An Interface for Black Box Learning in Probabilistic Programs

Jan-Willem van de Meent **Brooks Paige** **David Tolpin** **Frank Wood**
Department of Engineering Science, University of Oxford

We define a family of parameter learning methods for probabilistic program systems (PPSs) [1–10] that perform stochastic gradient ascent using sample-based estimates of the gradient. These algorithms are specified in terms of the interaction between a back end $\mathcal{B}$ and a probabilistic program $\mathcal{P}$, which is treated as a black box computation. The result is a language-agnostic infterface for learning of probabilistic programs. The described methods can be used to perform both variational Bayes (VB) [11–13], which approximates a posterior distribution, and empirical Bayes (EB) [14, 15], which maximizes the marginal likelihood with respect to the prior hyperparameters. Moreover, EB estimation is equivalent to (upper-level) policy search [15, 16] in programs where the exponent of the reward takes the place of the likelihood [17, 18].

In this abstract we are interested in algorithms that combine inference with learning. As a motivating example we consider a program (see Figure 1), written in the language Anglican [7], which simulates the Canadian traveler problem (CTP) domain. In the CTP, an agent must travel along a graph, which represents a network of roads, to get from the start node (green) to the target node (red). Due to bad weather some roads are blocked, but the agent does not know which in advance. The agent performs depth-first search along the graph, which will require a varying number of steps, depending on which edges are closed, and incurs a cost for the traveled distance. The program in Figure 1 defines two types of policies for the CTP. For the policy where edges are chosen at random, we may perform online planning by simulating future actions and outcomes, also known as rollouts, and choosing the action that minimizes the expected cost. Alternatively we may learn a policy that, after an initial training period, can be applied without calculating rollouts. To do so we consider a deterministic policy for which we learn a set of parameters (the edge preferences).

Structural operational semantics for probabilistic programming languages typically specify the result and associated probability for each elementary step in an execution. We here specify an interface between a stateful computation $\mathcal{P}$, which performs all deterministic steps in the execution, and a back end $\mathcal{B}$, which handles 3 types of events: 1. `sample` signifies that the back end must supply a value for a random variable 2. `learn` is the same as sample, but additionally indicates that this is the back end must learn hyperparameters for the distribution on the random variable. 3. `factor` assigns a probability to the execution.

We write θ_b for the random variables in a program $\mathcal{P}$ that are considered parameters and z_a for all other random variables. Here $a \in A$ and $b \in B$ are unique identifiers, also known as addresses, in sets A and B that can vary from execution to execution. We assume that factor statements have computable densities w_c with $c \in C$. A program $\mathcal{P}$ then defines a density $\pi_{\mathcal{P}}(z, \theta) = \gamma_{\mathcal{P}}(z, \theta)/Z$, where

$$\gamma_{\mathcal{P}}(z, \theta) := \prod_{a \in A} f_a(z_a \,|\, \phi_a) \prod_{b \in B} f_b(\theta_b \,|\, \eta_b) \prod_{c \in C} w_c.$$

Here, the run-time densities $f_a(\cdot \,|\, \phi_a)$ and $f_b(\cdot \,|\, \eta_b)$ for each variable may vary from execution to execution, and it is assumed that their dependency on previous random variables is not known to the back end.

We define the density of a variational program $\mathcal{Q}_\lambda$ as the unconditioned variant of $\mathcal{P}$ in which learned variational parameters λ_b replace run-time parameters η_b

$$p_{\mathcal{Q}_\lambda}(z, \theta) := \prod_{a \in A} f_a(z_a \,|\, \phi_a) \prod_{b \in B} f_b(\theta_b \,|\, \lambda_b).$$

We now define the inference semantics of an importance sampler that targets $\gamma_{\mathcal{P}}$ by proposing from $p_{\mathcal{Q}_\lambda}$:

- Initially $\mathcal{B}$ calls $\mathcal{P}$ with no arguments $\mathcal{P}()$
- A call to $\mathcal{P}$ returns one of four responses to $\mathcal{B}$:
  1. $(\texttt{sample}, a, f, \phi)$: $\mathcal{B}$ samples $z_a \sim f_a(\cdot \,|\, \phi_a)$. Execution continues by calling $\mathcal{P}(z_a)$.
  2. $(\texttt{learn}, b, f, \eta)$: $\mathcal{B}$ samples $\theta_b \sim f_b(\cdot \,|\, \lambda_b)$, registers $w_b = f_b(\theta_b \,|\, \eta_b)/f_b(\theta_b \,|\, \lambda_b)$ and calls $\mathcal{P}(\theta_b)$.
  3. $(\texttt{factor}, c, w)$: $\mathcal{B}$ registers w_c and calls $\mathcal{P}()$.
  4. $(\texttt{return}, v)$: $\mathcal{P}$ terminates, returning v.

Repeated execution of $\mathcal{P}$ through this interface results in a sequence of weighted samples $(w^{[n]}, \theta^{[n]}, z^{[n]})$, whose importance weight $w^{[n]}$ is defined as

$$w^{[n]} := \gamma_{\mathcal{P}}(z^{[n]}, \theta^{[n]}) \;/\; p_{\mathcal{Q}_\lambda}(z^{[n]}, \theta^{[n]}) = \prod_{b \in B} w_b \prod_{c \in C} w_c.$$

```
(defquery ctp
 [problem base-prob make-policy]
 (let [graph (get problem :graph)
       start (get problem :s)
       target (get problem :t)
       sub-graph (sample-weather graph base-prob)
       [path dist counts]
        (dfs-agent sub-graph s t (make-policy))]
  (factor (- (or dist inf)))
  (predict :path path)
  (predict :distance dist)
  (predict :counts counts))))

(defm dfs-agent
 [graph start target policy]
 (loop [path [start]
        counts {}
        dist 0.0]
  (let [u (peek path)]
   (if (= u target)
    [path dist counts]
    (let [unvisited
          (filter
           (fn [v] (not (get counts #{u v})))
           (adjacent graph u))]
     (if (empty? unvisited)
      (if (empty? (pop path))
       [nil dist counts]
       (let [v (peek (pop path))]
        (recur (pop path)
               (assoc counts #{u v} 2)
               (+ dist (distance graph u v)))))
      (let [v (policy u unvisited)]
       (recur (conj path v)
              (assoc counts #{u v} 1)
              (+ dist
                 (distance graph u v))))))))))
```

```
(defm make-random-policy []
 (fn policy [u vs]
  (sample
   (categorical
    (zipmap vs (repeat (count vs) 1.))))))

(defm make-edge-policy []
 (let [Q (mem (fn [u v]
               (sample
                (learn [u v] (gamma 1. 1.)))))]
  (fn policy  [u vs]
   (argmax
    (zipmap vs (map (fn [v] (Q u v)) vs))))))
```

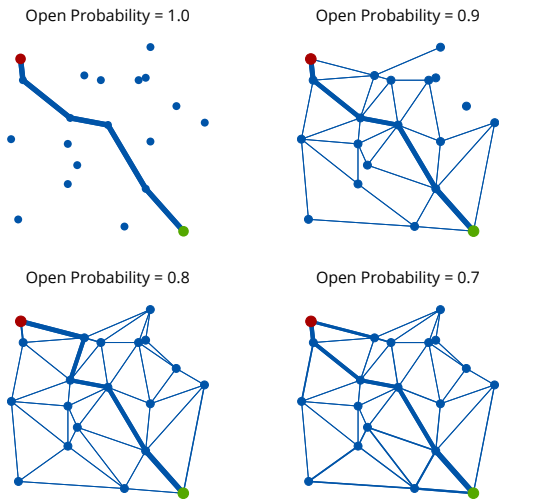

Figure 1: A Canadian traveler problem (CTP) implementation in Anglican. The function `dfs-agent` walks the graph by performing depth-first search, calling a function `policy` to choose the next destination based on the current and unvisited locations. The function `make-random-policy` returns a `policy` function that selects destinations uniformly at random, whereas `make-edge-policy` constructs a `policy` that selects according to sampled edge preferences `(Q u v)`. By learning a distribution on each value `(Q u v)` through gradient ascent on the marginal likelihood, we obtain a heuristic offline policy that follows the shortest path when all edges are open, and explores more alternate routes as more edges are closed.

This importance sampling protocol allows us to perform black box variational inference (BBVI) [12, 13, 15], which optimizes a lower bound $\mathcal{L}_\lambda$ on $\log Z$

$$\begin{aligned}\mathcal{L}_\lambda &= \mathbb{E}_{p_{\mathcal{Q}_\lambda}}[\log \gamma_{\mathcal{P}}(z,\theta) - \log p_{\mathcal{Q}_\lambda}(z,\theta)],\\ &= \log Z - D_{\mathrm{KL}}(p_{\mathcal{Q}_\lambda}(z,\theta)\,||\,\gamma_{\mathcal{P}}(z,\theta)/Z) \leq \log Z,\end{aligned}$$

by calculating a sample-based estimate of $\nabla_\lambda \mathcal{L}_\lambda$

$$\hat{\nabla}_{\lambda_b}\mathcal{L}_\lambda = \sum_{n\in\{n:b\in B_n\}} \nabla_{\lambda_b} \log p_{\mathcal{Q}_\lambda}(z^{[n]},\theta^{[n]})(\log w^{[n]} - \hat{b}_i),$$

in which $w^{[n]}$ is the importance weight defined above, and $\hat{b}_b$ is a control variate (see [12, 13, 15] for details). We note that this estimate only requires calculation of $\nabla_{\lambda_b} \log p_{\mathcal{Q}_\lambda}(z,\theta)$, which is trivial since λ-dependent terms in $p_{\mathcal{Q}_\lambda}(z,\theta)$ are independent by construction.

This gradient estimator can be used in 3 types of learning algorithms. In VB estimation we consider updates

$$\lambda_{k+1} = \lambda_k + \rho_k \hat{\nabla}_\lambda \mathcal{L}_\lambda\big|_{\lambda=\lambda_k},$$

where ρ_k is a Robbins-Monro sequence of step sizes[19].

In EB estimation we define the lower bound $\mathcal{L}_{\lambda,\lambda_k}$ in terms of a density $\gamma_{Q_{\lambda_k}}$ where the learned distribution $p_{\mathcal{Q}_{\lambda_k}}$ replaces the prior, and perform updates

$$\lambda_{k+1} = \lambda_k + \rho_k \hat{\nabla}_\lambda \mathcal{L}_{\lambda,\lambda_k}\big|_{\lambda=\lambda_k}.$$

In models where $w_c = \exp(R_c)$ is defined in terms of the reward R_c, EB estimation is equivalent to policy search, and maximizes the expected reward [15].

In summary, we have specified the inference semantics for a family of learning algorithms for probabilistic programs, in which importance sampling semantics are used as a basis for computation of a sample-based estimate of the gradient. This estimate can be calculated in any stystem that (a) implements the importance sampling interface defined above and (b) implements derivatives for the density of each primitive distribution type in the language. Since no automatic differentiation implementation is needed for this gradient estimate, this family of algorithms provides a natural "light-weight" baseline for learning in programs.

References

[1] Brian Milch, Bhaskara Marthi, Stuart Russell, David Sontag, Daniel L. Ong, and Andrey Kolobov. BLOG : Probabilistic Models with Unknown Objects. In *IJCAI*, 2005.

[2] Luc De Raedt, Angelika Kimmig, and Hannu Toivonen. ProbLog: A probabilistic prolog and its application in link discovery. *IJCAI International Joint Conference on Artificial Intelligence*, pages 2468–2473, 2007.

[3] Noah Goodman, Vikash Mansinghka, Daniel M Roy, Keith Bonawitz, and Joshua B Tenenbaum. Church: a language for generative models. In *Proc. 24th Conf. Uncertainty in Artificial Intelligence (UAI)*, pages 220–229, 2008.

[4] Avi Pfeffer. Figaro: An object-oriented probabilistic programming language. Technical report, 2009.

[5] a McCallum, K Schultz, and S Singh. Factorie: Probabilistic programming via imperatively defined factor graphs. In *Advances in Neural Information Processing Systems*, volume 22, pages 1249–1257, 2009.

[6] T Minka, J Winn, J Guiver, and D Knowles. Infer .NET 2.4, Microsoft Research Cambridge, 2010.

[7] F Wood, JW van de Meent, and V Mansinghka. A new approach to probabilistic programming inference. In *Artificial Intelligence and Statistics*, pages 1024–1032, 2014.

[8] Brooks Paige and Frank Wood. A Compilation Target for Probabilistic Programming Languages. *International Conference on Machine Learning (ICML)*, 32, mar 2014.

[9] Vikash Mansinghka, Daniel Selsam, and Yura Perov. Venture: a higher-order probabilistic programming platform with programmable inference. *arXiv*, page 78, mar 2014. URL `http://arxiv.org/abs/1404.0099`.

[10] Stan Development Team. Stan: A C++ Library for Probability and Sampling, Version 2.4, 2014.

[11] Martin J Wainwright and Michael I Jordan. Graphical Models, Exponential Families, and Variational Inference. *Foundations and Trends in Machine Learning*, 1(1âĂŞ2):1–305, 2008.

[12] David Wingate and Theo Weber. Automated variational inference in probabilistic programming. *arXiv preprint arXiv:1301.1299*, pages 1–7, 2013. URL `http://arxiv.org/abs/1301.1299`.

[13] Rajesh Ranganath, Sean Gerrish, and David M Blei. Black Box Variational Inference. In *Artificial Intelligence and Statistics*, 2014. URL `http://arxiv.org/abs/1401.0118`.

[14] J S Maritz and T Lwin. *Empirical Bayes methods*, volume 35. Chapman and Hall, London, 1989. ISBN 0412277603.

[15] Jan-Willem van de Meent, David Tolpin, Brooks Paige, and Frank Wood. Black-Box Policy Search with Probabilistic Programs. pages 1–22, 2015. URL `http://arxiv.org/abs/1507.04635`.

[16] Marc Peter Deisenroth, Gerhard Nuemann, and Jan Peters. A Survey on Policy Search for Robotics. *Foundations and Trends in Robotics*, 2(2011):1–142, 2011.

[17] Marc Toussaint, Stefan Harmeling, and Amos Storkey. Probabilistic inference for solving (PO)MDPs. *Neural Computation*, 31(December):357–373, 2006.

[18] Matthew Botvinick and Marc Toussaint. Planning as inference. *Trends in Cognitive Sciences*, 16(10):485–488, 2012.

[19] H Robbins and S Monro. A Stochastic Approximation Method. *The Annals of Mathematical Statistics*, 22(3):400–407, 1951. URL `http://www.jstor.org/stable/10.2307/2236626`.