We define a family of parameter learning methods for probabilistic program systems (PPSs) \[11, 10\] 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 \( B \) and a probabilistic program \( P \), which is treated as a black box computation. The result is a language-agnostic interface 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 which learned variational parameters can be used to perform both variational and empirical Bayes estimation.

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 \( P \), which performs all deterministic steps in the execution, and a back end \( B \), which handles 3 types of events: 1. \textbf{sample} signifies that the back end must supply a value for a random variable. 2. \textbf{learn} is the same as \textbf{sample}, but additionally indicates that this is the back end must learn hyperparameters for the distribution on the random variable. 3. \textbf{factor} assigns a probability to the execution.

We write \( \theta_b \) for the random variables in a program \( P \) 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 \( P \) then defines a density \( \pi_P (z, \theta) \) that can vary from execution to execution. We assign a probability to the execution.

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

\[ w[n] := \gamma_P (z[n], \theta[n]) / p_{Q_{\lambda}} (z[n], \theta[n]) = \prod_{b \in B} w_b \prod_{c \in C} w_c. \]
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 \( \log Z \)

\[
\mathcal{L}_\lambda = \mathbb{E}_{p_{Q_\lambda}} \left[ \log \gamma_p(z, \theta) - \log p_{Q_\lambda}(z, \theta) \right],
\]

\[
= \log Z - D_{KL}(p_{Q_\lambda}(z, \theta) \parallel \gamma_p(z, \theta)/Z) \leq \log Z,
\]

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

\[
\nabla_\lambda \mathcal{L}_\lambda = \sum_{n \in \{x \in B_n\}} \nabla_\lambda \log p_{Q_\lambda}(z^{[n]}, \theta^{[n]}) (\log w^{[n]} - \hat{w}),
\]

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

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

\[
\lambda_{k+1} = \lambda_k + \rho_k \nabla_\lambda \mathcal{L}_\lambda|_{\lambda=\lambda_k},
\]

where \( \rho_k \) 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 \( p_{Q_{\lambda_k}} \) replaces the prior, and perform updates

\[
\lambda_{k+1} = \lambda_k + \rho_k \nabla_\lambda \mathcal{L}_{\lambda, \lambda_k}|_{\lambda=\lambda_k}.
\]

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 system 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


