Design Doc: Making Pyro compatible with the PyTorch JIT

Authors: Eli Bingham, Fritz Obermeyer
First Edit 2018-04-27 - Last Edit 2018-04-30

Objectives

Jit compile the coarsest possible parts of Pyro: optimally svi.step() and hmc's leapfrog step.

Background

The PyTorch team have been working on a tracing just-in-time compiler for PyTorch computation graphs (see usage overview and source code). The JIT works by recording a trace of all of the PyTorch tensor operations that occur within an execution of a function or Module and converting that trace into a program that runs in a custom runtime.

Design overview

SVI

SVI offers several potential targets for compilation. The largest potential chunk is an entire SVI.step(), or possibly even multiple steps.

HMC

HMC also offers multiple targets for compilation. In HMC, because of the MH step, the largest potential chunks are the computation of the MH acceptance ratio or the leapfrog integrator in the kernel. Due to the dynamic nature of NUTS, the largest potential chunk is a single step of the leapfrog integrator.

Detailed design

Because the JIT only works with pure functions with tensor input and output, some of Pyro's internals will have to be refactored to conform to such a specification before they can take advantage of JIT acceleration.

The torch.jit.compile decorator works roughly as follows (see also: its source and docstring):

1. User writes a python function declaring all data dependencies (as tensors) and returning a tensor or list of tensors.
2. User decorates the python function with @torch.jit.compile(nderivs=n)
3. On first call of the decorated function, pytorch will record the ids of the tensor inputs, record a compute graph of all tensor operations, and record the ids of the tensor outputs.
4. If nderivs>0 PyTorch waits until .backward() or grad() is called nderivs times before stopping recording. TODO learn about details here.
5. PyTorch creates a simple Python-free version of the function that replays the recorded straight-line trace, but pointed at the new input tensors. There is currently only a little bit of optimization: fusing some pointwise operations.
6. On subsequent calls of the decorated function, PyTorch checks that tensor shapes match and if so uses its optimized version instead. This eliminates all Python overhead.

A general pattern: JIT-ing parametrized functions

The JIT only works with pure tensor functions and nn.Modules. It supports closing over static, read-only Python variables that aren't tensors, but most Pyro functions are closures over the parameter store. To bridge this gap, we'll need a utility for exposing parameters to the JIT as

function arguments or as parameters in a Module. With this in hand it would very easy to write loss functions or other operations that include calls to the parameter store, e.g.:

```
@pyro.ops.jit.compile(nderivs=1)
def simple_elbo(*args):
    guide_tr = trace(guide).get_trace(*args)
    model_tr = trace(replay(model, trace=guide_tr)).get_trace(*args)
    return model_tr.log_prob_sum() - guide_tr.log_prob_sum()
```

See an implementation of `pyro.ops.jit.compile` here

Work plan

Pyro.infer.SVI refactoring

See the TraceEnum_ELBO implementation in Pyro's torch-jit branch for example.

Pyro.infer.HMC refactoring

Modify the leapfrog integrators or their components to be compatible with the JIT

PyTorch feature requests

Support broadcasting in gradient computations. Currently we are blocked by the error:
`RuntimeError: saved_variables() needed but not implemented in ExpandBackward`

Benchmarking

Implement some performance benchmarks to understand the performance improvements associated with the JIT. As a simple approximation we can

```
pytest --duration=0 tests/infer/test_jit.py  # in the torch-jit branch
```

Accelerating slow unit and integration tests

Assuming TraceGraph_ELBO can be JIT-compiled, the Gaussian pyramid integration tests and other slow SVI tests are prime candidates for massive speedups