The Journal of Open Source Software

torch_blue: A Flexible Python Package for Bayesian Neural Networks in PyTorch

Arvid Weyrauch [1], **Lars H. Heyen** [1], **Juan Pedro Gutiérrez Hermosillo Muriedas** [1], **Pei-Hsuan Hsia** [1], **Asena Karolin Özdemir** [1], **Achim Streit** [1], **Markus Götz** [1,2], **and Charlotte Debus** [1]

1 Karlsruhe Institute of Technology, Germany **2** Helmholtz AI

DOI: 10.21105/joss.09415

Software

- Review
- Repository
- Archive

Editor: Tristan Miller

Reviewers:

- @Bobby-Huggins
- @arnauqb
- @JSchmie

Submitted: 08 September 2025
Published: 23 January 2026

Summary

Bayesian Neural Networks (BNN) integrate uncertainty quantification in all steps of the training and prediction process, thereby enabling better-informed decisions (Arbel et al., 2023). Among the different approaches to implementing BNNs, Variational Inference (VI) (Hoffman et al., 2013) specifically strikes a balance between the ability to consider a large variety of distributions while maintaining low enough compute requirements to allow scaling to larger models.

However, setting up and training BNNs is quite complicated, and existing libraries all either lack flexibility, lack scalability, or tackle Bayesian computation in general, adding even more complexity and therefore a huge barrier to entry. Moreover, no existing framework directly supports straightforward BNN model prototyping by offering pre-programmed Bayesian network layer types, similar to PyTorch's `nn` module. This forces any BNNs to be implemented from scratch, which can be challenging even for non-Bayesian networks.

`torch_blue` addresses this by providing an interface that is almost identical to the widely used PyTorch (Ansel et al., 2024) for basic use, providing a low barrier to entry, as well as an advanced interface designed for exploration and research. Overall, this allows users to set up models and even custom layers without worrying about the Bayesian intricacies under the hood.

Statement of need

To represent uncertainty, BNNs do not consider their weights as point values, but random variables, i.e., distributions. The optimization goal becomes adapting the weight distributions to minimize their distance to the true distribution. This requires two assumptions. For one, the distance between distributions needs to be defined, for which the Kullback-Leibler divergence (Kullback & Leibler, 1951) is typically used. Secondly, optimizing an object as complex as a distribution is a non-trivial task. To overcome this, VI specifies a parametrized distribution and optimizes its parameters. Thus, the Kullback-Leibler criterion can be simplified to the ELBO (**E**vidence **L**ower **BO**und) loss (Jordan et al., 1999):

$$\mathrm{ELBO} = \mathbb{E}_{W\sim q}[\underbrace{\log p(Y|X,W)}_{\text{Data fitting}} - \underbrace{(\log q(W|\lambda) - \log p(W))}_{\text{Prior matching}}] \quad ,$$

where (X, Y) are the training inputs and labels, W the network weights, q the variational distribution and λ its current best fit parameters.

While interest in uncertainty quantification and BNNs has been growing, support for users with little to no experience in Bayesian statistics is still limited. Probabilistic programming languages, such as Pyro (Bingham et al., 2019) and Stan (Stan Development Team, 2025),

are very powerful and versatile, allowing the implementation of many approaches beyond VI. However, their interfaces are structured around Bayesian concepts – like plate notation – which will be unfamiliar to many primary machine learning users.

```python
from torch import nn
from torch_blue import vi
from torch_blue.vi.distributions import Categorical


# Set up three layer Bayesian MLP
class NeuralNetwork(vi.VIModule):
    def __init__(self) -> None:
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = vi.VISequential(
            vi.VILinear(28 * 28, 512),
            nn.ReLU(),
            vi.VILinear(512, 512),
            nn.ReLU(),
            vi.VILinear(512, 10),
        )

    def forward(self, x_: Tensor) -> Tensor:
        x_ = self.flatten(x_)
        logits = self.linear_relu_stack(x_)
        return logits

model = NeuralNetwork().to(device)
model.return_log_probs = True

# Set up classification task
predictive_distribution = Categorical()
loss_fn = vi.KullbackLeiblerLoss(
    predictive_distribution, dataset_size=len(training_data)
)
```

Figure 1: Code example of a three-layer Bayesian MLP with cross-entropy loss in `torch_blue`. The highlight colors relate user-facing components to their position in Figure 2.

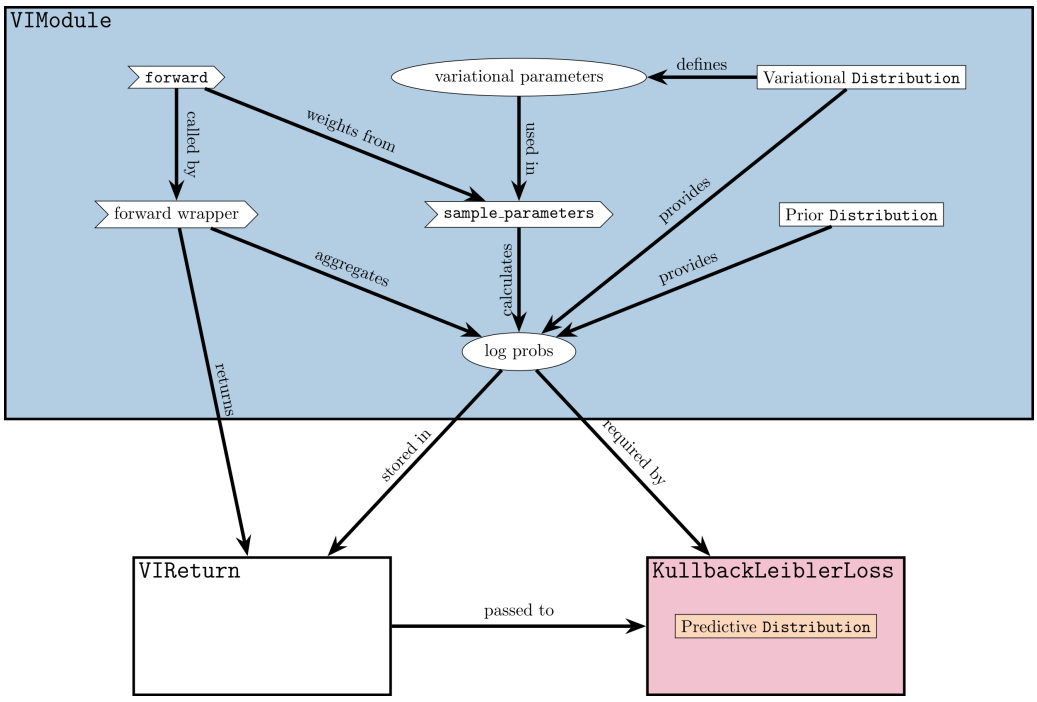

Figure 2: Design graph of `torch_blue`. Colored highlights correspond to their practical applications in the code example (Figure 1).

`torch_blue` sacrifices this extreme flexibility to allow nearly fully automatic VI with reparametrization (Bayes by Backprop) (Blundell et al., 2015). The ability to use multiple independent sampling dimensions is removed, which allows for fully automating a single sampling dimension in the outermost instance of the new base class `VIModule`. To control

the number of samples this module also captures the optional keyword argument `samples`. The log likelihoods typically needed for loss calculation are automatically calculated whenever weights are sampled, aggregated, and returned once again by the outermost `VIModule`.

Core design and features

`torch_blue` is designed around two core aims:

1. Ease of use, even for users with little to no experience with Bayesian statistics
2. Flexibility and extensibility as required for research and exploration

While ease of use influences all design decisions, it features most prominently in the PyTorch-like interface. While currently only the most common layer types provided by PyTorch are supported, corresponding Bayesian layers follow an analogous naming pattern and accept the same arguments as their PyTorch version. Additionally, while there are minor differences, the process of implementing custom layers is also very similar to PyTorch. To illustrate this Figures 1 and 2 show an application example and internal interactions of `torch_blue` with the colors connecting the abstract and applied components.

The additional arguments required to modify the Bayesian aspects of the layers are collected on a common group of keyword arguments called `VIkwargs`. The default settings use mean field Gaussian variational inference with a Gaussian prior, allowing beginner users to implement simple, unoptimized models without worrying about Bayesian settings.

An overview of the currently supported user-facing components is given in Figure 3. While modular priors and predictive distributions are quite common even for packages with simpler interfaces, flexible variational distributions are much more challenging and are often restricted to mean-field Gaussian. This is likely due to the fact that a generic variational distribution might require any number of different parameters, and the number and shape of weight matrices can only be determined with knowledge of the specific combination of layer and variational distribution. This is overcome in `torch_blue` by having the layer provide the names and shapes of the required random variables (e.g., mean and bias) and dynamically creating the associated class attributes during initialization, when the variational distribution is known. The modules also provide methods to sample from the variational distribution and access its parameters.

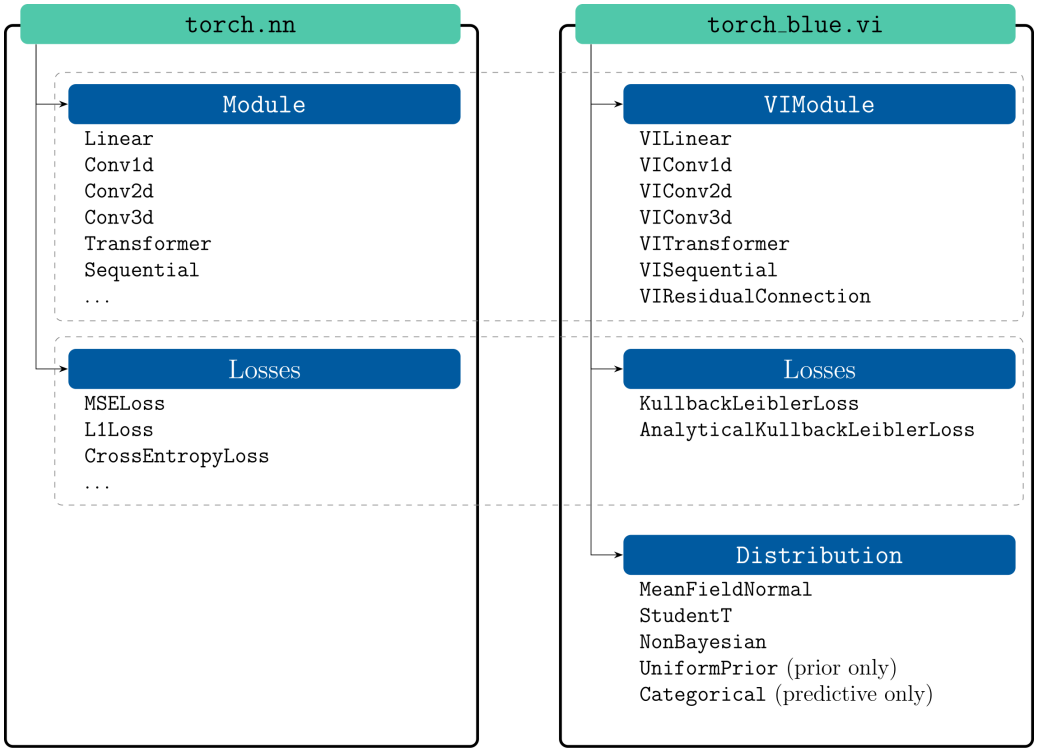

Figure 3: Overview of the major components of `torch_blue` and corresponding non-Bayesian components of PyTorch.

Another challenge is introduced by the prior term of the ELBO loss. It can only be calculated analytically for a very limited set of priors and variational distributions. However, like the rest of the ELBO it can be estimated from the log probability of the sampled weights under these two distributions. Therefore, `torch_blue` provides the option to return these as part of the forward pass in the form of a `Tensor` containing an additional `log_probs` attribute similar to gradient tracking. As a result, the only requirement on custom distributions is that there needs to be a method to differentiably sample from a variational distribution and, for both priors and variational distributions, a method to compute the log probability of a given sample.

Finally, in the age of large neural networks, scalability and efficiency are always a concern. While BNNs are not currently scaled to very large models and this is not a primary target of `torch_blue`, it is kept in mind wherever possible. A core feature for this purpose is GPU compatibility, which comes with the challenge of various backends and device types. We address this by performing all core operations, in particular the layer forward passes, with the methods from `torch.nn.functional`. This outsources backend maintenance to a large, community-supported library.

Another efficiency optimization is the automatic vectorization of the sampling process. `torch_blue` adds an additional wrapper around the forward pass, which catches the optional `samples` argument, creates the specified number of samples (default: 10), and vectorizes the forward pass via `PyTorchs vmap` method.

Acknowledgements

This work is supported by the German Federal Ministry of Research, Technology and Space under the 01IS22068 - EQUIPE grant. The authors gratefully acknowledge the computing time made available to them through the HAICORE@KIT partition and on the high-performance computer HoreKa at the NHR Center KIT. This center is jointly supported by the Federal

Ministry of Education and Research and the state governments participating in the NHR (www.nhr-verein.de/en/our-partners).

References

Ansel, J., Yang, E., He, H., Gimelshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P., Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A., DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., … Chintala, S. (2024). PyTorch 2: Faster machine learning through dynamic Python bytecode transformation and graph compilation. *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, 929–947. https://doi.org/10.1145/3620665.3640366

Arbel, J., Pitas, K., Vladimirova, M., & Fortuin, V. (2023). A primer on Bayesian neural networks: Review and debates. *arXiv Preprint arXiv:2309.16314*. https://doi.org/10.48550/arXiv.2309.16314

Bingham, E., Chen, J. P., Jankowiak, M., Obermeyer, F., Pradhan, N., Karaletsos, T., Singh, R., Szerlip, P. A., Horsfall, P., & Goodman, N. D. (2019). Pyro: Deep universal probabilistic programming. *Journal of Machine Learning Research*, *20*(28), 28:1–28:6. http://jmlr.org/papers/v20/18-403.html

Blundell, C., Cornebise, J., Kavukcuoglu, K., & Wierstra, D. (2015). Weight uncertainty in neural network. In F. Bach & D. Blei (Eds.), *Proceedings of the 32nd international conference on machine learning* (Vol. 37, pp. 1613–1622). PMLR. https://proceedings.mlr.press/v37/blundell15.html

Hoffman, M. D., Blei, D. M., Wang, C., & Paisley, J. (2013). Stochastic variational inference. *Journal of Machine Learning Research*, *14*(4), 1303–1347. http://jmlr.org/papers/v14/hoffman13a.html

Jordan, M. I., Ghahramani, Z., Jaakkola, T. S., & Saul, L. K. (1999). An introduction to variational methods for graphical models. *Machine Learning*, *37*(2), 183–233. https://doi.org/10.1023/A:1007665907178

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, *22*(1), 79–86. https://doi.org/10.1214/aoms/1177729694

Stan Development Team. (2025). *Stan reference manual, 2.37*. https://mc-stan.org