Using Probabilistic Programs as Proposals

Marco F. Cusumano-Towner and Vikash K. Mansinghka
MIT Probabilistic Computing Project

Overview

- Monte Carlo inference has nice asymptotic guarantees, but can be slow when using generic proposals.
- Handcrafted proposals that rely on user knowledge about the posterior distribution ('*posterior knowledge*') can be efficient, but are difficult to derive and implement.
- We propose to let users express posterior knowledge in the form of *proposal programs*, which are samplers written in probabilistic programming languages.
- Proposal programs can combine domain-specific heuristic algorithms with amortized inference networks, bridging the gap between automated and custom inference.

Proposal programs

- A proposal program $\mathcal{P}$ has a subset of its random choices designated as *output choices* (z). Other random choices are *internal choices* (y). Arguments are denoted x.
- Evaluating the proposal probability $p(z; x)$ requires marginalizing over internal choices. We show that *estimates* of the proposal probability can be used instead in importance sampling and Metropolis-Hastings without sacrificing consistency.
- Proposal programs are easily implemented on top of a sampling-based probabilistic programming runtime.

A standard interface for proposal distributions

procedure SIMULATE($\mathcal{P}$, x)
  $z \sim p(\cdot; x)$ ▷ Execute $\mathcal{P}$
  $\xi \leftarrow p(z; x)$ ▷ Compute output probability
  return (z, ξ)
end procedure

procedure ASSESS($\mathcal{P}$, x, z)
  $\xi \leftarrow p(z; x)$ ▷ Compute output probability
  return ξ
end procedure

The new interface for proposal programs

procedure SIMULATE($\mathcal{P}$, x, K)
  $y_1, z \sim p(\cdot, \cdot; x)$ ▷ Execute $\mathcal{P}$
  for $k \leftarrow 2 \ldots K$ **do**
    $y_k \sim p(\cdot; x)$ ▷ Execute $\mathcal{P}$ (excluding outputs)
  end for
  $\hat{\xi} \leftarrow \frac{1}{K} \sum_{k=1}^{K} p(z; y_k, x)$
  return $\left(z, \hat{\xi}\right)$
end procedure

procedure ASSESS($\mathcal{P}$, x, z, K)
  for $k \leftarrow 1 \ldots K$ **do**
    $y_k \sim p(\cdot; x)$ ▷ Execute $\mathcal{P}$ (excluding outputs)
  end for
  $\hat{\xi} \leftarrow \frac{1}{K} \sum_{k=1}^{K} p(z; y_k, x)$
  return $\hat{\xi}$
end procedure

Offline optimization of proposal programs

- Proposal programs can be optimized offline to maximize objective $J(\theta)$ for training distribution $r(x, z)$.

$$\min_\theta \mathbb{E}_{x \sim r(\cdot)} \left[\mathrm{D}_{\mathrm{KL}}(r(z|x) || p(z; x, \theta)) \right] \quad \text{is equivalent to}$$

$$\max_\theta J(\theta) = \max_\theta \mathbb{E}_{z,x \sim r(\cdot,\cdot)} \left[\log p(z; x, \theta) \right]$$

- Because proposal programs can include internal random choices, we instead optimize the following lower bound:

$$J^K(\theta) := \mathbb{E}_{x,z \sim r(\cdot,\cdot)} \left[\mathbb{E}_{y_{1:K} \overset{\text{i.i.d.}}{\sim} p(\cdot;x,\theta)} \left[\log \hat{\xi}(y_{1:K}, z, x, \theta) \right] \right] \leq J(\theta)$$

- We use a stochastic gradient estimator based on the multiple-sample baseline of Mnih and Rezende (2016).

Example: Using RANSAC for posterior inference

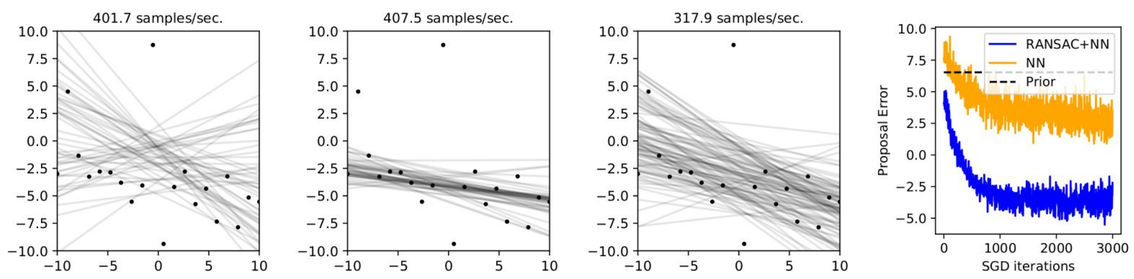
Probabilistic program for linear regression model

```
@probabilistic function linear_regression_model(xs)
    slope = @choice(normal(0, 1), "slope")
    intercept = @choice(normal(0, 2), "intercept")
    for (i, x) in enumerate(xs)
        outlier = @choice(flip(PRIOR_PROB_OUTLIER), "outlier-$i")
        var = outlier ? OUTLIER_VAR : INLIER_VAR
        @choice(normal(slope * x + intercept, sqrt(var)), "y-$i")
    end
end
```

RANSAC-based proposal program (RANSAC+NN)

```
@probabilistic function ransac_proposal(xs, ys, params)

    # Generate parameters for RANSAC using learned parameters
    epsilon = @choice(gamma(exp(params.eps_alpha),
                            exp(params.eps_beta)), "epsilon")
    num_iters = @choice(categorical(params.iter_dist), "iters")
    ransac_params = RANSACParams(num_iters, epsilon)

    # Run RANSAC (uses many un-annotated random choices)
    slope_guess, int_guess = ransac(xs, ys, ransac_params)

    # Predict output variability using learned neural network
    nn_hidden = ewise(sigmoid, params.h_weights * vcat(xs, ys)
                               + params.h_biases)
    nn_out = params.out_weights * nn_hidden + params.out_biases
    slope_scale, int_scale = (exp(nn_out[1]), exp(nn_out[2]))

    # Add noise
    slope = @choice(cauchy(slope_guess, slope_scale), "slope")
    intercept = @choice(cauchy(int_guess, int_scale),"intercept")

    # Generate outlier statuses from conditional distribution
    for (i, (x, y)) in enumerate(zip(xs, ys))
        p_outier = conditional_outlier(x, y, slope, intercept)
        @choice(flip(p_outlier), "outlier-$i")
    end
end
```

Importance sampling (IS) results for different proposals



(a) IS (prior) **(b)** IS (RANSAC+NN) **(c)** IS (NN) **(d)** Training objective