Hijacking Simulators with Universal Probabilistic Programming

Paper ID: 1010

Abstract

Notes on how to develop a formalism to perform inference in population based simulators.

1 Introduction

One challenge in performing inference in such models is the combinatorial increase in *paths* that can be taken within the given probabilistic model. This is because each member of the population has a bounded, but large number of decisions to make. In addition to this, each member of the population can interact with other members of the population. To model a single population member quantum-based methods leveraging linear superposition and entanglement provide an ability to explore multiple paths simultaneously maintaining the whole program, *state*, in one iteration. Unfortunately, analytically writing the functional form of the state is, in this instance, impossible. The usability of current classical inference methods in this context is also problematic. However, by combining hijacking and amortizing parts of the simulator, in particular rejection sampling loops, we can not only correctly perform inference in such models, as the rejection sampling loops are now replaced with learnt functions, but we can also perform inference much more efficiently. This is because the rejection sampling loops are now replaced with learnt functions which enables us to correctly construct the density and as they are amortized, evaluation is cheap and only has to be performed once per call. This significantly reduces memory consumption and running time.

Without the amortization procedure, as updates are made sequentially MCMC methods are typically not-well suited this type of recursive estimation problem. As each time we get a new data point y_{t+1} we have to run new MCMC simulations for $P(x_{0:t_1}|y_{0:t+1})$, we also cannot directly reuse the samples $\{x^i_{0:t}\}$. However, as we can only run the simulator `forward()` we cannot return back to a previous state and re-run. So adapting existing inference techniques is the only viable option. MCMC methods can be effective with a well designed amortized inference procedure (**Note: This is not guaranteed**). Non-MCMC based methods such as importance sampling (IS) and sequential monte carlo (SMC), may also be sensible inference engines, although we shall discuss the limitations within the simulator context in the proceeding sections.

However, to implement this in practice we are required to do the following:

- Create a sensible training procedure
- Have 2 additional functions implemented within pyprob
    1. 1) That stores only the necessary raw samples from the desired rejection sampling addresses to be used for training.
    2. 2) A function at *test time* that replaces those given addresses with a learnt surrogate.
- Finer grained control over the actual latent variables of interest, to avoid unnecessary memory consumption and ensure that inference problems remain tractable.

There is no doubt that implementing some fo these features will be more invasive to the underlying simulator code than previously just hijacking the raw-random number draws, but, it would enable

Preprint. Under review.

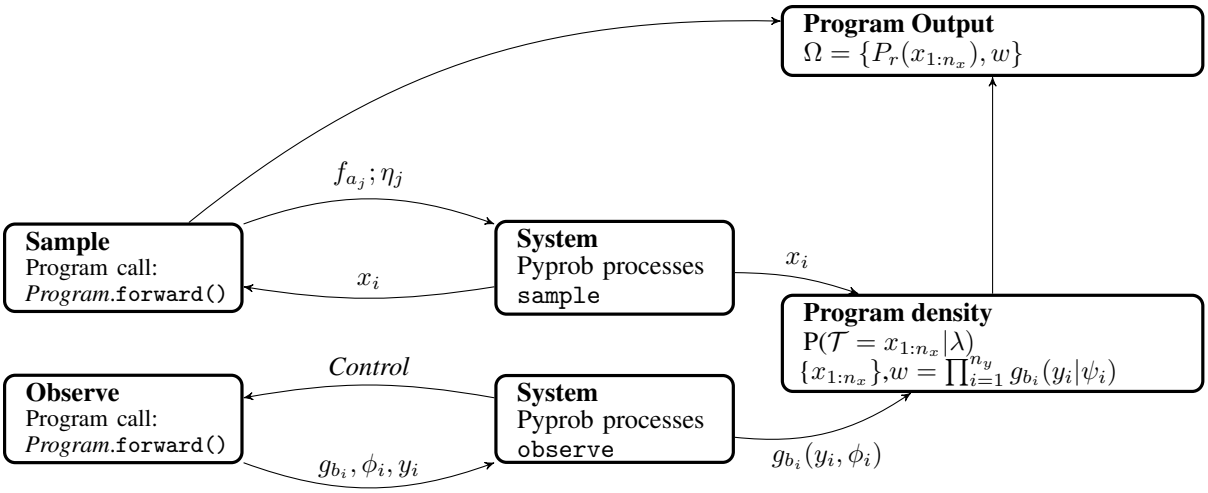

Figure 1: An overview of the hijacking system. x_i represent latent variables. f_{a_i} represent raw sample calls, g_{b_j} represent conditioning statements. $\mathcal{T}$ corresponds to the program trace. w represents the collective weights. r represents run number. n_x represents the number of latent variables and n_y represents the number of observations, which will always be fixed, but is model dependent. λ represents all the other variables generated by running the simulator `.forward()`.

us to attack difficulties in the inference procedures that are currently not amenable to the current approach.

2 Limitations with different inference engines

2.1 Prior based sampling

In prior based sampling we take our existing program and look directly at the product of all the raw sample calls, denoted $f_{a_i}(x_i|\eta)$ and construct a proposal that is is dependent on those. In this sampling scheme our proposal is defined as:

$$q(x_{1:n_x}) = \begin{cases} \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta) \texttt{ if } \mathcal{B}(x_{1:n_x}) = 1 \\ 0 \end{cases} \tag{1}$$

where $\mathcal{B}(\cdot)$ is satisfied by the simulator and $f_{a_i}(\cdot)$ represents a raw random sample, η is a subset of the variables in-scope at the point of sampling and n_x is the number of latent variables. This is true in all inference engines used in this hijacking framework. However, this makes use of no conditioning and is not sufficient for performing inference in complex, population-based simulations.

3 Non-prior Based Sampling

In non-prior based sampling we utilise conditioning too, which enables us to explore spaces more efficiently. In the current hijacking set-up, we can only do conditioning at the end of each `forward()` run of the simulator. This does inhibit certain inference engines and makes aspects of performing inference within this setting more complicated. Nonetheless, there are several inference schemes that we can exploit and utilise. We shall now give an Introduction to each of these schemes.

3.1 Importance Sampling

In the generalised probabilistic programming setting an importance sampling scheme over the program raw samples and observations can be defined as follows:

$$p(\mathcal{T} = x_{1:n_x}|\lambda) = \begin{cases} \prod_{i=1}^{n_x} f_{a_i}(x_i|\eta) \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi) \texttt{ if} \mathcal{B}(x_{1:n_x}) = 1 \\ 0 \end{cases} \tag{2}$$

where the observe statements are defined by $g_{b_j}(.)$, ϕ plays the same role as η and n_y represents the number of observations. The expectation over the whole program can be defined as:

$$\mathbb{E}_{p(\Omega|\lambda)}[h(\Omega)] = \mathbb{E}_{p(x_{1:n_x}|\lambda)}[h(P_r(x_{1:n_x}))] \tag{3}$$

Note: It is a little weird how we define this, as the the program output Ω should also include the product of the weights

where r represents a run of the program, $P_r(\cdot)$ is the entirety of the program output, $h(\cdot)$ is just a deterministic function.

As the importance sampler updates the weights according to the following formula:

$$w(x_{1:n_x}) = \prod_{i=1}^{n_x} \frac{f_{a_i}(x_i|\eta_i)}{q_i(x_i|f_{a_i}, \eta_i)} \prod_{j=1}^{n_y} g_{b_j}(y_j|\phi_j) \tag{4}$$

When the number of observations is fixed, the combined weight is equal to $w(x_{1:n_x}) = \prod_{j=1}^{n_y} w_j$. To generate the weight up to each observe we can define $K(j)$ to be a function that returns the least i before hitting an observe j. This enables us to write the j-th weight as follows:

$$w_j(x1 : K(j)) = g_{b_j}(y_j|\psi_j) \prod_{K(j-1)+1}^{K(j)} v_i \tag{5}$$

where we define $v_j = \frac{f_{a_j}(x_j|\eta_j)}{q_j(x_j|f_{a_j},\eta_j)}$

Taking these definitions we can construct a target density over the latents that is proportional to the products of `sample` and `observe` statements **Note: Add update posterior here**. Thus, given the proposal $q(x_{1:K(t)})$ we define the importance weight as:

$$\begin{aligned} w(x_{1:t}) &= \prod_{j=1}^{t} w_j(x_{1:K(j)}) = \\ &\frac{p_t(x_{1:K(t)})}{q(x_{1:K(t)})} = \begin{cases} \prod_{j=1}^{t} g_{b_j}(y_j|w_i) \prod^{K}(j)_{K(j-1)+1} v_i \texttt{ if } B_t(x_1 : K(t)) = 1 \\ 0 \end{cases} \end{aligned} \tag{6}$$

One problem with importance sampling is in its ability to scale and condition upon multiple observations. | Choosing a *good* prior remains challenging.

3.2 Sequential Monte Carlo

Given multiple observations, we can ake the inference more computationally efficient by employing sequential monte carlo (SMC), although as we can only condition at the end of one full run of the simulation, computational speed increases are minimal. SMC allows for conditioning on multiple observations, but the number of observations must be fixed. **Note: If we could condition on observations within the simulator, we could get dramatic speed increases over standard importance sampling.**

To implement SMC in the current set-up we round require a forking method and the

3.3 Random-walk Metropolis Hastings

Due to the current set-up performing Random-walk Metropolis Hastings (RMH) is not feasible as the rejection sampling blocks within the simulator mean that we cannot correctly perform RMH as we do not have a functional form for all parts of the simulator to construct a target density that we can evaluate. **Note: Ask tom what: one can sometimes achieve improvements by instead using the type of the distribution object associated with x_m (what does this first part mean?) to automatically construct a valid random walk proposal that allows for improved hill climbing behavior on the individual updates.** .

Algorithm 1 SMC for Population Based Simulators - well any general program - not new

procedure SMC(T, M,**x**, **w**,**v**, **g**)
  for $t = 1 : T$ **do**
    for $m = 1 : M$ **do**
      $w^m = 1$
      for $k = K(t-1)+1 : K(t)$ **do**
        $x_k^m \sim q_k(x_k^m | f_{a_k^m}, \eta_j^m)$
        $w_t^m \leftarrow w_m^t v_j^m(x_k^m)$
      end for
    end for
    $w_t^m \leftarrow w_t^m g_{b_t^m}(y_t^m | \psi_t^m)$
    $\{x_{1:t}^{1:m}\} \leftarrow$ `resample`($\{x_{1:t}^{1:m}\}$, $\{w_{1:t}^{1:m}\}$)
  end for
end procedure

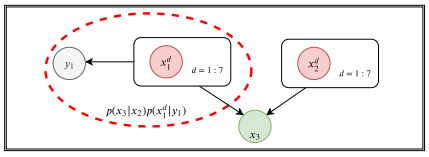

Figure 2: The DAG of the Parasite model.

4 An Example: Modelling Parasite Densities in Simulation

The parasite model by Smith el al. **?** is a simple stochastic model for determining the number of infections within a given population demographic. The models hypothesised in **?** were found, after extensive testing and development, to not characterise well the number of infections among different demographics, especially those under 15 years old. By using probabilistic programming we can immediately understand the expressiveness of the proposed model and can exact important insights for extending the proposed models, which increases ones ability to develop improved models to estimate parasite densities, given our observations of the seasonal entomological inoculation rate (EIR). In addition to this we can greatly reduce the computational costs, whilst improving model comprehension and expressivity. Although a statistical model is not formally defined within **?** we define the unknown *disease dynamics* parameters with a set of latent variables$\{x_i\}^N$ and construct the model as follows:

$$\mu_i = N \tag{7}$$
$$S_\infty \sim \mathcal{N}(\mu_\infty, \sigma^2_\infty) \tag{8}$$
$$E_{crit} \sim \mathcal{N}(\mu_{crit}, \sigma^2_{crit}) \tag{9}$$
$$S_{imm} \sim \mathcal{N}(\mu_{imm}, \sigma^2_{imm}) \tag{10}$$
$$X_{p_{crit}} \sim \mathcal{N}(\mu_{p_{crit}}, \sigma^2_{p_{crit}}) \tag{11}$$
$$\gamma_p \sim \mathcal{N}(\mu_p, \sigma^2_p) \tag{12}$$
$$\rho_i \sim \mathcal{U}(\mathbb{L}_i, \mathbb{U}_i) \tag{13}$$
$$for\ t = 1 : t_{max} : \tag{14}$$
$$E_a = \frac{E_{max_t}}{A_{max}} \times h_{\mu_{SA}}() \tag{15}$$
$$X_p = E_a + X_p \tag{16}$$
$$S_1 = S_\infty + \frac{(1 - S_\infty)}{1 + \frac{E_a}{E_{crit}}} \tag{17}$$
$$S_2 = S_{imm} + \frac{1 - S_{imm}}{1 + (\frac{X_p}{X_{pcrit}})^{\gamma_p}} \tag{18}$$
$$S_p = S_1 \times S_2 \tag{19}$$
$$\lambda = S_p \times E_a \tag{20}$$
$$h \sim Pos(\lambda) \tag{21}$$
$$\texttt{if}\ h > 0 : \tag{22}$$
$$n_{infect} = 1 \tag{23}$$
TODO: add prevelance construction below (24)

TODO: Add some of the plots generated from this model and an accompanying analysis to them.

5 Additional TODOs

- Before implementing such tools into the simulator, spend time looking for simpler stochastic simulators that can be hijacked but have target densities that can be constructed analytically.

- If we are not utilizing the additional information stored in the trace, which is only used during the inference compilation training procedure, then we do not need it all, as for importance sampling we only require the weights after each `forward()` update.

- If we cannot find any pre-existing simulators that are amenable to this approach then, we need to create our own, with a rejection sampling loop, that can be exploited by the new machinery. This would enable us to debug, provide a simple example for the paper and understand the feasibility of the entire approach.

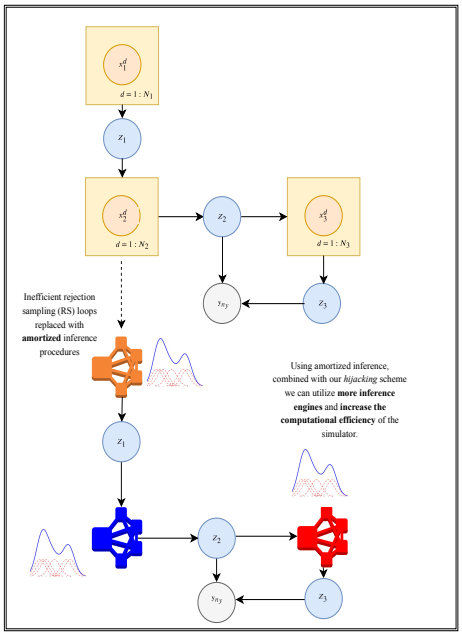

Figure 3: Using PyProb we can locate all regions of inefficient sampling and replace those points in the code with learnt functions that represent the distributions of the rejection sampling statements. Z_i represents the marginal and by learning surrogate functions for each of the rejection sampling loops. In doing so we can learn approximate raw sample distributions $f(x_i|\eta)$ for each rejection sampling loop. This means that we can evaluate the function at each part in the simulator, enabling us to construct a target density that can be evaluated when using MCMC inference schemes such as RMH.