Meta-Learning an Inference Algorithm for Probabilistic Programs

Gwonsoo Che
KAIST
Daejeon, Korea
gche@kaist.ac.kr

Hongseok Yang
KAIST
Daejeon, Korea
hongseok.yang@kaist.ac.kr

Abstract

We present a meta-algorithm for learning a posterior-inference algorithm for restricted probabilistic programs. Our meta-algorithm takes a training set of probabilistic programs that describe models with observations, and attempts to learn an efficient method for inferring the posterior of a similar program. A key feature of our approach is the use of what we call a white-box inference algorithm that analyses the given program sequentially using multiple neural networks to compute an approximate posterior. The parameters of these networks are learnt from a training set by our meta-algorithm. We empirically demonstrate that the learnt inference algorithm generalises well to programs that are new in terms of both parameters and model structures, and report cases where our approach achieves greater test-time efficiency than alternatives such as HMC.

1 Introduction

The development of performant probabilistic programming systems[Goodman et al., 2008, Wood et al., 2014, Mansinghka et al., 2014, Minka et al., 2018, Narayanan et al., 2016, Salvatier et al., 2016, Carpenter et al., 2017, Tran et al., 2016, Ge et al., 2018, Bingham et al., 2018] also revealed the difficulty of achieving efficiency and universality simultaneously, and the need for equipping probabilistic programming languages (PPLs) with mechanisms for customising inference or learning algorithms to a given domain. In fact, recent PPLs include constructs for specifying conditional independence in a model [Bingham et al., 2018] or defining proposals [Ritchie et al., 2015, Siddharth et al., 2017, Bingham et al., 2018, Tran et al., 2018, Cusumano-Towner et al., 2019], all enabling users to help the algorithms. In this paper, we present a meta-algorithm for learning a posterior-inference algorithm itself from a given set of restricted probabilistic programs. The meta-algorithm aims at constructing a customised inference algorithm for the given set of models, while ensuring universality to the extent that the constructed algorithm generalises to similar but unseen programs.

The distinguished feature of our approach is the use of what we call a white-box inference algorithm, which is equipped with multiple neural networks, one for each type of atomic command in a PPL, and computes an approximate posterior for a given program by analysing individual atomic commands in it sequentially using these networks. For instance, our white-box inference algorithm regards the program in Fig. 1 as a sequence of the five atomic commands, initialises its internal state $h \in \mathbb{R}^m$ with h_0, and transforms the state over the sequence. The internal state h is the encoding of an approximate posterior at the current program point, which corresponds to an approximate filtering distribution of a state-space model. The update of this state for each atomic command is directed by neural networks. Our meta-algorithm trains the parameters of these networks by trying to make the inference algorithm compute accurate posterior approximations over a training set of probabilistic programs.

We discuss related work [Andrieu and Thoms, 2008, Hoffman and Gelman, 2014, Wang et al., 2018, Gong et al., 2019, Wu et al., 2020, Gordon et al., 2019, Iakovleva et al., 2020, Minka, 2001,

$mass \sim \mathcal{N}(5, 10)$; // log of the mass of Milky Way
$g_1 \sim \mathcal{N}(mass \times 2, 5)$; $\texttt{obs}(\mathcal{N}(g_1, 1), 10)$; // observed velocity vel_1=10 of the first satellite galaxy
$g_2 \sim \mathcal{N}(mass + 5, 2)$; $\texttt{obs}(\mathcal{N}(g_2, 1), 3)$ // observed velocity vel_2=3 of the second satellite galaxy

Figure 1: Probabilistic program for a model for Milky Way and its two satellite galaxies. The `obs` statements refer to the observations of (unnamed) random variables vel_1 and vel_2.

Wainwright and Jordan, 2008, Jitkrittum et al., 2015, Gershman and Goodman, 2014, Le et al., 2017, Paige and Wood, 2016, Stuhlmüller et al., 2013, Kingma and Welling, 2013, Mnih and Gregor, 2014, Rezende et al., 2014, Ritchie et al., 2016, Marino et al., 2018, Zaremba and Sutskever, 2014, Bieber et al., 2020, Reed and de Freitas, 2016] in Appendix A.

Our contributions are as follows: (i) we present a white-box posterior-inference algorithm, which works directly on model description and can be customised to a given model class; (ii) we describe a meta-algorithm for learning the parameters of the inference algorithm; (iii) we empirically analyse our approach with different model classes, and show the promise as well as the remaining challenges.

2 Our Approach

Setup Our results assume a simple PPL without loop and with a limited form of conditional statement. The syntax of the language is given by the following grammar, where r represents a real number, z and v_i variables storing a real, and p the name of a procedure taking two real-valued parameters and returning a real number:

$$\begin{aligned} \textit{Programs } C &::= A \mid C_1; C_2 \\ \textit{Atomic Commands } A &::= z \sim \mathcal{N}(v_1, v_2) \mid \texttt{obs}(\mathcal{N}(v_0, v_1), r) \mid v_0 := \texttt{if}\ (v_1 > v_2)\ v_3\ \texttt{else}\ v_4 \\ &\quad\mid\ v_0 := r \mid v_0 := v_1 \mid v_0 := p(v_1, v_2) \end{aligned}$$

Programs in the language are constructed by sequentially composing atomic commands. The last atomic command $v_0 := p(v_1, v_2)$ is a call to one of the known deterministic procedures, which may be standard binary operations such as addition and multiplication, or complex non-trivial functions that are used to build advanced, non-conventional models. When p is a standard binary operation, we use the usual infix notation and write, for example, $v_1 + v_2$, instead of $+(v_1, v_2)$. We provide full discussion of (our choice of) the PPL in Appendix B. The formal semantics of the PPL is in Appendix C. Our white-box inference algorithm aims at computing the approximate posterior and marginal likelihood estimate for a given program C, when the unnormalised density p_C (denoted by C) has a finite non-zero marginal likelihood and, as a result, a well-defined posterior density.

White-Box Inference Algorithm Consider, for presentation, programs that sample n-many latent variables $z_1, \ldots, z_n$ and use at most m-many variables ($m \geq n$). Let $\mathbb{V}$ be $[0, 1]^m$, the space of the one-hot encodings of those m variables, and $\mathbb{P}$ the set of procedure names. Given a program $C = (A_1; \ldots; A_k)$, our white-box inference algorithm computes an approximate posterior and a marginal likelihood estimate for C by sequentially processing the A_i's. The computation for each A_i is directed by one of the neural networks defined below, according to the type of A_i:

$$\begin{aligned} &nn_{\mathrm{sa},\phi_1} : \mathbb{V}^3 \times \mathbb{R}^s \to \mathbb{R}^s, && nn_{\mathrm{ob},\phi_2} : \mathbb{V}^2 \times \mathbb{R} \times \mathbb{R}^s \to \mathbb{R}^s, && nn_{\mathrm{if},\phi_3} : \mathbb{V}^5 \times \mathbb{R}^s \to \mathbb{R}^s, \\ &nn^{\mathrm{c}}_{:=,\phi_4} : \mathbb{V} \times \mathbb{R} \times \mathbb{R}^s \to \mathbb{R}^s, && nn^{\mathrm{v}}_{:=,\phi_5} : \mathbb{V}^2 \times \mathbb{R}^s \to \mathbb{R}^s, && nn_{p,\phi_p} : \mathbb{V}^3 \times \mathbb{R}^s \to \mathbb{R}^s \text{ for } p \in \mathbb{P}, \\ &nn_{\mathrm{de},\phi_6} : \mathbb{R}^s \to (\mathbb{R} \times \mathbb{R})^n, && nn_{\mathrm{intg},\phi_7} : \mathbb{V}^2 \times \mathbb{R} \times \mathbb{R}^s \to \mathbb{R}, \end{aligned}$$

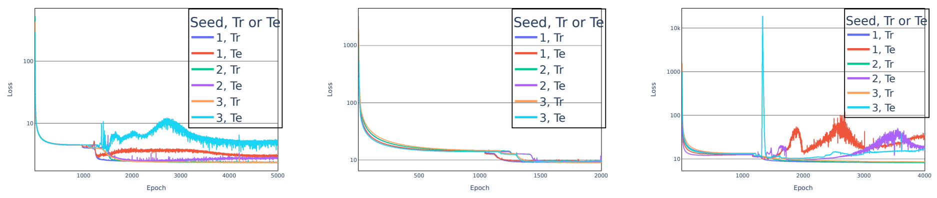
where $\phi_{1:7}$ and ϕ_p for $p \in \mathbb{P}$ are network parameters. The top six networks are for the six types of atomic commands, nn_{de,ϕ_6} is a decoder that maps the states h to the probability densities (means and variances in our setup) over latent variables, and $nn_{\mathrm{intg},\phi_7}$ is used when the algorithm updates the marginal likelihood estimate for an observe statement $\texttt{obs}(\mathcal{N}(v_0, v_1), r)$. The full details for $nn_{\mathrm{intg},\phi_7}$ and the derivation of the marginal likelihood are provided in Appendix D.

Given a program $C = (A_1; \ldots; A_k)$ that draws n samples (and so uses latent variables $z_1, \ldots, z_n$), the algorithm approximates the posterior and marginal likelihood of C as follows:

$$\begin{aligned} \textsc{infer}(C) \;=\; &\textbf{let } (h_0, Z_0) = (\vec{0}, 1) \textbf{ and } (h_k, Z_k) = (\textsc{infer}(A_k) \circ \ldots \circ \textsc{infer}(A_1))(h_0, Z_0) \textbf{ in} \\ &\textbf{let } ((\mu_1, \sigma_1^2), \ldots, (\mu_n, \sigma_n^2)) = nn_{\mathrm{de},\phi_6}(h_k) \textbf{ in return } \Big(\textstyle\prod_{i=1}^n \mathcal{N}(z_i \mid \mu_i, \sigma_i^2),\ Z_k \Big), \end{aligned}$$

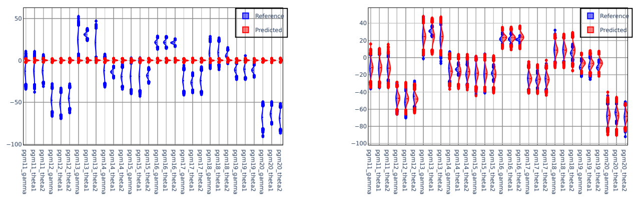
where $\textsc{infer}(A_i) : \mathbb{R}^s \times \mathbb{R} \to \mathbb{R}^s \times \mathbb{R}$ picks an appropriate neural network based on the type of A_i, and uses it to transform h and Z:

$$\textsc{infer}(\texttt{obs}(\mathcal{N}(v_0, v_1), r))(h, Z) = (nn_{\mathrm{ob}}(\overline{v_{0:1}}, r, h), Z \times nn_{\mathrm{intg}}(\overline{v_{0:1}}, r, h)),$$



(a) gauss (b) hierl (c) milky

Figure 2: Average training and test losses under three random seeds. The y-axes are log-scaled. The increases in later epochs of Fig. 2c were due to only one or a few test programs out of 50.



(a) Before training (b) After 2K epochs

Figure 3: Comparisons of predicted and reference marginal posteriors recorded before training and after 2K epochs.

$\text{INFER}(v_0 := \texttt{if}\ (v_1 > v_2)\ v_3\ \texttt{else}\ v_4)(h, Z) = (nn_{\text{if}}(\overline{v_{0:4}}, h), Z),$
$\text{INFER}(v_0 := r)(h, Z) = (nn^{\text{c}}_{:=}(\overline{v_0}, r, h), Z),\ \ \text{INFER}(z \sim \mathcal{N}(v_1, v_2))(h, Z) = (nn_{\text{sa}}(\overline{z}, \overline{v_{1:2}}, h), Z),$
$\text{INFER}(v_0 := v_1)(h, Z) = (nn^{\text{v}}_{:=}(\overline{v_{0:1}}, h), Z),\ \ \text{INFER}(v_0 := p(v_1, v_2))(h, Z) = (nn_p(\overline{v_{0:2}}, h), Z).$

where $\overline{v_{0:k}}$ refers to the sequence of the one-hot encodings of variables $v_0, \ldots, v_k$.

Meta-Learning Parameters Given a training set of programs $\mathcal{D} = \{C_1, \ldots, C_N\}$, we learn the parameters $\phi = (\phi_{1:7}, (\phi_p)_{p \in \mathbb{P}})$ by solving the following optimisation problem:[1]

$$\arg\min_{\phi} \sum_{C \in \mathcal{D}} \text{KL}[\pi_C(z_{1:n}) || q_C(z_{1:n})] + \frac{\lambda}{2}(N_C - Z_C)^2$$

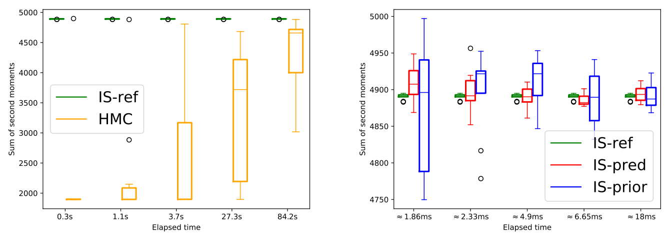
where $\lambda > 0$ is a hyper-parameter, N_C is the marginal likelihood $\int p_C(z_{1:n}) dz_{1:n}$ for p_C, the next $\pi_C(z_{1:n})$ is the normalised posterior $p_C(z_{1:n})/N_C$ for C, and the last q_C and Z_C are the approximate posterior and marginal likelihood estimate computed by the inference algorithm (that is, $(q_C(z_{1:n}), Z_C) = \text{INFER}(C)$). We optimise the objective by stochastic gradient descent, where the gradients can be approximated by sampling techniques. We describe the full algorithm in Appendix E.

3 Empirical Results

This section describes our empirical results. See Appendix F for the full list of our model classes, Appendix G for the detailed experimental setup, and Appendix N for limitations and future work.

Generalisation to New Model Parameters and Observations We evaluated our approach on six model classes: Gaussian (gauss), two hierarchical (hierl and hierd), clustering (cluster), Milky Way (milky and milkyo), and Rosenbrock models (rb) (see Appendix F.1). For each model class, we used 400 programs to meta-learn an inference algorithm, and applied the learnt algorithm to 50 unseen test programs. We measured the average test loss over the 50 test programs, and also compared the marginal posteriors predicted by our learnt inference algorithm with the reference marginal posteriors that were computed analytically, or approximately by HMC using 500K samples after 50K warmups. In case of HMC, we computed the marginal sample means and standard deviations using one of the 10 independent, converged Markov chains. All training and test programs were automatically generated by a random program generator. The training setup is described in Appendix G.1.

[1]Strictly speaking, we assume that the marginal likelihood of any $C \in \mathcal{D}$ is non-zero and finite.



(a) Moments by HMC. (b) Moments by IS-pred and IS-prior.

Figure 4: Comparisons of test-time efficiency in terms of moments estimation.

Fig. 2 shows the training and test losses for gauss, hierl, and milky under three random seeds. The losses for the other model class are in Appendix H. The training loss was averaged over the training set and over the 8 batch updates. The test loss was averaged over the test set. The decrease of training losses led to the downturns of the test losses. The later part of Fig. 2c shows cases where the test loss increases. This was because the loss of only a few programs in the test set (of 50 programs) became large; the losses of the rest remained small. Fig. 3 compares, for 10 test programs in hierl, the reference marginal posteriors (blue) and their predicted counterparts (red) by the learnt inference algorithm instantiated before training and after 2K epochs. It shows that the learnt inference algorithm predicts the reference posteriors precisely for most of the variables. We observed similar patterns for the other cases, except for cluster and rb (see Appendix I).

Generalisation to New Model Structures We let two kinds of model structure vary across programs: the dependency (or data-flow) graph for the variables of a program and the position of a nonlinear function $\mathrm{nl}(x) = 50/\pi \times \arctan(x/10)$ in the program. We specifically considered two classes of models, ext1 (four-variable models that are grouped into 12 types according to the dependency graph and the position of nl) and ext2 (seven-variable models that are grouped into five types according to the dependency graph). See Appendix F.2 for full description. We ran seven experiments for ext1; three evaluated generalisation to unseen dependency graphs, and four evaluated generalisation to unseen positions of nl. We ran five experiments for ext2, each of which evaluated generalisation to an unseen dependency graph. All the experiments were repeated three times under different random seeds, and so the total numbers of experiment runs were 21 and 15 for ext1 and ext2, respectively. The rest of the setup was similar to the generalisation-to-new-parameters case, and we detail the full setup in Appendix G.2.

In 17 runs (out of 21) for ext1, the test losses stabilised or reduced as the training losses decreased, even when the test losses were high and fluctuated in earlier training epochs. In 8 runs (out of 15) for ext2, the test losses were stabilised as the training losses decreased; in 4 runs out of the other 7, the test losses increased only slightly. Appendix J and K show all the losses. In terms of predicted posteriors, we observed highly accurate predictions in 8 runs of ext1. For ext2, the predicted posteriors were precise in 7 runs. Overall, the learnt algorithms generalised to unseen types of models well or fairly well in many cases.

Test-Time Efficiency in Comparison with Alternatives We considered three-variable models (mulmod) where two latent variables follow normal distributions and the other stores the value of the function $\mathrm{mm}(x) = 100 \times x^3/(10 + x^4)$. The models are grouped into three types defined by their dependency graphs and the positions of mm in the programs (see Appendix F.3). We ran our meta-algorithm using 600 programs from all three types using importance samples (not HMC samples). Then for 60 test programs from the last model type, we measured ESS and the sum of second moments along the wall-clock time using three approaches: importance sampling (IS-pred; ours) with the predicted posteriors as proposal, importance sampling (IS-prior) with prior as proposal, and HMC. As the reference sampler, we used importance sampling (IS-ref) with prior as proposal using 5M samples. All the approaches were repeated 10 times.

We report the results for a program (see Appendix L) in the test set. Fig. 4a and 4b show the moments estimated by HMC and IS-{pred, prior}, respectively, in comparison with the same (across the two figures) reference moments by IS-ref. The estimates by IS-pred (red) converged to the reference (green) more quickly (18ms) than those by HMC (orange; over 84s) and IS-prior (blue; higher variance). We obtained similar results for ESS, and the results are in Appendix M.

References

L. Ambrogioni, G. Silvestri, and M. van Gerven. Automatic variational inference with cascading flows. *arXiv preprint arXiv:2102.04801*, 2021.

C. Andrieu and J. Thoms. A tutorial on adaptive MCMC. *Stat. Comput.*, 18(4):343–373, 2008.

D. Bieber, C. Sutton, H. Larochelle, and D. Tarlow. Learning to execute programs with instruction pointer attention graph neural networks. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

E. Bingham, J. P. Chen, M. Jankowiak, F. Obermeyer, N. Pradhan, T. Karaletsos, R. Singh, P. Szerlip, P. Horsfall, and N. D. Goodman. Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, 2018.

B. Carpenter, A. Gelman, M. D. Hoffman, D. Lee, B. Goodrich, M. Betancourt, M. Brubaker, J. Guo, P. Li, and A. Riddell. Stan: A probabilistic programming language. *Journal of statistical software*, 76(1), 2017.

M. F. Cusumano-Towner, F. A. Saad, A. K. Lew, and V. 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, Phoenix, AZ, USA, June 22-26, 2019*, pages 221–236. ACM, 2019.

S. Duane, A. D. Kennedy, B. J. Pendleton, and D. Roweth. Hybrid Monte Carlo. *Physics letters B*, 195(2):216–222, 1987.

H. Ge, K. Xu, and Z. Ghahramani. Turing: Composable inference for probabilistic programming. In *International Conference on Artificial Intelligence and Statistics, AISTATS 2018, 9-11 April 2018, Playa Blanca, Lanzarote, Canary Islands, Spain*, volume 84 of *Proceedings of Machine Learning Research*, pages 1682–1690. PMLR, 2018.

S. Gershman and N. Goodman. Amortized inference in probabilistic reasoning. In *Proceedings of the annual meeting of the cognitive science society*, volume 36, 2014.

W. Gong, Y. Li, and J. M. Hernández-Lobato. Meta-learning for stochastic gradient MCMC. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

J. Goodman and J. Weare. Ensemble samplers with affine invariance. *Communications in applied mathematics and computational science*, 5(1):65–80, 2010.

N. D. Goodman, V. K. Mansinghka, D. Roy, K. Bonawitz, and J. B. Tenenbaum. Church: a language for generative models. In *Proceedings of the Twenty-Fourth Conference on Uncertainty in Artificial Intelligence*, pages 220–229, 2008.

J. Gordon, J. Bronskill, M. Bauer, S. Nowozin, and R. E. Turner. Meta-learning probabilistic inference for prediction. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

M. D. Hoffman and A. Gelman. The No-U-Turn sampler: adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.*, 15(1):1593–1623, 2014.

E. Iakovleva, J. Verbeek, and K. Alahari. Meta-learning with shared amortized variational inference. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event*, volume 119 of *Proceedings of Machine Learning Research*, pages 4572–4582. PMLR, 2020.

W. Jitkrittum, A. Gretton, N. Heess, S. M. A. Eslami, B. Lakshminarayanan, D. Sejdinovic, and Z. Szabó. Kernel-based just-in-time learning for passing expectation propagation messages. In *Proceedings of the Thirty-First Conference on Uncertainty in Artificial Intelligence, UAI 2015, July 12-16, 2015, Amsterdam, The Netherlands*, pages 405–414. AUAI Press, 2015.

D. P. Kingma and J. Ba. Adam: A method for stochastic optimization. In *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

D. P. Kingma and M. Welling. Auto-encoding variational Bayes. *arXiv preprint arXiv:1312.6114*, 2013.

T. A. Le, A. G. Baydin, and F. Wood. Inference compilation and universal probabilistic programming. In *Artificial Intelligence and Statistics*, pages 1338–1348. PMLR, 2017.

V. Mansinghka, D. Selsam, and Y. Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv preprint arXiv:1404.0099*, 2014.

J. Marino, Y. Yue, and S. Mandt. Iterative amortized inference. In *International Conference on Machine Learning*, pages 3403–3412, 2018.

L. Martino, V. Elvira, D. Luengo, and J. Corander. Layered adaptive importance sampling. *Statistics and Computing*, 27(3):599–623, 2017.

L. Mazare. ocaml-torch: OCaml bindings for pytorch, 2018. URL `https://github.com/LaurentMazare/ocaml-torch`.

T. Minka, J. Winn, J. Guiver, Y. Zaykov, D. Fabian, and J. Bronskill. /Infer.NET 0.3, 2018. Microsoft Research Cambridge. http://dotnet.github.io/infer.

T. P. Minka. Expectation propagation for approximate Bayesian inference. In *UAI '01: Proceedings of the 17th Conference in Uncertainty in Artificial Intelligence, University of Washington, Seattle, Washington, USA, August 2-5, 2001*, pages 362–369. Morgan Kaufmann, 2001.

A. Mnih and K. Gregor. Neural variational inference and learning in belief networks. In *International Conference on Machine Learning*, pages 1791–1799, 2014.

P. Narayanan, J. Carette, W. Romano, C. Shan, and R. Zinkov. Probabilistic inference by program transformation in hakaru (system description). In *International Symposium on Functional and Logic Programming - 13th International Symposium, FLOPS 2016, Kochi, Japan, March 4-6, 2016, Proceedings*, pages 62–79. Springer, 2016.

F. Pagani, M. Wiegand, and S. Nadarajah. An n-dimensional rosenbrock distribution for mcmc testing. *arXiv preprint arXiv:1903.09556*, 2019.

B. Paige and F. Wood. Inference networks for sequential Monte Carlo in graphical models. In *International Conference on Machine Learning*, pages 3040–3049, 2016.

S. E. Reed and N. de Freitas. Neural programmer-interpreters. In *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings*, 2016.

D. J. Rezende, S. Mohamed, and D. Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International Conference on Machine Learning*, pages 1278–1286, 2014.

D. Ritchie, B. Mildenhall, N. D. Goodman, and P. Hanrahan. Controlling procedural modeling programs with stochastically-ordered sequential Monte Carlo. *ACM Trans. Graph.*, 34(4), July 2015. ISSN 0730-0301.

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

J. Salvatier, T. V. Wiecki, and C. Fonnesbeck. Probabilistic programming in python using pymc3. *PeerJ Computer Science*, 2:e55, 2016.

N. Siddharth, B. Paige, J.-W. van de Meent, A. Desmaison, N. D. Goodman, P. Kohli, F. Wood, and P. Torr. Learning disentangled representations with semi-supervised deep generative models. In *Advances in Neural Information Processing Systems 30*, pages 5927–5937. Curran Associates, Inc., 2017.

A. Stuhlmüller, J. Taylor, and N. Goodman. Learning stochastic inverses. In *Advances in neural information processing systems*, pages 3048–3056, 2013.

D. Tran, A. Kucukelbir, A. B. Dieng, M. Rudolph, D. Liang, and D. M. Blei. Edward: A library for probabilistic modeling, inference, and criticism. *arXiv preprint arXiv:1610.09787*, 2016.

D. Tran, M. D. Hoffman, D. Moore, C. Suter, S. Vasudevan, and A. Radul. Simple, distributed, and accelerated probabilistic programming. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 7609–7620, 2018.

J.-W. van de Meent, B. Paige, H. Yang, and F. Wood. An introduction to probabilistic programming. *arXiv preprint arXiv:1809.10756*, 2018.

M. J. Wainwright and M. I. Jordan. Graphical models, exponential families, and variational inference. *Found. Trends Mach. Learn.*, 1(1-2):1–305, 2008.

H. Wang and J. Li. Adaptive gaussian process approximation for Bayesian inference with expensive likelihood functions. *Neural computation*, 30(11):3072–3094, 2018.

T. Wang, Y. Wu, D. Moore, and S. J. Russell. Meta-learning MCMC proposals. In *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 4150–4160, 2018.

F. Wood, J. W. van de Meent, and V. Mansinghka. A new approach to probabilistic programming inference. In *Proceedings of the 17th International conference on Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

M. Wu, K. Choi, N. D. Goodman, and S. Ermon. Meta-amortized variational inference and learning. In *AAAI*, pages 6404–6412, 2020.

Y. Yan, K. Swersky, D. Koutra, P. Ranganathan, and M. Hashemi. Neural execution engines: Learning to execute subroutines. In *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

W. Zaremba and I. Sutskever. Learning to execute. *CoRR*, abs/1410.4615, 2014. URL `http://arxiv.org/abs/1410.4615`.

A Related Work

The difficulty of developing an effective posterior-inference algorithm has motivated active research on adapting key components of an inference algorithm. Techniques for adjusting an MCMC proposal [Andrieu and Thoms, 2008] or an HMC integrator [Hoffman and Gelman, 2014] to a given inference task were implemented in popular tools. Recently, methods for meta-learning these techniques themselves from a collection of inference tasks have been developed [Wang et al., 2018, Gong et al., 2019]. The meta-learning approach also features in the work on stochastic variational inference (VI) where a variational distribution receives observations for each inference task and is trained with a collection of datasets of observations [Wu et al., 2020, Gordon et al., 2019, Iakovleva et al., 2020]. For a message-passing-style VI, such as expectation propagation [Minka, 2001, Wainwright and Jordan, 2008], Jitkrittum et al. [2015] studied the problem of learning a mechanism to pass a message for a given *single* inference task. A natural follow-up is to meta-learn such a mechanism from a dataset of *multiple* inference tasks that can generalise to *unseen* models. Our white-box inference algorithm can be viewed as a message-passing-style VI that can meta-learn the representation of messages and a mechanism for passing them for given probabilistic programs.

Amortised inference and inference compilation [Gershman and Goodman, 2014, Le et al., 2017, Paige and Wood, 2016, Stuhlmüller et al., 2013, Kingma and Welling, 2013, Mnih and Gregor, 2014, Rezende et al., 2014, Ritchie et al., 2016, Marino et al., 2018] are closely related to our approach in that they also attempt to learn a form of a posterior-inference algorithm. However, the learnt algorithm by them and that by ours have different scopes. The former is designed to work for unseen inputs or observations of a *single* model, while the latter for *multiple* models with different structures.

The idea of running programs with learnt neural networks also appears in the work on training neural networks to execute programs [Zaremba and Sutskever, 2014, Bieber et al., 2020, Reed and de Freitas, 2016]. As far as we know, however, we are the first to frame the problem of learning a posterior-inference algorithm as the one of learning to execute.

B Full Discussion about the Choice of the PPL

$$\begin{array}{rl} \textit{Programs } C ::= & A \mid C_1; C_2 \\ \textit{Atomic Commands } A ::= & z \sim \mathcal{N}(v_1, v_2) \mid \texttt{obs}(\mathcal{N}(v_0, v_1), r) \mid v_0 := \texttt{if } (v_1 > v_2)\, v_3 \texttt{ else } v_4 \\ & \mid \ v_0 := r \mid v_0 := v_1 \mid v_0 := p(v_1, v_2) \end{array}$$

Programs in the language are constructed by sequentially composing atomic commands. The language supports six types of *atomic commands*. The first type is $z \sim \mathcal{N}(v_1, v_2)$, which draws a sample from the normal distribution with mean v_1 and variance v_2, and assigns the sampled value to z. The second command, $\texttt{obs}(\mathcal{N}(v_0, v_1), r)$, states that a random variable is drawn from $\mathcal{N}(v_0, v_1)$ and its value is observed to be r. The next is a restricted form of a conditional statement that selects one of v_3 and v_4 depending on the result of the comparison $v_1 > v_2$. The following two commands are different kinds of assignments, one for assigning a constant and the other for copying a value from one variable to another.

We permit only the programs where a variable does not appear more than once on the left-hand side of the $:=$ and $\sim$ symbols. This means that no variable is updated twice or more, and it corresponds to the so-called static single assignment assumption in the work on compilers. This restriction lets us regard variables updated by $\sim$ as latent random variables. We denote those variables by $z_1, \ldots, z_n$.

We use this simple language for two reasons. First, the restriction imposed on our language enables the simple definition of our white-box inference algorithm. For instance, the language supports only a limited form of conditional statements and restricts the syntactic forms of atomic commands; the arguments to a normal distribution or to a procedure p should be variables, not more general expression forms such as addition of two variables. As we will show soon, this restriction makes it easy to exploit information about the type of each atomic command in our inference algorithm; we use different neural networks for different types of atomic commands in the algorithm. Second, the language is intended to serve as an intermediate language of a compiler for a high-level PPL, not the one to be used directly by the end user. The compilation scheme in, for instance, [van de Meent et al., 2018] from high-level probabilistic programs with general conditional statements and for loops to graphical models can be adopted to compile such programs into our language.

Programs with recursion or while loops cannot generally be translated into our intermediate language, since such programs may go into infinite loops while the programs in our language always terminate. Programs with for loops and general branches can in theory be translated into a less expressive language such as ours. For example, van de Meent et al. [2018] explain a language called FOPPL (Section 2), which has for loops and branches, and the translation of FOPPL into graphical models (Section 3). We think that these graphical models can be translated into programs in our language. Of course, this does not mean that the learnt inference algorithm would interact well with the compilation; the interaction between compilation and inference in the context of meta-learning is something to be explored in future work.

C Formal Semantics of the PPL

Probabilistic programs in the language denote unnormalised probability densities over $\mathbb{R}^n$ for some n. Specifically, for a program C, if $z_1, \ldots, z_n$ are all the variables assigned by the sampling statements $z_i \sim \mathcal{N}(\ldots)$ in C in that order and C contains m observe statements with observations $r_1, \ldots, r_m$, then C denotes an unnormalised density p_C over the real-valued random variables $z_1, \ldots, z_n$:

$$p_C(z_{1:n}) = p_C(x_{1:m} = r_{1:m} | z_{1:n}) \times \prod_{i=1}^{n} p_C(z_i | z_{1:i-1})$$

where $x_1, \ldots, x_m$ are variables not appearing in C and are used to denote observed variables. This density is defined inductively over the structure of C.

Here $z_1, \ldots, z_n$ are all the variables assigned by the sampling statements $z_i \sim \mathcal{N}(\ldots)$ in C in that order, the program C contains m observe statements with observations $r_1, \ldots, r_m$, and these observed random variables are denoted by $x_1, \ldots, x_m$. The goal of this section is to provide the details of our statement. That is, we describe the formal semantics of our probabilistic programming language, and from it, we derive a map from programs C to unnormalised densities p_C.

To define the formal semantics of programs in our language, we need a type system that tracks information about updated variables and observations, and also formalises the syntactic conditions that we imposed informally in the main text. The type system lets us derive the following judgements for programs C and atomic commands A:

$$(S, V, \alpha) \vdash_1 C : (T, W, \beta), \quad (S, V, \alpha) \vdash_2 A : (T, W, \beta),$$

where S and T are sequences of distinct variables, V and W are sets of variables that do not appear in S and T, respectively, and α and β are sequences of reals. The first judgement says that if before running the program C, the latent variables in S are sampled in that order, the program variables in V are updated by non-sample statements, and the real values in the sequence α are observed in that order, then running C changes these three data to T, W, and β. The second judgement means the same thing except that we consider the execution of A, instead of C. The triples (S, V, α) and (T, W, β) serve as types in this type system.

The rules for deriving the judgements for C and A follow from the intended meaning just explained. We show these rules below, using the notation @ for the concatenation operator for two sequences and also $\mathrm{set}(S)$ for the set of elements in the sequence S:

$$\frac{(R, U, \alpha) \vdash_1 C_1 : (S, V, \beta) \quad (S, V, \beta) \vdash_1 C_2 : (T, W, \gamma)}{(R, U, \alpha) \vdash_1 (C_1; C_2) : (T, W, \gamma)} \qquad \frac{(S, V, \alpha) \vdash_2 A : (T, W, \beta)}{(S, V, \alpha) \vdash_1 A : (T, W, \beta)}$$

$$\frac{z \notin \mathrm{set}(S) \cup V \quad v_1, v_2 \in \mathrm{set}(S) \cup V}{(S, V, \alpha) \vdash_2 (z \sim \mathcal{N}(v_1, v_2)) : (S@[z], V, \alpha)} \qquad \frac{v_0, v_1 \in \mathrm{set}(S) \cup V}{(S, V, \alpha) \vdash_2 \mathtt{obs}(\mathcal{N}(v_0, v_1), r) : (S, V, \alpha@[r])}$$

$$\frac{v_0 \notin \mathrm{set}(S) \cup V \quad v_1, v_2, v_3, v_4 \in \mathrm{set}(S) \cup V}{(S, V, \alpha) \vdash_2 (v_0 := \mathtt{if}\ (v_1 > v_2)\ v_3\ \mathtt{else}\ v_4) : (S, V \cup \{v_0\}, \alpha)}$$

$$\frac{v_0 \notin \mathrm{set}(S) \cup V}{(S, V, \alpha) \vdash_2 (v_0 := r) : (S, V \cup \{v_0\}, \alpha)} \qquad \frac{v_0 \notin \mathrm{set}(S) \cup V \quad v_1 \in \mathrm{set}(S) \cup V}{(S, V, \alpha) \vdash_2 (v_0 := v_1) : (S, V \cup \{v_0\}, \alpha)}$$

$$\frac{v_0 \notin \mathrm{set}(S) \cup V \quad v_1, v_2 \in \mathrm{set}(S) \cup V}{(S, V, \alpha) \vdash_2 (v_0 := p(v_1, v_2)) : (S, V \cup \{v_0\}, \alpha)}$$

We now define our semantics, which specifies mappings from judgements for C and A to mathematical entities. First, we interpret each type (S, V, α) as a set, and it is denoted by $[\![(S, V, \alpha)]\!]$:

$$\begin{aligned}[\![(S, V, \alpha)]\!] = \{(p, f, l) \mid\ & p \text{ is a (normalised) density on } \mathbb{R}^{|S|},\ f = (f_v)_{v \in \mathrm{set}(S) \cup V}, \\ & \text{each } f_v \text{ is a measurable map from } \mathbb{R}^{|S|} \text{ to } \mathbb{R}, \\ & l \text{ is a measurable function from } \mathbb{R}^{|S|} \times \mathbb{R}^{|\alpha|} \text{ to } \mathbb{R}_+\},\end{aligned}$$

where $|S|$ and $|\alpha|$ are the lengths of the sequences S and α, and $\mathbb{R}_+$ means the set of positive reals. Next, we define the semantics of the judgements $(S, V, \alpha) \vdash_1 C : (T, W, \beta)$ and $(S, V, \alpha) \vdash_2 A : (T, W, \beta)$ that can be derived by the rules from above. The formal semantics of these judgements, denoted by the $[\![-]\!]$ notation, are maps of the following type:

$$[\![(S, V, \alpha) \vdash_1 C : (T, W, \beta)]\!] : [\![(S, V, \alpha)]\!] \to [\![(T, W, \beta)]\!],$$
$$[\![(S, V, \alpha) \vdash_2 A : (T, W, \beta)]\!] : [\![(S, V, \alpha)]\!] \to [\![(T, W, \beta)]\!].$$

The semantics is given by induction on the size of the derivation of each judgement, under the assumption that for each procedure name $p \in \mathbb{P}$, we have its interpretation as a measurable map from $\mathbb{R}^2$ to $\mathbb{R}$:

$$[\![p]\!] : \mathbb{R}^2 \to \mathbb{R}.$$

We spell out the semantics below, first the one for programs and next that for atomic commands.

$$\begin{aligned}[\![(S, V, \alpha) \vdash_1 A : (T, W, \beta)]\!](p, f, l) &= [\![(S, V, \alpha) \vdash_2 A : (T, W, \beta)]\!](p, f, l), \\ [\![(R, U, \alpha) \vdash_1 (C_1; C_2) : (T, W, \gamma)]\!](p, f, l) &= ([\![(S, V, \beta) \vdash_2 C_2 : (T, W, \gamma)]\!] \\ &\qquad \circ [\![(R, U, \alpha) \vdash_2 C_1 : (S, V, \beta)]\!])(p, f, l).\end{aligned}$$

Let $\mathcal{N}(a; b, c)$ be the density of the normal distribution with mean b and variance c when $c > 0$ and 1 when $c \leq 0$. For a family of functions $f = (f_v)_{v \in V}$, a variable $w \notin V$, and a function f'_w, we write $f \oplus f'_w$ for the extension of f with a new w-indexed member f'_w.

$$\begin{aligned}&[\![(S, V, \alpha) \vdash_2 z \sim \mathcal{N}(v_1, v_2) : (S@[z], V, \alpha)]\!](p, f, l) = (p', f', l') \\ &\qquad (\text{where } p'(a_{1:|S|+1}) = p(a_{1:|S|}) \times \mathcal{N}(a_{|S|+1}; f_{v_1}(a_{1:|S|}), f_{v_2}(a_{1:|S|})), \\ &\qquad\qquad f'_v(a_{1:|S|+1}) = f_v(a_{1:|S|}) \text{ for all } v \in V,\ f'_z(a_{1:|S|+1}) = a_{|S|+1}, \\ &\qquad\qquad l'(a_{1:|S|+1}, b_{1:|\alpha|}) = l(a_{1:|S|}, b_{1:|\alpha|})),\end{aligned}$$

$$\begin{aligned}&[\![(S, V, \alpha) \vdash_2 \mathtt{obs}(\mathcal{N}(v_0, v_1), r) : (S, V, \alpha@[r])]\!](p, f, l) = (p, f, l') \\ &\qquad (\text{where } l'(a_{1:|S|}, b_{1:|\alpha|+1}) = l(a_{1:|S|}, b_{1:|\alpha|}) \times \mathcal{N}(b_{|\alpha|+1}; f_{v_1}(a_{1:|S|}), f_{v_2}(a_{1:|S|})),\end{aligned}$$

$$\begin{aligned}&[\![(S, V, \alpha) \vdash_2 (v_0 := \mathtt{if}\ (v_1 > v_2)\ v_3\ \mathtt{else}\ v_4) : (S, V \cup \{v_0\}, \alpha)]\!](p, f, l) = (p, f \oplus f'_{v_0}, l) \\ &\qquad (\text{where } f'_{v_0}(a_{1:|S|}) = \text{if } (f_{v_1}(a_{1:|S|}) > f_{v_2}(a_{1:|S|})) \text{ then } f_{v_3}(a_{1:|S|}) \text{ else } f_{v_4}(a_{1:|S|})),\end{aligned}$$

$$\begin{aligned}&[\![(S, V, \alpha) \vdash_2 (v_0 := r) : (S, V \cup \{v_0\}, \alpha)]\!](p, f, l) = (p, f \oplus f'_{v_0}, l) \\ &\qquad (\text{where } f'_{v_0}(a_{1:|S|}) = r),\end{aligned}$$

$$\begin{aligned}&[\![(S, V, \alpha) \vdash_2 (v_0 := v_1) : (S, V \cup \{v_0\}, \alpha)]\!](p, f, l) = (p, f \oplus f'_{v_0}, l) \\ &\qquad (\text{where } f'_{v_0}(a_{1:|S|}) = f_{v_1}(a_{1:|S|})),\end{aligned}$$

$$\begin{aligned}&[\![(S, V, \alpha) \vdash_2 (v_0 := p'(v_0, v_1)) : (S, V \cup \{v_0\}, \alpha)]\!](p, f, l) = (p, f \oplus f'_{v_0}, l) \\ &\qquad (\text{where } f'_{v_0}(a_{1:|S|}) = [\![p']\!](f_{v_0}(a_{1:|S|}), f_{v_1}(a_{1:|S|}))).\end{aligned}$$

Finally, we define p_C for the well-initialised well-typed programs C, i.e., programs C for which we can derived

$$([], \emptyset, []) \vdash_1 C : (S, V, \alpha).$$

For such a C, the definition of p_C is given below:

$$p_C(z_{1:|S|}) = p(z_{1:|S|}) \times l(z_{1:|S|}, \alpha)$$

where $(p, \_, l) = [\![([], \emptyset, []) \vdash_1 C : (S, V, \alpha)]\!](p_0, f_0, l_0)$ for the constant-1 functions p_0 and l_0 of appropriate types and the empty family f_0 of functions.

D Marginal Likelihood Computation: Derivation and Correctness

What $nn_{\text{intg},\phi}$ Aims at Computing When we write the meaning of this observe statement as the likelihood $\mathcal{N}(r; v_0, v_1)$, and the filtering distribution for v_0 and v_1 under (the decoded density of) the current state h as $p_h(v_0, v_1)$,[2] the neural network $nn_{\text{intg},\phi}$ computes the following approximation:

$$nn_{\text{intg},\phi}(\overline{v_{0:1}}, r, h) \approx \int \mathcal{N}(r; v_0, v_1) p_h(v_0, v_1) dv_0 dv_1$$

where $\overline{v_{0:1}}$ mean the one-hot encoded variables $v_{0:1}$.

Derivation and Correctness of the Marginal Likelihood Let x_n be the random variable (RV) that is observed by the command $\texttt{obs}(\mathcal{N}(v_0, v_1), r)$ and $x_{1:(n-1)}$ be the $(n-1)$ RVs that are observed before the command. When our algorithm is about to analyse this observe command, we have (an estimate of) $p(x_{1:(n-1)})$ by induction. Then, the marginal likelihood of $x_{1:n}$ can be computed as follows:

$$
\begin{aligned}
&p(x_{1:(n-1)}, x_n) \\
&= \iint p(x_{1:(n-1)}, x_n, v_0, v_1)\, dv_0\, dv_1 \\
&= \iint p(x_{1:(n-1)})\, p(v_0, v_1 | x_{1:(n-1)})\, p(x_n | x_{1:(n-1)}, v_0, v_1)\, dv_0\, dv_1 \\
&\approx p(x_{1:(n-1)}) \iint p_h(v_0, v_1)\, p(x_n | x_{1:(n-1)}, v_0, v_1)\, dv_0\, dv_1 \\
&\qquad \text{// The filtering distribution } p(v_0, v_1 | x_{1:(n-1)}) \text{ is approximated by } p_h. \\
&= p(x_{1:(n-1)}) \iint p_h(v_0, v_1)\, p(x_n | v_0, v_1)\, dv_0\, dv_1 \\
&\qquad \text{// The RV } x_n \text{ is conditionally independent of } x_{1:(n-1)} \text{ given } v_0, v_1. \\
&= p(x_{1:(n-1)}) \iint p_h(v_0, v_1)\, \mathcal{N}(r; v_0, v_1)\, dv_0\, dv_1 \\
&\qquad \text{// } p(x_{1:(n-1)}) \text{ is } Z \text{ in the description of } \textsc{infer}(A_i) \text{ in the main text, and the neural network} \\
&\qquad \text{// } nn_{\text{intg},\phi_7} \text{ aims at approximating the integral term accurately.}
\end{aligned}
$$

This derivation leads to the first equation in this appendix section.

In a setting of probabilistic programming where observations are allowed to be different in true and false branches, the marginal likelihood may fail to be defined, and such a setting is beyond the scope of our language. Using variables multiple times or having observe commands spread out in the program does not make differences in the derivation above.

E Description of Our Optimisation Algorithm

We optimise the objective by stochastic gradient descent. The key component of the optimisation is a gradient estimator derived as follows: $(\nabla_\phi \sum_{C \in \mathcal{D}} \mathrm{KL}[\pi_C || q_C] + \frac{\lambda}{2}(N_C - Z_C)^2) = (\sum_{C \in \mathcal{D}} \mathbb{E}_{z_{1:n} \sim \pi_C}[-\nabla_\phi \log q_C(z_{1:n})] - \lambda(N_C - Z_C)\nabla_\phi Z_C) \approx \sum_{C \in \mathcal{D}} -\widehat{L_{C,\phi}} - \lambda(\widehat{N_C} - Z_C)\nabla_\phi Z_C$. Here $\widehat{L_{C,\phi}}$ and $\widehat{N_C}$ are sample estimates of $\mathbb{E}_{z_{1:n} \sim \pi_C}[\nabla_\phi \log q_C(z_{1:n})]$ and the marginal likelihood, respectively. Both estimates can be computed using standard Monte-Carlo algorithms. For instance, we can run the self-normalising importance sampler with prior as proposal, and generate weighted samples $\{(w^{(j)}, z_{1:n}^{(j)})\}_{1 \leq j \leq M}$ for the unnormalised posterior p_C. Then, we can use these samples to compute the required estimates: $\widehat{N_C} = \frac{1}{M}\sum_{j=1}^{M} w^{(j)}$ and $\widehat{L_{C,\phi}} = \frac{1}{M}\sum_{j=1}^{M}(w^{(j)} \nabla_\phi \log q_C(z_{1:n}^{(j)}))/\widehat{N_C}$. Alternatively, we may run Hamiltonian Monte Carlo [Duane et al., 1987] to generate posterior samples, and use those samples to draw weighted importance samples using, for instance, the layered adaptive importance sampler [Martino et al., 2017]. Then, we compute $\widehat{L_{C,\phi}}$ using posterior samples,

[2]The $p_h(v_0, v_1)$ is a filtering distribution, not prior.

Table 1: Full list of the model classes in the empirical evaluation.

Evaluation	Model class	Description	Detail
Generalisation to new model parameters and observations	gauss	Gaussian models with a latent variable and an observation where the mean of the Gaussian likelihood is an affine transformation of the latent.	Appendix F.1.1
	hierl	Hierarchical models with three hierarchically structured latent variables.	Appendix F.1.2
	hierd	Hierarchical or multi-level models with both latent variables and data structured hierarchically where data are modelled as a regression of latent variables of different levels.	Appendix F.1.3
	cluster	Clustering models where five observations are clustered into two groups.	Appendix F.1.4
	milky and milkyo	Milky Way models, and their multiple-observations extension where five observations are made for each satellite galaxy.	Appendix F.1.5
	rb	Models with the Rosenbrock function, which is expressed as an external procedure.	Appendix F.1.6
Generalisation to new model structures	ext1	Models with three Gaussian variables and one deterministic variable storing the value of the function $\mathrm{nl}(x) = 50/\pi \times \arctan(x/10)$, where the models have 12 different types — four different dependency graphs of the variables, and three different positions of the deterministic nl variable for each of these graphs.	Appendix F.2.1 (and Fig. 5)
	ext2	Models with six Gaussian variables and one nl variable, which are grouped into five model types based on their dependency graphs.	Appendix F.2.2 (and Fig. 6)
Test-time efficiency in comparison with alternatives	mulmod	Three-variable models where two latent variables follow normal distributions and the other stores the value of the function $\mathrm{mm}(x) = 100 \times x^3/(10 + x^4)$. The models in this class are grouped into three types defined by their dependency graphs and the positions of mm in the programs.	Appendix F.3 (and Fig. 7)

and $\widehat{N_C}$ using weighted importance samples. Note that neither π_C in $\mathbb{E}_{z_{1:n}\sim\pi_C}[-\nabla_\phi \log q_C(z_{1:n})]$ nor N_C depends on the parameters ϕ. Thus, for each $C \in \mathcal{D}$, N_C needs to be estimated only once throughout the entire optimisation process, and the posterior samples from π_C need to be generated only once as well. We use this fact to optimise the computation of each gradient-update step.

F Detailed Descriptions for Probabilistic Models Used in the Empirical Evaluation

Table 1 shows the full list of the model classes that we considered in our empirical evaluation (§3). We detail the program specifications for the classes using our probabilistic programming language, and then describe how our program generator generated programs from those classes randomly.

In the program specifications to follow, randomly-generated constants are written in the Greek alphabets (θ), and latent and other program variables in the English alphabets. Also, we often use more intuitive variable names instead of using z_i for latent variables and v_i for the other program variables, to improve readability. When describing random generation of the parameter values, we let $\mathrm{U}(a, b)$ denote the uniform distribution whose domain is $(a, b) \subset \mathbb{R}$; we use this only for describing the random program generation process itself, not the generated programs (only normal distributions are used in our programs, with the notation $\mathcal{N}$).

F.1 Generalisation to New Model Parameters and Observations

F.1.1 gauss

The model class is described as follows:

$$\begin{aligned}&m_z := \theta_1;\ v_z := \theta_2';\ c_1 := \theta_3;\ c_2 := \theta_4;\ v_x := \theta_5';\\&z_1 \sim \mathcal{N}(m_z, v_z);\ z_2 := z_1 \times c_1;\ z_3 := z_2 + c_2;\\&\texttt{obs}(\mathcal{N}(z_3, v_x), o)\end{aligned}$$

For each program of the class, our random program generator generated the parameter values as follows:

$$\begin{aligned}&\theta_1 \sim \mathrm{U}(-5,5),\ \theta_2 \sim \mathrm{U}(0,20),\ \theta_2' = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(-3,3)\\&\theta_4 \sim \mathrm{U}(-10,10),\ \theta_5 \sim \mathrm{U}(0.5,10),\ \theta_5' = (\theta_5)^2\end{aligned}$$

and then generated the observation o by running the program forward where the value for z_1 was sampled from $z_1 \sim \mathrm{U}(m_z - 2 \times \sqrt{v_z}, m_z + 2 \times \sqrt{v_z})$.

F.1.2 hierl

The model class is described as follows:

$$\begin{aligned}&m_g := \theta_1;\ v_g := \theta_2';\ v_{t_1} := \theta_3';\ v_{t_2} := \theta_4';\ v_{x_1} := \theta_5';\\&v_{x_2} := \theta_6';\ g \sim \mathcal{N}(m_g, v_g);\ t_1 \sim \mathcal{N}(g, v_{t_1});\ t_2 \sim \mathcal{N}(g, v_{t_2});\\&\texttt{obs}(\mathcal{N}(t_1, v_{x_1}), o_1);\ \texttt{obs}(\mathcal{N}(t_2, v_{x_2}), o_2)\end{aligned}$$

For each program of the class, our generator generated the parameter values as follows:

$$\begin{aligned}&\theta_1 \sim \mathrm{U}(-5,5),\ \theta_2 \sim \mathrm{U}(0,50),\ \theta_2' = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(0,10)\\&\theta_3' = (\theta_3)^2,\ \theta_4 \sim \mathrm{U}(0,10),\ \theta_4' = (\theta_4)^2,\ \theta_5 \sim \mathrm{U}(0.5,10)\\&\theta_5' = (\theta_5)^2,\ \theta_6 \sim \mathrm{U}(0.5,10),\ \theta_6' = (\theta_6)^2\end{aligned}$$

and then generated the observations o_1 and o_2 by running the program (i.e., simulating the model) forward.

F.1.3 hierd

The model class is described as follows:

$$\begin{aligned}&m_{a_0} := \theta_1;\ v_{a_0} := \theta_2';\ v_{a_1} := \theta_3';\ v_{a_2} := \theta_4';\ m_b := \theta_5;\\&v_b := \theta_6';\ d_1 = \theta_7;\ d_2 = \theta_8;\ v_{x_1} := \theta_9';\ v_{x_2} := \theta_{10}';\\&a_0 \sim \mathcal{N}(m_{a_0}, v_{a_0});\ a_1 \sim \mathcal{N}(a_0, v_{a_1});\ a_2 \sim \mathcal{N}(a_0, v_{a_2});\\&b \sim \mathcal{N}(m_b, v_b);\\&t_1 := b \times d_1;\ t_2 := a_1 + t_1;\ \texttt{obs}(\mathcal{N}(t_2, v_{x_1}), o_1);\\&t_3 := b \times d_2;\ t_4 := a_2 + t_3;\ \texttt{obs}(\mathcal{N}(t_4, v_{x_2}), o_2)\end{aligned}$$

For each program of the class, our generator generated the parameter values as follows:

$$\begin{aligned}&\theta_1 \sim \mathrm{U}(-10,10),\ \theta_2 \sim \mathrm{U}(0,100),\ \theta_2' = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(0,10)\\&\theta_3' = (\theta_3)^2,\ \theta_4 \sim \mathrm{U}(0,10),\ \theta_4' = (\theta_4)^2,\ \theta_5 \sim \mathrm{U}(-5,5)\\&\theta_6 \sim \mathrm{U}(0,10),\ \theta_6' = (\theta_6)^2,\ \theta_7 \sim \mathrm{U}(-5,5),\ \theta_8 \sim \mathrm{U}(-5,5)\\&\theta_9 \sim \mathrm{U}(0.5,10),\ \theta_9' = (\theta_9)^2,\ \theta_{10} \sim \mathrm{U}(0.5,10),\ \theta_{10}' = (\theta_{10})^2\end{aligned}$$

and then generated the observations o_1 and o_2 by running the program forward where the values for a_0, a_1, a_2, and b in this specific simulation were sampled as follows:

$$\begin{aligned}&a_0 \sim \mathrm{U}(m_{a_0} - 2 \times \sqrt{v_{a_0}},\ m_{a_0} + 2 \times \sqrt{v_{a_0}})\\&a_1 \sim \mathrm{U}(a_0 - 2 \times \sqrt{v_{a_1}},\ a_0 + 2 \times \sqrt{v_{a_1}})\\&a_2 \sim \mathrm{U}(a_0 - 2 \times \sqrt{v_{a_2}},\ a_0 + 2 \times \sqrt{v_{a_2}})\\&b \sim \mathrm{U}(m_b - 2 \times \sqrt{v_b},\ m_b + 2 \times \sqrt{v_b})\end{aligned}$$

F.1.4 cluster

The model class is described as follows:

$$\begin{aligned}
&m_{g_1} := \theta_1;\ v_{g_1} := \theta'_2;\ m_{g_2} := \theta_3;\ v_{g_2} := \theta'_4;\ v_x := \theta'_5;\\
&g_1 \sim \mathcal{N}(m_{g_1}, v_{g_1});\ g_2 \sim \mathcal{N}(m_{g_2}, v_{g_2});\\
&\mathit{zero} := 0;\ \mathit{hund} := 100;\\
&t_1 \sim \mathcal{N}(\mathit{zero}, \mathit{hund});\ m_1 := \texttt{if}\ (t_1 > \mathit{zero})\ g_1\ \texttt{else}\ g_2;\\
&\texttt{obs}(\mathcal{N}(m_1, v_x), o_1);\\
&t_2 \sim \mathcal{N}(\mathit{zero}, \mathit{hund});\ m_2 := \texttt{if}\ (t_2 > \mathit{zero})\ g_1\ \texttt{else}\ g_2;\\
&\texttt{obs}(\mathcal{N}(m_2, v_x), o_2);\\
&t_3 \sim \mathcal{N}(\mathit{zero}, \mathit{hund});\ m_3 := \texttt{if}\ (t_3 > \mathit{zero})\ g_1\ \texttt{else}\ g_2;\\
&\texttt{obs}(\mathcal{N}(m_3, v_x), o_3);\\
&t_4 \sim \mathcal{N}(\mathit{zero}, \mathit{hund});\ m_4 := \texttt{if}\ (t_4 > \mathit{zero})\ g_1\ \texttt{else}\ g_2;\\
&\texttt{obs}(\mathcal{N}(m_4, v_x), o_4);\\
&t_5 \sim \mathcal{N}(\mathit{zero}, \mathit{hund});\ m_5 := \texttt{if}\ (t_5 > \mathit{zero})\ g_1\ \texttt{else}\ g_2;\\
&\texttt{obs}(\mathcal{N}(m_5, v_x), o_5)
\end{aligned}$$

For each program of the class, our generator generated the parameter values as follows:

$$\begin{aligned}
&\theta_1 \sim \mathrm{U}(-15, 15),\ \theta_2 \sim \mathrm{U}(0.5, 50),\ \theta'_2 = (\theta_2)^2\\
&\theta_3 \sim \mathrm{U}(-15, 15),\ \theta_4 \sim \mathrm{U}(0.5, 50),\ \theta'_4 = (\theta_4)^2\\
&\theta_5 \sim \mathrm{U}(0.5, 10),\ \theta'_5 = (\theta_5)^2
\end{aligned}$$

and then generated the observations $o_{1:5}$ by running the program forward.

F.1.5 milky **and** milkyo

The model class milky is described as follows:

$$\begin{aligned}
&m_{\mathit{mass}} := \theta_1;\ v_{\mathit{mass}} := \theta'_2;\ c_1 := \theta_3;\ v_{g_1} := \theta'_4;\ c_2 := \theta_5;\\
&v_{g_2} := \theta'_6;\ v_{x_1} := \theta'_7;\ v_{x_2} := \theta'_8;\\
&\mathit{mass} \sim \mathcal{N}(m_{\mathit{mass}}, v_{\mathit{mass}});\\
&\mathit{mass}_1 := \mathit{mass} \times c_1;\ g_1 \sim \mathcal{N}(\mathit{mass}_1, v_{g_1});\\
&\mathit{mass}_2 := \mathit{mass} + c_2;\ g_2 \sim \mathcal{N}(\mathit{mass}_2, v_{g_2});\\
&\texttt{obs}(\mathcal{N}(g_1, v_{x_1}), o_1);\ \texttt{obs}(\mathcal{N}(g_2, v_{x_2}), o_2)
\end{aligned}$$

For each program of milky, our generator generated the parameter values as follows:

$$\begin{aligned}
&\theta_1 \sim \mathrm{U}(-10, 10),\ \theta_2 \sim \mathrm{U}(0, 30),\ \theta'_2 = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(-2, 2)\\
&\theta_4 \sim \mathrm{U}(0, 10),\ \theta'_4 = (\theta_4)^2,\ \theta_5 \sim \mathrm{U}(-5, 5),\ \theta_6 \sim \mathrm{U}(0, 10)\\
&\theta'_6 = (\theta_6)^2,\ \theta_7 \sim \mathrm{U}(0.5, 10),\ \theta'_7 = (\theta_7)^2,\ \theta_8 \sim \mathrm{U}(0.5, 10)\\
&\theta'_8 = (\theta_8)^2
\end{aligned}$$

and then generated the observations o_1 and o_2 by running the program forward.

Everything remained the same for the milkyo class, except that the two obs commands were extended to $\texttt{obs}(\mathcal{N}(g_1, v_{x_1}), [o_1, o_2, o_3, o_4, o_5])$ and $\texttt{obs}(\mathcal{N}(g_2, v_{x_2}), [o_6, o_7, o_8, o_9, o_{10}])$, respectively, and all the observations were generated similarly by running the extended model forward.

F.1.6 rb

The model class rb is described as follows:

$$\begin{aligned}
&m_{z_1} := \theta_1;\ v_{z_1} := \theta'_2;\ m_{z_2} := \theta_3;\ v_{z_2} := \theta'_4;\ v_x := \theta'_5;\\
&z_1 \sim \mathcal{N}(m_{z_1}, v_{z_1});\ z_2 \sim \mathcal{N}(m_{z_2}, v_{z_2});\ r := \mathrm{Rosenbrock}(z_1, z_2);\\
&\texttt{obs}(\mathcal{N}(r, v_x), o)
\end{aligned}$$

where $\mathrm{Rosenbrock}(z_1, z_2) = 0.05 \times (z_1 - 1)^2 + 0.005 \times (z_2 - {z_1}^2)^2$. The function is often used to evaluate learning and inference algorithms [Goodman and Weare, 2010, Wang and Li, 2018, Pagani et al., 2019]. For each program of the class, our generator generated the parameter values as follows:

$$\theta_1 \sim \mathrm{U}(-8, 8),\ \theta_2 \sim \mathrm{U}(0, 5),\ \theta_2' = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(-8, 8)$$
$$\theta_4 \sim \mathrm{U}(0, 5),\ \theta_4' = (\theta_4)^2,\ \theta_5 \sim \mathrm{U}(0.5, 10),\ \theta_5' = (\theta_5)^2$$

and then generated the observation o by running the program forward where the values for z_1 and z_2 in this specific simulation were sampled as follows:

$$z_1 \sim \mathrm{U}(m_{z_1} - 1.5 \times \sqrt{v_{z_1}},\ m_{z_1} + 1.5 \times \sqrt{v_{z_1}})$$
$$z_2 \sim \mathrm{U}(m_{z_2} - 1.5 \times \sqrt{v_{z_2}},\ m_{z_2} + 1.5 \times \sqrt{v_{z_2}})$$

F.2 Generalisation to New Model Structures

For readability, we present canonicalised dependency graphs where variables are named in the breadth-first order. In the experiments reported in the corresponding part of the main text, we used a minor extension of our probabilistic programming language with procedures taking one parameter.

F.2.1 ext1

Fig. 5 shows the dependency graphs for all model types in ext1. The variables $z_0, z_1, \ldots$ and $x_1, x_2, \ldots$ represent latent and observed variables, respectively, and observed variables are colored in gray. The red node in each graph represents the position of the nl variable.

Our program generator in this case generates programs from the whole model class ext1; it generates programs of all twelve different types in ext1. We explain this generation process for the model type (1,1) in Fig. 5, while pointing out that the similar process is applied to the other eleven types. To generate programs of the model type (1,1), we use the following program template:

$$m_{z_0} := \theta_1;\ v_{z_0} := \theta_2';\ v_{z_2} := \theta_3';\ v_{z_3} := \theta_4';\ v_{x_1} := \theta_5';$$
$$z_0 \sim \mathcal{N}(m_{z_0}, v_{z_0});\ z_1 := \mathrm{nl}(z_0);\ z_2 \sim \mathcal{N}(z_1, v_{z_2});\ z_3 \sim \mathcal{N}(z_2, v_{z_3});$$
$$\mathtt{obs}(\mathcal{N}(z_3, v_{x_1}), o_1)$$

where $\mathrm{nl}(z) = 50/\pi \times \arctan(z/10)$. The generation involves randomly sampling the parameters of this template, converting the template into a program in our language, and creating synthetic observations. Specifically, our generator generates the parameter values as follows:

$$\theta_1 \sim \mathrm{U}(-5, 5),\ \theta_2 \sim \mathrm{U}(0, 20),\ \theta_2' = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(0, 20),\ \theta_3' = (\theta_3)^2$$
$$\theta_4 \sim \mathrm{U}(0, 20),\ \theta_4' = (\theta_4)^2,\ \theta_5 \sim \mathrm{U}(0.5, 10),\ \theta_5' = (\theta_5)^2$$

and generates the observation o_1 by running the program forward where the values for $z_{0:3}$ in this specific simulation were sampled (and fixed to specific values) as follows:

$$z_0 \sim \mathrm{U}(m_{z_0} - 2 \times \sqrt{v_{z_0}},\ m_{z_0} + 2 \times \sqrt{v_{z_0}})$$
$$z_1 = \mathrm{nl}(z_0)$$
$$z_2 \sim \mathrm{U}(z_1 - 2 \times \sqrt{v_{z_2}},\ z_1 + 2 \times \sqrt{v_{z_2}})$$
$$z_3 \sim \mathrm{U}(z_2 - 2 \times \sqrt{v_{z_3}},\ z_2 + 2 \times \sqrt{v_{z_3}}).$$

The generator uses different templates for the other eleven model types in ext1, while sharing the similar process for generation of the parameters and observations.

F.2.2 ext2

Fig. 6 shows the dependency graphs for all five model types in ext2. Programs of these five types are randomly generated by our program generator. As in the ext1 case, we explain the generator only for one model type, which corresponds to the first dependency graph in Fig. 6. To generate programs of this type, we use the following program template:

$$m_{z_0} := \theta_1;\ v_{z_0} := \theta_2';\ v_{z_1} := \theta_3';\ v_{z_3} := \theta_4';\ v_{z_4} := \theta_5';\ v_{z_5} := \theta_6';\ v_{z_6} := \theta_7';$$
$$v_{x_1} := \theta_8';\ v_{x_2} := \theta_9';\ v_{x_3} := \theta_{10}';\ v_{x_4} := \theta_{11}';$$

$$z_0 \sim \mathcal{N}(m_{z_0}, v_{z_0});\ z_1 \sim \mathcal{N}(z_0, v_{z_1});\ z_2 := \mathrm{nl}(z_0);\ z_3 \sim \mathcal{N}(z_0, v_{z_3});$$
$$z_4 \sim \mathcal{N}(z_1, v_{z_4});\ z_5 \sim \mathcal{N}(z_1, v_{z_5});\ z_6 \sim \mathcal{N}(z_2, v_{z_6});$$
$$\texttt{obs}(\mathcal{N}(z_4, v_{x_1}), o_1);\ \texttt{obs}(\mathcal{N}(z_5, v_{x_2}), o_2);\ \texttt{obs}(\mathcal{N}(z_6, v_{x_3}), o_3);\ \texttt{obs}(\mathcal{N}(z_3, v_{x_4}), o_4)$$

In order to generate a program of this model type and observations, our generator instantiates the parameters of the template as follows:

$$\theta_1 \sim \mathrm{U}(-5,5),\ \theta_2 \sim \mathrm{U}(0,10),\ \theta_2' = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(0,10),\ \theta_3' = (\theta_3)^2,\ \theta_4 \sim \mathrm{U}(0,10),\ \theta_4' = (\theta_4)^2$$
$$\theta_5 \sim \mathrm{U}(0,10),\ \theta_5' = (\theta_5)^2,\ \theta_6 \sim \mathrm{U}(0,10),\ \theta_6' = (\theta_6)^2,\ \theta_7 \sim \mathrm{U}(0,10),\ \theta_7' = (\theta_7)^2$$
$$\theta_8 \sim \mathrm{U}(0,10),\ \theta_8' = (\theta_8)^2,\ \theta_9 \sim \mathrm{U}(0,10),\ \theta_9' = (\theta_9)^2,\ \theta_{10} \sim \mathrm{U}(0,10),\ \theta_{10}' = (\theta_{10})^2$$
$$\theta_{11} \sim \mathrm{U}(0,10),\ \theta_{11}' = (\theta_{11})^2.$$

Then, it generates the observations $o_{1:4}$ by running the program forward where the values for $z_{0:6}$ in this specific simulation were sampled (and fixed to specific values) as follows:

$$\begin{aligned}
z_0 &\sim \mathrm{U}(m_{z_0} - 2 \times \sqrt{v_{z_0}},\ m_{z_0} + 2 \times \sqrt{v_{z_0}}) \\
z_1 &\sim \mathrm{U}(z_0 - 2 \times \sqrt{v_{z_1}},\ z_0 + 2 \times \sqrt{v_{z_1}}) \\
z_2 &= \mathrm{nl}(z_0) \\
z_3 &\sim \mathrm{U}(z_0 - 2 \times \sqrt{v_{z_3}},\ z_0 + 2 \times \sqrt{v_{z_3}}) \\
z_4 &\sim \mathrm{U}(z_1 - 2 \times \sqrt{v_{z_4}},\ z_1 + 2 \times \sqrt{v_{z_4}}) \\
z_5 &\sim \mathrm{U}(z_1 - 2 \times \sqrt{v_{z_5}},\ z_1 + 2 \times \sqrt{v_{z_5}}) \\
z_6 &\sim \mathrm{U}(z_2 - 2 \times \sqrt{v_{z_6}},\ z_2 + 2 \times \sqrt{v_{z_6}}).
\end{aligned}$$

The generator uses different templates for the other four model types in ext2, while sharing the similar process for generation of the parameters and observations.

F.3 Test-Time Efficiency in Comparison with Alternatives

This section details the mulmod class, which has three different model types. Fig. 7 shows the dependency graphs for all the model types. The red node in each graph represents the position of the mm variable. We used all the three types in training, applied the learnt inference algorithm to programs in the third model type, and compared the results with those returned by HMC.

We similarly explain the generator only using the model type corresponding to the first dependency graph in Fig. 7. To generate programs of this type, we use the following program template:

$$m_{z_0} := \theta_1;\ v_{z_0} := \theta_2';\ v_{z_1} := \theta_3';\ v_{x_1} := \theta_4';$$
$$z_0 \sim \mathcal{N}(m_{z_0}, v_{z_0});\ z_1 \sim \mathcal{N}(z_0, v_{z_1});\ z_2 := \mathrm{mm}(z_1);\ \texttt{obs}(\mathcal{N}(z_2, v_{x_1}), o_1)$$

where $\mathrm{mm}(x) = 100 \times x^3/(10 + x^4)$. For each program in this model type, our generator instantiates the parameter values as follows:

$$\theta_1 \sim \mathrm{U}(-5,5),\ \theta_2 \sim \mathrm{U}(0,20),\ \theta_2' = (\theta_2)^2,\ \theta_3 \sim \mathrm{U}(0,20),\ \theta_3' = (\theta_3)^2$$
$$\theta_4 \sim \mathrm{U}(0.5,10),\ \theta_4' = (\theta_4)^2$$

and synthesises the observation o_1 by running the program forward where the values for $z_{0:2}$ in this specific simulation were sampled (and fixed to specific values) as follows:

$$\begin{aligned}
z_0 &\sim \mathrm{U}(m_{z_0} - 2 \times \sqrt{v_{z_0}},\ m_{z_0} + 2 \times \sqrt{v_{z_0}}) \\
z_1 &\sim \mathrm{U}(z_0 - 2 \times \sqrt{v_{z_1}},\ z_0 + 2 \times \sqrt{v_{z_1}}) \\
z_2 &= \mathrm{mm}(z_1).
\end{aligned}$$

The generator uses different templates for the other two model types in mulmod, while sharing the similar process for instantiation of the parameters and observations.

G Detailed Evaluation Setup

We implemented our inference algorithm and meta-algorithm using ocaml-torch [Mazare, 2018], an OCaml binding for PyTorch. For HMC, we used the Python interface for Stan [Carpenter et al., 2017]. We used a Ubuntu server with Intel(R) Xeon(R) Gold 6234 CPU @ 3.30GHz with 16 cores, 32 threads, and 263G memory.

In our evaluation, the dimension s of the internal state h was 10 (i.e., $h \in \mathbb{R}^{10}$). We used the same neural network architecture for all the neural network components of our inference algorithm INFER. Each neural network had three linear layers and used the $\tanh$ activation. The hidden dimension was 10 for each layer in all the neural networks except for nn_{de} where the hidden dimensions were 50. The hyper-parameter in our optimisation objective was set to $\lambda = 2$ in the evaluation. For HMC, we used the NUTS sampler [Hoffman and Gelman, 2014].

G.1 Detailed Training Setup for Generalisation to New Model Parameters and Observations in §3

For each training program, our meta-algorithm used 2^{15} samples from the analytic (for gauss) or approximate (for the rest, by HMC) posterior distribution for the program.[3] Similarly, our meta-algorithm computed the marginal likelihood analytically (for gauss) or approximately (for the rest) using layered adaptive importance sampling [Martino et al., 2017] where the proposals were defined by an HMC chain. We performed mini-batch training with the batch size 2^{12}. We used Adam [Kingma and Ba, 2015] with its hyperparameters $\{\beta_1 = 0.9,\ \beta_2 = 0.999,\ \text{weight\_decay} = 0\}$ and 0.001 as the initial learning rate. We repeated the same experiments three times using different random seeds, and the training stopped when the average training loss converged enough.

G.2 Detailed Setup for Generalisation to New Model Structures in §3

We let two structural aspects vary across programs: the dependency (or data-flow) graph for the variables of a program and the position of a nonlinear function in the program. Specifically, we considered two kinds of models: (1) models (ext1) with three Gaussian variables and one deterministic variable storing the value of the function $\mathrm{nl}(x) = 50/\pi \times \arctan(x/10)$, where the models have 12 different types — four different dependency graphs of the variables, and three different positions of the deterministic nl variable for each of these graphs; and (2) models (ext2) with six Gaussian variables and one nl variable, which are grouped into five types based on their dependency graphs. See Fig. 5 and 6 in the appendix for visualisation of the different types in ext1 and ext2, respectively.

Before running our inference algorithm, we canonicalise the names of variables in a given program based on its dependency (i.e., data-flow) graph. Although not perfect, this preprocessing removes a superficial difference between programs caused by different variable names, and enables us to avoid the unnecessary complexity caused by variable-renaming symmetries at training and inference times.

For ext1, we ran seven different experiments. Four of them evaluated generalisation to unseen dependency graphs. Among all the four possible dependency graphs of programs we considered, these experiments used only three of them during training, and the other for testing: the dependency graph of each training program is one of these three (with the nl variable positioned in all the three possible places in the graph), and that of each test program is the other dependency graph. The remaining three experiments evaluated generalisation to unseen positions of the nl variable where we fixed the position of the nl variable in the four dependency graphs, and used programs with the variable in one of those fixed positions for testing, while using programs with the variables in all the other positions for training. For ext2, we ran five different experiments where each of them tested generalisation to an unseen dependency graph after training with the other four types of dependency graphs. All of these experiments were repeated three times under different random seeds. So, the total numbers of experiment runs were $21\ (= 7 \times 3)$ and $15\ (= 5 \times 3)$ for ext1 and ext2, respectively.

In each experiment run for ext1, we used 720 programs for training, and 90 (when generalising to new graphs) or 100 (when generalising to new positions of the nl variable) unseen programs for testing. In each run for ext2, we used 600 programs for training and tested the learnt inference algorithm on 50 unseen programs. We ran HMC to estimate posteriors and marginal likelihoods, and used 200K samples after 10K warmups to compute reference posteriors. We stopped training after giving enough time for convergence within a limit of computational resources. The rest was the same as in Appendix G.1.

[3] Except for rb; see the discussion on Rosenbrock models in Appendix I.

H Losses for hierd, cluster, milkyo, and rb

Fig. 8 shows the average training and test losses under three random seeds for hierd, cluster, milkyo, and rb. The later part of Fig. 8a, 8c and 8d shows cases where the test loss surges. This was when the loss of only a few programs in the test set (of 50 programs) became large. Even in this situation, the losses of the rest remained small. We give analyses for cluster and rb separately in §I.

I Multimodal Posteriors: cluster and rb

The cluster and rb classes posed another challenge: the models often had multimodal posteriors, and it was significantly harder for our meta-algorithm to learn an optimal inference algorithm. To make the evaluation partially feasible for rb, we changed two parts of our meta-algorithm slightly, as well as increasing the size of the test set from 50 to 100. First, we used importance samples instead of samples by HMC, which often failed to converge, to learn an inference algorithm. Second, our random program generator placed some restriction on the programs it generated (e.g., by using tight boundaries on some model parameters), guided by the analysis of the geometry of the Rosenbrock function [Pagani et al., 2019]. Consequently, HMC (with 500K samples after 50K warmups) failed to converge for only one fifth of the test programs.

Fig. 9 shows the similar comparison plots between reference and predicted marginal posteriors for 10 test programs of the rb type, after 52.4K epochs. Our inference algorithm computed the posteriors precisely for most of the programs except two (pgm75 and pgm79) with significant multimodality. The latent variable pgm75_z0 had at least two modes at around -10 (visible in the figure) and around 10 (hidden in the figure)[4]. Our inference algorithm showed a mode-seeking behavior for this latent variable. Similarly, the variable pgm79_z0 had at least two modes in the similar domain region (one shown and one hidden), but this time our inference algorithm showed a mode-covering behavior.

The multimodality issue raises two questions. First, how can our meta-algorithm generate samples from the posterior more effectively so that it can optimise the inference algorithm for classes of models with multimodal posteriors? For example, our current results for cluster suffer from the fact that the samples used in the training are often biased (i.e., only from a single mode of the posterior). One possible direction would be to use multiple Markov chains simultaneously and apply ideas from the mixing-time research. Second, how can our white-box inference algorithm catch more information from the program description and find non-trivial properties that may be useful for computing the posterior distributions having multiple modes? We leave the answers for future work.

J Training and Test Losses for ext1

Fig. 10 shows the average training and test losses in the ext1 experiment runs (under different random seeds).

K Training and Test Losses for ext2

Fig. 11 shows the average training and test losses in the ext2 experiment runs (under different random seeds).

L Program from mulmod That was Reported in the Test-Time Comparisons in §3

Fig. 12 shows the program that is reported in the test-time comparisons in our evaluation, written in our probabilistic programming language.

[4]The blue reference plots were drawn using an HMC chain, but the HMC chain got stuck in the mode around -10 for this variable.

Table 2: ESS by IS-pred, compared with those by HMC and IS-prior. For HMC, two kinds of ESS were computed (bulk and tail [Carpenter et al., 2017]) for each latent variable, and we report the maximum among them. For IS-{pred, prior}, ESS was averaged over the 10 trials. The elapsed time for all the three approaches was averaged over 10 trials.

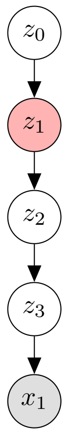
	HMC	IS-pred	IS-prior
ESS	80.0	$\mathbf{16,030.1}$	$1,364.1$
Elapsed time	84.2s	$\approx$ **18ms**	≈ 18ms
Sample size	1M	70K	100K

M ESS Results in the Comparisons with Alternatives in Terms of Test-Time Efficiency in §3

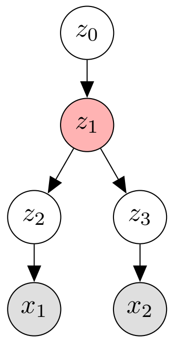
Table 2 shows ESS by the same runs of the three approaches as in the last columns of Fig. 4a and 4b in §3. IS-pred produced over 16K effective samples in 18ms, while HMC generated only 80 effective samples even after 84s. Similarly, IS-pred produced effective samples 10 times more than IS-prior in the approximately same elapsed time. In fact, Fig. 13 shows that as the time increases, the gap between the ESSes of IS-pred and IS-prior gets widen, because the former increases at a rate significantly higher than the latter.

N Limitation and Future Work

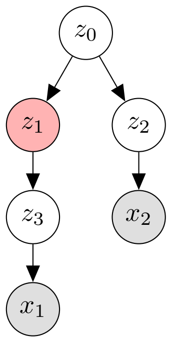
A learnt inference algorithm in this work does not generalise to programs with different sizes [Yan et al., 2020], e.g., from clustering models with two clusters to those with ten clusters. Each model class assumes a fixed number of variables, and the neural networks crucially exploit the assumption. Second, our meta-algorithm does not scale in practice to large programs, e.g., state-space models with a few hundred time steps, and cannot learn an optimal inference algorithm within a reasonable amount of time. Third, it is a good future work to relax the (meanfield Gaussian) assumption on the approximating distribution in our inference algorithm, and to meta-learn the form of the approximation itself by, e.g., incorporating the ideas proposed by Ambrogioni et al. [2021]. This extension is closely related to automatic guide generation in the Pyro community [Bingham et al., 2018].



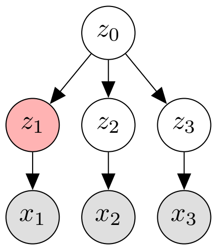
(a) Model type (1,1).



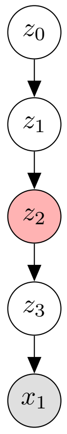
(b) Model type (1,2).



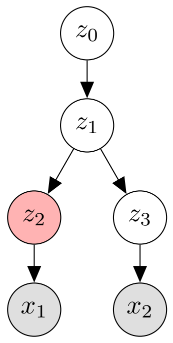
(c) Model type (1,3).



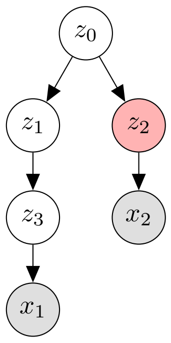
(d) Model type (1,4).



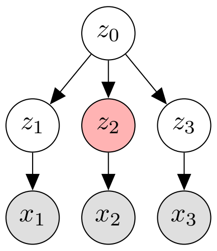
(e) Model type (2,1).



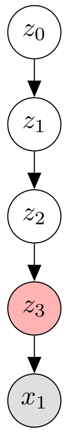
(f) Model type (2,2).



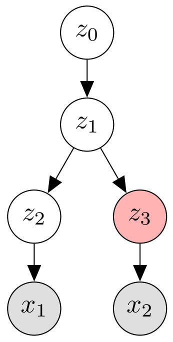
(g) Model type (2,3).



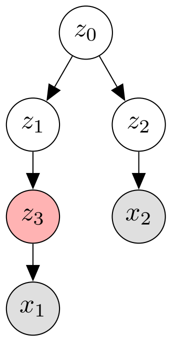
(h) Model type (2,4).



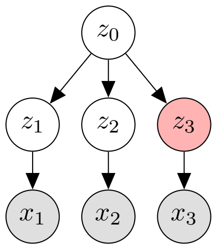
(i) Model type (3,1).



(j) Model type (3,2).

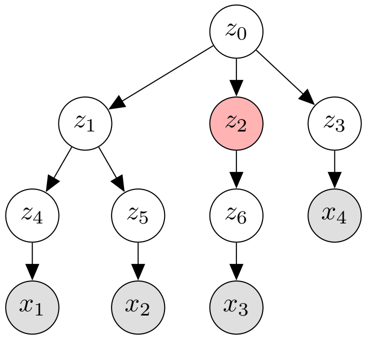


(k) Model type (3,3).

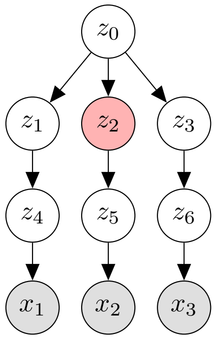


(l) Model type (3,4).

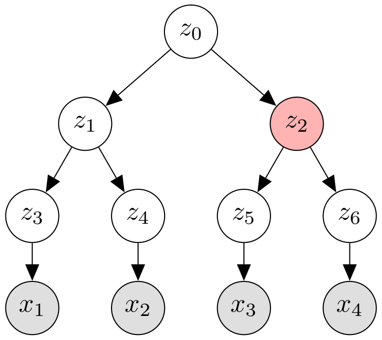
Figure 5: Canonicalised dependency graphs for all 12 model types in ext1. The rows are for different positions of the nl variable, and the columns are for different dependency graphs: the model type (i,j) means one of the 12 model types in ext1 that corresponds to the i-th position of the nl variable and j-th dependency graph.



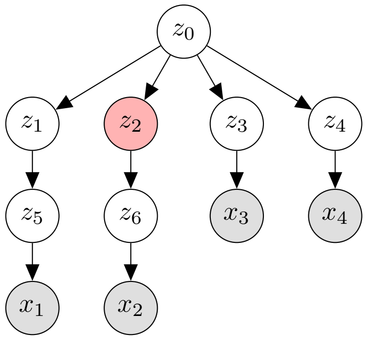
(a) Model type for the 1st dependency graph.



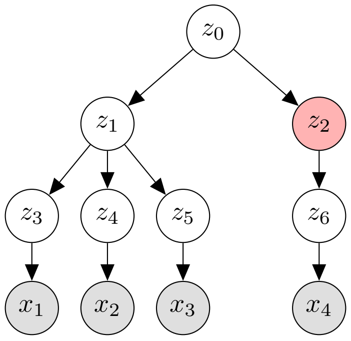
(b) Model type for the 2nd dependency graph.



(c) Model type for the 3rd dependency graph.

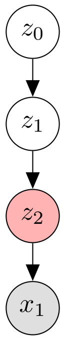


(d) Model type for the 4th dependency graph.

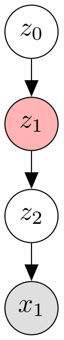


(e) Model type for the 5th dependency graph.

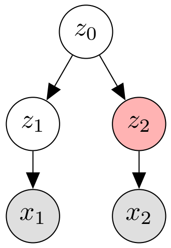
Figure 6: Canonicalised dependency graphs for all five model types in ext2.



(a) 1st model type.

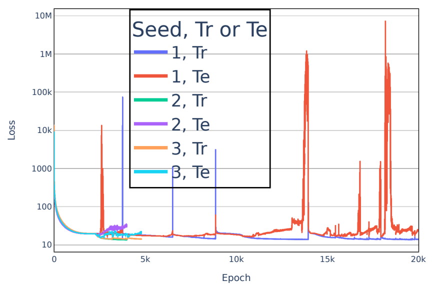


(b) 2nd model type.

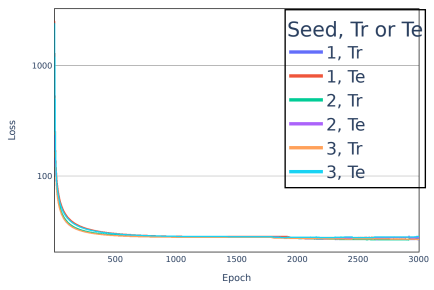


(c) 3rd model type.

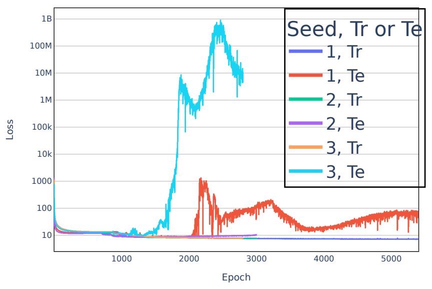
Figure 7: Canonicalised dependency graphs for all three model types in the mulmod class.



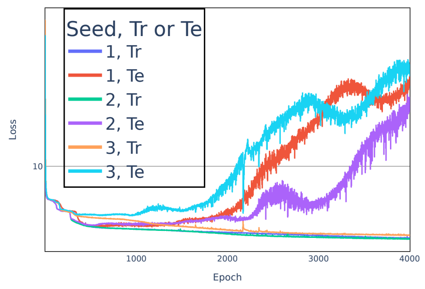
(a) hierd



(b) cluster



(c) milkyo



(d) rb

Figure 8: The y-axes are log-scaled. The surges in later epochs of Fig. 8a, 8c and 8d were due to only a single or a few test programs out of 50.

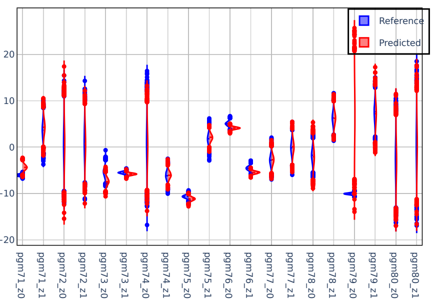
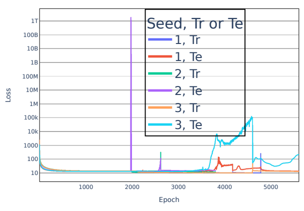
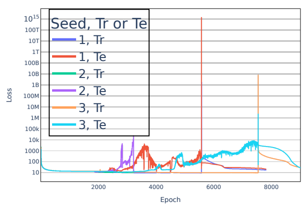


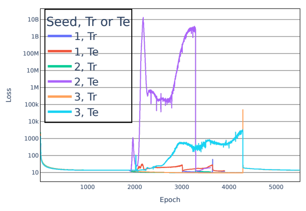
Figure 9: Comparisons of reference and predicted marginal posteriors for 10 programs in the rb test set.



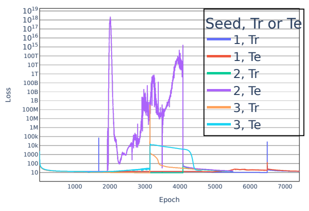
(a) To 1st dep. graph.



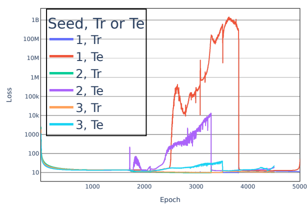
(b) To 2nd dep. graph.



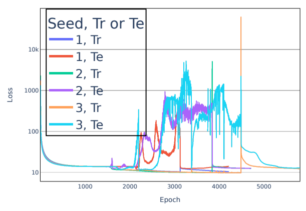
(c) To 3rd dep. graph.



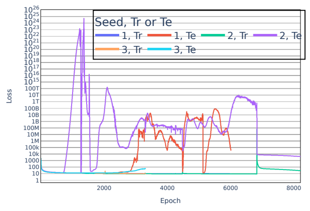
(d) To 4th dep. graph.



(e) To 1st nl position.

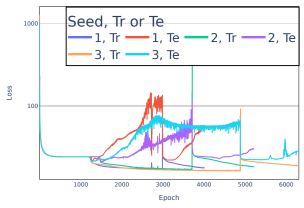


(f) To 2nd nl position.

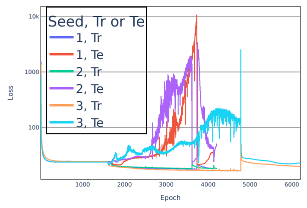


(g) To 3rd nl position.

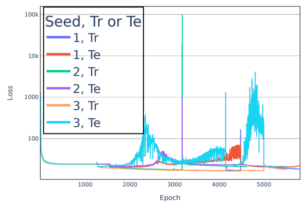
Figure 10: Average training and test losses for generalisation in ext1. The y-axes are log-scaled.



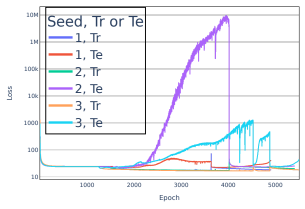
(a) To 1st dep. graph.



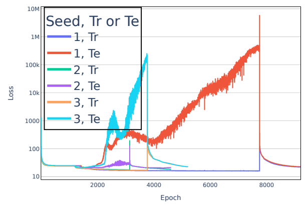
(b) To 2nd dep. graph.



(c) To 3rd dep. graph.



(d) To 4th dep. graph.



(e) To 5th dep. graph.

Figure 11: Average training and test losses for ext2. The y-axes are log-scaled.

$a := 3.93;\ b := 348.16;\ c := 57.5;\ d := 14.04;\ e := 40.34;$
$z_1 \sim \mathcal{N}(a, b);\ z_2 \sim \mathcal{N}(z_1, c);\ z_3 := \mathrm{mm}(z_1);$
$\texttt{obs}(\mathcal{N}(z_2, d), 53.97);\ \texttt{obs}(\mathcal{N}(z_3, e), 0.12)$

Figure 12: The program in mulmod that is reported in the test-time comparisons, written in our probabilistic programming language.

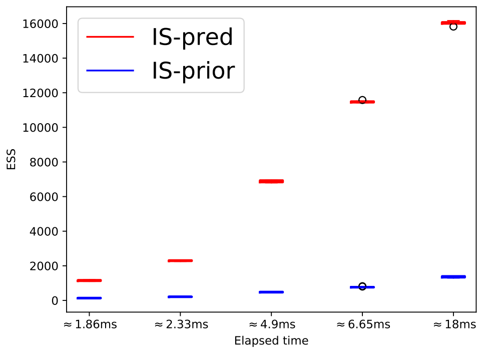


Figure 13: ESS by IS-pred and IS-prior for the test program from mulmod in the test-time comparisons in §3.