




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

Arvid Weyrauch¹, Lars H. Heyen¹, Juan Pedro Gutiérrez Hermosillo Muriedas¹, Peihuan Hsia¹, Asena K. Özdemir¹, Achim Streit¹, Markus Götz^{1,2}, and Charlotte Debus¹

¹ Karlsruhe Institute of Technology, Germany  ² Helmholtz AI

DOI: [10.xxxxxx/draft](https://doi.org/10.xxxxxx/draft)

Software

- [Review](#) 
- [Repository](#) 
- [Archive](#) 

Editor: 

Submitted: 08 September 2025

Published: unpublished

License

Authors of papers retain copyright and release the work under a Creative Commons Attribution 4.0 International License ([CC BY 4.0](https://creativecommons.org/licenses/by/4.0/)).

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 variants to implement 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 potentially allow scaling to larger models.

However, setting up and training BNNs is quite complicated, and existing libraries all either lack flexibility, scalability, or tackle Bayesian computation in general, adding even more complexity and therefore a huge entry barrier. Moreover, no existing framework directly supports direct 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_bayesian 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 entry barrier, 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 perfect 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 utilizes a parametrized distribution and optimizing its parameters. Thus, the Kullback-Leibler criterion can be simplified to the ELBO (Evidence Lower BOund) loss ([Jordan et al., 1999](#)):

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

where (X, Y) are the training inputs and labels, W are 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.

```
from torch import nn
from torch_bayesian import vi
from torch_bayesian.vi.predictive_distributions import (
    CategoricalPredictiveDistribution
)

# Set up three layer Bayesian MLP
class NeuralNetwork(vi.VModule):
    def __init__(self) -> None:
        super().__init__()
        self.flatten = nn.Flatten()
        self.linear_relu_stack = vi.VSequential(
            vi.VLinear(28 * 28, 512),
            nn.ReLU(),
            vi.VLinear(512, 512),
            nn.ReLU(),
            vi.VLinear(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 = CategoricalPredictiveDistribution()

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_bayesian. The highlight colors relate user-facing components to their position in Figure 2.

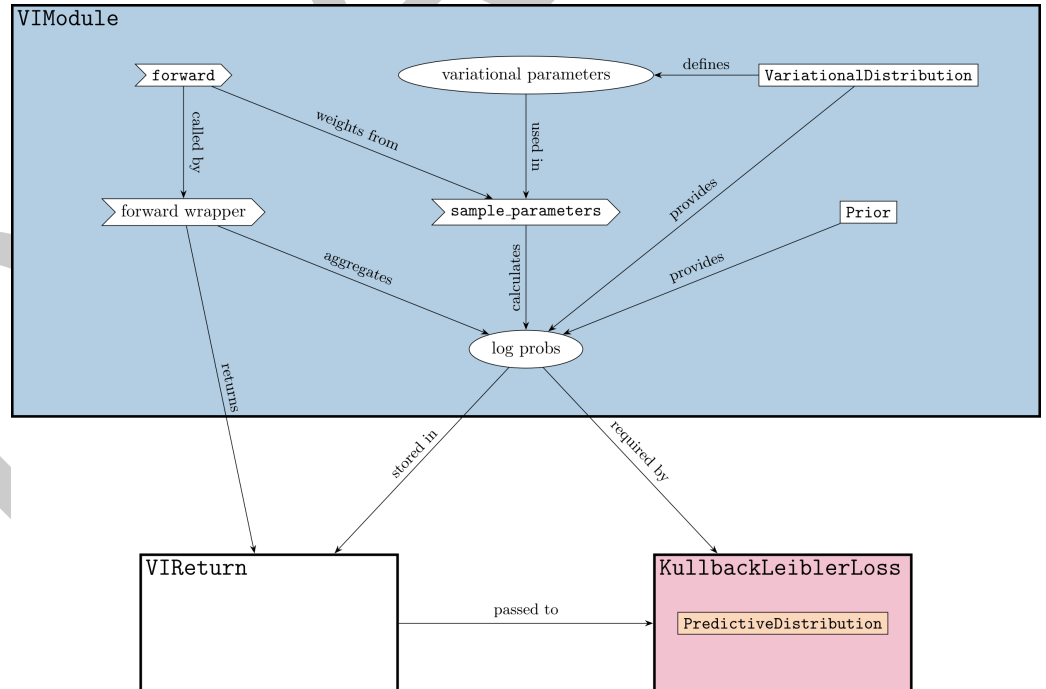


Figure 2: Design graph of torch_bayesian colored highlights correspond to their practical applications in the code example (Figure 1).

torch_bayesian sacrifices this extreme flexibility to allow nearly fully automating VI with reparametrization (Bayes by Backprop) (Blundell et al., 2015). The ability to use multiple

independent sampling dimensions is removed, which allows to fully automate a single sampling dimensions in the outermost instance of the new base class `VIModule`, which captures the optional keyword argument `samples` specifying the number of 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_bayesian` 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 colors 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 [Figure 1](#) and [2](#) show an application example and internal interactions of `torch_bayesian` 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`. These all use settings for mean field Gaussian variational inference with Gaussian prior as defaults 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 a simpler interface, 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_bayesian` 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 either return samples of all random variables or the name of each attribute for direct access.

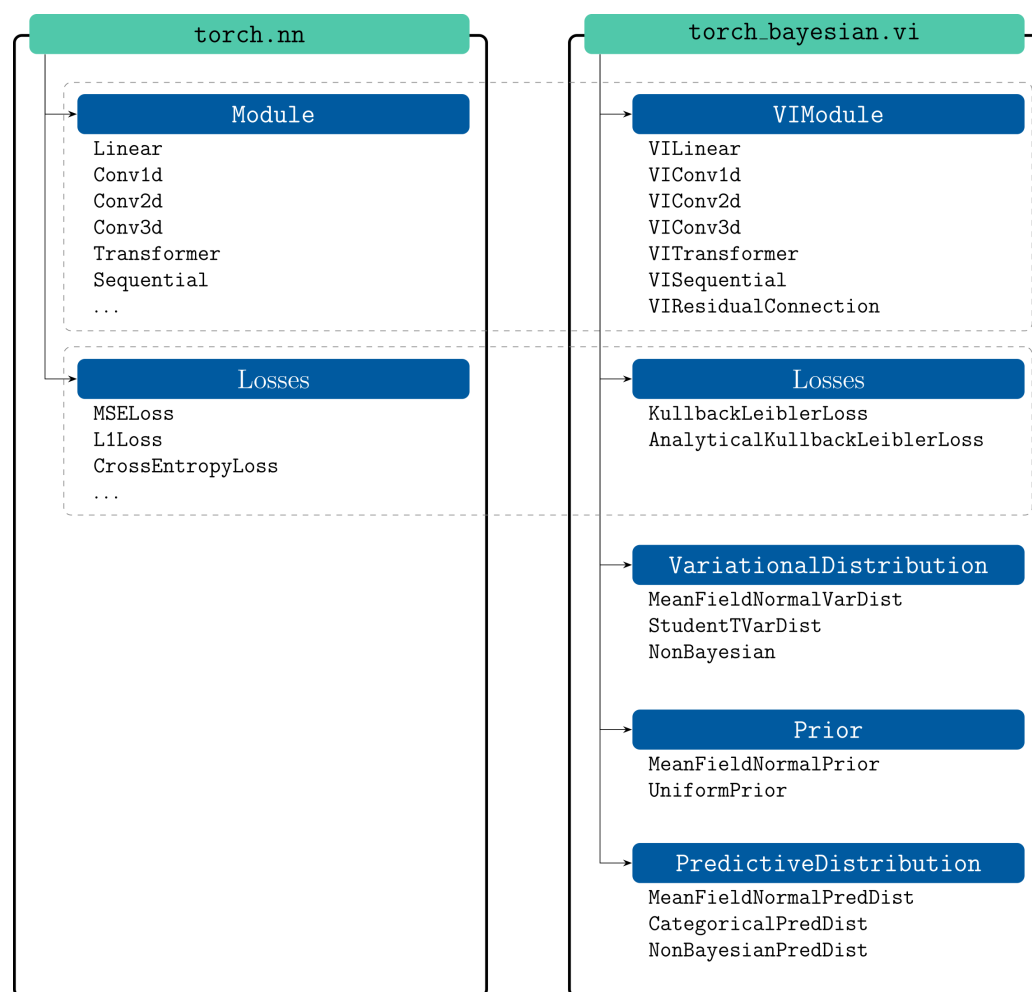


Figure 3: Overview of the major components of torch_bayesian and corresponding non-Bayesian components of PyTorch.

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_bayesian 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 differentially sample from a variational distribution and that for both priors and variational distributions, the log probability of a given sample can be computed.

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_bayesian, 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_bayesian 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 PyTorch's vmap method.

93 Acknowledgements

94 This work is supported by the German Federal Ministry of Research, Technology and Space
95 under the 01IS22068 - EQUIPE grant. The authors gratefully acknowledge the computing time
96 made available to them through the HAICORE@KIT partition and on the high-performance
97 computer HoreKa at the NHR Center KIT. This center is jointly supported by the Federal
98 Ministry of Education and Research and the state governments participating in the NHR
99 (www.nhr-verein.de/en/our-partners).

100 Ansel, J., Yang, E., He, H., Gimselshein, N., Jain, A., Voznesensky, M., Bao, B., Bell, P.,
101 Berard, D., Burovski, E., Chauhan, G., Chourdia, A., Constable, W., Desmaison, A.,
102 DeVito, Z., Ellison, E., Feng, W., Gong, J., Gschwind, M., ... Chintala, S. (2024). PyTorch
103 2: Faster machine learning through dynamic python bytecode transformation and graph
104 compilation. *Proceedings of the 29th ACM International Conference on Architectural
105 Support for Programming Languages and Operating Systems, Volume 2*, 929–947. <https://doi.org/10.1145/3620665.3640366>
106

Arbel, J., Pitas, K., Vladimirova, M., & Fortuin, V. (2023). A primer on bayesian neural
networks: Review and debates. *arXiv Preprint 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. *J. Mach. Learn. Res.*, 20, 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](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.

Kullback, S., & Leibler, R. A. (1951). On information and sufficiency. *The Annals of Mathematical Statistics*, 22(1), 79–86.

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