `AdvancedHMC.jl` [Xu et al., 2020]. To ensure a fair comparison we use the same setup and hyperparameters for both EPT's backend and standard, non-target-aware AnIS. We also compare directly to MCMC targeting the posterior

and using the same type of transition kernel as AnIS and EPT. This transition kernel is MH in Section 6.5.1 and HMC elsewhere. Detailed configurations are given in Appendix C.4.

To compare the performance of the estimators we look at the effective sample size (ESS) and the relative squared error (RSE) $\hat{\delta} := (\hat{\mu} - \mu)^2/\mu^2$, where μ denotes the ground-truth value and $\hat{\mu}$ is the estimate. All our experiments correspond to target functions which are always positive, so we use Z_1 to refer to Z_1^+ as $Z_1^- = 0$. Appendix C.3 shows how EPT can avoid computation for Z_1^- when possible.

Both EPT's backend and AnIS produce unnormalized weighted samples w_i so the ESS is calculated as $(\sum_i w_i)^2/\sum_i w_i^2$. For MCMC we focus solely on the RSE, as MCMC ESS estimates, which are based on autocorrelation [Vehtari et al., 2020], are not directly comparable with ESS estimates based on importance weights, and unreliable if the RSE is large.

For EPT we can look at the ESS of the two AnIS runs which we denote ESS_{Z_1} and ESS_{Z_2}. Similarly, ESS_{AnIS} is the ESS for AnIS. We use $\text{ESS}_{\text{AnIS retargeted}}$ to denote what the ESS would be if we use the AnIS samples to estimate Z_1 by multiplying the weights generated by AnIS with $f(\theta)$.

When comparing our estimators based on RSE and ESS we ensure that we give each estimator the same computational budget i.e. same number of likelihood evaluations. Thus if we have an AnIS estimator with n intermediate distributions and K samples, then we compare it to the EPT estimator which uses n intermediate distributions and $K/2$ samples for each of the two separate terms. In turn, we compare AnIS and EPT to an MCMC estimator with $n \cdot K$ samples.

6.5.1 Gaussian Posterior Predictive

The first problem considered is calculating the posterior predictive distribution of a Gaussian model with an unknown mean, where $\gamma(\theta) = \mathcal{N}(\theta; 0, I)\,\mathcal{N}(\mathbf{y}; \theta, I)$ and $f(\theta) = \mathcal{N}(-\mathbf{y}; \theta, \frac{1}{2}I)$ are the unnormalised density and target function, respectively. We assume our observed data is $\mathbf{y} = (3.5/\sqrt{10})\mathbf{1}$ where $\mathbf{1}$ is a 10-dimensional vector of ones. Using EPT we can express this expectation in just 5 lines of code—the full model is given in Appendix C.8. This problem is amenable to an analytic solution so allows us to compute the error of the estimates. Figure 6.3 compares the performance of EPT, AnIS, and MCMC (here MH). We see a clear benefit to using the target-aware inference algorithm to estimate the expectation. EPT achieves a lower RSE, and the ESS highlights the advantage of using separate marginal likelihood estimators for Z_1 and Z_2.

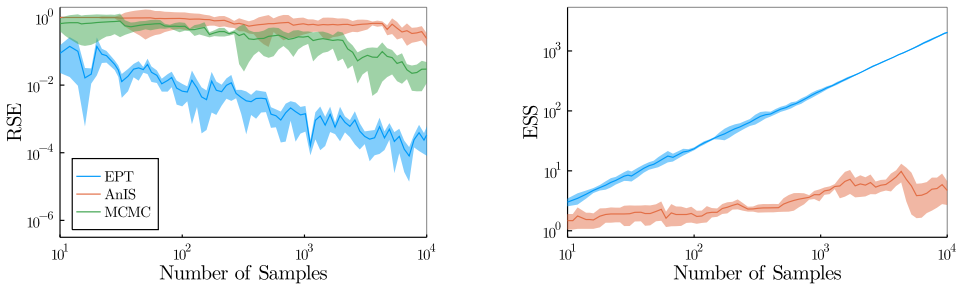


Figure 6.3: Relative squared error and effective sample size for the Gaussian posterior predictive experiment. The solid lines show the median of the estimator while the shaded region show the 25 % and 75 % quantiles. Medians and quantiles are computed over 10 separate runs with different random seed for the posterior predictive problem. For the ESS plot we are plotting $\min(\mathrm{ESS}_{Z_1}, \mathrm{ESS}_{Z_2})$ for EPT and $\min(\mathrm{ESS}_{\text{AnIS retargeted}}, \mathrm{ESS}_{\text{AnIS}})$ for AnIS.

6.5.2 SIR Epidemiological Model

Our second problem setting is a more applied example based on the Susceptible-Infected-Recovered (SIR) model of Kermack et al. [1927] from the field of epidemiology. Assume we face a disease outbreak. The government has provided us with a function yielding the expected cost of the disease which depends on the basic reproduction rate R_0, which indicates the expected number of people one infected person will infect in a population where everyone is susceptible. We seek to infer R_0 and the expected cost of the outbreak.

The SIR model divides the population into three compartments: people who are susceptible to the disease, those who are currently infected, and those who have already recovered. The dynamics of the outbreak are modelled by a set of differential equations

$$\frac{dS}{dt} = -\beta S\frac{I}{N}, \quad \frac{dI}{dt} = \beta S\frac{I}{N} - \gamma I, \quad \frac{dR}{dt} = \gamma I, \tag{6.19}$$

with parameters β and γ. S, I and R correspond to the number of people susceptible, infected and recovered, respectively. The size of the total population is $N = S+I+R$. Roughly, β models the constant rate of infectious contact between people, while γ is the constant recovery rate of infected individuals. From these parameters we can calculate the basic reproduction rate $R_0 = \beta/\gamma$. We assume γ to be known, and we want to infer β and the initial number of infected people I_0. The full statistical model along with the cost function which is defined with respect to R_0 is given in Appendix C.5.

This scenario is a good use case for EPT because we are interested in estimating a specific expectation with high accuracy. Furthermore, our cost function has some

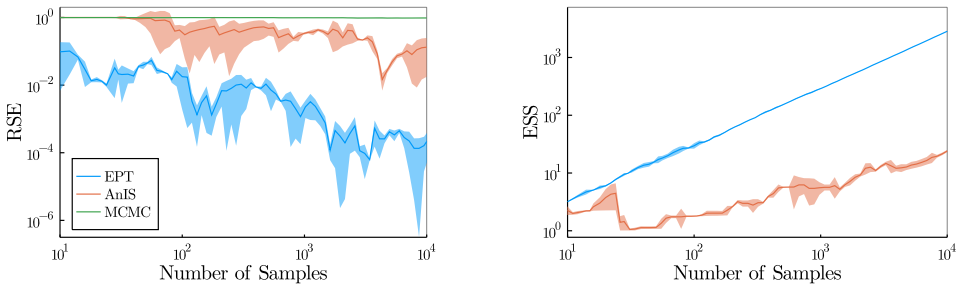


Figure 6.4: Relative squared error and effective sample size for the SIR experiment. Conventions as in Figure 6.3. Medians and quantiles are computed over 5 runs for different random seeds.

outcomes which might have low probability under the posterior but which incur a very high cost. These outcomes are liable to be missed by non-target-aware schemes, leading to extremely skew estimators that almost always underestimate the expectation.

Figure 6.4 compares the performance of the estimators. Since this problem is not amenable to an analytic solution, we estimate the ground-truth using a customized IS estimator with orders of magnitude more samples than estimates presented in the plot (see Appendix C.4). We see that our approach substantially improves on the baselines, with MCMC (here HMC) failing to provide any meaningful estimate; it produces no samples where $f(\theta)$ is significant. EPT is able to overcome this through its use of a separate estimator for $\gamma(\theta)f(\theta)$. The fact that MCMC does far worse than AnIS, despite neither being target-aware, stems from the latter producing a greater diversity of (weighted) samples, a small number of which land in regions of high $f(\theta)$ by chance, see Appendix C.10 for discussion.

6.5.3 Hierarchical Radon Concentration Model

Our third problem setting is a Bayesian hierarchical model for the radon concentration in households in different counties, adapted from Gelman and Hill [2006]. For the j^{th} house in county i, we would like to predict the log radon concentration y_{ij} inside the house. For each house we have a covariate x_{ij} which is 0 if the house has a basement, and 1 if it does not. With this setup, the model is defined as

$$\mu_\alpha \sim \mathcal{N}(0, 10), \qquad \alpha_i \sim \mathcal{N}(\mu_\alpha, 0.12), \tag{6.20}$$

$$\mu_\beta \sim \mathcal{N}(0, 10), \qquad \beta_i \sim \mathcal{N}(\mu_\beta, 0.22), \tag{6.21}$$

$$\epsilon \sim \text{HalfCauchy}(0, 5), \qquad y_{ij} \sim \mathcal{N}(\alpha_i + \beta_i x_{ij}, \epsilon). \tag{6.22}$$

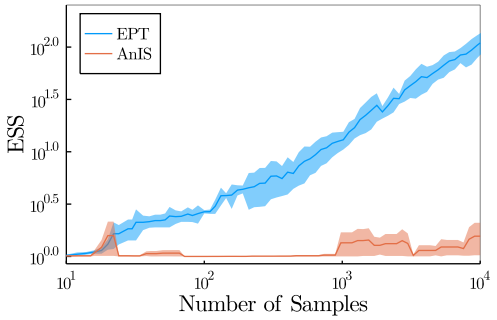


Figure 6.5: ESS plots for the Radon experiment. Conventions as in Figure 6.3; estimates based on 10 runs/seeds.

Table 6.1: Final estimates for the Radon experiments. Mean and standard deviation estimated over 10 runs.

Method	Final Estimate
EPT	3.74e−8 ± 2.39e−9
AnIS	1.15e−9 ± 3.02e−9
MCMC	7.79e−18 ± 2.46e−17

We now want to find out whether the radon level in *all* households is below an acceptable level, taking this threshold to be 4pCi/L. The probability of this event is equal to the expectation under the posterior of a step function $f(\theta)$. However, to allow the use of HMC transition kernels we use a logistic function as a continuous relaxation of this step function. See Appendix C.6 for more details.

This problem cannot be solved analytically and estimating the ground-truth with sufficient accuracy is computationally infeasible. We, therefore, resort to comparing EPT and AnIS based on their ESSs, noting that a low ESS almost exclusively means a poor inference estimate, while a high ESS is a strong (but not absolute) indicator of good performance. As we can see in Figure 6.5, EPT outperforms standard AnIS by several orders of magnitude. Additionally, Table 6.1 presents the final expectation estimates for each method. All methods differ in their estimates. However, EPT is the only one where the standard deviation of the estimate is small relative to its mean estimate, which, coupled with our ESS results, provides strong evidence that it is significantly outperforming the baselines. In particular, it seems clear that the MCMC (here HMC) estimate is very poor, as its estimate is many orders of magnitude smaller than the others, confirming that it has failed to produce any samples in regions where $f(\theta)$ is non-negligible.

6.6 Conclusion

We have introduced the concept of expectation programming which describes the process of encoding expectations programmatically and automating their estimation in an efficient, *target-aware* manner. This concept is realized in EPT, which is based on Turing, and we have shown that EPT estimates expectations effectively in practice through a combination of program transformations and target-aware

estimators. The modularity of EPT means that it can easily be built on by others through alternative base inference algorithms or target-aware strategies.

As EPT relies on TABI as its main estimation engine, the potential computational gains that can be achieved by TABI (in comparison to traditional inference pipelines) will increase with the mismatch between the posterior density and the target function.

Future work. We believe the introduction of the expectation programming concept can pave the way for exciting future advances. EPT focuses on the automation of TABI estimators but other implementations focusing on different approaches are conceivable, for example, systems targeting the automatic synthesis of control variates for a given input program, just as there are different PPSs focusing on distinct inference algorithms.

7 Conclusion

We began this thesis by describing science as a continuous process of: 1) constructing models of the world, 2) calibrating the models to observed data, 3) testing the model predictions against unseen data, and going back to step 1 to revise the model. Probabilistic programming systems provide us with a powerful framework to specify probabilistic models. However, seemingly simple programming constructs such as branching and loops can lead to new types of inference problems that require new algorithms to tackle them. Not only that, while estimating the posterior is the fundamental computational challenge in a Bayesian framework, the overall scientific workflow has many other aspects that need to be considered when building models in practice. Hence, when designing our probabilistic programming systems, it is worth not only to consider how to automate inference but also to think more holistically about the overall workflow of building and using models. In this thesis, we touched on multiple aspects on how to improve and expand the existing set of tools and algorithms for this.

In Chapter 4, we showed that breaking down the program into its individual program paths provides a powerful mechanism to construct flexible variational families. By decomposing the variational family as a mixture over program paths, we can consequently decompose the ELBO objective, resulting in an optimization process that can be easily parallelized over the program paths.

Chapter 5 started the theme of thinking more deeply about practical applications of probabilistic programming. When running inference on real-world data, we need to consider the fact that our models are misspecified. By relating the posterior in programs with stochastic support to Bayesian model averages, we showed that this

class of programs is particularly sensitive to model misspecification. Specifically, the posterior weights assigned to individual program paths can vary highly with small changes in the input data. We can produce more stable weights by optimizing the path weights based on predictive objectives instead of weighting the paths proportional to their normalization constant.

Chapter 6 demonstrated that it is worth thinking beyond the standard task of inference. The end goal of many workflows is to estimate expectations and if we can define the expectation in question ahead of time, we can leverage target-aware inference algorithms that lead to more accurate estimates. The framework of expectation programming allows users to automate the application of these target-aware algorithms, with big computational gains in practice.

7.1 The Place of Probabilistic Programming in a Deep Learning World

To conclude, it makes sense to step back and think about how probabilistic programming fits into the current wider machine learning research landscape. A lot of machine learning research has been driven by the goal of building systems that can reason and understand the world in a human-like way.

Broadly speaking, there are two dominant schools of thought on how to achieve this goal: the first believes that *deep learning* architectures based solely on neural networks are enough to reach this goal [LeCun et al., 2015, Schmidhuber, 2015], while the second believes that deep learning is fundamentally insufficient and instead we need to build systems which explicitly reason over "symbols" [Tenenbaum et al., 2011, Lake et al., 2017]. Proponents of the second school of thought often argue that probabilistic programming systems provide a natural framework to build such reasoning systems because Bayesian reasoning resembles how humans learn new concepts [Lake et al., 2015].

In the last couple of years, the "deep learning" school has demonstrated remarkable success in many domains, most notably in building so called "foundation models" [Bommasani et al., 2021] which are big deep learning models trained on large and potentially multi-modal datasets [Team Gemini et al., 2023, Achiam et al., 2023, Touvron et al., 2023]. Rather than building models that can incorporate more domain knowledge or reasoning tools, the biggest progress has been made by simply scaling up flexible base architectures and training them on the largest possible datasets [Vaswani et al., 2017, Rae et al., 2021, Brown et al., 2020]. A key principle has been to constrain the deep learning architectures only with a minimal set of

"inductive biases" and then let the learning algorithm—most often stochastic gradient descent—figure out the rest, instead of providing an extensive list of hard-coded rules.

On the outset, it seems then that the deep learning approach has won the argument for how to build intelligent systems and that elaborate systems that reason over symbols are not needed. However, despite their success, current foundation models, especially large language models (LLMs) for natural language processing, also have critical flaws, e.g. their tendency to produce hallucinations and incoherent reasoning [Achiam et al., 2023, Bubeck et al., 2023]. One potential avenue to tackle these flaws is to integrate deep learning architectures with programming abstractions and Bayesian reasoning (as argued for by e.g. Wong et al. [2023]). Probabilistic programming systems could then be used as a layer on top of pre-trained foundation models to imbue the deep learning architectures with reasoning capabilities.

Whether the probabilistic programming approach can truly make a difference in this domain still has to be proven, though. Crucially, it might fall victim to the "Bitter Lesson" described by Rich Sutton [Sutton, 2019]: seemingly elegant and clever methods are time and time again beaten by the "brute-force" approach of simply throwing more data and computing power at a problem. However, it is at least a research direction that is worth exploring and could potentially be an area in which probabilistic programming will be able to make a significant impact.

Not all of machine learning research is focused on building models that can reason and understand the world in a human-like way, though. There is a whole host of problem domains outside the classical AI tasks of vision and natural language processing where probabilistic programming can thrive and is already successful.

In many statistical tasks, we are often actually limited by the amount and quality of data we have access to, e.g. in election forecasting, clinical trials, or epidemiology. Probabilistic programming has been tremendously successful in these settings, as a way to champion the usage of Bayesian statistics and allowing practitioners to incorporate their domain knowledge into their models. These domains also continue to be a fruitful area for future research in probabilistic programming, as they are always in need to run inference in more complex models.

Additionally, in some scientific applications, the useful inductive biases that we need might have to be encoded in the form of simulator models. Compared to tasks such as language generation, many scientific prediction tasks satisfy well-known constraints and rules. For example, we know that for weather and climate predictions the overall underlying physical system conserves energy. While deep learning has recently shown initial successes in atmospheric modelling on short-term weather prediction tasks [Bi et al., 2022, Kurth et al., 2023, Lam et al., 2023], these deep

learning models often do not account for physical constraints which makes them fail to reliably generalize in settings outside their training regime [Kochkov et al., 2023].

A solution to this could lie in combining the strengths of deep learning with the physical constraints of climate models, by having simulators in which parts of the program are specified based on domain knowledge and others that are learned from data. Early examples of this approach, notably Neural General Circulation Models [Kochkov et al., 2023], have shown promising results in the context of weather modelling. This is similar to earlier work in PPSs which has already advocated for so-called *deep* probabilistic programming [Tran et al., 2017, Bingham et al., 2019, van de Meent et al., 2018] in which parts of a program are learnable components and this could prove to be a promising approach in the future.

Finally, while deep learning techniques provide us with extremely powerful mechanisms to make predictions from data, they often do not necessarily advance our understanding of the underlying systems. Fundamentally, probabilistic programming systems are tools to build *probabilistic models* of the world. Then, as long as humans are interested in improving their understanding of the world, probabilistic programming will continue to be a valuable tool for scientists to express their models.

Appendices

A

Appendix: Rethinking Variational Inference for Probabilistic Programs with Stochastic Support

A.1 Details on Resource Allocation

A.1.1 Background on Successive Halving

Successive Halving (SH) divides a total budget of T iterations into $L = \lceil \log_2(K) \rceil + 1$ phases and starts by optimizing each of K candidates, in our case the SLPs, for $\lfloor T/(KL) \rfloor$ iterations. It then ranks each of the candidates in terms of their performance, in our case the values of $\exp(\mathcal{L}_k)$, before eliminating the bottom half. This process then repeats, with each of the remaining candidates run for $2^{\ell-1}T/(KL)$ iterations at the ℓ-th phase. This results in an exponential distribution of resources allocated to the different candidates, with more resources allocated to those that are more promising after intermediate evaluation.

Adapting it to our setting of treating the problem as a top-m identification is done by simply using $L = \lceil \log_2(K) - \log_2(m) \rceil + 1$ phases instead of $L = \lceil \log_2(K) \rceil + 1$.

A.1.2 Online Resource Allocation

Here, we present an online version of Algo. 4, where the term 'online' refers to the fact that the algorithm considers more and more SLPs as the computational budget increases. The online variant of the algorithm is useful if a user is unsure about the total iteration budget that they want to spend on the input program.

Algorithm 6 Online SDVI

Require: Target program γ, iteration budget per SH run T, minimum no. of SH candidates m, parameter controlling $\alpha > 0$ exploration

Extract SLPs $\{\gamma_k\}_{k=1}^{K}$ from γ and set $\mathcal{C} = \{1, \ldots, K\}$
Formulate guide q_k for each SLP and initialize parameters ϕ_k
$t_k = 0$ for all $k \in \mathcal{C}$
while Stopping criteria not satisfied **do**
    $\mathcal{C}' \leftarrow \mathcal{C}$
    Phases in successive halving $L = \lceil \log_2(|\mathcal{C}|) - \log_2(m) \rceil + 1$
    for $l = 1, \ldots, L$ **do**
        Number of iterations $n_l = \lfloor \frac{T}{L|\mathcal{C}'|} \rfloor$
        for $k \in \mathcal{C}'$ **do**
            Perform n_l optimization iterations of ϕ_k targeting $\mathcal{L}_{\text{surr},k}(\phi_k)$
            Estimate $\mathcal{L}_{\text{surr},k}(\phi_k)$ using Monte Carlo estimate of Eq. (4.22)
            $t_k = t_k + n_l$
        end for
        Remove $\min(\lfloor |\mathcal{C}'|/2 \rfloor, |\mathcal{C}'| - m)$ SLPs from $\mathcal{C}'$ with the lowest $\exp(\alpha\, \mathcal{L}_{\text{surr},k}(\phi_k))/t_k$
    end for
    Extract new SLPs from γ and add them to $\mathcal{C}$, set $t_{k'} = 0$ for each new SLP with index k'
end while
Truncate q_k outside of SLP support, Θ_k, using Eq. (4.24)
Estimate each $\mathcal{L}_k(\phi_k)$ using Monte Carlo estimate of Eq. (4.13)
Calculate $q(k; \lambda)$ according to Eq. (4.15) and return $q(\theta; \phi, \lambda)$ as per Eq. (4.6)

This user might want to run SDVI with an initial iteration budget T_1 and after having observed the results, they might decide that they want to keep further optimizing the guide parameters. We therefore need to adapt Algo. 4 so that it can be 'restarted' after it has terminated. A naive approach to this would be to simply run Algo. 4 again but re-use the q_k's for the SLPs that have already been discovered and only initialize the q_k from scratch for SLPs which have not been seen before. However, this scheme is limited as it disproportionately favours SLPs which were discovered in the previous run. This is because for those SLPs the local ELBOs will already be relatively large compared to the newly added SLPs. As a consequence, SH will not assign significant computational budget to the SLPs that were added after the algorithm was restarted.

To safeguard against this behaviour we instead propose an online version of SDVI in Algo. 6 which is using a modified 'reward' for SH. Instead of ranking the different SLPs according to $\mathcal{L}_k(\phi_k(t_k))$ we instead propose the objective $\exp(\alpha\mathcal{L}_k(\phi_k(t_k)))/t_k$ where $0 < \alpha \leqslant 1$. The reward is scaled by the reciprocal of t_k because we are no

longer aiming to select the SLPs with the highest $\mathcal{L}_k(\phi_k(t_k))$ but instead aim to choose the SLPs which have been 'underselected' compared to other SLPs, assuming we should have selected them in proportion to $\exp(\alpha\mathcal{L}_k(\phi_k(t_k)))$. The scaling by the scalar α is a further mechanism to encourage more exploration, with setting $\alpha = 0$ equivalent to uniform sampling in the limit of repeated SH runs. Since this adapted objective takes into account the computational budget that was spent on each SLP, it is a more suitable objective when running SH repeatedly.

A.2 Details for Training Local Guides

A.2.1 Density Estimation of the Prior

Before we can define the KL divergence we first have to carefully define global and local prior distributions. We first define what we informally call the global "prior" distribution of the program as the product of all the terms added to the program density by the `sample` statements

$$\pi_{\text{prior}}(\theta_{1:n_\theta}) := \prod_{i=1}^{n_\theta} g_{a_i}(\theta_i|\eta_i). \tag{A.1}$$

However, here we are using the term prior only informally, since (A.1) is not a prior in the conventional Bayesian sense since the η_i can be functions of the observed data $\mathbf{y}$. Note that here n_θ in (A.1) is again a random variable since the raw random draws $\theta_{1:n_\theta}$ of the program do not necessarily have fixed length. Then similarly we define local 'prior' distributions

$$\pi_{\text{prior},k}(\theta_{1:n_k}) := \frac{\mathbb{I}[\theta_{1:n_k} \in \Theta_k] \prod_{i=1}^{n_k} g_{A_k[i]}(\theta_i|\eta_i)}{Z_{\text{prior},k}} = \frac{\mathbb{I}[\theta_{1:n_k} \in \Theta_k]\pi_{\text{prior}}(\theta_{1:n_k})}{Z_{\text{prior},k}},, \tag{A.2}$$

where

$$Z_{\text{prior},k} := \int_\Theta \mathbb{I}[\theta \in \Theta_k]\, \pi_{\text{prior}}(\theta) d\theta. \tag{A.3}$$

Note that for our purposes we will never actually have to estimate $Z_{prior,k}$, we only defined it to ensure that $\pi_{prior,k}$ is a normalized density. This allows us to define the forward KL divergence which we would like to optimize with respect to ϕ_k

$$\text{KL}(\pi_{\text{prior},k}(\theta) \parallel \tilde{q}_k(\theta;\phi_k)) = \mathbb{E}_{\pi_{\text{prior},k}(\theta)}\left[\log \frac{\pi_{\text{prior},k}(\theta)}{\tilde{q}_k(\theta;\phi_k)}\right] \tag{A.4}$$

which we can rewrite as

$$= \mathbb{E}_{\pi_{\text{prior},k}(\theta)}\left[\log \pi_{\text{prior},k}(\theta)\right] - \mathbb{E}_{\pi_{\text{prior},k}(\theta)}\left[\log \tilde{q}_k(\theta;\phi_k)\right]. \tag{A.5}$$

The first term is a constant with respect to ϕ_k and therefore does not affect the optimization

$$\propto \mathbb{E}_{\pi_{\text{prior},k}(\theta)}\left[-\log \tilde{q}_k(\theta;\phi_k)\right], \tag{A.6}$$

then by the definition of $\pi_{\text{prior},k}(\theta)$ in Eq. (A.2) this is equivalent to

$$= -\frac{1}{Z_{\text{prior},k}}\mathbb{E}_{\pi_{\text{prior}}(\theta)}\left[\mathbb{I}[\theta \in \Theta_k]\log \tilde{q}_k(\theta;\phi_k)\right]. \tag{A.7}$$

Finally, $Z_{prior,k}$ is a constant with respect to ϕ_k and can be dropped

$$\propto \mathbb{E}_{\pi_{\text{prior}}(\theta)}\left[-\mathbb{I}[\theta \in \Theta_k]\log \tilde{q}_k(\theta;\phi_k)\right]. \tag{A.8}$$

We can estimate the gradients of the objective in Eq. (A.8) using a Monte Carlo estimator

$$\nabla_{\phi_k}\mathbb{E}_{\pi_{\text{prior}}(\theta)}\left[-\mathbb{I}[\theta \in \Theta_k]\log \tilde{q}_k(\theta;k,\phi_k)\right] \approx \frac{1}{N}\sum_{j=1}^{N}\mathbb{I}[x^{(j)} \in \Theta_k]\nabla_{\phi_k}\log \tilde{q}_k(x^{(j)};k,\phi_k) \tag{A.9}$$

where $x^{(j)}$ are raw random draws generated by executing the input program forward. These gradient estimates can then be used in a stochastic gradient descent optimization procedure. In our experiments, we generate a fixed set of N samples and re-use the same set of samples for the entire optimization process. Other approaches are also possible such as periodically collecting a new set of samples and using local MCMC moves to collect samples instead of repeatedly sampling from the prior.

A.3 Additional Details for Experiments

For all experiments that rely on optimization we use the Adam optimizer [Kingma and Ba, 2015]. The experiments were executed on an internal cluster which uses a range of different computer architectures.

A.3.1 Model From Figure 4.1

Listing A.1: Pyro Code for Figure 4.1.

```
import pyro
import pyro.distributions as dist

def model():
    x = pyro.sample("x", dist.Normal(0, 1))
    if x < 0:
        z1 = pyro.sample("z1", dist.Normal(-3, 1))
```

```
    else:
        z1 = pyro.sample("z2", dist.Normal(3, 1))

    x = pyro.sample(
        "x", dist.Normal(z1, 2), obs=torch.tensor(2.0)
    )

guide = pyro.infer.autoguide.AutoNormalMessenger(model)
optim = pyro.optim.Adam({"lr": 0.01})
svi = pyro.infer.SVI(
    model, guide, optim, loss=pyro.infer.Trace_ELBO()
)

for j in range(2000):
    svi.step()
```

The full Pyro code for the model in Fig. 4.1, including automatically generating and training the guide is given in Listing A.1. The code for BBVI and SDVI is provided in the code supplementary. For Pyro's AutoGuide and BBVI we run the optimization for 2000 iterations with a learning rate of 0.01. Similarly, for SDVI we have a total iteration budget of $T = 2000$ and use a learning rate of 0.01; we set the minimum number of SH candidates to $m = 2$

A.3.2 Program with Normal Distributions

For SDVI, we use 10^3 samples from the prior to discover SLPs. To train the local guides to place support within the SLP boundaries we collect 10^2 samples per SLP and optimize the objective in Equation (4.23) for 10^3 iterations. We run Algorithm 4 with a total budget of $T = 10^5$ with 5 particles for the ELBO and to estimate the final SLP weights we use 10^3 samples per SLP. We use a learning rate of 0.01.

For Pyro AutoGuide, we run the optimization for 10^5 steps with 1 ELBO particle. For BBVI, we run the optimization for 10^4 steps with 10 ELBO particles. For both we use a learning rate of 0.01.

A.3.3 Infinite Gaussian Mixture Model

For SDVI, we use 10^3 samples from the prior to discover SLPs, run Algorithm 4 with a total budget of $T = 2 * 10^4$ with 10 particles for the ELBO and to estimate the final SLP weights we use 10^2 samples per SLP. We use a learning rate of 0.1.

For BBVI, we run for $2 * 10^4$ iterations using 10 particles for the ELBO and a learning rate of 0.1. In the guide, we use a categorical distribution for number of components K over the range $K \in [1, 25]$. We ran initial experiments with

instead using a Poisson distribution paramterized by the rate but we found this leads to an explosion in the number of components in the guide which resulted in the program running out of memory. For each μ_k the variational approximation is a diagonal Normal distribution parameterized by the mean and the diagonal entries in the covariance matrix.

For DCC, we run for 200 iterations, at each iteration we run 10 independent RMH chains generating 10 samples and to get a marginal likelihood estimate we use PI-MAIS [Martino et al., 2017] which places a proposal distribution (in our case a Gaussian) on the outputs of the RMH chains and samples from this proposal M times; we set $M = 10$.

A.3.4 Inferring Gaussian Process Kernels

A.3.4.1 Model Details

Our probabilistic context-free grammar for the kernel structure has the production rules

$$\mathcal{K} \rightarrow \text{SE} \mid \text{RQ} \mid \text{PER} \mid \text{LIN} \mid \mathcal{K} \times \mathcal{K} \mid \mathcal{K} + \mathcal{K}. \tag{A.10}$$

with the production probabilities $[0.2, 0.2, 0.2, 0.2, 0.1, 0.1]$. On each base kernel hyperparameter we place an InverseGamma$(\alpha = 2, \beta = 1)$ prior. For each base kernel the specific hyperparameters we wish to do inference over are:[1]

- Squared Exponential (SE): Lengthscale
- Rational Quadratic (RQ): Lengthscale, Scale Mixture
- Periodic (PER): Lengthscale, Period
- Linear (LIN): Bias

Assuming we have N observations with inputs $\mathbf{x} \in \mathbb{R}^N$ and outputs $\mathbf{y} \in \mathbb{R}^N$ our model can then be written as

$$\mathcal{K} \sim \text{PCFG}(), \quad \sigma \sim \text{HalfNormal}(0, 1), \quad \mathbf{y} \sim \mathcal{N}(0, \mathcal{K}(\mathbf{x}) + \sigma^2 \text{I}) \tag{A.11}$$

where PCFG() samples a kernel (and its hyperparameters) from the probabilistic context-free grammar and $\mathcal{K}(\mathbf{x})$ is the $N \times N$ covariance matrix computed from kernel $\mathcal{K}$.

[1] We use the same naming conventions as the Pyro Docs at https://docs.pyro.ai/en/stable/contrib.gp.html#module-pyro.contrib.gp.kernels.

A.3.4.2 Algorithm Configurations

For SDVI, we use 10^3 samples from the prior to discover SLPs, run Algorithm 4 with a total budget of $T = 10^6$ with 1 particles for the ELBO and to estimate the final SLP weights we use 10^2 samples per SLP. We use a learning rate of 0.005.

For BBVI, we run for 10^5 iterations using 10 particles for the ELBO and a learning rate of 0.005. The guide uses a log-normal distribution for the kernel hyperparameters and the observation noise, and for the discrete variables which influence the kernel structure we use categorical distributions. For DCC, we run for 10^3 iterations and otherwise use the exact same hyperparameters as in the Gaussian Mixture Model experiment.

A.4 Additional Experimental Results

A.4.1 Program with Normal Distributions

For completeness we include here the results for DCC on the model from Sec. 4.6.1. DCC does not have the same fundamental limitations as the BBVI baselines therefore is competitive with SDVI and provides a similar squared error for the SLP weights. In fact, it is quite impressive that SDVI is able to match the performance of DCC because DCC leverages marginal likelihood estimators which asymptotically converge to the true marginal likelihood whereas SDVI calculates the weights based on the ELBO. This is therefore a further indicator that for this model SDVI is able to provide good posterior approximations for each SLP.

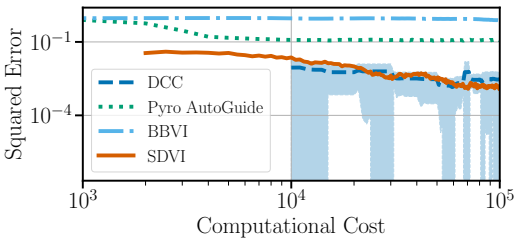


Figure A.1: Squared error for the model in § 4.6.1 with DCC baseline. Conventions as in Fig. 4.2a.

A.5 Difficulties of Parameter Learning for Models with Stochastic Support

In static support settings, one often uses variational bounds not only as a mechanism for inference, but also for training model parameters themselves [Kingma and Welling, 2014, Rezende et al., 2014]. Using our notation from Sec. 4.2.2, this setting corresponds to having model parameters, ψ, that we wish to optimize alongside the variational parameters, ϕ, such that the unnormalized density can be written as

$\gamma(\theta;\psi)$, with corresponding normalization constant $Z(\psi)$. The ELBO then depends on both the variational and model parameters $\mathcal{L}(\phi,\psi) := \mathbb{E}_{q(\theta;\phi)}\left[\log {}^{\gamma(\theta;\psi)}/_{q(\theta;\phi)}\right]$. Provided $Z(\psi)$ is differentiable with respect to ψ, both ϕ and ψ can then, at least in principle, be simultaneously optimized using stochastic gradient ascent.

However, similar as to the case of pure inference, naively extending this scheme to models with stochastic support is non-trivial and quickly runs into both conceptual and practical problems.

Parameters, ψ_k, that are inherently local to only a single SLP can be dealt with straightforwardly: as $\nabla_{\psi_k}\mathcal{L}_\ell = 0\ \forall \ell \neq k$ for such parameters, we can simply ignore parameters not associated with the SLP we are updating, that is we only take a gradient step for $\{\phi_k, \psi_k\}$ on Line 5 of Algo. 4.

Problems start to occur, though, in the more common scenario where parameters are shared between SLPs, in the sense that they influence more than one γ_k. Consider, for example, the GP model from Sec. 4.6.3 and assume that instead of doing inference over the observation noise, σ, we instead wish to treat this as a learnable parameter instead. Here σ could be seen as a 'global' model parameter as it appears in every SLP, so could be viewed as shared between them.

This now creates an issue in 'balancing' updates from different SLPs; the need to learn a shared ψ breaks the separability between inference problems for individual SLPs. Consequently, we can no longer directly treat how often we update each SLPs as just a resource allocation problem: making more updates on a given SLP now increases the influence that SLP has on the ψ which are learned. This problem is unlikely to be insurmountable—one could maintain a running estimate of $q(k;\lambda)$ during training and then use this to either directly control the resource allocation or scale the updates of ψ depending on how often the corresponding SLP has been used—but it does represent a notable complication that would require its own careful consideration.

Beyond this specific practical challenge, there is also a more fundamental and general issue for parameter learning under stochastic support: should shared parameters be treated globally when we are learning them? Going back to the example of the observation noise, σ, in our GP example, it will actually be quite inappropriate here to learn a single global value for σ, as the optimal observation noise will be different depending on the kernel structure. Thus, though the variable is shared between SLPs in the program itself, it would be advantageous to learn separate values for it for each SLP, regardless of the inference approach we take.

The natural solution to this issue would be to perform parameter learning separately for each SLP, e.g. learning a separate σ_k for each SLP in the GP example

above. However, this raises a variety of issues in its own, not least the fact that the inference algorithm will now start to influence the model itself: SDVI and BBVI will learn fundamentally different models. There may also be settings where it is important for a parameter to be truly global and thus shared across the SLPs, e.g. because such sharing is an explicit prior assumption we wish to make.

Further problems occur when we consider that it is also feasible for learnable parameters to influence the control flow of the program, or even the set of possible SLPs. For example, a learnable parameter could impact the maximum possible recursion depth of a recursive program. This will create challenging interactions between SLPs: updates of one will influence the desirable behaviour for the variational approximation of another. In turn, this can substantially complicate the resource allocation process and even the SLP discovery process itself.

Together, these aforementioned issues demonstrate that parameter learning for models with stochastic support is a complex issue, requiring specialist consideration beyond the scope of the current work.

A.6 Issues with Directly Training Local Guides

A natural question one might ask with the SDVI method is why do we not directly train q_k to (4.13) by treating it as an implicit variational approximation defined by $\tilde{q}_k$? Namely, we can express (4.13) in terms of $\tilde{q}_k$ as follows

$$\mathcal{L}_k(\phi_k) = \log \tilde{Z}_k(\phi_k) + \frac{1}{\tilde{Z}_k(\phi_k)} \mathbb{E}_{\tilde{q}_k(\theta;\phi_k)} \left[\mathbb{I}\left[x \in \Theta_k\right] \log \frac{\gamma_k(\theta)}{\tilde{q}_k(\theta;\phi_k)} \right], \tag{A.12}$$

which, in principle, could be directly optimized with respect to ϕ_k.

There are unfortunately two reasons that make this impractical. Firstly, though $\tilde{Z}_k(\phi_k)$ can easily be estimated using Monte Carlo, we actually cannot generate conventional unbiased estimates of $\log \tilde{Z}_k(\phi_k)$ and $1/\tilde{Z}_k(\phi_k)$ (or their gradients) because mapping the Monte Carlo estimator induces a bias. Second, this objective applies no pressure to learn a $\tilde{q}_k$ with a high acceptance rate, i.e. which actually concentrates on SLP k, such that it can easily learn a variational approximation that is very difficult to draw truncated samples from at test time.

By contrast, using our surrogate objective in (4.22) allows us to produce unbiased gradient estimates. Because of the mode seeking behaviour of variational inference, it also naturally forces us to learn a variational approximation with a high acceptance rate, provided we use a suitably low value of c. If desired, one can even take $c \to 0$ during training to learn an approximation which only produces samples from the

target SLP without requiring any rejection. Figure A.2 shows that empirically we learn a $\tilde{q}_k$ with a very high acceptance rates for the problem in Section 6.1.

Note that the surrogate and true ELBOs are exactly equal for any variational approximation that is confined to the SLP (as these have $\tilde{Z}_k(\phi_k) = 1$). This does not always necessarily mean that they have the same optima in ϕ_k for restricted variational families, even in the limit $c \to 0$. However, such differences originate from the fact that the truncation can itself actually generalize the variational family (e.g. if $\tilde{q}_k$ is Gaussian, then q_k will be a truncated Gaussians). As such, any hypothetical gains from targeting (4.13) directly will always be offset against drops in the acceptance rate of the rejection sampler.

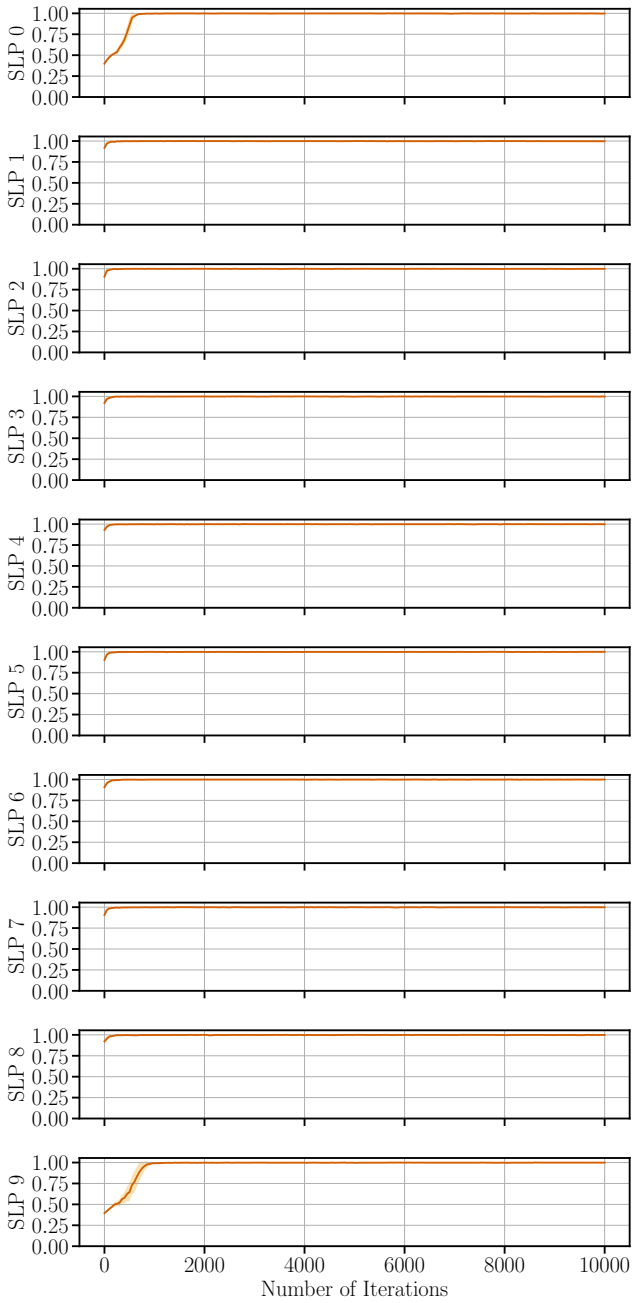


Figure A.2: Acceptance rates for evaluating the local ELBOs in each SLP for the model from Sec. 4.6.1. Each plot represents a separate SLP; the plot with "SLP i" corresponds to the SLP with $z = \text{i}$ in Eq. (4.6.1). We can see that for all SLPs the acceptance rate approaches 1 with more iterations, confirming the mode seeking behaviour that arises when maximizing the surrogate ELBO in Eq. (4.22).

B

Appendix: Beyond BMA over Paths

B.1 Probabilistic Programs without Predictive Distribution

```
def model(y):
  # Input data is a list of length 2
  if y[0] > 10:
    x = sample("x1", Normal(10, 1))
    sample("y1", Normal(x, 10), obs=y[0])
    sample("y2", Normal(y[0], sqrt(y[0])), obs=y[1])
  else:
    x = sample("x2", Normal(0, 1))
    sample("y1", Normal(x, 1), obs=y[0])
    sample("y2", Normal(y[0], 1), obs=y[1])
```

Figure B.1: Example of a probabilistic program without a predictive distribution.

Conventionally, in Bayesian statistics the modeller defines a prior $p(\theta)$ and predictive distribution $p(y_i \mid \theta)$. Then for a dataset $\mathbf{y} = (y_1, \ldots, y_N)$, these two ingredients together define the joint density $p(\theta, \mathbf{y}) = \prod_{i=1}^{N} p(y_i \mid \theta)\, p(\theta)$ from which can compute the posterior $p(\theta \mid \mathbf{y}) \propto p(\theta, \mathbf{y})$. In order to predict new data $\tilde{y}$, we can then use the posterior predictive distribution defined as

$$p(\tilde{y} \mid \mathbf{y}) = \int p(\tilde{y} \mid \theta)\, p(\theta \mid \mathbf{y}) dx. \tag{B.1}$$

However, more generally, the joint density does not necessarily factorize in this manner which makes it more difficult to automatically deduce a prediction task from the model alone. Additionally, in the setting of universal probabilistic programming

languages the input data can directly influence the model definition as well. Take for example the Pyro program in Fig. B.1. Here, the input data y directly influences what `sample` statements we encounter during execution. Furthermore, in both `sample` statements with address `"y2"` the distribution depends on the first data point `y[0]`. Therefore, just from this program definition alone, it is not clear what exactly a reasonable predictive distribution for a new data point would be.

B.2 PSIS-LOO Approximation

We here only give a brief description of the PSIS-LOO approximation as it is a common procedure. We refer the reader to Vehtari et al. [2017] for full details on the approximation and to Vehtari et al. [2015] for more details on PSIS. We want to approximate the local posterior predictive

$$\rho_k(y_i \mid y_{-i}) = \int_{\Theta_k} h_k(y_i \mid \theta)\, \pi_k(\theta \mid y_{-i}) d\theta \tag{B.2}$$

which can be rewritten in terms of the posterior of the full dataset as

$$= \int_{\Theta_k} h_k(y_i \mid \theta)\, \frac{\pi_k(\theta \mid y_{-i})}{\pi_k(\theta \mid y_{1:N})}\, \pi_k(\theta \mid y_{1:N}) d\theta. \tag{B.3}$$

Importantly, the ratio in that integral is proportional to a term that only depends on the individual predictive density

$$r_i := \frac{1}{h_k(y_i \mid \theta)} \propto \frac{\pi_k(\theta \mid y_{-i})}{\pi_k(\theta \mid y_{1:N})}. \tag{B.4}$$

Hence, we can use a self-normalized importance sampler to get an estimate of Eq. (B.3) as follows

$$\rho_k(y_i \mid y_{-i}) \approx \frac{\sum_{s \in I_k} r_i^s\, h_k(y_i \mid \theta_s)}{\sum_{s \in I_k} r_i^s} \tag{B.5}$$

where $r_i^s = 1/h_k(y_i \mid \theta_s)$ and θ_s are (approximate) samples from the posterior. However, these ratios r_i^s can often have high variance since $\pi_k(\theta \mid y_{1:N})$ will usually have lower variance and thinner tails than $\pi_k(\theta \mid y_{-i})$, in turn, leading to unstable estimates. To avoid these instabilities, the PSIS-LOO approximation replaces the ratios r_i^s with smoothed importance weights ν_i^s. The importance weights are computed by fitting a generalized Pareto distribution to the raw ratios r_i^s and replacing them with the expected order statistics of the fitted Pareto distribution. This then leads to the estimate

$$\hat{\rho}_k(y_i \mid y_{-i}) = \frac{\sum_{s \in I_k} \nu_i^s\, h_k(y_i \mid \theta_s)}{\sum_{s \in I_k} \nu_i^s}. \tag{B.6}$$

```
def pyro_subset_regression(X, y, X_val, y_val):
    k = pyro.sample(
        "k",
        dist.Categorical(torch.ones(X.shape[1]) / X.shape[1]),
        infer={"branching": True},
    )
    X = X[:, k]
    beta = pyro.sample(
        f"beta_{k.item()}", dist.Normal(0, np.sqrt(10)))
    sigma = pyro.sample("sigma", dist.Gamma(0.1, 0.1))
    mean = X * beta
    with pyro.plate("data", X.shape[0]):
        pyro.sample("y", dist.Normal(mean, sigma), obs=y)

    X_val = X_val[:, k]
    mean_val = X_val * beta
    return dist.Normal(mean_val, sigma).log_prob(y_val)
```

Figure B.2: Pyro program for the experiments in Sec. 5.6.2.

B.3 Implementation details

To be able to evaluate the stacking objective defined in Eq. (5.19) we need access to $\hat{\rho}_k(\tilde{y}_\ell)$, the estimates of the predictive densities. These, in turn, depend on the evaluations of the predictive densities. Hence, given validation data $\tilde{y}_1, \ldots, \tilde{y}_L$ and posterior samples $\theta_1, \ldots, \theta_S$, for stacking we require the evaluations $g(\tilde{y}_\ell \mid \theta_s)$. Note that the local posterior predictive distributions $\rho_k(\tilde{y}_\ell)$ are expectations under the posterior.

This is important because previous work noted that in the context of probabilistic programming these types of expectations can be formalized as the *expected return values* of a program [Gordon et al., 2014, Zinkov and Shan, 2017, Reichelt et al., 2022b, Lew et al., 2023]. Then to apply stacking, we require the user to define a program in which the return values for a given sample θ_s are the predictive density evaluations $g(\tilde{y}_1 \mid \theta_s), \ldots, g(\tilde{y}_L \mid \theta_s)$. For our Pyro implementation this means that we require the user to define a program which returns an L-dimensional vector which correspond to the (log) predictive density on the validation data points. Fig. B.2 shows how this can be done for the subset regression model from Sec. 5.6.2.

To actually compute the stacking weights we rely on Pyro's `Trace`[1] data structure which saves important metadata of each program execution such as the distribution type of each `sample` statement, the corresponding sampled value, its address, and, crucially, also the return value of the program. Algorithm 5 can then be implemented as a simple function which takes as input a list of posterior samples in the form

[1] https://docs.pyro.ai/en/stable/poutine.html#trace

of Pyro `Traces`. Because each `Trace` stores the address path of a given run, we can easily determine the set of SLPs and associate each sample with its SLP. From the `Trace` data structure we can then also extract the return values i.e. the evaluations $\log g(\tilde{y}_\ell \mid \theta_s)$. From these evaluations we can calculate the estimates $\hat{\rho}_k(\tilde{y}_\ell)$ which are needed in our stacking objective in Eq. (5.19). The optimization of the objective is done using SciPy's [Virtanen et al., 2020] implementation of the L-BFGS-B optimizer [Liu and Nocedal, 1989].

Similarly, to Zhou et al. [2020] and Reichelt et al. [2022a] we also allow the user to annotate specific discrete sampling statements to indicate that they influence the SLP of the program. In our implementation, this is done by passing `{"branching": True}` to the `infer` keyword argument of Pyro's `sample` statement. Using this annotation the inference backend can then control which SLP of the program is sampled by conditioning these `sample` statements on specific values. Furthermore, these annotations allow the inference backend to deterministically enumerate all SLPs. In our implementation, we give the user to enumerate all SLPs using breadth-first search, i.e. in the enumeration procedure we run the program as normal but if we encounter an annotated sampling statement we enumerate the whole support of the distribution and put each possible sampled value onto a queue. Once enumeration of a sample site has finished, execution resumes by popping a value from the front of the queue and continuing executing the program from the `sample` statement from which the value was sampled. This enumeration procedure is similar to how marginalization of discrete sample sites is implemented in Pyro. Note that we are assuming that each annotated discrete `sample` site has finite support. Future implementations could deal with finite support by truncating the sampling distribution.

B.3.1 Alternative Interface for Stacking

```
def model_average(models, model_args, model_kwargs):
    k = pyro.sample(
        "k",
        dist.Categorical(torch.ones(len(models)) / len(models)),
        infer={"branching": True},
    )
    return models[k](*model_args, **model_kwargs)
```

Figure B.3: Alternative interface to enable stacking of a list of user-specified models.

The main aim of our paper is to highlight the shortcomings of the default BMA weights in posteriors of probabilistic programs with stochastic support. However,

as a side-effect our Pyro implementation also provides a convenient mechanism for users to utilize stacking. Probabilistic programs with stochastic support allow users to encode a large class of problem settings, including the traditional BMA setting. However, in some cases, for convenience, users might not wish to write a single program with stochastic support which encodes all the different models in the BMA. Instead, the user might want to write K separate programs, each with static support, and then want to apply either BMA or stacking to that list of models. However, this interface is easily encoded within our framework because the user only needs to write another program which combines all the models together. This program is shown in Fig. B.3, it takes as input a list of Pyro models, and the arguments that should be passed to these models. The program then samples an index k and chooses one of the candidate models. Running standard inference in this program would correspond to BMA. If the user instead wants to run stacking instead of BMA, they can simply choose our stacking implementation and apply it to this program.

B.4 Experimental Details and Additional Results

To make the stacking approach widely applicable we implemented a version of the DCC framework in Pyro [Bingham et al., 2019]. For our implementation of DCC, we assume that all the stochastic branching happens on variables with discrete support and that the remaining latent variables in the program are continuous. Models of this form then permit the usage of Pyro's built-in Hamiltonian Monte Carlo [Neal, 2011, Hoffman and Gelman, 2014, Betancourt, 2018] as the local inference algorithm. To improve efficiency, our implementation also leverages Pyro's just-in-time (JIT) compilation ability to compile each SLP. Note that leveraging Pyro's JIT compilation is possible due to breaking down the original program into its SLPs because by default the JIT compilation cannot handle programs with stochastic support.

Our experiments were conducted on an internal compute cluster which consists of a mix of Intel Broadwell, Haswell, and Cascade Lake CPUs. In general, we use 16 cores to parallelize our computation and running a single replication of an experiment finishes in a matter of minutes if not seconds. An exception is the function induction experiment in Sec. 5.6.3 which takes around 3 hours for a single replication, using 32 cores.

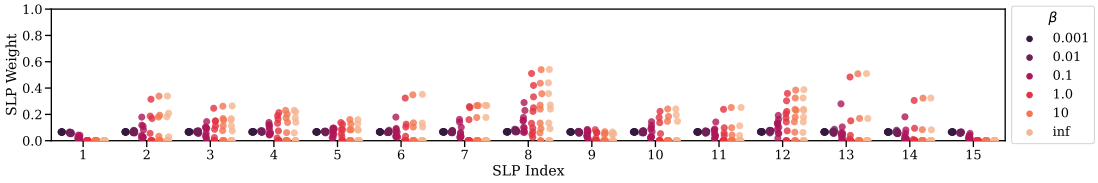


Figure B.4: Impact of varying regularization parameter β on the computed SLP weights. Smaller values of β lead to more regularization, pushing the SLP weights more towards the uniform distribution over SLPs.

B.4.1 When is Stacking Helpful?

For each setting we collect 10^3 HMC samples from each SLP with 400 burn-in samples. To estimate the normalization constant for the dominant SLP setting we use a proposal of $q(\theta_1, \theta_2) = \mathcal{N}(\theta_1; 0, 1)\,\Gamma(\theta_2; 1, 1)$ with 10^6 samples.

Overall, the experiments show that stacking is particularly useful if there are multiple SLPs which fit the data roughly equally well (with respect to the local normalization constant). In these cases, the SLP weights are at risk of being unstable and stacking can provide a more robust weight estimation procedure. On the other hand, if there is one dominant SLP which clearly performs better than all the other SLPs then we would expect there to be less to gain from stacking, as the weights will already be fairly stable.

B.4.2 Subset Regression

Following Breiman [1996] and Yao et al. [2018], we generate the regression coefficients according to $\beta_d = \eta\,(\zeta_d(4) + \zeta_d(8) + \zeta_d(12))$ with $\zeta_d(a) := \mathbb{I}\,[|d - a < h|]\ (h - |d - a|)^2$. The parameter h determines the number of "strong" coefficients. Following Yao et al. [2018] we set $h = 5$ which leads to 15 weak coefficients and set η such that the signal-to-noise ratio $\mathbb{V}\left[\sum_{d=1}^{15} \beta_d X_d\right] / (1 + \mathbb{V}\left[\sum_{d=1}^{15} \beta_d X_d\right]) = 4/5$ where X_d is the random variable for the dth covariate. Our input program has 15 SLPs and each SLP has the unnormalized density $\gamma_k(\theta_1, \theta_2, \theta_3) = \mathbb{I}\,[\theta_3 = k]$ $\prod_{i=1}^{N} \mathcal{N}(y_i; \theta_1 x_{i,k}, \theta_2^2)\,\mathcal{N}(\theta_1; 0, 10)\,\Gamma(\theta_2; 0.1, 0.1)\,\text{DiscreteUniform}(\theta_3; 1, 15)$. For DCC inference, for each SLP we collect 10^3 HMC samples with 400 burn-in samples. For RJMCMC our transition kernel samples a new θ_3, the variable controlling the covariate that is selected, from a uniform categorical distribution and new θ_1, the local regression coefficient, from a standard normal distribution. The noise variable is independently updated using a Metropolis-Hastings kernel.

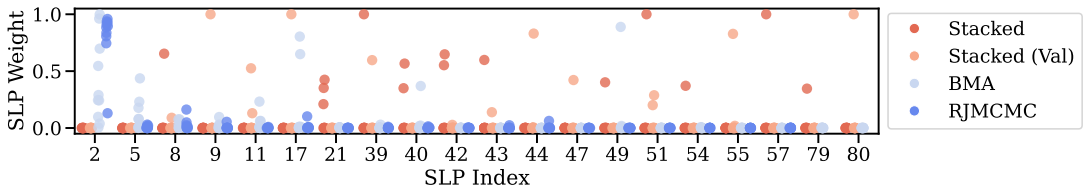


Figure B.5: SLP weights for the function induction model with the misspecified PCFG. We only plot SLPs which have achieved a weight > 0.3 in any run for any method.

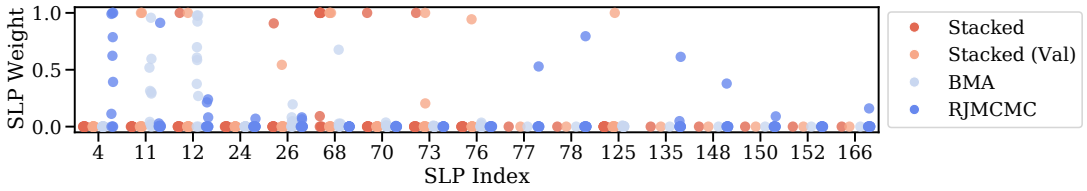


Figure B.6: SLP weights for the function induction model with the well-specified PCFG. We only plot SLPs which have achieved a weight > 0.3 in any run for any method. The SLP indices 11 and 12 both correspond to the true function (due to the symmetry in the $+$ operator there are two SLPs which represent the true function).

B.4.3 Function Induction

We are using the probabilistic context-free grammar $e \to \{x \mid \sin(a{*}e) \mid a{*}e{+}b{*}e\}$. In our model, we use prior production probabilities $[0.4, 0.4, 0.2]$ and for each of the sampled coefficients (denoted a and b in the PCFG) we use a $\mathcal{N}(0, 10)$ prior. Using the PCFG we can sample an expression for a function $f : \mathbb{R} \to \mathbb{R}$, then informally our model can be written as

$$f \sim \text{PCFG}(); \quad \sigma \sim \Gamma(\sigma; 1, 1); \quad y_i \sim \mathcal{N}(f(x_i), \sigma^2) \;\text{ for }\; i = 1, \ldots, N. \tag{B.7}$$

For DCC inference, we run 4 chains with 500 HMC samples and 200 burn-in samples for each SLP. The sampling distribution of SLP weights are shown in Fig. B.5. For RJMCMC, our transition kernel represents each expression as a PCFG parse tree. To propose a new expression, the transition kernel picks a random node in the parse tree and replaces that node with a sampled expression from the prior PCFG. The standard deviation of the likelihood, σ, is updated independently with a Metropolis-Hastings transition kernel. For RJMCMC, we collect the same number of total samples as for DCC to ensure a fair comparison.

Due to recursion in the PCFG rules, the program actually defines an infinite number of SLPs. Similar to Zhou et al. [2020], to avoid infinite recursion we restrict the underlying inference engine to only consider a finite number of SLPs. Our inference engine enumerates the possible PCFG expressions using breadth-first search and only considers the first 128 expressions.

B.4.4 Variable Selection

```
def variable_selection_model(X, y,):
    n_features = X.shape[1]
    features_included = torch.zeros((n_features,))
    for ix in range(n_features):
        features_included[ix] = pyro.sample(
            f"feature_{ix}", dist.Bernoulli(0.5),
            infer={"branching": True}
        )
    X_selected = X[:, features_included]
    num_selected = X_selected.shape[1]
    noise_var = pyro.sample(
        "noise_var", dist.InverseGamma(2.0, 1.0))
    with pyro.plate("features", num_selected):
        w = pyro.sample(
            "weights", dist.Normal(0, noise_var.sqrt()))
    means = w @ X_selected.T
    with pyro.plate("data", y.shape[0]):
        pyro.sample(
            "obs", dist.Normal(means, noise_var.sqrt()), obs=y)
```

Figure B.7: Pyro program for the variable selection experiments.

We assume we are given data $y_{1:N}$ and an associated matrix of covariates $X \in \mathbb{R}^{N \times D}$. We consider both regression and classification problems. The problem of variable selection is to find a subset of the features $\mathcal{D} \subseteq \{1, \ldots, D\}$ to make predictions given our data. For regression task, we have $y_i \in \mathbb{R}$ and our model for a specific subset $\mathcal{D}$ of the features is given by

$$\sigma^2 \sim \Gamma^{-1}(2, 1), \tag{B.8}$$

$$\beta_d \sim \mathcal{N}(0, \sigma^2) \quad \text{for } d \in \mathcal{D}, \tag{B.9}$$

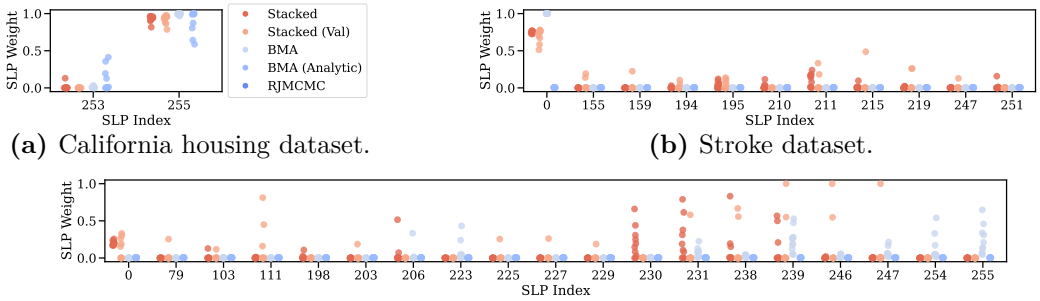
$$y_i \sim \mathcal{N}(\textstyle\sum_{d \in \mathcal{D}} \beta_d x_{i,d}, \sigma^2). \tag{B.10}$$

This form of the regression model allows us to analytically calculate the marginal likelihood (see e.g. Ch. 3.5 in Bishop [2006]). For classification tasks, we instead have $y_i \in \{0, 1\}$ and use the logistic regression model

$$\beta_d \sim \mathcal{N}(0, 1), \text{ for } d \in \mathcal{D} \text{ and } y_i \sim \mathcal{B}(S(\textstyle\sum_{d \in \mathcal{D}} \beta_d x_{i,d})),$$

where $S(x) = 1/(1 + \exp(-x))$ and $\mathcal{B}$ is the Bernoulli distribution.

The three different datasets we consider are: *California* housing ($N = 20,650$, $D = 8$; 50 % train, 50 % test) [Pace and Barry, 1997], a regression dataset where the goal is to predict the median house prices for districts of California; *Diabetes* ($N = 768$, $D = 8$; 80 % train, 20 % test) [Smith et al., 1988], a classification dataset



(a) California housing dataset.

(b) Stroke dataset.

(c) Diabetes dataset.

Figure B.8: SLP weights for the models in Sec. 5.6.4 (conventions as in Fig. B.5). The Stroke and Diabetes problems do not permit an analytic solution to the BMA weights.

on if a person has diabetes or not; and *Stroke* ($N = 4908$, $D = 8$; 80 % train, 20 % test), a classification dataset on if a person will have a stroke or not. For DCC inference, we collect 10^3 HMC samples with 400 burn-in samples for each SLP. The sampling distribution of the SLP weights are plotted in Fig. B.8. For RJMCMC, the transition kernel randomly selects a feature dimension d and for that dimension flips the inclusion, i.e. if the feature was previously included it will be excluded and vice versa. For a previously excluded feature, a new coefficient is sampled from the prior. For all other coefficients, a new coefficient is proposed from a standard normal centred at the current value. To ensure a fair comparison to DCC, we collect the same number of samples from RJMCMC as we do for DCC.

B.4.5 Modelling Radon Contamination in US Counties

As mentioned in the main text, our program encodes different modelling choices for both the intercept(s), α, and the slope parameter(s), β, of the regression relation $y_i = \alpha + \beta\, x_i$. Below we describe in detail the four different modelling choices for the intercept term. The modelling choices for the slope parameter are analogous, except we do not consider using a group-level predictor for β. The full Pyro program for this experiment is shown in Fig. B.9.

Pooling. This model corresponds to the SLP denoted "P, P" in Fig. 5.3.

$$\alpha \sim \mathcal{N}(0, 10), \qquad \beta \sim \mathcal{N}(0, 10), \tag{B.11}$$

$$\sigma \sim \text{Exponential}(5), \qquad y_i \sim \mathcal{N}(\alpha + \beta\, x_i, \sigma^2). \tag{B.12}$$

No pooling. Here $c[i]$ refers to the county index of the ith house. This model

corresponds to the SLP denoted "NP, P" in Fig. 5.3.

$$\alpha_c \sim \mathcal{N}(0, 10) \text{ for each county } c, \tag{B.13}$$
$$\beta \sim \mathcal{N}(0, 10), \tag{B.14}$$
$$\sigma \sim \text{Exponential}(5), \tag{B.15}$$
$$y_i \sim \mathcal{N}(\alpha_{c[i]} + \beta\, x_i, \sigma^2). \tag{B.16}$$

Hierarchical. This model corresponds to the SLP denoted "H, P" in Fig. 5.3. We are using a non-centred parameterization to allow for better sampling performance from the HMC sampler.

$$\sigma_\alpha \sim \text{Exponential}(1), \tag{B.17}$$
$$\mu_\alpha \sim \mathcal{N}(0, 10), \tag{B.18}$$
$$\epsilon_c \sim \mathcal{N}(0, 1) \quad \text{for each county } c, \tag{B.19}$$
$$\alpha_c = \mu_\alpha + \sigma_\alpha\, \epsilon_c, \tag{B.20}$$
$$\beta \sim \mathcal{N}(0, 10), \tag{B.21}$$
$$\sigma \sim \text{Exponential}(5), \tag{B.22}$$
$$f_i = \alpha_{c[i]} + \beta\, x_i, \tag{B.23}$$
$$y_i \sim \mathcal{N}(f_i, \sigma^2) \tag{B.24}$$

Group-level predictor. This model corresponds to the SLP denoted "G, P" in Fig. 5.3.

$$\gamma_0 \sim \mathcal{N}(0, 10), \tag{B.25}$$
$$\gamma_1 \sim \mathcal{N}(0, 10), \tag{B.26}$$
$$\sigma_\alpha \sim \text{Exponential}(1), \tag{B.27}$$
$$\epsilon_c \sim \mathcal{N}(0, 1) \quad \text{for each county } c, \tag{B.28}$$
$$\alpha_c = \mu_{\alpha,c} + \sigma_\alpha\, \epsilon_c, \tag{B.29}$$
$$\beta \sim \mathcal{N}(0, 10), \tag{B.30}$$
$$\sigma \sim \text{Exponential}(5), \tag{B.31}$$
$$y_i \sim \mathcal{N}(f_i, \sigma^2) \tag{B.32}$$

where $\mu_{\alpha,c} = \gamma_0 + \gamma_1\, u_c$ and $f_i = \alpha_{c[i]} + \beta\, x_i$.

To ensure we have a balanced representation of data in each county in both the training and the testing data we apply stratified sampling: for each county, we hold out 20 % of the observations for evaluation. For this dataset we do not run stacking

with a validation set because of the limited amount of data available per county. For DCC inference, we collect 2000 HMC samples with 2000 samples for burn-in for each SLP. For RJMCMC, the transition kernel picks a modelling choice of α and β and then samples the local parameters for each modelling choice from the prior. The standard deviation σ is updated separately with a Metropolis-Hastings transition kernel. To ensure a fair comparison between DCC and RJMCMC we collect the same total number of samples.

B.4.6 Stacked RJMCMC

We conducted further experiments to determine the impact of running stacking on top of RJMCMC, we call this *Stacked RJMCMC*. The results can be viewed in Fig. B.10 where we plot the difference in LPPD between RJMCMC and other methods, i.e. $\text{LPPD}_{\text{Diff}} = \text{LPPD}_{\text{Other}} - \text{LPPD}_{\text{RJMCMC}}$. Note, compared to the main paper we here evaluate the difference in LPPD to RJMCMC and not to stacking on top of DCC. This is because we care about investigating the performance improvement relative to RJMCMC. We observe in Fig. B.10 that Stacked RJMCMC generally leads to higher LPPD than RJMCMC. The difference is positive for all problem settings. Further, the difference is statistically significant under a Wilcoxon-signed rank test in all problems, except for *California* and *Stroke*.

B.5 PAC-Bayes and Stacking

For completeness, we first give a brief introduction into PAC-Bayes which is mostly based on Morningstar et al. [2022]. Morningstar et al. [2022] assume a setting in which data is sampled i.i.d. from $\tilde{y}_\ell \sim p_{\text{true}}(\tilde{y})$, that we have a parameterized probability model $p(\tilde{y} \mid \theta)$ and we want to find a mechanism to fit a distribution, $q(\theta)$, over the parameters of the probability model.

The **true predictive risk** is given by

$$\mathcal{P}(q) := -\mathbb{E}_{p_{\text{true}}(\tilde{y})}[\log \mathbb{E}_{q(\theta)}[p(\tilde{y} \mid \theta)]] \tag{B.33}$$

and in most applications is the quantity we care about in the end because it directly measures the quality of our predictions. However, in practice we cannot evaluate the true predictive risk because we do not have access to the true data generating distribution. The goal of PAC-Bayes methods is then to provide a stochastic upper bound on $\mathcal{P}(q)$ which can be used to train $q(\theta)$ [Masegosa, 2020, Morningstar et al., 2022].

The **empirical predictive risk**

$$\overline{\mathcal{P}}(q) := -\frac{1}{L}\sum_{\ell=1}^{L}\log \mathbb{E}_{q(\theta)}[p(\tilde{y}_\ell \mid \theta]. \tag{B.34}$$

is an empirical estimate of the true predictive risk. *Ensemble methods (which include stacking) directly minimize the empirical predictive risk.* For example, our stacking objective in Eq. (5.19) is a particular instantiation of the empirical predictive risk. We will explain the connection in more detail below.

The **true inferential risk**

$$\mathcal{R}(q) := -\mathbb{E}_{p_{\text{true}}(\tilde{y})}[\mathbb{E}_{q(\theta)}[\log p(\tilde{y} \mid \theta)]] \tag{B.35}$$

is an *upper bound on the true predictive risk* (by applying Jensen's inequality). In the case that our model is well specified, i.e. $\exists\theta'$ s.t. $p_{\text{true}}(\cdot) = p(\cdot \mid \theta')$, then $\operatorname{argmin}\mathcal{P}(q) = \operatorname{argmin}\mathcal{R}(q)$. Hence, when our model is well specified minimizing the true inferential risk is equivalent to minimizing the true predictive risk.

The **empirical inferential risk**

$$\overline{\mathcal{R}}(q) := -\frac{1}{L}\mathbb{E}_{q(\theta)}[\log p(\tilde{y}_\ell \mid \theta)] \tag{B.36}$$

is the empirical estimate of the true inferential risk. *Minimizing this risk directly is equivalent to maximum likelihood estimation.* By adding an extra regularization term to the empirical inferential risk we get the **PAC-inferential risk**, given by

$$\widetilde{\mathcal{R}}(q; r, \beta) := \mathbb{E}_{q(\theta)}\left[-\frac{1}{L}\sum_{\ell=1}^{L}\log p(\tilde{y}_\ell \mid \theta) + \frac{1}{\beta L}\log\frac{q(\theta)}{r(\theta)}\right] \tag{B.37}$$

where $r(\theta)$ is a user specified prior distribution on the parameters θ. This is a stochastic upper bound on the true predictive risk.

Morningstar et al. [2022] use results from Burda et al. [2016] to tighten the PAC-inferential risk and introduce **the PAC^M bound**

$$\widetilde{\mathcal{P}}_{M,L}(q; r, \beta) := -\frac{1}{L}\sum_{\ell=1}^{L}\mathbb{E}_{q(\theta^M)}\left[\log\left(\frac{1}{M}\sum_{j=1}^{M}p(\tilde{y}_\ell \mid \theta_j)\right)\right] + \frac{1}{\beta L}\mathrm{KL}(q(\theta) \,\|\, r(\theta)). \tag{B.38}$$

Notably, as the PAC-inferential risk, this bound contains a regularization term which is meant to prevent overfitting.

Theorem 1 (Morningstar et al. [2022])**.** *For all $q(\theta)$ absolutely continuous with respect to $r(\theta)$, $\tilde{y}_\ell \sim p_{true}(\tilde{y})$ i.i.d., $\beta \in (0, \infty)$, $L, M \in \mathbb{N}$, $p(\tilde{y} \mid \theta) \in (0, \infty)$ for all*

$\{\theta \in \Theta \mid p_{true}(\tilde{y}) > 0\} \times \{\theta \in \Theta \mid r(\theta) > 0\}$, *and* $\xi \in (0,1)$, *then with probability at least* $1-\xi$,

$$\mathcal{P}(q) \leqslant \widetilde{\mathcal{P}}_{M,L}(q; r, \beta) + \psi(p_{true}, \beta, M, L, r, \xi) - \frac{1}{\beta\, M\, L} \log \xi \tag{B.39}$$

and furthermore (unconditionally)

$$\widetilde{\mathcal{P}}_{M+1,L}(q; r, \beta) \leqslant \widetilde{\mathcal{P}}_{M,L}(q; r, \beta) \tag{B.40}$$

where $\widetilde{\mathcal{P}}_{M,L}(q; r, \beta)$ *as in Eq.* (B.38) *and:*

$$\psi(p_{true}, \beta, M, L, r, \xi) := \frac{1}{\beta\, M\, L} \log \mathbb{E}_{p_{true}(\tilde{y}^L)} \mathbb{E}_{r(\theta^M)} \left[\exp(\beta L M \Delta(\tilde{y}^L, \theta^M))\right], \tag{B.41}$$

$$\Delta(\tilde{y}^L, \theta^M) := \frac{1}{L} \sum_{\ell=1}^{L} \log \left(\frac{1}{M} \sum_{j=1}^{M} p(\tilde{y}_\ell \mid \theta_j) \right) \tag{B.42}$$

$$- \mathbb{E}_{p_{true}(\tilde{y})} \left[\log \left(\frac{1}{M} \sum_{j=1}^{M} p(\tilde{y} \mid \theta_j) \right) \right]. \tag{B.43}$$

For a proof see Appendix C.3 of Morningstar et al. [2022].

B.5.1 From PAC-Bayes to Regularized Stacking

In order to connect the ideas from Morningstar et al. [2021] with the stacking objective for probabilistic programs we need to define a specific form for the parameterized probability model and the distribution q which is meant to be optimized. We choose the latent variable of our probability model to be the random variable k which indexes into the SLPs. Our probabilistic model then turns out to be $p(\tilde{y} \mid k) = \rho_k(\tilde{y})$ and our approximate posterior $q(k)$ is a categorical distribution over $\{1, \ldots, K\}$ parameterized by the weights w, i.e. $q(k) = \text{Categorical}(w_1, \ldots, w_K)$. In this formulation the true predictive risk is given by

$$\mathcal{P}(q) = -\mathbb{E}_{p_{\text{true}}(\tilde{y})}[\log \mathbb{E}_{q(k)}[\rho_k(\tilde{y})]] \tag{B.44}$$

$$= -\mathbb{E}_{p_{\text{true}}(\tilde{y})} \left[\log \left(\sum_{k=1}^{K} w_k \rho_k(\tilde{y}) \right) \right] \tag{B.45}$$

which is equivalent to our stacking objective (Eq. (5.11) in the main paper). The PAC^M bound becomes

$$\widetilde{\mathcal{P}}_{M,L}(q; r, \beta) = -\frac{1}{L} \sum_{\ell=1}^{N} \mathbb{E}_{q(k^M)} \left[\log \left(\frac{1}{M} \sum_{j=1}^{M} \rho_{k_j}(\tilde{y}_\ell) \right) \right] + \frac{1}{\beta L} \text{KL}(q(k) \parallel r(k)). \tag{B.46}$$

Note that $\beta = 1$ can be interpreted as the "standard Bayesian" setting as it provides an equal weighting of the prior term $\mathrm{KL}(q(k) \parallel r(k))$ and likelihood term $\sum_{\ell=1}^{N} \mathbb{E}_{q(k^M)}\left[\log\left(\frac{1}{M}\sum_{j=1}^{M} \rho_{k_j}(\tilde{y}_\ell)\right)\right]$. We can rewrite the sum inside the log as a sum over SLPs, as follows

$$\widetilde{\mathcal{P}}_{M,L}(q; r, \beta) = -\frac{1}{L}\sum_{\ell=1}^{L} \mathbb{E}_{q(k^M)}\left[\log\left(\sum_{k=1}^{K} \frac{|I_k|}{M}\rho_k(\tilde{y}_\ell)\right)\right] + \frac{1}{\beta L}\mathrm{KL}(q(k) \parallel r(k)) \quad \text{(B.47)}$$

where $I_k := \{k_j \mid j = \{1, \ldots, M\}, k_j = k\}$ is the set of all parameter samples that are equal to k. Now by the law of large numbers (LLN) as $m \to \infty$ this converges to

$$\widetilde{P}_{\infty,L}(q; r, \beta) = R_\beta(w_{1:K}) = -\frac{1}{L}\sum_{\ell=1}^{L} \log\left(\sum_{k=1}^{K} w_k \rho_k(\tilde{y}_\ell)\right) + \frac{1}{\beta L}\mathrm{KL}(q(k) \parallel r(k)). \quad \text{(B.48)}$$

The first term in this objective is now the stacking objective and the second term is a regularization term pushing the distribution over weights to the prior $r(k)$. In other words, **this is our estimated stacking objective from Eq. (5.19) with an added regularization term.**

There is one caveat with pushing $M \to \infty$, as shown by Morningstar et al. [2021] the slack term ψ that controls the tightness of the bound grows linearly in M. Hence, the bound becomes vacuous for infinite M. However, there are several reasons why in our setting it is still reasonable to work with the objective $\widetilde{P}_{\infty,L}(q; r, \beta)$. First of all, in our specific setting the special case $M = 1$ is the degenerate setting in which we collapse onto a single SLP. This is exactly the behaviour we are trying to avoid in the first place. Additionally, Morningstar et al. [2022] have shown that performance increases with using larger M and show empirically that using $\widetilde{P}_{\infty,L}(q; r, \beta)$ can be beneficial when it is possible to do so. The only practical consideration they mention for increasing M are issues with gradient variance which are not applicable in our setting.

To choose the prior one could be inclined to use the posterior SLP weights $Z_k / \sum_{k'} Z_{k'}$, as they are our Bayesian beliefs about which SLP has generated the data. However, as we have argued in the main text these weights can be very unstable and expensive to estimate. Hence, a reasonable default choice could be to use the discrete uniform distribution. This will further discourage stacking from collapsing onto a single SLP. For the special case of the prior $r(k)$ being chosen to be the uniform distribution the objective further simplifies to

$$R_\beta(w_{1:K}) = \frac{1}{L}\sum_{\ell=1}^{L} \log\left(\sum_{k=1}^{K} w_k \rho_k(\tilde{y}_\ell)\right) - \frac{1}{\beta L}\left(\mathrm{H}[q(k; w_{1:K})] + \log K\right) \quad \text{(B.49)}$$

where H denotes the Shannon entropy.

```
def radon_model(log_radon, floor_ind, county, n_counties, uranium):
    alpha_choice = pyro.sample(
        "alpha_choices", dist.Categorical(torch.ones(4)),
        infer={"branching": True})
    if alpha_choice == 0:
        # Pooled model
        alpha = pyro.sample("alpha", dist.Normal(0, 10))
    elif alpha_choice == 1:
        # County specific intercepts
        with pyro.plate("num_alpha", n_counties):
            alpha = pyro.sample("alpha", dist.Normal(0, 10))
        alpha = alpha[..., county]
    elif alpha_choice == 2 or alpha_choice == 3:
        if alpha_choice == 2:
            # Partially pooled model
            mean_a = pyro.sample("mean_a", dist.Normal(0, 1))
        elif alpha_choice == 3:
            # Uranium context
            gamma_0 = pyro.sample("gamma_0", dist.Normal(0, 10))
            gamma_1 = pyro.sample("gamma_1", dist.Normal(0, 10))
            mean_a = gamma_0 + gamma_1 * uranium
        std_a = pyro.sample("std_a", dist.Exponential(1))
        with pyro.plate("num_alpha", n_counties):
            z_a = pyro.sample("z_a", dist.Normal(0, 1))
        alpha = mean_a + std_a * z_a
        alpha = alpha[..., county]
    beta_choice = pyro.sample(
        "beta_choices", dist.Categorical(torch.ones(3)),
        infer={"branching": True})
    if beta_choice == 0:
        # Pooled model
        beta = pyro.sample("beta", dist.Normal(0, 10))
    elif beta_choice == 1:
        # County specific slopes
        with pyro.plate("num_beta", n_counties):
            beta = pyro.sample("beta", dist.Normal(0, 10))
        beta = beta[..., county]
    elif beta_choice == 2:
        # Partially pooled model
        mean_b = pyro.sample("mean_b", dist.Normal(0, 1))
        std_b = pyro.sample("std_b", dist.Exponential(1))
        with pyro.plate("num_beta", n_counties):
            z_b = pyro.sample("z_b", dist.Normal(0, 1))
        beta = mean_b + std_b * z_b
        beta = beta[..., county]
    theta = alpha + beta * floor_ind
    sigma = pyro.sample("sigma", dist.Exponential(5))
    with pyro.plate("data", log_radon.shape[0]):
        pyro.sample("ys", dist.Normal(theta, sigma), obs=log_radon)
```

Figure B.9: Pyro program for the radon model.

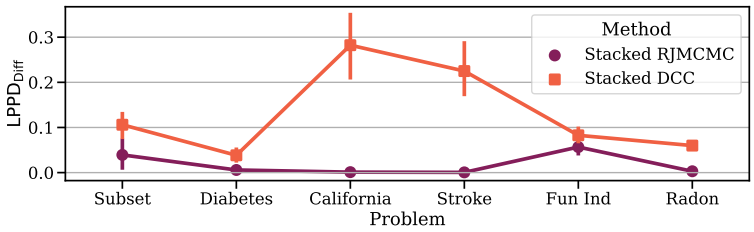


Figure B.10: Difference in LPPD between RJMCMC and other methods (higher is better).

```
def distinct(y):
    model1 = pyro.sample("model1", dist.Bernoulli(0.5))
    if model1:
        z = pyro.sample("z1", dist.Normal(0, 1))
        with pyro.plate("data", y.shape[0]):
            pyro.sample("obs", dist.Normal(z, 0.62), obs=y)
    else:
        z = pyro.sample("z2", dist.Normal(0, 1))
        with pyro.plate("data", y.shape[0]):
            pyro.sample("obs", dist.Normal(z, 2.0), obs=y)

def overlap(X, y):
    model1 = pyro.sample("model1", dist.Bernoulli(0.5))
    if model1:
        w = pyro.sample(
            "w1", dist.Normal(0, 1).expand([2]).to_event(1),
        )
        mean = w @ X[:, [0, 2]].T
    elif model2:
        w = pyro.sample(
            "w2", dist.Normal(0, 1).expand([2]).to_event(1),
        )
        mean = w @ X[:, [0, 3]].T
    with pyro.plate("data", X.shape[0]):
        pyro.sample("obs", dist.Normal(mean, 1.0), obs=y)

def dominating(X, y):
    model1 = pyro.sample("model1", dist.Bernoulli(0.5))
    if model1:
        w = pyro.sample("w", dist.Normal(0, 1))
        fs = w * X
    else:
        sin_w = pyro.sample("sin_w", dist.Normal(0, 1))
        fs = torch.sin(sin_w * X)
    sigma = pyro.sample("sigma", dist.Gamma(1, 1))
    with pyro.plate("data", X.shape[0]):
        pyro.sample("obs", dist.Normal(fs, sigma), obs=y)
```

Figure B.11: Pyro program for experiments in Sec. 5.6.1

C

Appendix: Expectation Programming

C.1 Estimating Expectations in Turing

C.1.1 Standard approach

```
@model function model(y=2)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
end

num_samples = 1000
posterior_samples = sample(model(), NUTS(0.65), num_samples)

f(x) = x^3
posterior_x = Array(posterior_samples[:x])
expectation_estimate = mean(map(f, posterior_x))
```

Full example of the estimation of an expectation with the Turing language. The user first defines the model, then conditions it on some observed data, computes posterior samples and then uses these samples to compute a Monte Carlo estimate of the expectation.

C.1.2 Using generated quantities function

When we designed the API Turing largely ignored the `return` statements in the model definition. In the meantime Turing introduced a convenience function `generated_quantities`. Given a model and N samples it returns a list of the N return values generated by running the program on each sample. Note that

`generated_quantities` reruns the entire `model` function for each posterior sample to compute the return value. This means that for models which have an expensive likelihood computation the use of `generated_quantities` might incur a significant overhead.

It is important to note that `generated_quantities` is merely a convenience function and does not change how Turing interprets model definitions. In fact, the `generated_quantities` function provides complimentary functionality and Turing models generated with EPT can use this function without problems.

The example from Section C.1.1 can be rewritten to use `generated_quantities`:

```
@model function model(y=2)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end

num_samples = 1000
posterior_samples = sample(model(), NUTS(0.65), num_samples)

expectation_estimate = mean(generated_quantities(model(),
↪  posterior_samples))
```

C.2 Full Example of Macro Transformation

The expectation

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end
```

gets transformed into

```
@model function gamma1_plus(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    tmp = x^3
    if _context isa Turing.DefaultContext
        @addlogprob!(log(max(tmp, 0)))
    end
    return tmp
end

@model function gamma1_minus(y)
```

```
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    tmp = x^3
    if _context isa Turing.DefaultContext
        @addlogprob!(log(-min(tmp, 0)))
    end
    return tmp
end

@model function gamma2(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^3
end

expt_prog = Expectation(
    gamma1_plus,
    gamma1_minus,
    gamma2
)
```

The type `Expectation` is simply used to have one common object which stores the three different Turing models. Notice that for `gamma2` the function body is identical to the original function.

For `gamma1_plus` and `gamma1_minus` we also have to check in what `_context` the model is executed in. Turing allows to execute the model with different contexts which change the model behaviour. For example, there is a `PriorContext` which essentially ignores the tilde statements which have observed data on the LHS. This is useful for evaluating the prior probability of some parameters. However, by default the `@addlogprob` macro ignores the model context. As a consequence if a Turing model includes an `@addlogprob` macro and is executed with a `PriorContext` then it no longer calculates the log prior probability but instead the log prior probability plus whatever value was added with the `@addlogprob` statement. Since we want to use the Turing model with Annealed Importance Sampling we need to be able to extract the prior from our model and hence we need to ensure that we do not call `@addlogprob` when executed in a `PriorContext`. This is what the added `if` clause ensures.

C.3 Different Estimators for Z_1^+, Z_1^- and Z_2

The target function $f(x) = x^2$ in the following expectation is always positive:

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x^2
end
```

Therefore, we already know that $Z_1^- = 0$, so it would be wasteful to spend computational resources on estimating Z_1^-. EPT allows users to specify the marginal likelihood estimator for each of the terms in TABI separately which means if the user knows that the target function is always positive they can specify that 0 samples should be used to estimate Z_1^-:

```
expct_estimate, diagnostics = estimate_expectation(
    expt_prog(2), TABI(
        TuringAlgorithm(AnIS(), num_samples=1000), # Z_1^+
        TuringAlgorithm(AnIS(), num_samples=0),    # Z_1^-
        TuringAlgorithm(AnIS(), num_samples=1000)  # Z_2
))
```

It is easy to see how this can be adapted to the case in which we have $Z_1^+ = 0$. This interface is not just useful for avoiding unnecessary computation, in some cases the user might also want to have different marginal likelihood estimators for each term. This allows user to further tailor the inference algorithm for the given target function $f(\theta)$.

C.4 Hyperparameters for Experiments

EPT runs standard annealed importance sampling twice: one time to estimate Z_1^+ and the other time to estimate Z_2. For each of the problems we always use the same hyperparameters for the annealed importance sampling algorithm both to run AnIS and for the two estimates in EPT. Running times for the individual scripts are given in the code supplementary.

C.4.1 Posterior Predictive

For the annealed importance sampling, we use a MH transition kernel with an isotropic Gaussian with covariance $0.5I$ as a proposal and 5 MH steps on each annealing distribution. We use 100 uniformly spaced annealing distributions. For the MCMC, we collect $5 \cdot 10^6$ samples in total. To parallelise sampling we run 500 chains with 10^4 samples each in parallel, discarding the first 10^3 samples as burn-in. We use a MH transition kernel with standard Normal proposal.

C.4.2 SIR Model

For the annealed importance sampling estimators we use HMC transition kernels with a step size of 0.05, 10 leapfrog steps and 10 MCMC steps on each annealing distribution. We use 100 geometrically spaced annealing distributions.

For the MCMC model we collect 10^6 samples in total with Turing's implementation of NUTS and a target acceptance rate of 65%.[1] We parallelise sampling over 10^2 chains with 10^4 samples and discard the first 10^3 samples as burn-in.

The ground truth is computed using importance sampling with 10^8 samples and the prior as a proposal distribution. See Equation (C.1) for the full SIR model including the priors. The observed data was generated from the model described in (C.1) with $\beta = 0.25$, $I_0 = 100$, $N = 10^4$ and $\phi = 10$ as the overdispersion parameter of the SIR model. We generate data for 15 time steps.

C.4.3 Radon model

We run EPT and AnIS with 200 intermediate distributions and one step of the dynamic HMC transition kernel [Betancourt, 2018, Hoffman and Gelman, 2014] on each intermediate distribution with a step size of 0.044. The step size was informed by running adaptive MCMC on the target distribution.

C.5 SIR Experiment

We assume we are given data in the form of observations y_i, the number of observed newly infected people on day i. Fixing $\gamma = 0.25$, this gives us the statistical model

$$\beta \sim \text{TruncatedNormal}(2, 1.5^2, [0, \infty]), \tag{C.1a}$$
$$I_0 \sim \text{TruncatedNormal}(100, 100^2, [0, 10000]), \tag{C.1b}$$
$$S_0 = 10000 - I_0, \tag{C.1c}$$
$$R_0 = 0, \tag{C.1d}$$
$$\mathbf{x} = \texttt{ODESolve}(\beta, \gamma, S_0, I_0, R_0), \tag{C.1e}$$
$$y_i \sim \text{NegativeBinomial}(\mu = x_i, \phi = 0.5). \tag{C.1f}$$

Here `ODESolve` indicates a call to a numerical ODE solver which solves the set of equations (6.19). It outputs x_i, the predicted number of newly infected people on day i. We assume the observation process is noisy and model it using a negative binomial distribution, which is parametrised by a mean μ and an overdispersion

[1] https://turing.ml/dev/docs/library/#Turing.Inference.NUTS

coefficient ϕ. For an in-depth discussion about doing Bayesian parameter inference in the SIR model we refer the reader to the case study of Grinsztajn et al. [2020].

We are further given a cost function in terms of R_0, $\text{cost}(R_0) = 10^{12} * \text{logistic}(10R_0 - 30)$. Intuitively, the cost initially increases exponentially with R_0. However, the total cost also saturates for very large R_0 (as the entire population becomes infected).

C.6 Hierarchical Radon Model

The data for this problem was taken from: https://github.com/pymc-devs/pymc-examples/blob/main/examples/data/radon.csv (the repository uses an MIT license; the data contains no personally identifiable information). The original data contains information about houses in 85 counties. In order to make estimating normalization constants more tractable we reduce the number of counties to 20.

Our target function is a function of predicted radon levels y_i for a typical house with a basement (i.e. $x_i = 0$) in county i; y_i is calculated using the predictive equation given in (6.22). We apply the function

$$f(y_i) = \frac{1}{1 + \exp(5(y_i - 4))}$$

to all the predicted radon levels and then take the product of all the f_i. Finally, to avoid floating point underflow we set a minimum value of 1e−200.

C.7 Multiple Expectations

The user is not restricted to defining only one expectation per model. By specifying multiple return values the user can specify multiple expectations. The `@exptectation` macro can recognise multiple return values and generates an expectation for each of them. The user can then estimate each expectation independently using `estimate_expectation`:

```
@expectation function expt_prog(y)
    x ~ Normal(0, 1)
    y ~ Normal(x, 1)
    return x, x^2, x^3
end
y_observed = 3
expt_prog1, expr_prog2, expt_prog3 = expt_prog
expct1 = expt_prog1(y_observed)
expct1_estimate, diagnostics = estimate_expectation(
  expct1,
  ↪  method=TABI(marginal_likelihood_estimator=TuringAlgorithm(
    AnIS(), num_samples=1000)))
```

C.8 Posterior Predictive Model in EPT

The expectation from Section 6.5.1 can be defined in just 5 lines of code with EPT:

```
@expectation function expt_prog(y)
    x ~ MvNormal(zeros(length(y)), I)        # x ~ 𝒩(x; 0, I)
    y ~ MvNormal(x, I)                       # y ~ 𝒩(y; x, I)
    return pdf(MvNormal(x, 0.5*I), -y)       # f(x) = 𝒩(−y; x, ½I)
end
```

C.9 Syntax Design

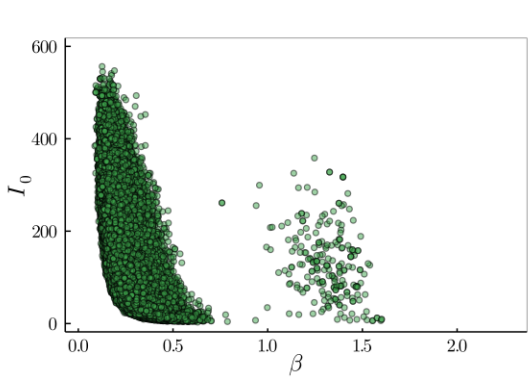
Prior works have considered two families of syntax design corresponding to the semantics required by EPT. Gordon et al. [2014] define the semantics for expectation computation via the syntax of probabilistic program's return expression, which is the approach we adopted in the design of EPT. Zinkov and Shan [2017] take a different route and define the expectation semantics via the use of syntax `expect(m, f)` where `m` is the program defining a measure and `f` is the target function.

While designing the interface of EPT we considered two different design for defining the target function: either letting users specify the target function implicitly through the return values of the function or allowing users to specify a target function `f` externally. The external function could then be passed to the `estimate_expectation` function explicitly.

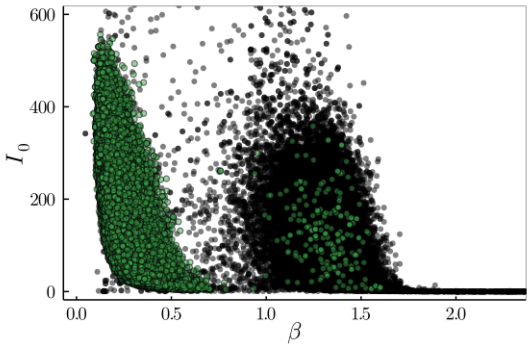
For EPT, we decided to adopt the former of the two designs mainly due to the simplicity of the resulting user interface and implementation. In particular, it allows for simple to execute program transformations of the `@expectation` macro into valid Turing programs to represent the individual densities, and thus the ability to use native Turing inference algorithms. Adopting the other approach would additionally require designing and specifying the interface between the function signature `f(.)` and the values of the named random draws performed by the model `m`. This would result in a more complex user-facing interface, at the slight advantage of improved compositionality of models and functions.

C.10 SIR Discussion

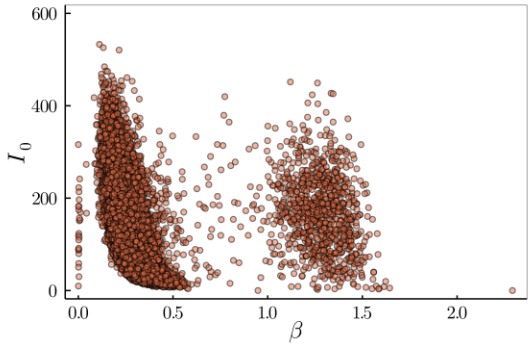
In the SIR experiment AnIS achieved a significantly lower RSE than MCMC even though both are non-target-aware. Figure C.1 shows samples from the different algorithms. The EPT samples for Z_1 visualise well in which regions of parameter space both the posterior and the target function have sufficient mass ($\beta \in [0.5, 2.0]$).



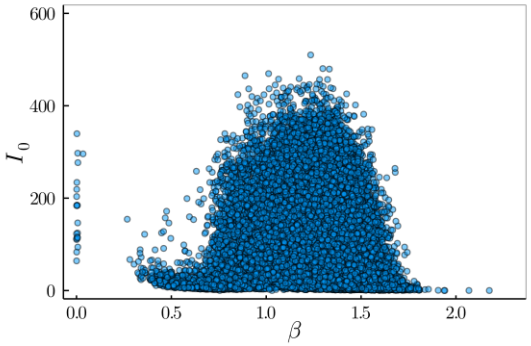
(a) MCMC samples.



(b) MCMC samples including burn-in samples (in black).



(c) AnIS samples.



(d) EPT samples for Z_1.

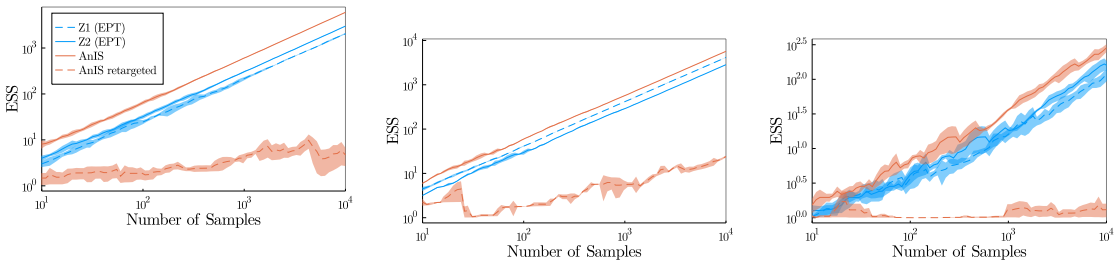
Figure C.1: Samples from the different algorithms for the SIR model. Note that for Figure C.1b some burn-in samples lie outside the boundaries of the plot but we adjusted the axis limits so that they are the same for all plots to allow for easier comparison.

The samples from AnIS and MCMC suggest that most of the posterior mass is located in the interval $\beta \in [0.3, 0.7]$. However, AnIS also generates a significant amount of samples in the parameter region $\beta \in [1.0, 1.5]$. The samples in this second "mode" are directly in the region of the target-aware samples. Further, the plots suggest that AnIS generates more samples in this regions than MCMC which is what allows AnIS to achieve a lower RSE. However, it seems that the AnIS represents the second "mode" disproportionally. Specifically looking at the burn-in samples from MCMC in Figure C.1b shows that MCMC will converge to the parameter space in $\beta \in [0.3, 0.7]$ even if the initial parameter samples are around $\beta \in [1.0, 1.5]$. This indicates that this is not a failure of MCMC to detect another mode but rather that there is negligible posterior mass in that parameter region. Therefore the better performance of AnIS compared to MCMC seems to occur mostly because AnIS got lucky by accidentally generating samples in the right parameter region.

C.10.1 A Note on MCMC ESS

The SIR experiment provides a good example of how the MCMC ESS [Vehtari et al., 2020] is unreliable for our use case. As detailed in Section C.4.2 for MCMC we run 100 chains with 10,000 samples each. This is replicated 5 times to get estimates on the variability in behaviour. After discarding the burn-in samples for each chain the 5 replications give us the following final ESS estimates: [631, 360; 805, 868; 873, 269; 665, 683; 5, 114]. We observe that all but one replication give disproportionally high ESS estimates. We found that the replication which gives a more conservative ESS estimate of 5,114 is the replication which generated samples in the parameter region $\beta \in [1.0, 1.5]$ (see Figure C.1a). More importantly, the MCMC ESS estimates do not seem to show any correlation with the RSE values (see Figure 6.4) which is the more important metric because it directly measures the error in our estimate. Therefore, we decided against using the MCMC ESS in our evaluation because it can give the impression that MCMC is performing well when it is actually failing dramatically (in terms of RSE).

C.11 Effective Sample Size



(a) Gaussian Posterior Predictive.

(b) SIR.

(c) Radon.

Figure C.2: Individual ESS values as defined in Section 5.6 for the three different experiments. Instead of taking $\min(\text{ESS}_{Z_1}, \text{ESS}_{Z_2})$ for EPT and $\min(\text{ESS}_{\text{AnIS retargeted}}, \text{ESS}_{\text{AnIS}})$ for AnIS we plot each value individually. Plotting each ESS value separately shows that the performance of AnIS is severely limited by its ability to generate samples in regions in which the target function $f(\theta)$ is large. This is indicated by the low values for $\text{ESS}_{\text{AnIS retargeted}}$.

C.12 Positive and Negative Target Functions

To demonstrate that EPT is also beneficial for target functions which are positive and negative we provide a brief description of a synthetic experiment. We assume the following model which gives us a banana shaped density (see Figure C.3):

```
@expectation function banana()
    x1 ~ Normal(0, 4)
    x2 ~ Normal(0, 4)
    @addlogprob!(banana_density(x1, x2))
    return banana_f(x1, x2)
end

banana_density(x1, x2) = -0.5*(0.03*x1^2+(x2/2+0.03*(x1^2-100))^2)
```

Note that there is no observed data in this experiment which is why we chose to express the banana distribution as an unnormalized density (i.e. use the `@addlogprob!` primitive). Our target function is given by

```
function banana_f(x1, x2)
    cond = 1 / (1 + exp(50 * (x2 + 5)))
    return cond * (x1 - 2)^3
end
```

Note that the target function can be positive and negative. Figure C.3 shows the RSE for EPT and AnIS. We used an MH transition kernel and 200 intermediate potentials for the Annealed Importance Sampling estimators. The RSE of AnIS does not improve because it fails to generate samples in the regions in which the target $f(\theta)$ is large. Rainforth et al. [2020] provide a comparison to MCMC on a similar problem so we omit it here.

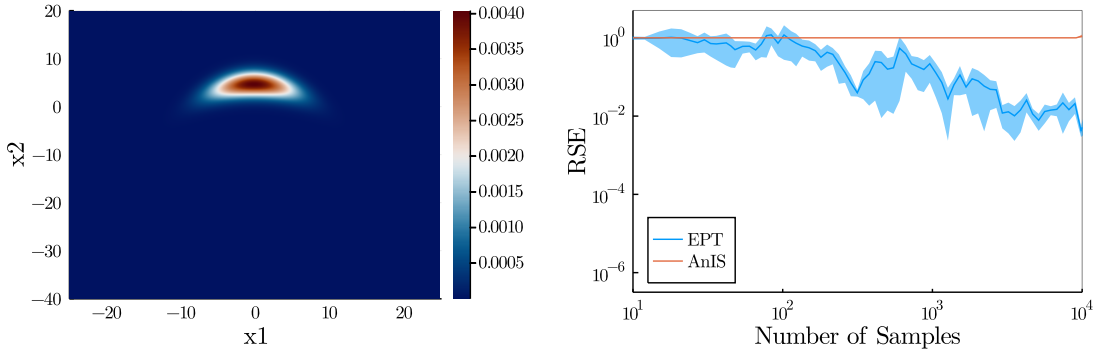


Figure C.3: Banana experiment. [Left] Heatmap of the density of the model. [Right] Relative Squared Error for EPT and AnIS.

Bibliography

Karl Popper. *The logic of scientific discovery*. Hutchinson, 1959.

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

George EP Box. Sampling and bayes' inference in scientific modelling and robustness. *Journal of the Royal Statistical Society Series A: Statistics in Society*, 143(4): 383–404, 1980.

Irving John Good. *Good thinking: The foundations of probability and its applications*. U of Minnesota Press, 1983.

Donald B Rubin. Bayesianly justifiable and relevant frequency calculations for the applied statistician. *The Annals of Statistics*, pages 1151–1172, 1984.

Deborah G Mayo. *Error and the growth of experimental knowledge*. University of Chicago Press, 1996.

Edwin T Jaynes. *Probability theory: The logic of science*. Cambridge university press, 2003.

Andrew Gelman and Cosma Rohilla Shalizi. Philosophy and the practice of Bayesian statistics: *Philosophy and the practice of Bayesian statistics*. *British Journal of Mathematical and Statistical Psychology*, 66(1):8–38, February 2013. ISSN 00071102. doi: 10.1111/j.2044-8317.2011.02037.x. URL http://doi.wiley.com/10.1111/j.2044-8317.2011.02037.x.

David M Blei. Build, compute, critique, repeat: Data analysis with latent variable models. *Annual Review of Statistics and Its Application*, 1:203–232, 2014.

Karl Friston. The free-energy principle: a unified brain theory? *Nature reviews neuroscience*, 11(2):127–138, 2010.

Joshua B Tenenbaum, Charles Kemp, Thomas L Griffiths, and Noah D Goodman. How to grow a mind: Statistics, structure, and abstraction. *science*, 331(6022): 1279–1285, 2011.

Brenden M Lake, Tomer D Ullman, Joshua B Tenenbaum, and Samuel J Gershman. Building machines that learn and think like people. *Behavioral and brain sciences*, 40:e253, 2017.

Eyke Hüllermeier and Willem Waegeman. Aleatoric and epistemic uncertainty in machine learning: An introduction to concepts and methods. *Machine learning*, 110(3):457–506, 2021.

Andrew Gelman, John B Carlin, Hal S Stern, David B Dunson, Aki Vehtari, and Donald B Rubin. *Bayesian Data Analysis*. CRC press, 2013.

George Casella and Roger Berger. *Statistical Inference*. Duxbury Resource Center, June 2001. ISBN 0534243126.

Christian P Robert et al. *The Bayesian choice: from decision-theoretic foundations to computational implementation*, volume 2. Springer, 2007.

Steve Brooks, Andrew Gelman, Galin Jones, and Xiao-Li Meng. *Handbook of Markov Chain Monte Carlo*. CRC Press, May 2011. ISBN 978-1-4200-7942-5.

David M. Blei, Alp Kucukelbir, and Jon D. McAuliffe. Variational Inference: A Review for Statisticians. *Journal of the American Statistical Association*, July 2017. ISSN 0162-1459. URL https://www.tandfonline.com/doi/full/10.1080/01621459.2017.1285773.

Andrew D. Gordon, Thomas A. Henzinger, Aditya V. Nori, and Sriram K. Rajamani. Probabilistic programming. In *Future of Software Engineering Proceedings*, FOSE 2014, pages 167–181, New York, NY, USA, May 2014. Association for Computing Machinery. ISBN 978-1-4503-2865-4. doi: 10.1145/2593882.2593900. URL https://doi.org/10.1145/2593882.2593900.

Jan-Willem van de Meent, Brooks Paige, Hongseok Yang, and Frank Wood. An Introduction to Probabilistic Programming. *arXiv:1809.10756 [cs, stat]*, September 2018.

Gilles Barthe, Joost-Pieter Katoen, and Alexandra Silva. *Foundations of probabilistic programming*. Cambridge University Press, 2020.

David J Lunn, Andrew Thomas, Nicky Best, and David Spiegelhalter. Winbugs-a bayesian modelling framework: concepts, structure, and extensibility. *Statistics and computing*, 10:325–337, 2000.

David Spiegelhalter, Andrew Thomas, Nicky Best, and Dave Lunn. Openbugs user manual. *Version*, 3(2):2007, 2007.

Daphne Koller and Nir Friedman. *Probabilistic graphical models: principles and techniques*. MIT Press, 2009.

Stuart Geman and Donald Geman. Stochastic relaxation, gibbs distributions, and the bayesian restoration of images. *IEEE Transactions on pattern analysis and machine intelligence*, pages 721–741, 1984.

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

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A Probabilistic Programming Language. *Journal of Statistical Software*, 76:1–32, January 2017. ISSN 1548-7660. doi: 10.18637/jss.v076.i01. URL https://doi.org/10.18637/jss.v076.i01.

Radford M. Neal. MCMC using Hamiltonian dynamics. In *Handbook of Markov Chain Monte Carlo*, pages 113–162. Chapman & Hall / CRC Press, 2011.

Michael Betancourt. A Conceptual Introduction to Hamiltonian Monte Carlo. *arXiv:1701.02434 [stat]*, July 2018. URL http://arxiv.org/abs/1701.02434.

Seth Flaxman, Swapnil Mishra, Axel Gandy, H Juliette T Unwin, Thomas A Mellan, Helen Coupland, Charles Whittaker, Harrison Zhu, Tresnia Berah, Jeffrey W Eaton, et al. Estimating the effects of non-pharmaceutical interventions on covid-19 in europe. *Nature*, 584(7820):257–261, 2020.

Ralf Herbrich, Tom Minka, and Thore Graepel. Trueskill: a bayesian skill rating system. *Advances in neural information processing systems*, 19, 2006.

Merlin Heidemanns, Andrew Gelman, and G. Elliott Morris. An Updated Dynamic Bayesian Forecasting Model for the US Presidential Election. *Harvard Data Science Review*, 2(4), oct 27 2020. https://hdsr.mitpress.mit.edu/pub/nw1dzd02.

Thomas William Gamlen Rainforth. *Automating Inference, Learning, and Design Using Probabilistic Programming*. http://purl.org/dc/dcmitype/Text, University of Oxford, 2017. URL https://ora.ox.ac.uk/objects/uuid:e276f3b4-ff1d-44bf-9d67-013f68ce81f0.

Sam Staton, Frank Wood, Hongseok Yang, Chris Heunen, and Ohad Kammar. Semantics for Probabilistic Programming: Higher-Order Functions, Continuous Distributions, and Soft Constraints. In *2016 31st annual acm/ieee symposium on logic in computer science (lics)*, pages 1–10. IEEE, 2016.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, UAI'08, pages 220–229, Arlington, Virginia, USA, July 2008. AUAI Press. ISBN 978-0-9749039-4-1.

Frank Wood, Jan Willem Meent, and Vikash Mansinghka. A New Approach to Probabilistic Programming Inference. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 1024–1032. PMLR, April 2014.

Noah D Goodman and Andreas Stuhlmüller. WebPPL - probabilistic programming for the web, 2014. URL http://webppl.org/.

Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 20(28):1–6, 2019. ISSN 1533-7928. URL http://jmlr.org/papers/v20/18-403.html.

Marco F. Cusumano-Towner, Feras A. Saad, Alexander K. Lew, and Vikash K. Mansinghka. Gen: A general-purpose probabilistic programming system with programmable inference. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 221–236, New York, NY, USA, June 2019. Association for Computing Machinery. ISBN 978-1-4503-6712-7. doi: 10.1145/3314221.3314642. URL https://doi.org/10.1145/3314221.3314642.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: A Language for Flexible Probabilistic Inference. In *International Conference on Artificial Intelligence and Statistics*, pages 1682–1690. PMLR, March 2018. URL http://proceedings.mlr.press/v84/ge18b.html.

Atilim Güneş Baydin, Lei Shao, Wahid Bhimji, Lukas Heinrich, Lawrence Meadows, Jialin Liu, Andreas Munk, Saeid Naderiparizi, Bradley Gram-Hansen, Gilles Louppe, Mingfei Ma, Xiaohui Zhao, Philip Torr, Victor Lee, Kyle Cranmer, Prabhat, and Frank Wood. Etalumis: Bringing probabilistic programming to scientific simulators at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pages 1–24, New York, NY, USA, November 2019. Association for Computing Machinery. ISBN 978-1-4503-6229-0. doi: 10.1145/3295500.3356180. URL https://doi.org/10.1145/3295500.3356180.

Atılım Güneş Baydin, Lukas Heinrich, Wahid Bhimji, Lei Shao, Saeid Naderiparizi, Andreas Munk, Jialin Liu, Bradley Gram-Hansen, Gilles Louppe, Lawrence Meadows, Philip Torr, Victor Lee, Prabhat, Kyle Cranmer, and Frank Wood. Efficient Probabilistic Inference in the Quest for Physics Beyond the Standard Model. *arXiv:1807.07706 [hep-ph, physics:physics, stat]*, February 2020. URL http://arxiv.org/abs/1807.07706.

Feras A. Saad, Marco F. Cusumano-Towner, Ulrich Schaechtle, Martin C. Rinard, and Vikash K. Mansinghka. Bayesian synthesis of probabilistic programs for automatic data modeling. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–32, January 2019. ISSN 24751421. doi: 10.1145/3290350. URL http://dl.acm.org/citation.cfm?doid=3302515.3290350.

Fredrik Ronquist, Jan Kudlicka, Viktor Senderov, Johannes Borgström, Nicolas Lartillot, Daniel Lundén, Lawrence Murray, Thomas B. Schön, and David Broman. Universal probabilistic programming offers a powerful approach to statistical phylogenetics. *bioRxiv*, page 2020.06.16.154443, December 2020. doi: 10.1101/2020.06.16.154443. URL https://www.biorxiv.org/content/10.1101/2020.06.16.154443v4.

Tan Zhi-Xuan, McCoy R Becker, and Vikash K Mansinghka. Genify. jl: Transforming julia into gen to enable programmable inference. In *Languages For Inference (LAFI) Workshop, 48th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2021.

Andrew Gelman, Aki Vehtari, Daniel Simpson, Charles C. Margossian, Bob Carpenter, Yuling Yao, Lauren Kennedy, Jonah Gabry, Paul-Christian Bürkner, and Martin Modrák. Bayesian Workflow. *arXiv:2011.01808 [stat]*, November 2020. URL http://arxiv.org/abs/2011.01808.

Andrew Gelman and Xiao-Li Meng. Simulating Normalizing Constants: From Importance Sampling to Bridge Sampling to Path Sampling. *Statistical science*, pages 163–185, 1998.

Simon Lacoste-Julien, Ferenc Huszár, and Zoubin Ghahramani. Approximate Inference for the Loss-Calibrated Bayesian. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 416–424, 2011.

Art B. Owen. *Monte Carlo Theory, Methods and Examples.* https://artowen.su.domains/mc/, 2013.

Adam Goliński, Frank Wood, and Tom Rainforth. Amortized Monte Carlo Integration. *arXiv:1907.08082 [cs, stat]*, July 2019. URL http://arxiv.org/abs/1907.08082.

Tom Rainforth, Adam Golinski, Frank Wood, and Sheheryar Zaidi. Target–Aware Bayesian Inference: How to Beat Optimal Conventional Estimators. *Journal of Machine Learning Research*, 21(88):1–54, 2020. URL http://jmlr.org/papers/v21/19-102.html.

Timothy Brooks Paige. *Automatic Inference for Higher-Order Probabilistic Programs.* http://purl.org/dc/dcmitype/Text, University of Oxford, 2016. URL https://ora.ox.ac.uk/objects/uuid:d912c4de-4b08-4729-aa19-766413735e2a.

David Wingate and Theophane Weber. Automated Variational Inference in Probabilistic Programming. *arXiv:1301.1299 [cs, stat]*, January 2013. URL http://arxiv.org/abs/1301.1299.

Jennifer A Hoeting, David Madigan, Adrian E Raftery, and Chris T Volinsky. Bayesian model averaging: a tutorial (with comments by m. clyde, david draper and ei george, and a rejoinder by the authors). *Statistical science*, 14(4):382–417, 1999.

Christopher M. Bishop. *Pattern Recognition and Machine Learning.* Springer-Verlag, Berlin, Heidelberg, 2006. ISBN 0387310738.

Richard McElreath. *Statistical rethinking: A Bayesian course with examples in R and Stan.* Chapman and Hall/CRC, 2018.

Kevin P. Murphy. *Probabilistic Machine Learning: An introduction.* MIT Press, 2022. URL probml.ai.

Howard Raiffa and Robert Schlaifer. *Applied statistical decision theory*, volume 78. John Wiley & Sons, 2000.

Christian Robert and George Casella. *Monte Carlo Statistical Methods*. Springer Texts in Statistics. Springer-Verlag, New York, second edition, 2004. ISBN 978-0-387-21239-5. doi: 10.1007/978-1-4757-4145-2. URL https://www.springer.com/gp/book/9780387212395.

Joris Bierkens, Paul Fearnhead, and Gareth O. Roberts. The zig-zag process and super-efficient sampling for bayesian analysis of big data. *The Annals of Statistics*, 2016. URL https://api.semanticscholar.org/CorpusID:51848020.

Alexandre Bouchard-Côté, Sebastian J Vollmer, and Arnaud Doucet. The bouncy particle sampler: A nonreversible rejection-free markov chain monte carlo method. *Journal of the American Statistical Association*, 113(522):855–867, 2018.

Christophe Andrieu and Samuel Livingstone. Peskun-tierney ordering for markov chain and process monte carlo: beyond the reversible scenario. *arXiv preprint arXiv:1906.06197*, 2019.

Saifuddin Syed, Alexandre Bouchard-Côté, George Deligiannidis, and Arnaud Doucet. Non-reversible parallel tempering: a scalable highly parallel MCMC scheme. *Journal of the Royal Statistical Society (Series B)*, 84:321–350, 2021.

Joris Bierkens, Sebastiano Grazzi, Kengo Kamatani, and Gareth Roberts. The boomerang sampler. In *International conference on machine learning*, pages 908–918. PMLR, 2020.

Nicholas Metropolis, Arianna W Rosenbluth, Marshall N Rosenbluth, Augusta H Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The journal of chemical physics*, 21(6):1087–1092, 1953.

W. K. Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970. ISSN 00063444. URL http://www.jstor.org/stable/2334940.

James E Gubernatis. Marshall rosenbluth and the metropolis algorithm. *Physics of plasmas*, 12(5), 2005.

Michael I Jordan, Zoubin Ghahramani, Tommi S Jaakkola, and Lawrence K Saul. An introduction to variational methods for graphical models. *Machine learning*, 37:183–233, 1999.

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

Thomas P. Minka. Expectation propagation for approximate bayesian inference. In *Proceedings of the Seventeenth Conference on Uncertainty in Artificial Intelligence*, UAI'01, page 362–369, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc. ISBN 1558608001.

Jörg Bornschein and Yoshua Bengio. Reweighted wake-sleep. *International Conference on Learning Representations*, 2015.

Christian Naesseth, Fredrik Lindsten, and David Blei. Markovian score climbing: Variational inference with kl(p || q). In H. Larochelle, M. Ranzato, R. Hadsell, M.F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 15499–15510. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/b20706935de35bbe643733f856d9e5d6-Paper.pdf.

Abhinav Agrawal, Daniel R Sheldon, and Justin Domke. Advances in black-box VI: Normalizing flows, importance weighting, and optimization. In H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, editors, *Advances in Neural Information Processing Systems*, volume 33, pages 17358–17369. Curran Associates, Inc., 2020. URL https://proceedings.neurips.cc/paper/2020/file/c91e3483cf4f90057d02aa492d2b25b1-Paper.pdf.

Tommi S Jaakkola and Michael I Jordan. A variational approach to bayesian logistic regression models and their extensions. In *Sixth International Workshop on Artificial Intelligence and Statistics*, pages 283–294. PMLR, 1997.

Zoubin Ghahramani and Matthew Beal. Propagation algorithms for variational bayesian learning. *Advances in neural information processing systems*, 13, 2000.

David Blei and John Lafferty. Correlated topic models. *Advances in neural information processing systems*, 18:147, 2006.

Michael Braun and Jon McAuliffe. Variational inference for large-scale models of discrete choice. *Journal of the American Statistical Association*, 105(489):324–335, 2010.

David Knowles and Tom Minka. Non-conjugate variational message passing for multinomial and binary regression. *Advances in Neural Information Processing Systems*, 24, 2011.

Herbert Robbins and Sutton Monro. A stochastic approximation method. *The annals of mathematical statistics*, pages 400–407, 1951.

Søren Asmussen and Peter W Glynn. *Stochastic simulation: algorithms and analysis*, volume 57. Springer, 2007.

John Paisley, David Blei, and Michael Jordan. Variational bayesian inference with stochastic search. *arXiv preprint arXiv:1206.6430*, 2012.

David J Nott, Siew Li Tan, Mattias Villani, and Robert Kohn. Regression density estimation with variational methods and stochastic approximation. *Journal of Computational and Graphical Statistics*, 21(3):797–820, 2012.

Tim Salimans and David A Knowles. Fixed-form variational posterior approximation through stochastic linear regression. 2013.

Diederik P. Kingma and Max Welling. Auto-Encoding Variational Bayes. *arXiv:1312.6114 [cs, stat]*, May 2014. URL http://arxiv.org/abs/1312.6114.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic Backpropagation and Approximate Inference in Deep Generative Models. *arXiv:1401.4082 [cs, stat]*, May 2014. URL http://arxiv.org/abs/1401.4082.

Rajesh Ranganath, Sean Gerrish, and David Blei. Black Box Variational Inference. In *Proceedings of the Seventeenth International Conference on Artificial Intelligence and Statistics*, pages 814–822. PMLR, April 2014. URL https://proceedings.mlr.press/v33/ranganath14.html.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. Monte Carlo Gradient Estimation in Machine Learning. *Journal of Machine Learning Research*, 21(132):1–62, 2020. ISSN 1533-7928. URL http://jmlr.org/papers/v21/19-346.html.

Jack P. C. Kleijnen and Reuven Y. Rubinstein. Optimization and sensitivity analysis of computer simulation models by the score function method. *European Journal of Operational Research*, 88(3):413–427, February 1996. ISSN 0377-2217. doi: 10.1016/0377-2217(95)00107-7. URL https://www.sciencedirect.com/science/article/pii/0377221795001077.

Ronald J. Williams. Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning. *Machine Language*, 8(3-4):229–256, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992696. URL https://doi.org/10.1007/BF00992696.

Andriy Mnih and Karol Gregor. Neural Variational Inference and Learning in Belief Networks. In *International Conference on Machine Learning*, pages 1791–1799. PMLR, June 2014. URL http://proceedings.mlr.press/v32/mnih14.html.

Michalis K. Titsias and Miguel Lázaro-Gredilla. Local expectation gradients for black box variational inference. *Advances in neural information processing systems*, 28, 2015.

P. Glasserman. *Monte Carlo Methods in Financial Engineering*. Applications of mathematics : stochastic modelling and applied probability. Springer, 2004. ISBN 9780387004518. URL https://books.google.co.uk/books?id=e9GWUsQkPNMC.

Ming Xu, Matias Quiroz, Robert Kohn, and Scott A Sisson. Variance reduction properties of the reparameterization trick. In *The 22nd international conference on artificial intelligence and statistics*, pages 2711–2720. PMLR, 2019.

Atilim Gunes Baydin, Barak A Pearlmutter, Alexey Andreyevich Radul, and Jeffrey Mark Siskind. Automatic differentiation in machine learning: a survey. *Journal of Marchine Learning Research*, 18:1–43, 2018.

Wonyeol Lee, Hangyeol Yu, and Hongseok Yang. Reparameterization Gradient for Non-differentiable Models. In *Advances in Neural Information Processing Systems*, 2018. URL https://papers.nips.cc/paper/2018/hash/b096577e264d1ebd6b41041f392eec23-Abstract.html.

Mikhail Figurnov, Shakir Mohamed, and Andriy Mnih. Implicit reparameterization gradients. *Advances in neural information processing systems*, 31, 2018.

Martin Jankowiak and Fritz Obermeyer. Pathwise derivatives beyond the reparameterization trick. In *International conference on machine learning*, pages 2235–2244. PMLR, 2018.

Alp Kucukelbir, Dustin Tran, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic differentiation variational inference. *J. Mach. Learn. Res.*, 18(1), Jan 2017.

George Papamakarios, Eric Nalisnick, Danilo Jimenez Rezende, Shakir Mohamed, and Balaji Lakshminarayanan. Normalizing Flows for Probabilistic Modeling and Inference. *Journal of Machine Learning Research*, 22(57):1–64, 2021. ISSN 1533-7928. URL http://jmlr.org/papers/v22/19-1028.html.

Matthew D Hoffman, David M. Blei, Chong Wang, and John Paisley. Stochastic Variational Inference. *Journal of Machine Learning Research*, 14(5), May 2013.

Michalis Titsias and Miguel Lázaro-Gredilla. Doubly Stochastic Variational Bayes for non-Conjugate Inference. In *Proceedings of the 31st International Conference on Machine Learning*, pages 1971–1979. PMLR, June 2014.

Danilo Rezende and Shakir Mohamed. Variational Inference with Normalizing Flows. In *Proceedings of the 32nd International Conference on Machine Learning*, pages 1530–1538. PMLR, June 2015. URL https://proceedings.mlr.press/v37/rezende15.html.

Warren Morningstar, Sharad Vikram, Cusuh Ham, Andrew Gallagher, and Joshua Dillon. Automatic differentiation variational inference with mixtures. In *International Conference on Artificial Intelligence and Statistics*, pages 3250–3258. PMLR, 2021.

Luca Ambrogioni, Kate Lin, Emily Fertig, Sharad Vikram, Max Hinne, Dave Moore, and Marcel van Gerven. Automatic structured variational inference. In *Proceedings of The 24th International Conference on Artificial Intelligence and Statistics*, pages 676–684. PMLR, March 2021. URL https://proceedings.mlr.press/v130/ambrogioni21a.html.

Joohwan Ko, Kyurae Kim, Woo Chang Kim, and Jacob R Gardner. Provably scalable black-box variational inference with structured variational families. *arXiv preprint arXiv:2401.10989*, 2024.

Leon Klein, Andreas Krämer, and Frank Noé. Equivariant flow matching. *Advances in Neural Information Processing Systems*, 36, 2024.

John Schulman, Nicolas Heess, Theophane Weber, and Pieter Abbeel. Gradient estimation using stochastic computation graphs. *Advances in neural information processing systems*, 28, 2015.

Chris J Maddison, Andriy Mnih, and Yee Whye Teh. The concrete distribution: A continuous relaxation of discrete random variables. *arXiv preprint arXiv:1611.00712*, 2016.

Eric Jang, Shixiang Gu, and Ben Poole. Categorical reparameterization with gumbel-softmax. *arXiv preprint arXiv:1611.01144*, 2016.

Chris J Oates, Mark Girolami, and Nicolas Chopin. Control Functionals for Monte Carlo Integration. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 3(79):695–718, 2017.

Geoffrey Roeder, Yuhuai Wu, and David K Duvenaud. Sticking the landing: Simple, lower-variance gradient estimators for variational inference. *Advances in Neural Information Processing Systems*, 30, 2017.

Jiaxin Shi, Yuhao Zhou, Jessica Hwang, Michalis Titsias, and Lester Mackey. Gradient estimation with discrete stein operators. *Advances in neural information processing systems*, 35:25829–25841, 2022.

Yingzhen Li and Richard E Turner. Rényi divergence variational inference. *Advances in neural information processing systems*, 29, 2016.

Robert Bamler, Cheng Zhang, Manfred Opper, and Stephan Mandt. Perturbative black box variational inference. *Advances in Neural Information Processing Systems*, 30, 2017.

Romain Lopez, Pierre Boyeau, Nir Yosef, Michael Jordan, and Jeffrey Regier. Decision-making with auto-encoding variational bayes. *Advances in Neural Information Processing Systems*, 33:5081–5092, 2020.

Tomas Geffner and Justin Domke. On the difficulty of unbiased alpha divergence minimization. *arXiv preprint arXiv:2010.09541*, 2020.

Manushi Welandawe, Michael Riis Andersen, Aki Vehtari, and Jonathan H Huggins. Robust, automated, and accurate black-box variational inference. *arXiv preprint arXiv:2203.15945*, 2022.

Mingxuan Yi and Song Liu. Sliced wasserstein variational inference. In *Asian Conference on Machine Learning*, pages 1213–1228. PMLR, 2023.

Ryan Giordano, Martin Ingram, and Tamara Broderick. Black box variational inference with a deterministic objective: Faster, more accurate, and even more black box. *Journal of Machine Learning Research*, 25(18):1–39, 2024.

Cheng Zhang, Judith Bütepage, Hedvig Kjellström, and Stephan Mandt. Advances in variational inference. *IEEE transactions on pattern analysis and machine intelligence*, 41(8):2008–2026, 2018.

Akash Kumar Dhaka, Alejandro Catalina, Manushi Welandawe, Michael Riis Andersen, Jonathan Huggins, and Aki Vehtari. Challenges and Opportunities in High-dimensional Variational Inference. *arXiv:2103.01085 null*, March 2021. URL http://arxiv.org/abs/2103.01085.

Richard Von Mises. *Probability, statistics, and truth*. Courier Corporation, 1981.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

David JC MacKay. Bayesian neural networks and density networks. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 354(1):73–80, 1995.

Edwin Fong and Chris Holmes. On the marginal likelihood and cross-validation. *arXiv:1905.08737 [stat]*, September 2019. URL http://arxiv.org/abs/1905.08737.

Andrew Gelman and Yuling Yao. Holes in Bayesian statistics *. *Journal of Physics G: Nuclear and Particle Physics*, 48(1):014002, December 2020. ISSN 0954-3899. doi: 10.1088/1361-6471/abc3a5. URL https://iopscience.iop.org/article/10.1088/1361-6471/abc3a5/meta.

Wessel Bruinsma, Andrew Y. K. Foong, and Richard E. Turner. What keeps a bayesian up at night? part 1: Day time, 2021. URL https://mlg.eng.cam.ac.uk/blog/2021/03/31/what-keeps-a-bayesian-awake-at-night-part-1.html.

Richard T Cox. Probability, frequency and reasonable expectation. *American journal of physics*, 14(1):1–13, 1946.

Leonard J Savage. *The foundations of statistics*. Courier Corporation, 1972.

Aad W Van der Vaart. *Asymptotic statistics*, volume 3. Cambridge university press, 2000.

James Aitchison. Goodness of prediction fit. *Biometrika*, 62(3):547–554, 1975.

Veronika Eyring, Sandrine Bony, Gerald A Meehl, Catherine A Senior, Bjorn Stevens, Ronald J Stouffer, and Karl E Taylor. Overview of the coupled model intercomparison project phase 6 (cmip6) experimental design and organization. *Geoscientific Model Development*, 9(5):1937–1958, 2016.

Joost Schymkowitz, Jesper Borg, Francois Stricher, Robby Nys, Frederic Rousseau, and Luis Serrano. The foldx web server: an online force field. *Nucleic acids research*, 33(suppl_2):W382–W388, 2005.

Yuan Zhou. *Automating inference for non–standard models*. PhD thesis, University of Oxford, 2020.

Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. Problog: A probabilistic prolog and its application in link discovery. In *IJCAI 2007, Proceedings of the 20th international joint conference on artificial intelligence*, pages 2462–2467. IJCAI-INT JOINT CONF ARTIF INTELL, 2007.

Luc De Raedt and Kristian Kersting. Probabilistic inductive logic programming. In *Probabilistic inductive logic programming: theory and applications*, pages 1–27. Springer, 2008.

Daan Fierens, Guy Van den Broeck, Joris Renkens, Dimitar Shterionov, Bernd Gutmann, Ingo Thon, Gerda Janssens, and Luc De Raedt. Inference and learning in probabilistic logic programs using weighted boolean formulas. *Theory and Practice of Logic Programming*, 15(3):358–401, 2015. doi: 10.1017/S1471068414000076.

Robin Manhaeve, Sebastijan Dumancic, Angelika Kimmig, Thomas Demeester, and Luc De Raedt. Deepproblog: Neural probabilistic logic programming. *Advances in neural information processing systems*, 31, 2018.

John Salvatier, Thomas V Wiecki, and Christopher Fonnesbeck. Probabilistic Programming in Python Using PyMC3. *PeerJ Computer Science*, 2:e55, 2016.

Dustin Tran, Alp Kucukelbir, Adji B Dieng, Maja Rudolph, Dawen Liang, and David M Blei. Edward: A Library for Probabilistic Modeling, Inference, and Criticism. *arXiv preprint arXiv:1610.09787*, 2016.

N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. Learning disentangled representations with semi-supervised deep generative models. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 5927–5937. Curran Associates, Inc., 2017.

Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Dexter Kozen. Semantics of probabilistic programs. In *20th Annual Symposium on Foundations of Computer Science (Sfcs 1979)*, pages 101–114, October 1979. doi: 10.1109/SFCS.1979.38.

Johannes Borgström, Ugo Dal Lago, Andrew D. Gordon, and Marcin Szymczak. A lambda-calculus foundation for universal probabilistic programming. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 33–46, New York, NY, USA, September 2016. Association for Computing Machinery. ISBN 978-1-4503-4219-3. doi: 10.1145/2951913.2951942. URL https://doi.org/10.1145/2951913.2951942.

Alexander K. Lew, Marco F. Cusumano-Towner, Benjamin Sherman, Michael Carbin, and Vikash K. Mansinghka. Trace types and denotational semantics for sound programmable inference in probabilistic languages. *Proceedings of the ACM on Programming Languages*, 4(POPL):19:1–19:32, December 2019. doi: 10.1145/3371087. URL https://doi.org/10.1145/3371087.

Carol Mak, C.-H. Luke Ong, Hugo Paquet, and Dominik Wagner. Densities of Almost Surely Terminating Probabilistic Programs are Differentiable Almost Everywhere. In Nobuko Yoshida, editor, *Programming Languages and Systems*, Lecture Notes in Computer Science, pages 432–461, Cham, 2021a. Springer International Publishing. ISBN 978-3-030-72019-3. doi: 10.1007/978-3-030-72019-3_16.

Carol Mak, Fabian Zaiser, and Luke Ong. Nonparametric Hamiltonian Monte Carlo. In *Proceedings of the 38th International Conference on Machine Learning*, pages 7336–7347. PMLR, July 2021b. URL https://proceedings.mlr.press/v139/mak21a.html.

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and Implementation of Probabilistic Programming Language Anglican. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages - IFL 2016*, pages 1–12, Leuven, Belgium, 2016. ACM Press. ISBN 978-1-4503-4767-9. doi: 10.1145/3064899.3064910. URL http://dl.acm.org/citation.cfm?doid=3064899.3064910.

Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: A higher-order probabilistic programming platform with programmable inference. *arXiv:1404.0099 [cs, stat]*, March 2014. URL http://arxiv.org/abs/1404.0099.

Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 2018. URL https://www.sciencedirect.com/science/article/pii/S1367578818301202.

Yuan Zhou, Hongseok Yang, Yee Whye Teh, and Tom Rainforth. Divide, Conquer, and Combine: A New Inference Strategy for Probabilistic Programs with Stochastic Support. In *Proceedings of the 37th International Conference on Machine Learning*, pages 11534–11545. PMLR, November 2020. URL https://proceedings.mlr.press/v119/zhou20e.html.

Yuan Zhou, Bradley J Gram-Hansen, Tobias Kohn, Tom Rainforth, Hongseok Yang, and Frank Wood. Lf-ppl: A low-level first order probabilistic programming language for non-differentiable models. In *The 22nd International Conference on Artificial Intelligence and Statistics*, pages 148–157. PMLR, 2019.

Gordon Plotkin and John Power. Adequacy for algebraic effects. In *International Conference on Foundations of Software Science and Computation Structures*, pages 1–24. Springer, 2001.

Gordon Plotkin and Matija Pretnar. Handlers of algebraic effects. In *European Symposium on Programming*, pages 80–94. Springer, 2009.

Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A Fresh Approach to Numerical Computing. *SIAM Review*, 59(1):65–98, January 2017. ISSN 0036-1445. doi: 10.1137/141000671. URL https://epubs.siam.org/doi/abs/10.1137/141000671.

Nils Lid Hjort, Chris Holmes, Peter Müller, and Stephen G Walker. *Bayesian nonparametrics*, volume 28 of *Cambridge Series in Statistical and Probabilistic Mathematics*. Cambridge University Press, 2010.

Carl Edward Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning*, volume 2. MIT Press, 2006. ISBN 978-0-262-18253-9.

Persi Diaconis and David Freedman. On the consistency of bayes estimates. *The Annals of Statistics*, pages 1–26, 1986.

David Freedman. On the bernstein-von mises theorem with infinite-dimensional parameters. *The Annals of Statistics*, 27(4):1119–1140, 1999. ISSN 00905364.

Peter Grünwald and John Langford. Suboptimal behavior of bayes and mdl in classification under misspecification. *Machine Learning*, 66:119–149, 2007.

Jeffrey W Miller and Matthew T Harrison. A simple example of dirichlet process mixture inconsistency for the number of components. *Advances in neural information processing systems*, 26, 2013.

Peter Orbanz and Yee Whye Teh. Bayesian Nonparametric Models. *Encyclopedia of machine learning*, 1:14, 2010.

Nathanael L. Ackerman, Cameron E. Freer, and Daniel M. Roy. On the computability of conditional probability. *J. ACM*, 66(3), jun 2019. ISSN 0004-5411. doi: 10.1145/3321699. URL https://doi.org/10.1145/3321699.

Daniel Roy, Vikash Mansinghka, Noah Goodman, and Josh Tenenbaum. A stochastic programming perspective on nonparametric bayes. *Nonparametric Bayesian Workshop, ICML*, 2008.

Daniel Murphy Roy. *Computability, inference and modeling in probabilistic programming*. PhD thesis, Massachusetts Institute of Technology, 2011.

David Wingate, Andreas Stuhlmueller, and Noah Goodman. Lightweight Implementations of Probabilistic Programming Languages Via Transformational Compilation. In *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics*, pages 770–778. JMLR Workshop and Conference Proceedings, June 2011. URL https://proceedings.mlr.press/v15/wingate11a.html.

Oleg Kiselyov. Problems of the Lightweight Implementation of Probabilistic Programming. Technical report, 2016. URL https://pps2016.luddy.indiana.edu/2015/12/23/problems-of-the-lightweight-implementation-of-probabilistic-programming/.

Chung-kil Hur, Aditya V. Nori, and Sriram K. Rajamani. A Provably Correct Sampler for Probabilistic Programs, 2015.

Lingfeng Yang, Patrick Hanrahan, and Noah Goodman. Generating efficient mcmc kernels from probabilistic programs. In *Artificial Intelligence and Statistics*, pages 1068–1076. PMLR, 2014.

Daniel Ritchie, Andreas Stuhlmüller, and Noah Goodman. C3: Lightweight incrementalized mcmc for probabilistic programs using continuations and callsite caching. In *Artificial Intelligence and Statistics*, pages 28–37. PMLR, 2016a.

Tuan Anh Le. *Inference for Higher Order Probabilistic Programs.* Master's Thesis, University of Oxford, 2016.

Carol Mak, Fabian Zaiser, and Luke Ong. Nonparametric involutive markov chain monte carlo. In Kamalika Chaudhuri, Stefanie Jegelka, Le Song, Csaba Szepesvári, Gang Niu, and Sivan Sabato, editors, *International Conference on Machine Learning, ICML 2022, 17-23 July 2022, Baltimore, Maryland, USA*, volume 162 of *Proceedings of Machine Learning Research*, pages 14802–14859. PMLR, 2022. URL https://proceedings.mlr.press/v162/mak22a.html.

Peter J. Green. Reversible jump Markov chain Monte Carlo computation and Bayesian model determination. *Biometrika*, 82(4):711–732, December 1995. ISSN 0006-3444. doi: 10.1093/biomet/82.4.711. URL https://doi.org/10.1093/biomet/82.4.711.

David A. Roberts, Marcus Gallagher, and Thomas Taimre. Reversible Jump Probabilistic Programming. In *Proceedings of the Twenty-Second International Conference on Artificial Intelligence and Statistics*, pages 634–643. PMLR, April 2019. URL https://proceedings.mlr.press/v89/roberts19a.html.

Kirill Neklyudov, Max Welling, Evgenii Egorov, and Dmitry Vetrov. Involutive MCMC: A Unifying Framework. In *Proceedings of the 37th International Conference on Machine Learning*, pages 7273–7282. PMLR, November 2020. URL https://proceedings.mlr.press/v119/neklyudov20a.html.

Marco Cusumano-Towner, Alexander K. Lew, and Vikash K. Mansinghka. Automating Involutive MCMC using Probabilistic and Differentiable Programming. *arXiv:2007.09871 [stat]*, July 2020. URL http://arxiv.org/abs/2007.09871.

L. Martino, V. Elvira, D. Luengo, and J. Corander. Layered adaptive importance sampling. *Statistics and Computing*, 27(3), May 2017.

Sriram Sankaranarayanan, Aleksandar Chakarov, and Sumit Gulwani. Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 447–458, New York, NY, USA, June 2013. Association for Computing Machinery. ISBN

978-1-4503-2014-6. doi: 10.1145/2491956.2462179. URL https://doi.org/10.1145/2491956.2462179.

Arun Chaganty, Aditya Nori, and Sriram Rajamani. Efficiently Sampling Probabilistic Programs via Program Analysis. In *Proceedings of the Sixteenth International Conference on Artificial Intelligence and Statistics*, pages 153–160. PMLR, April 2013. URL https://proceedings.mlr.press/v31/chaganty13a.html.

Raven Beutner, Luke Ong, and Fabian Zaiser. Guaranteed bounds for posterior inference in universal probabilistic programming. *PLDI 2022: International Conference on Programming Language Design and Implementation*, 2022.

Tom Rainforth, Yuan Zhou, Xiaoyu Lu, Yee Whye Teh, Frank Wood, Hongseok Yang, and Jan-Willem van de Meent. Inference Trees: Adaptive Inference with Exploration. *arXiv:1806.09550 [stat]*, June 2018. URL http://arxiv.org/abs/1806.09550.

Jan-Willem van de Meent, Brooks Paige, David Tolpin, and Frank Wood. Black-Box Policy Search with Probabilistic Programs. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics*, pages 1195–1204. PMLR, May 2016. URL https://proceedings.mlr.press/v51/vandemeent16.html.

Hagai Attias. Planning by Probabilistic Inference. In *International Workshop on Artificial Intelligence and Statistics*, pages 9–16. PMLR, January 2003. URL https://proceedings.mlr.press/r4/attias03a.html.

Sergey Levine. Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review. *arXiv:1805.00909 [cs, stat]*, May 2018. URL http://arxiv.org/abs/1805.00909.

Wonyeol Lee, Hangyeol Yu, Xavier Rival, and Hongseok Yang. Towards verified stochastic variational inference for probabilistic programs. *Proceedings of the ACM on Programming Languages*, 4(POPL):16:1–16:33, December 2019. doi: 10.1145/3371084. URL https://doi.org/10.1145/3371084.

Samuel J Gershman and Noah Goodman. Amortized Inference in Probabilistic Reasoning. *Proceedings of the Annual Meeting of the Cognitive Science Society*, page 6, 2014.

Tuan Anh Le, Atilim Gunes Baydin, and Frank Wood. Inference Compilation and Universal Probabilistic Programming. In *Artificial Intelligence and Statistics*, pages 1338–1348. PMLR, April 2017. URL http://proceedings.mlr.press/v54/le17a.html.

Sepp Hochreiter and Jürgen Schmidhuber. Long Short-Term Memory. *Neural Computation*, 9(8):1735–1780, November 1997. ISSN 0899-7667. doi: 10.1162/neco.1997.9.8.1735. URL https://doi.org/10.1162/neco.1997.9.8.1735.

William Harvey, Andreas Munk, Atılım Güneş Baydin, Alexander Bergholm, and Frank Wood. Attention for inference compilation. *arXiv preprint arXiv:1910.11961*, 2019.

Brooks Paige and Frank Wood. Inference networks for sequential monte carlo in graphical models. In *International Conference on Machine Learning*, pages 3040–3049. PMLR, 2016.

Daniel Ritchie, Paul Horsfall, and Noah D Goodman. Deep Amortized Inference for Probabilistic Programs. page 31, 2016b.

Praveen Narayanan, Jacques Carette, Wren Romano, Chung-chieh Shan, and Robert Zinkov. Probabilistic Inference by Program Transformation in Hakaru (System Description). In Oleg Kiselyov and Andy King, editors, *Functional and Logic Programming*, volume 9613, pages 62–79. Springer International Publishing, Cham, 2016. ISBN 978-3-319-29603-6 978-3-319-29604-3. doi: 10.1007/978-3-319-29604-3_5. URL http://link.springer.com/10.1007/978-3-319-29604-3_5.

Lawrence M. Murray and Thomas B. Schön. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control*, 46:29–43, 2018. ISSN 1367-5788. doi: https://doi.org/10.1016/j.arcontrol.2018.10.013. URL https://www.sciencedirect.com/science/article/pii/S1367578818301202.

Christopher Manning and Hinrich Schütze. *Foundations of Statistical Natural Language Processing*. MIT Press, Cambridge, MA, USA, May 1999. ISBN 978-0-262-13360-9.

Sylvia. Richardson and Peter J. Green. On Bayesian Analysis of Mixtures with an Unknown Number of Components (with discussion). *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 59(4):731–792, 1997. ISSN 1467-9868. doi: 10.1111/1467-9868.00095. URL https://onlinelibrary.wiley.com/doi/abs/10.1111/1467-9868.00095.

Bradley Gram-Hansen, Christian Schröder de Witt, Tom Rainforth, Philip H. S. Torr, Yee Whye Teh, and Atılım Güneş Baydin. Hijacking Malaria Simulators with Probabilistic Programming. *arXiv:1905.12432 [cs, stat]*, May 2019. URL http://arxiv.org/abs/1905.12432.

Alp Kucukelbir, Rajesh Ranganath, Andrew Gelman, and David M. Blei. Automatic Variational Inference in Stan. *arXiv:1506.03431 [stat]*, June 2015. URL http://arxiv.org/abs/1506.03431.

Stefan Webb, Adam Golinski, Robert Zinkov, N. Siddharth, Tom Rainforth, Yee Whye Teh, and Frank Wood. Faithful Inversion of Generative Models for Effective Amortized Inference. *arXiv:1712.00287 [cs, stat]*, November 2018. URL http://arxiv.org/abs/1712.00287.

Aditya V Nori, Chung-Kil Hur, Sriram K Rajamani, and Selva Samuel. R2: An Efficient MCMC Sampler for Probabilistic Programs. *AAAI Conference on Artificial Intelligence (AAAI)*, page 7, 2015.

Tor Lattimore and Csaba Szepesvári. *Bandit Algorithms*. Cambridge University Press, 2020. doi: 10.1017/9781108571401.

Zohar Karnin, Tomer Koren, and Oren Somekh. Almost optimal exploration in multi-armed bandits. In *International Conference on Machine Learning*, pages 1238–1246. PMLR, 2013.

Lisha Li, Kevin Jamieson, Giulia DeSalvo, Afshin Rostamizadeh, and Ameet Talwalkar. Hyperband: A novel bandit-based approach to hyperparameter optimization. *Journal of Machine Learning Research*, 18(185):1–52, 2018. URL http://jmlr.org/papers/v18/16-558.html.

Stefan Falkner, Aaron Klein, and Frank Hutter. BOHB: Robust and efficient hyperparameter optimization at scale. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1437–1446. PMLR, 10–15 Jul 2018. URL https://proceedings.mlr.press/v80/falkner18a.html.

S. M. Ali Eslami, Nicolas Heess, Theophane Weber, Yuval Tassa, David Szepesvari, Koray Kavukcuoglu, and Geoffrey E. Hinton. Attend, Infer, Repeat: Fast Scene Understanding with Generative Models. *arXiv:1603.08575 [cs]*, March 2016. URL http://arxiv.org/abs/1603.08575.

Brooks Paige and Frank Wood. A Compilation Target for Probabilistic Programming Languages. *arXiv:1403.0504 [cs, stat]*, July 2014. URL http://arxiv.org/abs/1403.0504.

Tom Rainforth, Christian Naesseth, Fredrik Lindsten, Brooks Paige, Jan-Willem Vandemeent, Arnaud Doucet, and Frank Wood. Interacting particle markov chain monte carlo. In Maria Florina Balcan and Kilian Q. Weinberger, editors, *Proceedings of The 33rd International Conference on Machine Learning*, volume 48 of *Proceedings of Machine Learning Research*, pages 2616–2625, New York, New York, USA, 20–22 Jun 2016. PMLR. URL https://proceedings.mlr.press/v48/rainforth16.html.

Andreas Stuhlmüller, Jacob Taylor, and Noah Goodman. Learning stochastic inverses. *Advances in neural information processing systems*, 26, 2013.

David Duvenaud, James Lloyd, Roger Grosse, Joshua Tenenbaum, and Ghahramani Zoubin. Structure Discovery in Nonparametric Regression through Compositional Kernel Search. In *International Conference on Machine Learning*, pages 1166–1174. PMLR, May 2013. URL http://proceedings.mlr.press/v28/duvenaud13.html.

David Janz, Brooks Paige, Tom Rainforth, Jan-Willem van de Meent, and Frank Wood. Probabilistic structure discovery in time series data. *arXiv preprint arXiv:1611.06863*, 2016.

G.E.P. Box, G.M. Jenkins, G.C. Reinsel, and G.M. Ljung. *Time Series Analysis: Forecasting and Control*. Wiley Series in Probability and Statistics. Wiley, 2015. ISBN 9781118674925. URL https://books.google.co.uk/books?id=rNt5CgAAQBAJ.

Wonyeol Lee, Xavier Rival, and Hongseok Yang. Smoothness analysis for probabilistic programs with application to optimised variational inference. *Proceedings of the ACM on Programming Languages*, 7(POPL):335–366, 2023.

Alexander K Lew, Mathieu Huot, Sam Staton, and Vikash K Mansinghka. Adev: Sound automatic differentiation of expected values of probabilistic programs. *Proceedings of the ACM on Programming Languages*, 7(POPL):121–153, 2023.

Dominik Wagner, Basim Khajwal, and C-H Luke Ong. Diagonalisation sgd: Fast & convergent sgd for non-differentiable models via reparameterisation and smoothing. *arXiv preprint arXiv:2402.11752*, 2024.

Yicheng Luo, Antonio Filieri, and Yuan Zhou. Symbolic parallel adaptive importance sampling for probabilistic program analysis. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1166–1177, 2021.

Thomas P Minka. Bayesian model averaging is not model combination. 2000.

Yuling Yao, Aki Vehtari, Daniel Simpson, and Andrew Gelman. Using Stacking to Average Bayesian Predictive Distributions (with Discussion). *Bayesian Analysis*, 13(3):917–1007, September 2018. ISSN 1936-0975, 1931-6690. doi: 10.1214/17-BA1091.

Oscar Oelrich, Shutong Ding, Måns Magnusson, Aki Vehtari, and Mattias Villani. When are bayesian model probabilities overconfident? *arXiv preprint arXiv:2003.04026*, 2020.

Jonathan H. Huggins and Jeffrey W. Miller. Reproducible Model Selection Using Bagged Posteriors, December 2021. URL http://arxiv.org/abs/2007.14845.

Ziheng Yang and Tianqi Zhu. Bayesian selection of misspecified models is overconfident and may cause spurious posterior probabilities for phylogenetic trees. *Proceedings of the National Academy of Sciences*, 115(8):1854–1859, February 2018. doi: 10.1073/pnas.1712673115. URL https://www.pnas.org/doi/full/10.1073/pnas.1712673115.

Jane T Key, Luis R Pericchi, and Adrian FM Smith. Bayesian model choice: what and why. *Bayesian statistics*, 6:343–370, 1999.

Aki Vehtari and Janne Ojanen. A survey of Bayesian predictive methods for model assessment, selection and comparison. *Statistics Surveys*, 6(none):142–228, January 2012. ISSN 1935-7516. doi: 10.1214/12-SS102.

Frank Smets and Rafael Wouters. Shocks and frictions in us business cycles: A bayesian dsge approach. *American economic review*, 97(3):586–606, 2007.

Alexander P Leff, Thomas M Schofield, Klass E Stephan, Jennifer T Crinion, Karl J Friston, and Cathy J Price. The cortical dynamics of intelligible speech. *Journal of Neuroscience*, 28(49):13209–13215, 2008.

David H. Wolpert. Stacked generalization. *Neural Networks*, 5(2):241–259, 1992. ISSN 0893-6080. doi: https://doi.org/10.1016/S0893-6080(05)80023-1. URL https://www.sciencedirect.com/science/article/pii/S0893608005800231.

Leo Breiman. Stacked regressions. *Machine learning*, 24(1):49–64, 1996.

Michael LeBlanc and Robert Tibshirani. Combining estimates in regression and classification. *Journal of the American Statistical Association*, 1996.

Andres Masegosa. Learning under model misspecification: Applications to variational and ensemble methods. *Advances in Neural Information Processing Systems*, 33:5479–5491, 2020.

Mohammad Saeed Masiha, Amin Gohari, Mohammad Hossein Yassaee, and Mohammad Reza Aref. Learning under distribution mismatch and model misspecification. In *2021 IEEE International Symposium on Information Theory (ISIT)*, pages 2912–2917. IEEE, 2021.

Warren R. Morningstar, Alex Alemi, and Joshua V. Dillon. Pacm-bayes: Narrowing the empirical risk gap in the misspecified bayesian regime. In Gustau Camps-Valls, Francisco J. R. Ruiz, and Isabel Valera, editors, *Proceedings of The 25th International Conference on Artificial Intelligence and Statistics*, volume 151 of *Proceedings of Machine Learning Research*, pages 8270–8298. PMLR, 28–30 Mar 2022. URL https://proceedings.mlr.press/v151/morningstar22a.html.

Pierre Alquier. User-friendly introduction to pac-bayes bounds, 2023.

José M Bernardo and Adrian FM Smith. *Bayesian theory*, volume 405. John Wiley & Sons, 2009.

Merlise Clyde and Edwin S Iversen. Bayesian model averaging in the M-open framework. In *Bayesian Theory and Applications*. Oxford University Press, 01 2013.

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

José M Bernardo. Expected information as expected utility. *the Annals of Statistics*, pages 686–690, 1979.

Tim Reichelt, Luke Ong, and Tom Rainforth. Rethinking variational inference for probabilistic programs with stochastic support. In *Advances in Neural Information Processing Systems*, 2022a.

Richard H Byrd, Peihuang Lu, Jorge Nocedal, and Ciyou Zhu. A limited memory algorithm for bound constrained optimization. *SIAM Journal on scientific computing*, 16(5):1190–1208, 1995.

Ciyou Zhu, Richard H Byrd, Peihuang Lu, and Jorge Nocedal. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. *ACM Transactions on mathematical software (TOMS)*, 23(4):550–560, 1997.

Aki Vehtari, Andrew Gelman, and Jonah Gabry. Practical bayesian model evaluation using leave-one-out cross-validation and waic. *Statistics and computing*, 27(5): 1413–1432, 2017.

Ravin Kumar, Colin Carroll, Ari Hartikainen, and Osvaldo Martin. Arviz a unified library for exploratory analysis of bayesian models in python. *Journal of Open Source Software*, 2019. doi: 10.21105/joss.01143.

Kristine Monteith, James L. Carroll, Kevin Seppi, and Tony Martinez. Turning bayesian model averaging into bayesian model combination. In *The 2011 International Joint Conference on Neural Networks*, 2011.

Hyun-Chul Kim and Zoubin Ghahramani. Bayesian classifier combination. In Neil D. Lawrence and Mark Girolami, editors, *Proceedings of the Fifteenth International Conference on Artificial Intelligence and Statistics*, volume 22 of *Proceedings of Machine Learning Research*, pages 619–627, La Palma, Canary Islands, 21–23 Apr 2012. PMLR. URL https://proceedings.mlr.press/v22/kim12.html.

Yuling Yao, Gregor Pirš, Aki Vehtari, and Andrew Gelman. Bayesian hierarchical stacking: Some models are (somewhere) useful. *Bayesian Analysis*, 17(4):1043–1071, 2022.

Isobel Claire Gormley and Sylvia Frühwirth-Schnatter. Mixture of experts models. In *Handbook of mixture analysis*, pages 271–307. Chapman and Hall/CRC, 2019.

Seymour Geisser and William F Eddy. A predictive approach to model selection. *Journal of the American Statistical Association*, 74(365):153–160, 1979.

Jonathan Warrell and Mark Gerstein. Higher-order generalization bounds: Learning deep probabilistic programs via pac-bayes objectives. *arXiv preprint arXiv:2203.15972*, 2022.

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757*, 2017.

Cameron E Freer, Vikash K Mansinghka, and Daniel M Roy. When are probabilistic programs probably computationally tractable. In *NIPS Workshop on Monte Carlo Methods for Modern Applications*, 2010.

M. Hoffman and A. Gelman. The No-U-turn Sampler: Adaptively Setting Path Lengths in Hamiltonian Monte Carlo. *Journal of Machine Learning Research*, 15: 1593–1623, 2014.

R Kelley Pace and Ronald Barry. Sparse spatial autoregressions. *Statistics & Probability Letters*, 33(3):291–297, 1997.

Jack W Smith, James E Everhart, WC Dickson, William C Knowler, and Robert Scott Johannes. Using the adap learning algorithm to forecast the onset of diabetes mellitus. In *Proceedings of the annual symposium on computer application in medical care*, page 261. American Medical Informatics Association, 1988.

Kaggle. Stroke prediction dataset. https://www.kaggle.com/datasets/fedesoriano/stroke-prediction-dataset, 2020. Accessed: 2023-10-05.

Andrew Gelman and Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Analytical Methods for Social Research. Cambridge University Press, 2006. doi: 10.1017/CBO9780511790942.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. http://dippl.org, 2014. Accessed: 2021-5-21.

Tom Minka, John M. Winn, John P. Guiver, Yordan Zaykov, Dany Fabian, and John Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

G.M. Torrie and J.P. Valleau. Nonphysical Sampling Distributions in Monte Carlo Free-Energy Estimation: Umbrella Sampling. *Journal of Computational Physics*, 23(2):187 – 199, 1977. ISSN 0021-9991. doi: https://doi.org/10.1016/0021-9991(77)90121-8. URL http://www.sciencedirect.com/science/article/pii/0021999177901218.

Timothy Classen Hesterberg. *Advances in Importance Sampling*. PhD thesis, Stanford University, 1988.

Robert L Wolpert. Monte Carlo Integration in Bayesian Statistical Analysis. *Contemporary Mathematics*, 115:101–116, 1991.

Man-Suk Oh and James O Berger. Adaptive Importance Sampling in Monte Carlo Integration. *Journal of Statistical Computation and Simulation*, 41(3-4):143–168, 1992.

Michael Evans, Tim Swartz, et al. Methods for Approximating Integrals in Statistics with Special Emphasis on Bayesian Integration Problems. *Statistical science*, 10 (3):254–272, 1995.

Xiao-Li Meng and Wing Hung Wong. Simulating Ratios of Normalizing Constants via a Simple Identity: a Theoretical Exploration. *Statistica Sinica*, pages 831–860, 1996.

Ming-Hui Chen, Qi-Man Shao, et al. On Monte Carlo Methods for Estimating Ratios of Normalizing Constants. *The Annals of Statistics*, 25(4):1563–1594, 1997.

Adam Golinski, Frank Wood, and Tom Rainforth. Amortized Monte Carlo Integration. In *International Conference on Machine Learning*, pages 2309–2318. PMLR, May 2019. URL http://proceedings.mlr.press/v97/golinski19a.html.

Radford M. Neal. Annealed Importance Sampling. *arXiv:physics/9803008*, September 1998. URL http://arxiv.org/abs/physics/9803008.

Robert Zinkov and Chung-Chieh Shan. Composing Inference Algorithms as Program Transformations. *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, page 10, 2017.

Timon Gehr, Samuel Steffen, and Martin Vechev. λpsi: Exact inference for higher-order probabilistic programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2020, pages 883–897, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450376136. doi: 10.1145/3385412.3386006. URL https://doi.org/10.1145/3385412.3386006.

Pierre Del Moral, Arnaud Doucet, and Ajay Jasra. Sequential Monte Carlo Samplers. *Journal of the Royal Statistical Society: Series B (Statistical Methodology)*, 68(3): 411–436, 2006.

Hanna M. Wallach, Iain Murray, Ruslan Salakhutdinov, and David Mimno. *Evaluation Methods for Topic Models*, pages 1105–1112. Association for Computing Machinery, New York, NY, USA, 2009. ISBN 9781605585161. URL https://doi.org/10.1145/1553374.1553515.

Ruslan Salakhutdinov and Hugo Larochelle. Efficient Learning of Deep Boltzmann Machines. In *International Conference on Artificial Intelligence and Statistics (AISTATS)*, pages 693–700, 2010.

Yuhuai Wu, Yuri Burda, R. Salakhutdinov, and Roger B. Grosse. On the Quantitative Analysis of Decoder-Based Generative Models. *International Conference on Learning Representations (ICLR)*, 2017.

The Turing Development Team. TuringLang/AdvancedMH.jl. The Turing Language, October 2020. URL https://github.com/TuringLang/AdvancedMH.jl.

Kai Xu, Hong Ge, Will Tebbutt, Mohamed Tarek, Martin Trapp, and Zoubin Ghahramani. AdvancedHMC.jl: A Robust, Modular and Efficient Implementation of Advanced HMC Algorithms. In *Symposium on Advances in Approximate Bayesian Inference (AABI)*, pages 1–10. PMLR, February 2020.

Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Bürkner. Rank-Normalization, Folding, and Localization: An Improved $\widehat{R}$ for Assessing Convergence of MCMC. *Bayesian Analysis*, 2020. doi: 10.1214/20-BA1221. URL https://doi.org/10.1214/20-BA1221. Advance publication.

William Ogilvy Kermack, A. G. McKendrick, and Gilbert Thomas Walker. A contribution to the mathematical theory of epidemics. *Proceedings of the Royal Society of London. Series A, Containing Papers of a Mathematical and Physical Character*, 115(772):700–721, August 1927. doi: 10.1098/rspa.1927.0118. URL https://royalsocietypublishing.org/doi/10.1098/rspa.1927.0118.

Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521 (7553):436–444, May 2015. ISSN 1476-4687. doi: 10.1038/nature14539. URL https://www.nature.com/articles/nature14539.

Jürgen Schmidhuber. Deep learning in neural networks: An overview. *Neural networks*, 61:85–117, 2015.

Brenden M Lake, Ruslan Salakhutdinov, and Joshua B Tenenbaum. Human-level concept learning through probabilistic program induction. *Science*, 350(6266): 1332–1338, 2015.

Rishi Bommasani, Drew A Hudson, Ehsan Adeli, Russ Altman, Simran Arora, Sydney von Arx, Michael S Bernstein, Jeannette Bohg, Antoine Bosselut, Emma Brunskill, et al. On the opportunities and risks of foundation models. *arXiv preprint arXiv:2108.07258*, 2021.

Team Gemini, Rohan Anil, Sebastian Borgeaud, Yonghui Wu, Jean-Baptiste Alayrac, Jiahui Yu, Radu Soricut, Johan Schalkwyk, Andrew M Dai, Anja Hauth, et al. Gemini: a family of highly capable multimodal models. *arXiv preprint arXiv:2312.11805*, 2023.

Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, et al. Gpt-4 technical report. 2023.

Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

Jack W Rae, Sebastian Borgeaud, Trevor Cai, Katie Millican, Jordan Hoffmann, Francis Song, John Aslanides, Sarah Henderson, Roman Ring, Susannah Young, et al. Scaling language models: Methods, analysis & insights from training gopher. *arXiv preprint arXiv:2112.11446*, 2021.

Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. Language models are few-shot learners. *Advances in neural information processing systems*, 33:1877–1901, 2020.

Sébastien Bubeck, Varun Chandrasekaran, Ronen Eldan, Johannes Gehrke, Eric Horvitz, Ece Kamar, Peter Lee, Yin Tat Lee, Yuanzhi Li, Scott Lundberg, et al. Sparks of artificial general intelligence: Early experiments with gpt-4. *arXiv preprint arXiv:2303.12712*, 2023.

Lionel Wong, Gabriel Grand, Alexander K Lew, Noah D Goodman, Vikash K Mansinghka, Jacob Andreas, and Joshua B Tenenbaum. From word models to world models: Translating from natural language to the probabilistic language of thought. *arXiv preprint arXiv:2306.12672*, 2023.

Richard S. Sutton. The bitter lesson. http://www.incompleteideas.net/IncIdeas/BitterLesson.html, 2019. Accessed: 25/03/2024.

Kaifeng Bi, Lingxi Xie, Hengheng Zhang, Xin Chen, Xiaotao Gu, and Qi Tian. Pangu-weather: A 3d high-resolution model for fast and accurate global weather forecast. *arXiv preprint arXiv:2211.02556*, 2022.

Thorsten Kurth, Shashank Subramanian, Peter Harrington, Jaideep Pathak, Morteza Mardani, David Hall, Andrea Miele, Karthik Kashinath, and Anima Anandkumar. Fourcastnet: Accelerating global high-resolution weather forecasting using adaptive fourier neural operators. In *Proceedings of the Platform for Advanced Scientific Computing Conference*, PASC '23, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400701900. doi: 10.1145/3592979.3593412. URL https://doi.org/10.1145/3592979.3593412.

Remi Lam, Alvaro Sanchez-Gonzalez, Matthew Willson, Peter Wirnsberger, Meire Fortunato, Ferran Alet, Suman Ravuri, Timo Ewalds, Zach Eaton-Rosen, Weihua Hu, et al. Learning skillful medium-range global weather forecasting. *Science*, 382(6677):1416–1421, 2023.

Dmitrii Kochkov, Janni Yuval, Ian Langmore, Peter Norgaard, Jamie Smith, Griffin Mooers, James Lottes, Stephan Rasp, Peter Düben, Milan Klöwer, et al. Neural general circulation models. *arXiv preprint arXiv:2311.07222*, 2023.

Aki Vehtari, Daniel Simpson, Andrew Gelman, Yuling Yao, and Jonah Gabry. Pareto smoothed importance sampling. *arXiv preprint arXiv:1507.02646*, 2015.

Tim Reichelt, Adam Goliński, Luke Ong, and Tom Rainforth. Expectation programming: Adapting probabilistic programming systems to estimate expectations efficiently. In *Uncertainty in Artificial Intelligence*, pages 1676–1685. PMLR, 2022b.

Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020. doi: 10.1038/s41592-019-0686-2.

Dong C Liu and Jorge Nocedal. On the limited memory bfgs method for large scale optimization. *Mathematical programming*, 45(1-3):503–528, 1989.

Yuri Burda, Roger Grosse, and Ruslan Salakhutdinov. Importance Weighted Autoencoders. *arXiv:1509.00519 [cs, stat]*, November 2016. URL http://arxiv.org/abs/1509.00519.

Leo Grinsztajn, Elizaveta Semenova, Charles C. Margossian, and Julien Riou. Bayesian workflow for disease transmission modeling in Stan, 2020. URL https://mc-stan.org/users/documentation/case-studies/boarding_school_case_study.html.