Comprehension, maps, and eager eval for differentiable probabilistic programs

Bob Carpenter

Center for Computational Mathematics

Flatiron Institute

CAPP 2022

Differentiable Prob Prog

- **differentiable**:
  code smooth function f and derive efficient ∇f, $\nabla\nabla f, \ldots$
  - cost is restrictions on coding (library / control flow)
  - e.g., Adol-C, Sacado, CppAD, **Stan Math**, TensorFlow, PyTorch, JAX, Zygote.jl, ...
- **differentiable probabilistic**:
  $f(\theta) = \log p(\theta \mid y) + \text{const.}$ is target log density
  plus sampling / optimization / variational approx.
  - e.g., AMB, **Stan**, PyMC, Pyro, Turing.jl, ...
- **most composable**: Zygote.jl, JAX; Turing.jl

Reverse-mode Autodiff

- **Constant cost** multiple for $f : \mathbb{R}^N \rightarrow \mathbb{R}$
- **Forward pass**: eval program to construct expression DAG
- **Reverse pass**: propagate derivatives in topological order
- **Cache misses** propagating subexpression derivatives

$$\frac{\partial \log p(\theta \mid y)}{\partial e_n} \mathrel{+}= \frac{\partial \log p(\theta \mid y)}{\partial f(e_1, \ldots, e_N)} \times \frac{\partial f(e_1, \ldots, e_N)}{\partial e_n}.$$

  - $e_1, \ldots, e_N$ **sequential** in memory after forward eval

- **efficiency** requires cache locality (& branch prediction)
- on miss, RAM fetch ≈ 150 clock cycles (!!!)

Map-Reduce for Likelihoods

- if data y **conditionally independent** given parameters θ,

$$p(y \mid \theta) \;=\; \sum_{n=1}^{N} \log p(y_n \mid \theta).$$

- **map** applies a function f to each element of a vector v,

$$\mathrm{map}(f, v) = \begin{bmatrix} f(v_1) & \cdots f(v_N) \end{bmatrix}^\top.$$

- Stan reduces map output with sum for likelihoods

$$\mathrm{reduce\_sum}(f, y, \theta_1, \ldots, \theta_K) = \sum_{n=1}^{N} f(y_n, \theta_1, \ldots, \theta_K).$$

Eager Subgraph Evaluation

- Runtime form of **partial evaluation**
- Eval $\nabla_x f(e_1(x), \ldots, e_N(x))$ any time after eval $f(\cdots)$
  - **Stan**: nested reverse-mode; Adept: forward-mode
- **Reduces memory footprint** for subexpressions to $\mathcal{O}(|x|)$
- Which increases **cache locality** and speeds up throughput
- **Parallel** evaluation of subexpression gradients
  - Stan **scales up** with threads (TBB) and **scales out** with MPI

- **communicate gradients** back in $\mathcal{O}(|x|)$
- MPI pushes data to **node local** on construction
- orthogonal to **GPU** usage

Autodiff Variable Locality

- Stan uses pointer to implementation for RAII

```
template <typename T>
struct var {
  vari* vi_;
};
```

```
template <typename T>
struct vari {
  T value_;  T adjoint_;
};
```

- Two ways to code matrices (vectors, tensors, etc.)

```
Eigen::Matrix<var<double, -1, -1>> A;

var<Eigen::Matrix<double, -1, -1>> B;
```

  - `A` can autodiff ‘Matrix¡T¿‘ algorithms
  - `B` is memory local for matrix derivatives

- only `A` supports lvalue indexing (element assignment)

Gaussian Process (GP)

- A GP is a non-parametric[1] nearest neighbors model
- Data size N requires $N \times N$ **covariance matrix** Σ
  - for **covariance function** κ, data x, params θ,

$$\Sigma_{i,j} = \kappa(i, j, x, \theta)$$

- In rich models, Σ is a **sum of covariance** matrices
- Adding large N covariance matrices is a **memory disaster**

[1] i.e., lots of parameters

- Want to **scale GPs** in Stan from $N < 1000$ to $N > 10,000$

Comprehensions

- From **set theory** (late 1800s), consistent in **ZF** (early 1900s)

  $B = \{x \in A : \phi(x)\}$ is a set if A is a set and $\phi : A \to \text{Bool}$

- Introduced to programming languages (early 1970s) and

  – from POP2 to Miranda to Haskell to Python to $\cdots$ Stan

- Python **list comprehension** is ordered

```
b = [x for x in A if phi(x)]
```

Matrix Comprehension

- **Stan** is adopting a **variadic** covariance function style

```
cov_matrix[N, N] B = comp_mat(f, a_1, ..., a_N);
```

  – defines `B` as if evaluated in the loop

```
for (i in 1:N)
  for (j in 1:N)
    B[i, j] = f(i, j, a1, ... aN);
```

- Just a specialized map
  – **partially evaluate** gradients $\frac{\partial f(i, j, a_1, \ldots, a_N)}{\partial a_n}$
  – **parallelize** eval and gradients over both loops
- For GPs, **add covariance functions** not covar matrices

Compiler/Runtime Automation

- autodetect when we can parallelize loops with map
- auto load balance parallel jobs
  - using Intel Thread Building Blocks (TBB) for pooling/allocation

Thanks for Listening

- Stan language transpiler (OCaml):
  `github.com/stan-dev/stanc3`
  - Carpenter, B., Gelman, A., Hoffman, M.D., Lee, D., Goodrich, B., Betancourt, M., Brubaker, M., Guo, J., Li, P. and Riddell, A., 2017. **Stan: A probabilistic programming language**. *Journal of Statistical Software*, 76(1).
- Stan math library (C++):
  `github.com/stan-dev/math`
  - Carpenter, B., Hoffman, M.D., Brubaker, M., Lee, D., Li, P. and Betancourt, M., 2015. **The Stan math library: Reverse-mode automatic differentiation in C++**. *arXiv* 1509.07164.