A Bayesian Computation Graph for High-Performance Gradient Evaluation on the JVM

AVI BRYANT

1 INTRODUCTION

Rainier is a Scala library for building fixed-structure, continuous-parameter generative models, to be sampled and optimized using gradient-based methods. Its core contribution is a static computation graph targeted at Bayesian model inference in a Java Virtual Machine production environment.

In particular, Rainier is designed to be deployed to large data processing clusters running Spark, Hadoop, or similar JVM-based systems, which are very commonly found in industry. These environments often discourage the use of native (as opposed to JVM) libraries, and do not have access to GPUs. As a result, Rainier must rely on a heavily optimizing compiler, directly targeting the JVM, to achieve high performance.

We will focus on two stages of the compiler: first, a partial evaluation stage that attempts to pre-compute as much as possible of the function represented by the graph, given fixed model inputs and observations (but still allowing parameter values to vary); second, the generation of low-level JVM bytecode designed to be easily compiled to efficient machine code by the JVM's just-in-time compiler.

We will also briefly discuss two novel features of the computation graph that are particularly helpful in the Bayesian context: log-density annotations on parameter nodes, and the use of interval arithmetic for tracking the support of each node of the graph.

2 RELATED WORK

Theano [5] and TensorFlow [1] both provide similar computation graphs based on individual mathematical operations. Both of them use a combination of Python and native libraries, and achieve their best performance on GPUs, making them challenging to deploy in a JVM context. They also target stochastic, mini-batch algorithms, which limits the opportunities for partial evaluation; by contrast, Rainier targets full-batch gradient algorithms like HMC and L-BFGS, enabling specialization of the computation to the entire dataset.

Stan [4] targets a very similar problem space: fixed-structure continuous-parameter models to be used with gradient-based methods. It similarly compiles to an intermediate form (C++ source code) that can be further compiled to produce high-performance machine code. However, it compiles the model without having access to any of the observation data, which limits its ability to perform partial evaluation optimizations. The requirement to have the C++ compiler available when building models also significantly increases the complexity of deploying it in a production setting, rather than for research use.

The marginalization described in section 3.3 is similar to that described in Autoconj [2], and in particular shares a similar canonical form.

3 PARTIAL EVALUATION

3.1 Motivation

Gradient-based samplers, such as HMC, must repeatedly evaluate a joint log-density function, $logp(\theta; x)$, along with its gradient with respect to θ. Each of these thousands or even millions of

Author's address: Avi Bryant, `avi@avibryant.com`.

evaluations will have different parameter values θ, but for any given sampling run, the structure of the function, and the observation values x, will not change.

Each model's *logp* function is unique. However, it is not typical to write a *logp* function from scratch using only mathematical primitives. Instead, this function will be assembled as a composition of many smaller, reusable functions, provided either by the standard library or by the user. One such function might be, for example, a $normal\_log\_pdf(\mu, \sigma, y)$. These functions will be written treating all of their parameters as free, eg,

$$normal\_log\_pdf(\mu, \sigma, y) = log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{1}{2}\left(\frac{y-\mu}{\sigma}\right)^2 \tag{1}$$

However, when used in a particular model, any (though usually not all) of those parameters will in fact be known constants. For example, when sampling from a $Normal(0, 1)$ random variable, and approximating π, the equation could be simplified as follows:

$$normal\_log\_pdf(0, 1, y) = \frac{-y^2}{2} - 0.9189 \tag{2}$$

If, on the other hand, μ and σ are random variables but y is known, it might be advantageous to distribute out the $(y - \mu)^2$ term, obtaining:

$$normal\_log\_pdf(\mu, \sigma, 3) = log\left(\frac{1}{\sigma\sqrt{2\pi}}\right) - \frac{9 - 6\mu + \mu^2}{2\sigma^2} \tag{3}$$

Simplifications can also become available because of function composition. For example, the general purpose poisson log PDF is:

$$poisson\_log\_pdf(\lambda, y) = ylog\lambda - \lambda - log(y!) \tag{4}$$

However, in the common case of a log link function, we will have $\lambda = e^x$ for some x. We'd like to be able to take advantage of that and simplify to $yx - \lambda - log(y!)$; and indeed, Stan provides a special `poisson_log` function for exactly this case.

3.2 Goal

Rainier's goal is to construct a similar simplification of the entire *logp* function, during model construction and compilation. In almost every model this will yield a significant reduction in the number of mathematical operations needed to compute the gradient, and removes the need for manual optimizations like `poisson_log`, as well as reducing the need for manual pre-processing of the observations outside of the model. In some common cases, this can also provide an asymptotic speed-up, simplifying a log-likelihood computation over N observations from $O(N)$ to $O(1)$.

3.3 Asymptotic Speedup

In particular, this asymptotic speed-up is available if the log-likelihood function $L(\theta; x)$ can be decomposed into $f(\theta) \cdot g(x)$, where f is a vector function on the parameter vector θ and g is a vector function on a single row of observations x. The sum of the log-likelihoods $\sum_{i=1}^{N} L(\theta; x_i)$ can then be decomposed into $f(\theta) \cdot k$, where $k = \sum_{i=1}^{N} g(x_i)$ and can be pre-computed, thus marginalizing the observations. This decomposition often falls naturally out of Rainier's partial evaluation system (for example, as in equation 3), and will be recognized and exploited when it does.

3.4 Approach

Rainier's approach to partial evaluation is unusual in that it is incremental, eagerly performing the partial evaluation as each new node is introduced to the graph. For example, consider the following Scala code:

```
val lambda = x.exp
val z = lambda.log * 10/(5*2)
```

In this simple case, while constructing the object to be assigned to z, Rainier will first strip away the *exp* node in response to the *log* operation, and then ignore the multiplication by 1, and return exactly the same node referenced by x. This keeps the graph always in a compact form, rather than allowing it to expand to a point where it would then take an infeasible amount of resouces to run an optimization pass.

Along with some special cases like in the example above, the key to this incremental approach is Rainier's choice of representation for arithmetic operation nodes.

A standard computation graph would provide binary nodes for arithmetic operators like $x + y$ or $x * y$, where x and y are references to child nodes. Rainier relies instead on a pair of richer node types. The first of these, the `Line` node, represents expressions of the following form:

$$a + \sum_{i=1}^{n} b_i x_i \tag{5}$$

Where a is a constant, b is a vector of constants of length n, and x is a vector of references to child nodes of length n.

Line nodes are closed under addition, and under multiplication by a constant, without changing the size of the graph. Although it is sometimes necessary to extend the size of the vector to accomodate references to child nodes that have not previously been seen, in most cases there is a finite upper bound on the number of distinct child nodes, which are always non-linear combinations and transformations of model parameters.

The second, which Rainier calls the `LogLine` node, represents expressions of the form:

$$\prod x_i^{b_i} \tag{6}$$

Similarly, LogLine nodes are closed under multiplication, and under exponentiation by a constant. In some cases they can also be effectively closed under addition, by using distribution rules on both sides of the addition to convert them into Line nodes with congruent terms.

The use of this representation has been remarkably effective in practice at achieving significant partial evaluation of a variety of models.

4 COMPILATION TO JVM BYTECODE

4.1 Target

Rainier compiles computation graphs to a restricted subset of the JVM which consists solely of static methods with the following properties:

- They never allocate any objects or arrays.
- They never exceed 8000 bytecodes in length.
- They accept exactly two arrays of doubles as parameters. The first one is treated as read-only, the second as read-write.
- Their return value is always a single double.
- They have no looping or recursion.

The first two items are important for performance: never allocating means that the garbage collector should never need to run during sampling; staying under 8000 bytecodes in length avoids a problem where, with default settings, the JVM's JIT compiler will avoid compiling methods greater than that length to machine code, running them in interpreted mode instead.

The remaining items are to provide a simple computational model that is easy to target and to reason about.

4.2 Compilation

A single compilation unit will consist of one, or usually more than one, computation graphs to be compiled in sequence. One top-level method will be produced for each of these, whose return value corresponds to the (necessarily scalar) root node of the graph. For example, it would be common to compile a graph representing a joint log-density, along with graphs representing each element of its automatically-derived gradient. (Note that these graphs will all share a considerable amount of structure.)

Any inputs that might vary between invocations (eg, model parameter values) are provided in the first, read-only array argument. The second, read-write array argument is used to memoize intermediate values which are needed by more than one compiled method. For example, if a value is first computed in the log-density method, but is also needed in one of the gradient methods, it will be stored in this array at the time it is first computed, and later retrieved as needed. Note that this means that it is essential to invoke these methods in the same order in which they were compiled: you cannot, for example, compute just the gradient without first computing the log-density, because the shared array will not be in the correct state.

To comply with the limitation on the length of individual methods, the compiler may split the compilation of a single computation graph into a tree of separate methods, with the top-level method calling one or more second-level methods, which may call one or more third-level methods, and so on, always passing both array arguments along. The same technique of writing into the second array is used to share memoized values between them.

4.3 Observations

Note that for small datasets, observations (or partially-evaluated transformations thereof) can simply be embedded as constants in the compiled method. For larger datasets, it's possible to treat the data instead as being similar to parameters, and produce compiled methods that deal with only a single observation at a time, reading its values from the input array. The higher levels of Rainier's API carefully manage the shared state arrays and invocation of the compiled methods in order to separately (sequentially or in parallel) compute and then sum the contributions to the log-density and the gradient of each observation.

See Appendix A for a representative sample of the code generated by the compiler.

5 PERFORMANCE

Rainier was benchmarked against Stan on the same hardware (2019 2.3Ghz MacBook Pro 16"), on a small number of models taken from Stan's example models repository [1]. They were transcribed as closely as possible to Rainier. Timings for Stan was taken from its self-reported gradient evaluation times; timings for Rainier were measured using the JMH benchmarking library [2]. See Table 1 for results.

[1]https://github.com/stan-dev/example-models/
[2]https://openjdk.java.net/projects/code-tools/jmh/

Table 1. Gradient evaluation time (μs)

Model	Stan	Rainier
eight_schools	24	1
ar_k	97	4
low_dim_gauss_mix	292	650
glmm_poisson	582	235

Table 2. kidiq evaluation time (μs)

N	Stan	Rainier
100	37	10
200	57	10
400	104	10

For the `kidiq_multi_preds` linear regression example, both Rainier and Stan were benchmarked using the first 100, 200, and 400 of the 434 observations in the examples repository. As you can see in Table 2, Rainier's partial evaluation was able to make this gradient evaluation asymptotically faster than the naive approach.

6 ANNOTATED PARAMETERS

6.1 Motivation

Computation graphs like Theano and TensorFlow were designed for constructing deep neural network models, which do not generally have explicit priors on parameters. A typical model will transform and combine parameters and input data to produce some output value, and then compute a single loss function by comparing that output value with a target value. Often, the actual parameter creation is encapsulated in some kind of function or module that returns an intermediate computed value and does not provide access to the raw parameters, but this doesn't pose any problems because the loss function at the root of the computation graph is the only thing that matters.

In a Bayesian context, a function or module that creates new parameters must not only return a computation involving those parameters, but must also in some way provide new terms for the joint log-density function that represent the priors for those parameters. Because it would be awkward to thread those terms through as return values, this is typically handled as some kind of side-effect: for example, by mutating a global `log_prob` value (as in Stan) or a thread-local model context (as in PyMC3 [3]). This use of global, mutable state to capture prior probabilities can make it difficult to reason about your code, most especially in an interactive environment such as a REPL or notebook.

6.2 Approach

Rainier avoids any such global state by allowing each parameter node in the graph to optionally be annotated with a `density` sub-graph, which, as a special-case, will cyclically refer back to the parameter node. A model is primarily defined by one or more log-likelihood computations, similar to the loss-functions used in a DNN context; however, these are summed with the log-densities that annotate any parameter in the graph that is reachable from the likelihood node, to assemble the full joint log-density. This allows fully immutable construction of Bayesian computation graphs (with

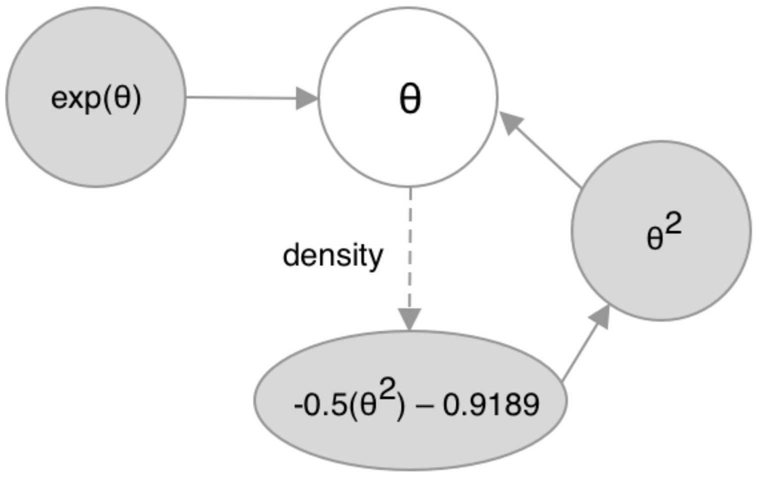

Fig. 1. The computation graph for a LogNormal(0,1) parameter

the use of a special primitive for setting up the cyclical annotation when creating a parameter). See Figure 1 for a simple example.

7 SUPPORT TRACKING

7.1 Motivation

It is commonplace when constructing a Bayesian model to use some random variable defined earlier in the model to parameterize a distribution: for example, to use a parameter σ to parameterize some $Normal(\mu, \sigma)$. Often, there are restrictions on the support of that random variable (eg in this case, $\sigma > 0$). Errors in specifying the model can lead to violations of those restrictions; the more clearly and directly those violations can be caught, the more easily the modeler will be able to discover and fix the error.

7.2 Approach

Rainier tracks the support of every random variable in the model - that is to say, every node in the computation graph - as a pair of double values representing lower and upper bounds of that support. As the base case, parameter nodes always have the bounds `(-inf,inf)`, and a node representing a constant value `k` will always have the bounds `(k,k)`. From there, nodes representing mathematical operations will have bounds obtained from straightforward interval arithmetic on the bounds of their terms. Most importantly, this allows distributions to check the support of their parameterization at construction time, and raise a warning or error if it is not appropriate. The availability of these bounds is also valuable for debugging and understanding models, in an interactive or visualization context.

ACKNOWLEDGMENTS

We would like to thank Stripe for funding and motivating much of this work, and to particularly acknowledge Roban Kramer, Mio Alter, and Andrew Metcalf for their feedback on Rainier's API and implementation.

REFERENCES

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. 2015. TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems. http://tensorflow.org/ Software available from tensorflow.org.

[2] Matthew D. Hoffman, Matthew J. Johnson, and Dustin Tran. 2018. Autoconj: Recognizing and Exploiting Conjugacy Without a Domain-Specific Language. *CoRR* abs/1811.11926 (2018). arXiv:1811.11926 http://arxiv.org/abs/1811.11926

[3] John Salvatier, Thomas V. Wiecki, and Christopher Fonnesbeck. 2016. Probabilistic programming in Python using PyMC3. *PeerJ Computer Science* 2 (apr 2016), e55. https://doi.org/10.7717/peerj-cs.55

[4] Stan Development Team. 2018. Stan Modeling Language Users Guide and Reference Manual, Version 2.18.0. http://mc-stan.org/

[5] Theano Development Team. 2016. Theano: A Python framework for fast computation of mathematical expressions. *arXiv e-prints* abs/1605.02688 (May 2016). http://arxiv.org/abs/1605.02688

A COMPILED FUNCTION EXAMPLE

This is a portion of the bytecode generated for a linear regression model, decompiled into its Java equivalent for readability (Rainier directly generates the JVM bytecode, and does not rely on any Java compiler.)

```
public static final double f0(double[] in, double[] g) {
    g[8] = in[1] * in[1];
    g[2] = in[0] * in[0];
    g[9] = (1.0 + g[2]) * 3.141592653589793;
    g[6] = in[2] * in[2];
    g[10] = in[3] * in[3];
    double d = -0.05263157894736842 * in[2];
    g[0] = -5.535908297792862E-4 * g[8] / g[2];
    g[1] = -5.535908297792862E-4 * in[1] * in[3] / g[2];
    g[3] = -5.535908297792862E-4 * in[3] * in[1] / g[2];
    g[4] = -5.535908297792862E-4 * g[10] / g[2];
    double d2 = -0.04518382122361609 * in[2];
    return (-0.0014607260940301225 +
    -3.459942686120538E-5 * g[8] +
    java.lang.Math.log(g[9]) * -6.919885372241076E-5 +
    -3.459942686120538E-5 * g[6] +
    -3.459942686120538E-5 * g[10] +
    -5.0038457168762935 * g[6] / g[2]
    + d * in[1] / g[2] +
    d * in[3] / g[2] +
    -0.05263157894736842 * in[1] * in[2] / g[2] +
    g[0] + g[1] +
    -0.05263157894736842 * in[3] * in[2] / g[2] +
    g[3] + g[4] +
    in[2] / g[2] +
    0.010518225765806436 * in[3] / g[2] +
    0.010518225765806436 * in[1] / g[2] +
    -3.6878820792274802 * g[6] / g[2] +
    d2 * in[1] / g[2] +
    d2 * in[3] / g[2] +
    f1(in, g) + g[0]);
}

public static final double f1(double[] in, double[] g) {
    return -0.04518382122361609 * in[1] * in[2] / g[2];
}
```