Journal of Machine Learning Research 20 (2019) 1-6 Submitted 6/18; Revised 1/19; Published 2/19

Pyro: Deep Universal Probabilistic Programming

Eli Bingham ELI.BINGHAM@UBER.COM
Jonathan P. Chen JPCHEN@UBER.COM
Martin Jankowiak JANKOWIAK@UBER.COM
Fritz Obermeyer FRITZO@UBER.COM
Neeraj Pradhan NPRADHAN@UBER.COM
Theofanis Karaletsos THEOFANIS@UBER.COM
Rohit Singh ROHITS@UBER.COM
Paul Szerlip PAS@UBER.COM
Uber AI Labs
Uber Technologies, Inc.
555 Market St., San Francisco, CA, 94103
Paul Horsfall HORSFALLP@GMAIL.COM
Noah D. Goodman NGOODMAN@STANFORD.EDU
Uber AI Labs
555 Market St., San Francisco, CA 94103
Stanford University, Stanford, CA, USA

Editor: Antti Honkela

Abstract

Pyro is a probabilistic programming language built on Python as a platform for developing advanced probabilistic models in AI research. To scale to large data sets and high-dimensional models, Pyro uses stochastic variational inference algorithms and probability distributions built on top of PyTorch, a modern GPU-accelerated deep learning framework. To accommodate complex or model-specific algorithmic behavior, Pyro leverages Poutine, a library of composable building blocks for modifying the behavior of probabilistic programs.

Keywords: Probabilistic programming, graphical models, approximate Bayesian inference, generative models, deep learning

```
def model():
  loc, scale = torch.zeros(20), torch.ones(20)
  z = pyro.sample("z", Normal(loc, scale))
  w, b = pyro.param("weight"), pyro.param("bias")
  ps = torch.sigmoid(torch.mm(z, w) + b)
  return pyro.sample("x", Bernoulli(ps))


def guide(x):
  pyro.module("encoder", nn_encoder)
  loc, scale = nn_encoder(x)
  return pyro.sample("z", Normal(loc, scale))
```

```
def conditioned_model(x):
  return pyro.condition(model, data={"x": x})()

optimizer = pyro.optim.Adam({"lr": 0.001})
loss = pyro.infer.Trace_ELBO()

svi = pyro.infer.SVI(model=conditioned_model,
                     guide=guide,
                     optim=optimizer,
                     loss=loss)

losses = []
for batch in batches:
    losses.append(svi.step(batch))
```

Figure 1: A complete Pyro example: the generative model (`model`), approximate posterior (`guide`), constraint specification (`conditioned_model`), and stochastic variational inference (`svi`, `loss`) in a variational autoencoder. `encoder` is a `torch.nn.Module` object. `pyro.module` calls `pyro.param` on every parameter of a `torch.nn.Module`.

1. Introduction

In recent years, richly structured probabilistic models have demonstrated promising results on a number of fundamental problems in AI (Ghahramani (2015)). However, most such models are still implemented from scratch as one-off systems, slowing their development and limiting their scope and extensibility. Probabilistic programming languages (PPLs) promise to reduce this burden, but in practice more advanced models often require high-performance inference engines tailored to a specific application. We identify design principles that enable a PPL to scale to advanced research applications while retaining flexibility, and we argue that these principles are fully realized together in the Pyro PPL.

2. Design Principles

First, a PPL suitable for developing state-of-the-art AI research models should be **expressive**: it should be able to concisely describe models with have data-dependent internal control flow or latent variables whose existence depends on the values of other latent variables, or models which only be defined in closed form as unnormalized joint distributions.

For a PPL to be practical, it must be **scalable**: its approximate inference algorithms must be able to seamlessly handle the large data sets and non-conjugate, high-dimensional models common in AI research, and should exploit compiler and hardware acceleration to approach the performance of optimized bespoke implementations of these models.

A PPL targeting research models should be **flexible**: in addition to scalability, many advanced models require inference algorithms with complex, model-specific behavior. A PPL should enable researchers to quickly and easily implement such behavior and should enforce a separation of concerns between between model, inference, and runtime implementations.

Finally, a PPL targeting researchers as users should strive to be **minimal**: in order to minimize cognitive overhead, it should share most of its syntax and semantics with existing languages and and systems and work well with other tools such as libraries for visualization.

As is clear from Table 2, these four principles are often in conflict, with one being achieved at the expense of others. For example, an overly flexible design may be very difficult to implement efficiently and scalably, especially while simultaneously integrating a new language with existing tools. Similarly, enabling development of custom inference algorithms may be difficult without limiting model expressivity. In this section, we describe the design choices we made in Pyro to balance between all four objectives.

Pyro is embedded in Python, and Pyro programs are written as Python functions, or callables, with just two extra language primitives (whose behavior is overridden by inference algorithms): `pyro.sample` for annotating calls to functions with internal randomness, and `pyro.param` for registering learnable parameters with inference algorithms that can change them. Pyro models may contain arbitrary Python code and interact with it in arbitrary ways, including expressing unnormalized models through the `obs` keyword argument to `pyro.sample`. Pyro's language primitives may be used with all of Python's control flow constructs, including recursion, loops, and conditionals. The existence of random variables in a particular execution may thus depend on any Python control flow construct.[1]

Pyro implements several generic probabilistic inference algorithms, including the No U-turn Sampler (Hoffman and Gelman (2014)), a variant of Hamiltonian Monte Carlo. However, the primary inference algorithm is gradient-based stochastic variational inference (SVI) (Kingma and Welling (2014)), which uses stochastic gradient descent to optimize Monte Carlo estimates of a divergence measure between approximate and true posterior distributions. Pyro scales to complex, high-dimensional models thanks to GPU-accelerated tensor math and reverse-mode automatic differentiation via PyTorch, and it scales to large data sets thanks to stochastic gradient estimates computed over mini-batches of data in SVI.

Some inference algorithms in Pyro, such as SVI and importance sampling, can use arbitrary Pyro programs (called *guides*, following webPPL) as approximate posteriors or proposal distributions. A guide for a given model must take the same input arguments as the model and contain a corresponding sample statement for every unconstrained sample statement in the model but is otherwise unrestricted. Users are then free to express complex hypotheses about the posterior distribution, e.g. its conditional independence structure. Unlike webPPL and Anglican, in Pyro guides may not depend on values inside the model.

Finally, to achieve flexibility and separation of concerns, Pyro is built on Poutine, a library of effect handlers (Kammar et al. (2013)) that implement individual control and book-keeping operations used for inspecting and modifying the behavior of Pyro programs, separating inference algorithm implementations from language details.

3. Project Openness and Development

Pyro's source code is freely available under an MIT license and developed by the authors and a community of open-source contributors at `https://github.com/uber/pyro` and documentation, examples, and a discussion forum are hosted online at `https://pyro.ai`. A comprehensive test suite is run automatically by a continuous integration service before code is merged into the main codebase to maintain a high level of project quality and usability.

We also found that while PyTorch was an invaluable substrate for tensor operations and automatic differentiation, it was lacking a high-performance library of probability

1. See the `geometric` example in `http://pyro.ai/examples/intro_part_i.html`

distributions. As a result, several of the authors made substantial open-source contributions upstream to PyTorch Distributions,[2] a new PyTorch core library inspired by TensorFlow Distributions (Dillon et al. (2017)).

4. Existing Systems

System	Expressivity: Dynamic control	Scalability: Subsampling, AD	Flexibility: Flexible inference	Minimality: Host language
Stan	Static control flow	Some, CPU	Automated	None
Church	Yes	No, None	Automated	Scheme
Venture	Yes	No, None	Yes	None
webPPL	Yes	No, CPU	Some	JavaScript
Edward	Static control flow	Yes, CPU/GPU	Yes	TensorFlow
Pyro	Yes	Yes, CPU/GPU	Yes	Python, PyTorch

Figure 2: Simplified summary of design principles of Pyro and some other PPLs.

Probabilistic programming and approximate inference are areas of active research, so there are many existing probabilistic programming languages and systems. We briefly mention several that were especially influential in Pyro's development, regretfully omitting (due to space limitations) many systems for which simple direct comparisons are more difficult. We also emphasize that, as in conventional programming language development, our design decisions are not universally applicable or desirable for probabilistic programming, and that other systems purposefully make different tradeoffs to achieve different goals.

Stan (Carpenter et al. (2017)) is a domain-specific language designed for describing a restricted class of probabilistic programs and performing high-quality automated inference in those models. Church (Goodman et al. (2008)), a probabilistic dialect of Scheme, was an early universal probabilistic programming language, capable of representing any computable probability distribution. Venture (Mansinghka et al. (2018)) is a universal language with a focus on expressiveness and flexibility and a custom syntax and virtual machine. Anglican (Tolpin et al. (2016)) and webPPL (Goodman and Stuhlmüller (2014)) are lightweight successors to Church, embedded as syntactic subsets (Wingate et al. (2011)) in the general-purpose programming languages Clojure and JavaScript. Edward (Tran et al. (2017)) is a PPL built on static TensorFlow graphs that features composable representations for models and inference. ProbTorch (Siddharth et al. (2017)) is a PPL built on PyTorch with a focus on deep generative models and developing new objectives for variational inference. Turing (Ge et al. (2018)) is a PPL embedded in Julia featuring composable MCMC algorithms.

5. Experiments

To demonstrate that Pyro meets our design goals, we implemented several state-of-the-art models.[3] Here we focus on the variational autoencoder (VAE; Kingma and Welling (2014))

2. `http://pytorch.org/docs/stable/distributions.html`
3. `http://pyro.ai/examples/`

# z	# h	PyTorch (ms)	Pyro (ms)
10	400	3.82 ± 0.02	6.79 ± 0.04
30	400	3.73 ± 0.07	6.67 ± 0.03
10	2000	7.65 ± 0.02	10.14 ± 0.06
30	2000	7.66 ± 0.02	10.19 ± 0.03

Figure 3: Times per update of VAE in Pyro versus PyTorch

# IAFs	Test ELBO
0 (theirs)	-6.93
0 (ours)	-6.87
1	-6.82
2	-6.80

Figure 4: Test ELBOs for DMM and extension with IAF guide

and the Deep Markov Model (DMM; Krishnan et al. (2017)), a non-linear state space model that has been used for several applications including audio generation and causal inference. The VAE is a standard example in deep probabilistic modeling, while the DMM has several characteristics that make it ideal as a point of comparison: it is a high-dimensional, non-conjugate model designed to be fit to large data sets; the number of latent variables in a sequence depends on the input data; and it uses a hand-designed approximate posterior.

The models and inference procedures derived by Pyro replicate the original papers almost exactly, except that we use Monte Carlo estimates rather than exact analytic expressions for KL divergence terms. We use the MNIST data set and 2-hidden-layer MLP encoder and decoder networks with varying hidden layer size $\#h$ and latent code size $\#z$ for the VAE and the same data set of digitized music[4] to train the DMM.

To demonstrate that Pyro's abstractions do not reduce its scalability by introducing too much overhead, we compared our VAE implementation with an idiomatic PyTorch implementation.[5] After verifying that they converge to the same test ELBO, we compared the wall-clock time taken to compute one gradient update, averaged over 10 epochs of GPU-accelerated mini-batch stochastic gradient variational inference (batch size 128) on a single NVIDIA GTX 1080Ti. Figure 3 shows that the relative performance gap between the Pyro and PyTorch versions is moderate, and, more importantly, that it shrinks as the total time spent performing tensor operations increases.

We used the DMM to evaluate Pyro's flexibility and expressiveness. As Figure 4 shows, we found that we were able to quickly and concisely replicate the exact DMM model and inference configuration and quantitative results reported in the paper after 5000 training epochs. Furthermore, thanks to Pyro's modular design, we were also able to build DMM variants with more expressive approximate posteriors via autoregressive flows (IAFs) (Kingma et al. (2016)), improving the results with a few lines of code at negligible computational cost.

Acknowledgments

We would like to acknowledge Du Phan, Adam Scibior, Dustin Tran, Adam Paszke, Soumith Chintala, Robert Hawkins, Andreas Stuhlmueller, our colleagues in Uber AI Labs, and the Pyro and PyTorch open-source communities for helpful contributions and feedback.

4. This is the JSB chorales data set from `http://www-etud.iro.umontreal.ca/~Eboulanni/icml2012`
5. We used the PyTorch example at `https://github.com/pytorch/examples/tree/master/vae`

References

Bob Carpenter, Andrew Gelman, Matthew D. Hoffman, Daniel Lee, Ben Goodrich, Michael Betancourt, Marcus Brubaker, Jiqiang Guo, Peter Li, and Allen Riddell. Stan: A probabilistic programming language. *Journal of Statistical Software*, 76(1), 2017.

Joshua V. Dillon, Ian Langmore, Dustin Tran, Eugene Brevdo, Srinivas Vasudevan, Dave Moore, Brian Patton, Alex Alemi, Matt Hoffman, and Rif A. Saurous. TensorFlow Distributions. *arXiv:1711.10604*, November 2017.

Hong Ge, Kai Xu, and Zoubin Ghahramani. Turing: a language for flexible probabilistic inference. In *AISTATS*, 2018.

Zoubin Ghahramani. Probabilistic machine learning and artificial intelligence. *Nature*, 521: 452–459, May 2015.

Noah D Goodman and Andreas Stuhlmüller. The Design and Implementation of Probabilistic Programming Languages. `http://dippl.org`, 2014.

Noah D. Goodman, Vikash K. Mansinghka, Daniel Roy, Keith Bonawitz, and Joshua B. Tenenbaum. Church: A language for generative models. In *UAI*, 2008.

Matthew D. Hoffman and Andrew Gelman. The no-U-turn sampler: Adaptively setting path lengths in Hamiltonian Monte Carlo. *J. Mach. Learn. Res.*, 15(1), January 2014.

Ohad Kammar, Sam Lindley, and Nicolas Oury. Handlers in action. In *ICFP*, 2013.

Diederik P Kingma and Max Welling. Auto-encoding variational Bayes. In *ICLR*, 2014.

Diederik P Kingma, Tim Salimans, Rafal Jozefowicz, Xi Chen, Ilya Sutskever, and Max Welling. Improved variational inference with inverse autoregressive flow. In *NIPS*. 2016.

Rahul G Krishnan, Uri Shalit, and David Sontag. Structured inference networks for nonlinear state space models. In *AAAI*, 2017.

Vikash K. Mansinghka, Ulrich Schaechtle, Shivam Handa, Alexey Radul, Yutian Chen, and Martin Rinard. Probabilistic programming with programmable inference. In *PLDI*, 2018.

N. Siddharth, Brooks Paige, Jan-Willem van de Meent, Alban Desmaison, Noah D. Goodman, Pushmeet Kohli, Frank Wood, and Philip Torr. Learning disentangled representations with semi-supervised deep generative models. In *NIPS*, 2017.

David Tolpin, Jan-Willem van de Meent, Hongseok Yang, and Frank Wood. Design and implementation of probabilistic programming language anglican. In *IFL*, 2016.

Dustin Tran, Matthew D. Hoffman, Rif A. Saurous, Eugene Brevdo, Kevin Murphy, and David M. Blei. Deep probabilistic programming. In *ICLR*, 2017.

David Wingate, Andreas Stuhlmller, and Noah Goodman. Lightweight implementations of probabilistic programming languages via transformational compilation. In *AISTATS*, 2011.