Pyro Breakdown

Zach Wolpe

March 2021

Pyro is a universal probabilistic programming language (PPL) written in Python and supported by PyTorch on the backend [1]. Pyro enables flexible and expressive deep probabilistic modeling, unifying the best of modern deep learning and Bayesian modeling [1]. It was designed with these key principles [1]:

- **Universal**: Pyro can represent any computable probability distribution.
- **Scalable**: Pyro scales to large data sets with little overhead.
- **Minimal**: Pyro is implemented with a small core of powerful, composable abstractions.
- **Flexible**: Pyro aims for automation when you want it, control when you need it.

Keywords: probabilistic programming, deep learning, Bayesian inference

Pyro vs STAN

The key difference between Pyro & other probabilistic programming languages is it's implemented from an engineer perspective as apposed to a statistical perspective. In Pyro everything is implemented as modular Python classes objects, whereas in STAN one specifies the full statistical model. Pyro is also built for deep learning - built off PyTorch - whereas other probabilistic frameworks are inherently centred around statistical models. In this regard, Pyro is most similar to Tensorflow Probability and PyMC3, and significantly different from STAN/Edward.

Models in Pyro: Stochastic Functions

Basic unit in probabilistic programming is the *stochastic function.* This is a Python callable that combines two components [1]:

- deterministic Python code; and
- primitive stochastic functions that call a random number generator

Stochastic functions are simple functions that rely on some stochastic (random) process, which is represented by sampling a probability distribution.

In Pyro, stochastic functions are referred to as models, since stochastic functions can be used to represent simplified or abstract descriptions of a process by which data are generated.

Distributions: Primitive Stochastic Functions

Primitive stochastic functions, or distributions, are an important class of stochastic functions for which we can explicitly compute the probability of the outputs given the inputs.

The *pyro.sample* Primitive

The *pyro.sample* function is a key feature when using Pyro. *pyro.sample* calls a primitive stochastic function (distribution) - returning a sample from the distribution. Each sample must have a unique name - a key feature utilized by the backend.

Universality: Stochastic Recursion, Higher-order Stochastic Functions, and Random Control Flow

Since Pyro is embedded in Python, stochastic functions can contain arbitrarily complex deterministic Python and randomness can freely affect control flow. For example, we can construct recursive functions that terminate their recursion nondeterministically, provided we take care to pass *pyro.sample* unique sample names whenever it's called. We are also free to define stochastic functions that accept as input or produce as output other stochastic functions.

The fact that Pyro supports arbitrary Python code like this - iteration, recursion, higher-order functions, etc. — in conjunction with random control flow means that Pyro stochastic functions are universal, i.e. they can be used to represent any computable probability distribution. This is incredibly powerful.

By this nature, we are able to specify any stochastic process or graphical model in Pyro.

It is worth emphasizing that this is one reason why Pyro is built on top of PyTorch: dynamic computational graphs are an important ingredient in allowing for universal models that can benefit from GPU-accelerated tensor math.

Inference in Pyro

Much of machine learning can be cast as conducting inference.

Conditional Dependence

A primitive stochastic function (distribution) represents some randomness about a variable, for example:

$$\begin{gathered} X|\tau \sim \mathcal{N}(\tau, 1) \\ Y|\tau, X \sim \mathcal{N}(X, 0.75) \end{gathered} \tag{1}$$

The real utility of probabilistic programming is in the ability to condition generative models on observed data and infer the latent factors that might have produced that data.

In Pyro, we separate the expression of conditioning from its evaluation via inference, making it possible to write a model once and condition it on many different observations.

A posterior is simple a the normalized joint distribution of the prior and likelihood conditioned on the data

Pyro supports constraining a model's internal sample statements to be equal to a given set of observations.

Suppose we want to sample from the distribution of X given input $\tau = 8.5$, but now we have observed that $Y == 9.5$. That is, we wish to infer the distribution:

$$(X|\tau, Y == 9.5) \sim ?$$

The *pyro.condition* Primitive

pyro.condition allows one to pass data to the stochastic function, conditioning the distribution on the sample. Taking the form:

$$pyro.condition(stochastic\_function,\ data = \{'X' : x\})$$

This allows one to pass an entire dataset. One can also condition a single sample on some value - which may be useful when writing complex stochastic functions. To do this, one uses the *obs* argument in the sample function:

$$pyro.sample('name',\ dist.Normal(X))$$

$$\begin{aligned} &\tau = 10.0 \\ &X = pyro.sample('X',\ dist.Normal(\tau, 1.0)) \\ &\#\ condition\ on\ Y == 9.5 \\ &Y = pyro.sample('Y',\ dist.Normal(X, 0.75),\ obs = 9.5) \end{aligned} \tag{2}$$

Flexible Approximate Inference with Guide Functions

Inference algorithms in Pyro, such as $pyro.infer.SVI$, allow us to use arbitrary stochastic functions, which we will call *guide functions* or *guides*, as approximate posterior distributions. Guide functions must satisfy these two criteria to be valid approximations for a particular model:

1. All unobserved (i.e., not conditioned) sample statements that appear in the model appear in the guide.
2. The guide has the same input signature as the model (i.e., takes the same arguments)

Guide functions can serve as programmable, data-dependent proposal distributions for importance sampling, rejection sampling, sequential Monte Carlo, MCMC, and independent Metropolis-Hastings, and as variational distributions or inference networks for stochastic variational inference. Currently, importance sampling, MCMC, and stochastic variational inference are implemented in Pyro.

Although the precise meaning of the guide is different across different inference algorithms, the guide function should generally be chosen so that, in principle, it is flexible enough to closely approximate the distribution over all unobserved sample statements in the model.

Parameterised Stochastic Functions and Variational Inference

In general it is intractable to analytically solve for a guide that is a good approximation to the posterior distribution of an arbitrary conditioned stochastic function.

What we can do instead is use the top-level function *pyro.param* to specify a family of guides indexed by named parameters, and search for the member of that family that is the best approximation according to some loss function. This approach to approximate posterior inference is called **variational inference**.

$$pyro.param()$$

For example, we can parameterise α and β in some prior distribution by:

$$\begin{aligned} &def\ some\_prior(\tau):\\ &\quad a = pyro.param('a',\ torch.tensor(\tau))\\ &\quad b = pyro.param('b',\ torch.tensor(\tau),\ constraint = constraints.positive)\\ &\quad return\ pyro.sample('prior',\ dist.Normal(a, b)) \end{aligned} \tag{3}$$

Stochastic Variational Inference

Pyro enables stochastic variational inference, a class of variational inference algorithms with three key characteristics:

1. Parameters are always real-valued tensors.
2. We compute Monte Carlo estimates of a loss function from samples of execution histories of the model and guide.
3. We use stochastic gradient descent to search for the optimal parameters.

Scalability: Efficient Computation

Combining stochastic gradient descent with PyTorch's GPU-accelerated tensor math and automatic differentiation allows us to scale variational inference to very high-dimensional parameter spaces and massive datasets.

Just as with *pytorch*, one:

1. Specifies the model: Loss, guide, trainable parameters, constraints.
2. Specifies the search algorithm (optimization technique).
3. defines the training loop.
4. execute.

Note that optimization will update the values of the guide parameters in the parameter store, so that once we find good parameter values, we can use samples from the guide as posterior samples for downstream tasks.

References

[1] Eli Bingham, Jonathan P. Chen, Martin Jankowiak, Fritz Obermeyer, Neeraj Pradhan, Theofanis Karaletsos, Rohit Singh, Paul Szerlip, Paul Horsfall, and Noah D. Goodman. Pyro: Deep Universal Probabilistic Programming. *Journal of Machine Learning Research*, 2018.