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 it first symbolically evaluates or compiles model code, then globally analyzes the code to create an inference update strategy. By contrast consider 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.

However 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 before control flow. We implement our approach using two effect handlers [Moore and Gorinova 2018; Pretnar 2015].

2 DELAYED SAMPLING

Delayed sampling [Murray et al. 2017] is an inference technique for automatic Rao-Blackwellization in sequential latent variable models. Delayed sampling was introduced in the Birch probabilistic programming language [Murray and Schön 2018].

3 FUNSORS

Funsors [Obermeyer et al. 2019] are a software abstraction generalizing Tensors and Distributions and supporting seminumerical computation including analytic integration.

4 DELAYED SAMPLING WITH FUNSORS

Consider an embedded probabilistic programming language, extending a host language with two primitive statements.

- The statement `x = sample(name,dist)` is a named stochastic statement, where `x` is a Funsor value, `name` is a unique identifier for the statement, and `dist` is a Funsor distribution.

Authors' addresses: Fritz Obermeyer, Uber AI, fritzo@uber.com; Eli Bingham, Uber AI, eli.bingham@uber.com.

- The statement `x = barrier(x)` eliminates any free variables from the recursive data structure `x`, which may contain Funsor values.

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()

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.

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

```
def model():
    z = sample("z_init",Normal(0,1))  # latent state
    k = 0                             # control
    cost = 0                          # cumulative cost of controller
    for t in range(1000):
        if z > 10:                    # control flow depends on z
            k -= 1
        elif z < -10:
            k += 1
        else:
            k = 0
        cost += abs(k)
        z = sample(f"z_{t}",Normal(z+k,1))
        x = sample(f"x_{t}",Normal(z,1))
    return cost
```

Now suppose we want to estimate the total controller cost given a sequence of observations `x`. One approach to inference is to apply 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; this is needed to communicate resampling decisions with the model's local state.

```
def model():
    z = sample("z_init",Normal(0,1))    # latent state
    k = 0 * z                           # control
    cost = 0 * z                        # cumulative cost of controller
    for t in range(1000):
        z,k,cost = barrier((z,k,cost))
        k = where(z > 10, k + 1, k)
        k = where(z < 10, k + 1, k)
        k = where(-10 <= z & z <= 10, 0 * k, k)
        z = sample(f"z_{t}",Normal(z+k,1))
        x = sample(f"x_{t}",Normal(z,1))
```

```
    return cost
```

See appendix for details of the effect handlers `condition` and `log_joint` 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, we require only:

- lazy versions of tensor operations such as *where*;
- a lazy interpretation for `sample` statements;
- a variable-eliminating interpretation of `barrier` statements; and
- a variable-eliminating computation of the final `log_joint` latent state.

See appendix for details of the effect handlers `condition` and `log_joint` to implement SMC inference. 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 partially Rao-Blackwellize our SMC inference, we can combine the two approaches, emitting lazily sampled values from `sample` statements and eagerly sampling delayed samples at `barrier` statements. In contrast to the variable-elimination interpretation of `barrier`, this interpretation guarantees all local state is ground and hence can be inspected by conditionals. See appendix for details of effect handlers `condition` and `log_joint` to implement delayed sampling.

5 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.

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).

Lawrence M Murray and Thomas B Schön. 2018. Automated learning with a probabilistic programming language: Birch. *Annual Reviews in Control* (2018).

Fritz Obermeyer, Eli Bingham, Martin Jankowiak, Du Phan, and Jonathan P Chen. 2019. Functional tensors for probabilistic programming. *arXiv preprint arXiv:1910.10775* (2019).

Matija Pretnar. 2015. An introduction to algebraic effects and handlers. invited tutorial paper. *Electronic Notes in Theoretical Computer Science* 319 (2015), 19–35.

6 APPENDIX: DETAILS OF EFFECT HANDLING

6.1 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, also implemented as an effect handler.

```
class StandardHandler:
    def __enter__(self):
```

```
        global HANDLER
        self.old_handler = HANDLER
        HANDLER = self
        return self

    def __exit__(self, type, value, traceback):
        global HANDLER
        HANDLER = self.old_handler

    def sample(self, name, dist):
        return dist.sample()

    def barrier(self, state):
        return state

HANDLER = StandardHandler()

def sample(name, dist):
    return HANDLER.sample(name, dist)

def barrier(state):
    return HANDLER.barrier(state)
```

Note that the global `sample` and `barrier` statements are late binding.

6.2 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, multiply sampling at `sample` statements, and resampling at `barrier` statements.

```
class SMC(StandardHandler):
    def __init__(self, data, num_particles=100):
        self.data = data
        self.log_joint = zeros(num_particles)
        self.num_particles = num_particles

    def sample(self, name, dist):
        if name in self.data:
            value = self.data[name]
        else:
            value = dist.sample(sample_shape=self.log_joint.shape)
        self.log_joint += dist.log_prob(self.data[name])
        return value

    def barrier(self, state):
        index = Categorical(logits=self.log_joint).sample()
        self.log_joint[:] = 0
        return [x[index] for x in state]
```

```
    def get_posterior(self, value):
        probs = exp(self.log_joint)
        probs /= probs.sum()
        return {"samples": value, "probs": probs}
```

6.3 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.

```
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):
        subs = self.log_joint.sample(self.num_particles)
        return value(**subs)
```

6.4 Effect handlers for inference via Delayed Sampling

We can now 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]
```