Delayed Sampling via Barriers and Funsors

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 an embedded probabilistic programming language using Funsors and effect handlers for `sample` statements and a `barrier` statement.

1 INTRODUCTION

Recently interest has grown in techniques to implement light-weight probabilistic programming languages (PPLs) as embedded domain specific languages (DSLs) in other popular languages used in industry, e.g. Figaro [?] and Ranier [?] in Scala, Edward [Tran et al. 2017] in Python+Tensorflow, Pyro [Bingham et al. 2018] in Python+PyTorch, Infergo [?] in Go, and the PPX protocol [?] for integrating Python+PyTorch+SHERPA. Among approaches to lightweight embedded PPLs, effect handlers have shown promise [??], allowing PPLs like Pyro and Edward2 [?] to apply program transformations without needing program analysis or even compilation. Here we show that delayed sampling can also be implemented in a light-weight embedded PPL using only a `barrier` statement and limited support from an underlying math library.

2 DELAYED SAMPLING

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. Say a strategy is *eager* if it eagerly executes model code, drawing samples from each latent variable. For example the gradient-based Monte Carlo inference algorithms in Pyro 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 lightweight embedded PPL 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.

3 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).

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

- The statement `state = barrier(state)` eliminates any free/delayed variables from the recursive observations structure `state`, which may contain Tensor or Funsor values (we will restrict attention to lists of Tensors/Funsors for ease of exposition).

We use Python for the host language in this paper.

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

```
with MyInferenceAlgorithm(observations=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.

4 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 $[-10, 10]$

```
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 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)`[1] as implemented in NumPy, PyTorch and TensorFlow). 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] https://docs.scipy.org/doc/numpy/reference/generated/numpy.where.html

```
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])  # inference may resample here
        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 6.2 for details of the effect handler 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 6.3 for details of the effect handler to implement variable elimination. This is the approach taken by Pyro's discrete enumeration inference, which leverages broadcasting in the host tensor DSL to simulate lazy sampling and lazy tensor ops.

To implement delayed sampling, and thereby 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 draw random samples 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 conditionals like `where`. See appendix 6.4 for details of effect handler 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[2] 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).

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).

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

[2] See the examples in http://pyro.ai/examples/enumeration.html

Dustin Tran, Matthew D Hoffman, Rif A Saurous, Eugene Brevdo, Kevin Murphy, and David M Blei. 2017. Deep probabilistic programming. *arXiv preprint arXiv:1701.03757* (2017).

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, 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
        global HANDLER
        HANDLER = self.old_handler

    def sample(self, name, dist):
        return dist.sample()  # by default, draw a random sample

    def barrier(self, state):
        return state  # by default do nothing

HANDLER = StandardHandler()
```

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

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

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

6.2 Effect handlers for inference via Sequential Monte Carlo

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

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

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

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

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

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

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

6.4 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)
        state = [x(**subs) for x in state]
        return state
```