

Delayed Sampling via Funsors and Barriers

FRITZ OBERMEYER, Uber AI

ELI BINGHAM, Uber AI

Delayed sampling is an inference technique for automatic Rao-Blackwellization in sequential latent variable models. Funsors are a software abstraction generalizing Tensors and Distributions and supporting seminumerical computation including analytic integration. We demonstrate how to easily implement delayed sampling in a Funsor-based probabilistic programming language using effect handlers for `sample` statements and a `barrier` statement.

1 INTRODUCTION

Let us distinguish two types of inference strategies in probabilistic programming, call them *lazy* and *eager*. Let us say a strategy is *lazy* if it first symbolically evaluates or compiles model code, then globally analyzes the code to create an inference update strategy. Say a strategy *eager* if it eagerly executes model code, drawing samples from each latent variable. For example the autograd-based inference algorithms in Pyro [Bingham et al. 2018] are eager in the sense that samples are eagerly created at each sample site.

It is often advantageous to combine lazy and eager strategies, performing local exact computations within small parts of a probabilistic model, but drawing samples to communicate between those parts. Examples include Rao-Blackwellized SMC filters and their generalization as implemented in Birch [Murray et al. 2017], and reactive probabilistic programming [Baudart et al. 2019].

This work addresses the challenge of implementing boundedly-lazy inference in a Pyro-like language where samples are eagerly drawn and control flow may depend on those sample values. Our approach is to use Funsors, a software abstraction generalizing Tensors, Distributions, and lazy compute graphs. The core idea is to allow lazy sample statements during program execution, and to trigger sampling of lazy random variables only at user-specified barrier statements, typically either immediately before control flow or immediately after a variable goes out of scope. We implement our approach using effect handlers [Moore and Gorinova 2018; Pretnar 2015].

2 EMBEDDED PROBABILISTIC PROGRAMMING LANGUAGES WITH EFFECTS

Consider an embedded probabilistic programming language, extending a host language by adding two primitive statements:

- The statement `x = sample(name, dist)` is a named stochastic statement, where `x` is a Tensor or Funsor value, `name` is a unique identifier for the statement, and `dist` is a distribution (possibly a Funsor).
- The statement `state = barrier(state)` eliminates any free/delayed variables from the recursive data structure `state`, which may contain Funsor values (we restrict attention to lists of Tensors/Funsors).

We use Python for the host language in this paper.

We will implement each inference algorithm as an effect handler. Each inference algorithm will input observed data, allow running of model code, and can then interpret nonstandard model outputs as posterior probability distributions, e.g.

```
with MyInferenceAlgorithm(data=observations) as inference:
```

```
    output = model() # executes with nonstandard interpretation
```

```
posterior = inference.get_posterior(output)
```

where `observations` is a dictionary mapping sample statement name to observed value, and the resulting posterior is some representation of the posterior distribution over latent variables, e.g. an importance-weighted bag of samples.

3 INFERENCE IN SEQUENTIAL MODELS

Consider a model of a stochastic control system with piecewise control, attempting to keep a latent state z within the interval $[-10, 10]$

```
1 def model():
2     z = sample("z_init", Normal(0,1)) # latent state
3     k = 0                             # control
4     cost = 0                          # cumulative cost of controller
5     for t in range(1000):
6         if z > 10:                    # control flow depends on z
7             k -= 1
8         elif z < -10:
9             k += 1
10        else:
11            k = 0
12        cost += abs(k)
13        z = sample(f"z_{t}", Normal(z+k,1))
14        x = sample(f"x_{t}", Normal(z,1))
15    return cost
```

Now suppose we want to estimate the total controller cost given a sequence of observations x . One approach to inference is Sequential Monte Carlo (SMC) filtering. To maintain a vectorized population of particles we can rewrite the model using a vectorized conditional (e.g. `where(cond, if_true, if_false)` as implemented in NumPy and PyTorch). Further we can support resampling of particle populations by adding a barrier statement to the model code; this is needed to communicate resampling decisions to the model's local state.

```
1 def model():
2     z = sample("z_init", Normal(0,1)) # latent state
3     k = 0 * z                         # control
4     cost = 0 * z                      # cumulative cost of controller
5     for t in range(1000):
6         z,k,cost = barrier([z,k,cost])
7         k = where(z > 10, k + 1, k)
8         k = where(z < -10, k + 1, k)
9         k = where(-10 <= z & z <= 10, 0 * k, k)
10        z = sample(f"z_{t}", Normal(z+k,1))
11        x = sample(f"x_{t}", Normal(z,1))
12    return cost
```

See appendix 5.2 for details of the effect handlers to implement SMC inference.

Notice that if there were no control k (or indeed if the control were linear), we could completely Rao-Blackwellize using a Kalman filter: inference via variable elimination would be exact. To implement linear-time exact inference would require only:

- lazy versions of tensor operations such as `where`;

- a lazy interpretation for `sample` statements;
- a variable-eliminating computation of the final `log_joint` latent state.

See appendix 5.3 for details of the effect handlers to implement variable elimination. This is the approach taken by Pyro’s discrete enumeration inference, which leverages broadcasting in the host tensor DSL to implement lazy sampling and lazy tensor ops.

To implement delayed sampling and partially Rao-Blackwellize our SMC inference, we can combine the two above approaches, emitting lazily sampled values from `sample` statements and eagerly sampling delayed samples at `barrier` statements, relying on a variable elimination engine to efficiently sample from the partial posterior. In contrast to the variable-elimination interpretation of `barrier`, this interpretation guarantees all local state is ground and hence can be inspected by e.g. `where`. See appendix 5.4 for details of effect handlers to implement delayed sampling.

4 CONCLUSION

We demonstrated flexible inference algorithms in an embedded probabilistic programming language with a new `barrier` statement and support for lazy computations represented as Funsors. While Funsor implementations are few, this same technique can be used for delayed sampling of discrete latent variables using only a Tensor library and a variable elimination engine such as `opt_einsum` [Smith and Gray 2018].

REFERENCES

- Guillaume Baudart, Louis Mandel, Eric Atkinson, Benjamin Sherman, Marc Pouzet, and Michael Carbin. 2019. Reactive probabilistic programming. *arXiv preprint arXiv:1989.07563* (2019).
- Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. 2018. Pyro: Deep Universal Probabilistic Programming. *arXiv preprint arXiv:1810.09538* (2018).
- Dave Moore and Maria I Gorinova. 2018. Effect handling for composable program transformations in Edward2. *arXiv preprint arXiv:1811.06150* (2018).
- Lawrence M Murray, Daniel Lundén, Jan Kudlicka, David Broman, and Thomas B Schön. 2017. Delayed sampling and automatic rao-blackwellization of probabilistic programs. *arXiv preprint arXiv:1708.07787* (2017).
- Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.
- Daniel G. A. Smith and Johnnie Gray. 2018. `opt_einsum` - A Python package for optimizing contraction order for `einsum`-like expressions. *Journal of Open Source Software* 3, 26 (2018), 753. <https://doi.org/10.21105/joss.00753>

5 APPENDIX: DETAILS OF EFFECT HANDLING

5.1 Effect handling framework

Before describing effect implementations, we provide a simple framework for effect handling embedded in Python. Let’s start with a standard interpretation, implemented as an effect handler base class.

```
class StandardHandler:
    def __enter__(self):
        # install this handler at the beginning of each with statement
        global HANDLER
        self.old_handler = HANDLER
        HANDLER = self
        return self

    def __exit__(self, type, value, traceback):
        # revert this handler at the end of each with statement
```

```

148         global HANDLER
149         HANDLER = self.old_handler
150
151     def sample(self, name, dist):
152         return dist.sample() # by default, draw a random sample
153
154     def barrier(self, state):
155         return state # by default do nothing
156
157     HANDLER = StandardHandler()
158 
```

Next we can define user facing statements with late binding to the active effect handler.

```

161     def sample(name, dist):
162         return HANDLER.sample(name, dist)
163
164     def barrier(state):
165         return HANDLER.barrier(state)
166 
```

5.2 Effect handlers for inference via Sequential Monte Carlo

We can now implement Sequential Monte Carlo inference by maintaining a vector `log_joint` of particle log weights, sampling independently each particle at `sample` statements, and resampling at `barrier` statements.

```

172     class SMC(StandardHandler):
173         def __init__(self, data, num_particles=100):
174             self.data = data
175             self.log_joint = zeros(num_particles)
176             self.num_particles = num_particles
177
178         def sample(self, name, dist):
179             if name in self.data:
180                 value = self.data[name]
181             else:
182                 value = dist.sample(sample_shape=self.log_joint.shape)
183             self.log_joint += dist.log_prob(self.data[name])
184             return value
185
186         def barrier(self, state):
187             index = Categorical(logits=self.log_joint).sample()
188             self.log_joint[:] = 0
189             return [x[index] for x in state]
190
191         def get_posterior(self, value):
192             probs = exp(self.log_joint)
193             probs /= probs.sum()
194             return {"samples": value, "probs": probs}
195 
```

5.3 Effect handlers for exact inference via Variable Elimination

We can implement variable elimination by leveraging lazy compute graphs and exact forward-backward computation of the Funsor library. This handler ignores barrier statements.

```

class VariableElimination(StandardHandler):
    def __init__(self, data, num_particles=100):
        self.data = data
        self.log_joint = funsor.Number(0)
        self.num_particles = num_particles

    def sample(self, name, dist):
        if name in self.data:
            value = self.data[name]
        else:
            value = funsor.Variable(name) # create a delayed sample
            self.log_joint += dist.log_prob(self.data[name])
        return value

    def get_posterior(self, value):
        return funsor.Expectation(log_joint, value)

```

5.4 Effect handlers for inference via Delayed Sampling

Finally we can implement delayed sampling by extending the VariableElimination handler to eagerly eliminate variables whenever a barrier statement is encountered.

```

class DelayedSampling(VariableElimination):
    def barrier(self, state):
        subs = self.log_joint.sample(state.inputs, self.num_samples)
        self.log_joint = self.log_joint(**subs)
        return [x(**subs) for x in state]

```