Variational Neural Networks implementation in Pytorch and JAX

Illia Oleksiienko*, Dat Thanh Tran† and Alexandros Iosifidis*
**Department of Electrical and Computer Engineering, Aarhus University, Denmark*
†Department of Computing Sciences, Tampere University, Finland
{io,ai}@ece.au.dk thanh.tran@tuni.fi

Abstract
Bayesian Neural Networks consider a distribution over the network's weights, which provides a tool to estimate the uncertainty of a neural network by sampling different models for each input. Variational Neural Networks (VNNs) consider a probability distribution over each layer's outputs and generate parameters for it with the corresponding sub-layers. We provide two Python implementations of VNNs with PyTorch and JAX machine learning libraries that ensure reproducibility of the experimental results and allow implementing uncertainty estimation methods easily in other projects.

Keywords
Bayesian Neural Networks, Bayesian Deep Learning, Uncertainty Estimation, PyTorch, JAX

Code metadata

Nr.	Code metadata description	Please fill in this column
C1	Current code version	v1.0.0
C2	Permanent link to code/repository used for this code version	https://github.com/iliiliiliili/vnn-pytorch-jax
C3	Permanent link to Reproducible Capsule	https://codeocean.com/capsule/2963476/tree
C4	Legal Code License	Apache-2.0 license
C5	Code versioning system used	Git
C6	Software code languages, tools, and services used	Python, R, LaTeX
C7	Compilation requirements, operating environments & dependencies	Pytorch/JAX (Tensorflow), plotnine, pandas, numpy, neural-tangents, scipy, scikit-learn, dm-haiku, dm-acme, chex, optax, fire, tqdm, tensorboard, tensorboardX, torchvision, absl-py, tensorflow
C8	If available Link to developer documentation/manual	
C9	Support email for questions	io@ece.au.dk

1. Introduction

Uncertainty estimation in neural networks provides an ability to detect possible errors in predictions that can be caused by the lack of training data or out-of-distribution samples. These failures may occur silently in regular neural networks and cause danger in critical tasks, such as medical image analysis or autonomous driving. To estimate the network's uncertainty in its predictions, several approaches have been proposed which are mostly following the Bayesian Neural Network (BNN) framework [1, 2, 3]. BNNs introduce a probability distribution over the neural network's weights. For each input, they sample a set of different models from the weights distribution which describe it from different points of view. The deviation in the prediction of these different models is an estimate of the uncertainty of the network for the specific input.

BNNs usually come with increased computational cost and memory requirements compared to neural networks providing point estimations for their inputs. The choice of the weights probability distribution results in

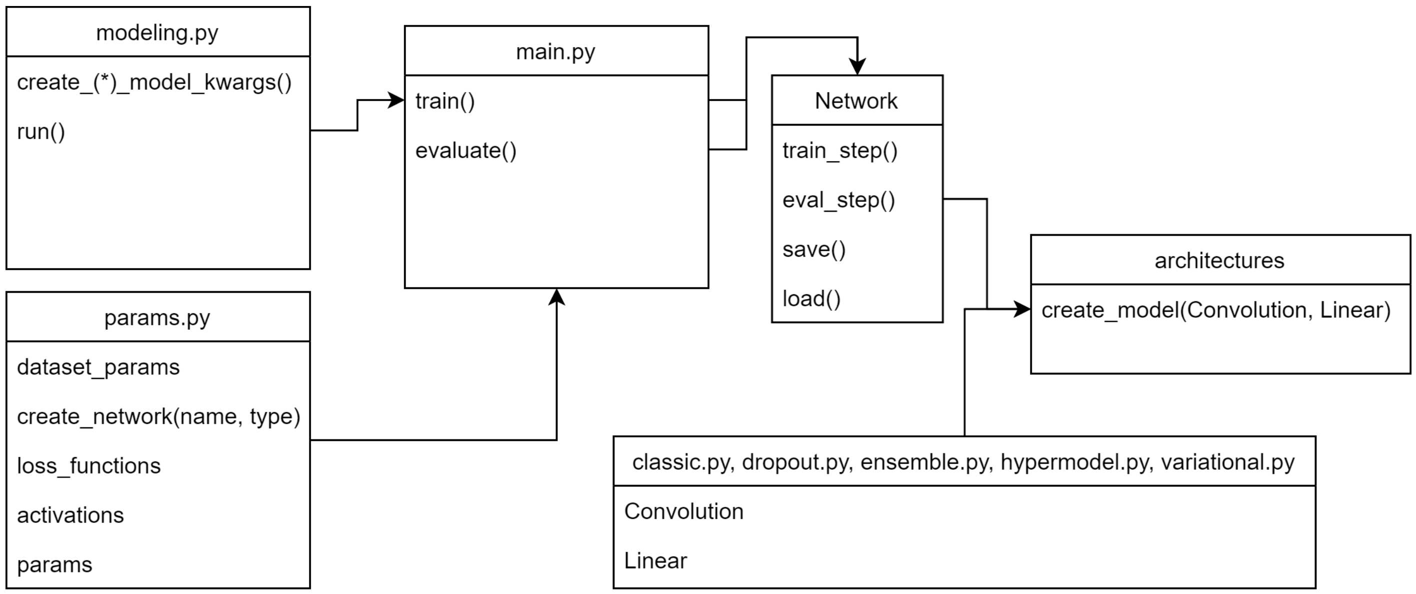

Figure 1: Structure of the PyTorch implementation of VNNs (R1).

different computational costs, but it also affects the statistical quality of the network. This is due to that more complex and parameterized distributions allow for greater freedom, but require more computations to train and execute the network, while simplified distributions may not represent the problem at hand. Existing methods explore a variet of distributions, including the Gaussian [4, 5], Bernoulli [6], and Categorical [7] distributions.

Variational Neural Networks (VNNs) [8] introduce a new type of uncertainty estimation for neural networks by considering a distribution over each layer's outputs and generate the distribution's parameters by processing inputs with corresponding sub-layers. To keep a low computational cost and memory requirements of VNNs, we consider the Gaussian distribution with its parameters, i.e., mean and variance, generated by the corresponding sub-layers. As shown in [8], VNNs achieve better uncertainty quality compared to Monte Carlo Dropout (MCD) [6] and Bayes By Backprop (BBB) [4], while having similar properties to those methods from the Bayesian Model Averaging perspective.

2. Impact Overview

The codebase of VNNs is split into two implementations with different purposes. The PyTorch implementation (R1) provides a general-purpose implementation of VNNs using Python 3 [9] and a machine learning framework PyTorch [10] together with code for image classification experiments on MNIST [11] and CIFAR-10 [12] datasets. The JAX implementation (R2) is built upon JAX [13] implementation [14] of Epistemic Neural Networks [15] and provides a JAX version of VNNs together with the uncertainty quality estimation experiments.

The software provides full reproducibility of the research results in [8] and includes code for generating plots and tables therein, which means that validating results and creating new experiments based on VNNs should be an easy task. The structure of the software is modular and allows for fast implementation of additional methods or usage of already existing methods in an existing project by replacing layers of a regular neural network with those provided by the software.

3. Functionalities and key features

The PyTorch implementation (R1) of VNNs is programmed in a modular manner, where each of the tested uncertainty estimation methods, including VNNs, Bayes By Backpropagation (BBB) [4], Monte Carlo Dropout (MCD) [6], Deep Ensembles [7] and Hypermodels [5], is implemented using a common training interface and isolated from the actual network architecture. The file structure of R1 is represented in Fig. 1 and consists of

- *main.py* is an entry point that implements *train* and *evaluate* functions to train and test models of interest.
- *network.py* implements a *Network* class that encapsulates all the functions need to train, run, load and save the model.
- *architectures* folder contains a set of scripts that implement model architecture classes, such as ResNet, DenseNet, and simpler architectures. These classes derive from the *Network* class and have convolutional and linear layers as generic parameters allowing the creation of models with different uncertainty methods for the same network architecture.
- *classic.py, dropout.py, ensemble.py, hypermodel.py, variational.py* implement the corresponding uncertainty methods by creating convolutional and linear layers of the method of interest or a full network generator. By providing these implementations into an architecture creator, a network for the selected uncertainty method and architecture will be generated.
- *params.py* contains data generators for MNIST and CIFAR-10 datasets, string-indexed dictionaries of all network creators, loss functions, activation functions and optimizers.
- *modeling.py* is an alternative entry point and is used to generate sets of experiments for each of the uncertainty methods and run them in parallel.
- *tools/create-grouped-tex-report.py* collects results from the trained models and generates a LaTeXtable with two best entries for each network type.

The presented repository structure allows for an easy expansion of the project:

- Adding a new network type requires implementing either convolutional and linear layers (as shown in *classic.py, dropout.py, variational.py*), or a network creator that takes an architecture as an input (as shown in *ensemble.py, hypermodel.py*).
- Adding a new architecture requires implementing it with convolutional and linear layer classes given as parameters.
- Adding a new layer type requires implementing versions of it for each of the network types, which is usually straightforward, following the already implemented ones.
- Using these methods for other projects can be achieved by copying layer implementations to the existing project with classical neural networks and replacing regular layers in the network with the desired ones.

The JAX implementation (R2) is based on the Epistemic Neural Networks repository [14] and provides a code to reproduce uncertainty quality experiments. All main files of R2 are located at *enn/experiments/neurips_2021* with the following file structure:

- *run_testbed.py* is an entry point that generates a problem for the selected experiment parameters and trains a set of models for the selected method.
- *agent_factories.py* contains model creators for each of the uncertainty methods and functions to create networks with different parameters to be trained.
- *enn/networks* folder contains implementations of the different uncertainty methods.
- *tools/create_enn_plots.py* is used to generate plots for each method, groups of experiments and a summary.

Acknowledgments

This work has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 871449 (OpenDR). This publication reflects the authors' views only. The European Commission is not responsible for any use that may be made of the information it contains.

References

[1] D. J. C. Mackay, “Probable networks and plausible predictions — a review of practical bayesian methods for supervised neural networks,” *Network: Computation in Neural Systems*, vol. 6, no. 3, pp. 469–505, 1995.

[2] A. G. Wilson and P. Izmailov, “Bayesian Deep Learning and a Probabilistic Perspective of Generalization,” *arXiv:2002.08791*, 2020.

[3] T. Charnock, L. Perreault-Levasseur, and F. Lanusse, “Bayesian neural networks,” *arXiv:2006.01490*, 2020.

[4] C. Blundell, J. Cornebise, K. Kavukcuoglu, and D. Wierstra, “Weight Uncertainty in Neural Networks,” *arXiv:1505.05424*, 2015.

[5] V. Dwaracherla, X. Lu, M. Ibrahimi, I. Osband, Z. Wen, and B. V. Roy, “Hypermodels for exploration,” *arXiv:2006.07464*, 2020.

[6] Y. Gal and Z. Ghahramani, “Dropout as a bayesian approximation: Representing model uncertainty in deep learning,” *arxiv:1506.02142*, 2016.

[7] I. Osband, J. Aslanides, and A. Cassirer, “Randomized prior functions for deep reinforcement learning,” *arXiv:1806.03335*, 2018.

[8] I. Oleksiienko, D. T. Tran, and A. Iosifidis, “Variational neural networks,” *arxiv:2207.01524*, 2022.

[9] G. Van Rossum and F. L. Drake, *Python 3 Reference Manual.* CreateSpace, 2009.

[10] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Kopf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, “Pytorch: An imperative style, high-performance deep learning library,” in *Advances in Neural Information Processing Systems 32.* Curran Associates, Inc., 2019, pp. 8024–8035.

[11] L. Deng, “The mnist database of handwritten digit images for machine learning research,” *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[12] A. Krizhevsky, “Learning multiple layers of features from tiny images,” Tech. Rep., 2009.

[13] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, C. Leary, D. Maclaurin, G. Necula, A. Paszke, J. VanderPlas, S. Wanderman-Milne, and Q. Zhang. (2018) JAX: composable transformations of Python+NumPy programs. [Online]. Available: http://github.com/google/jax

[14] I. Osband, Z. Wen, M. Asghari, M. Ibrahimi, X. Lu, and B. V. Roy. (2021) Epistemic neural networks implementation. [Online]. Available: https://github.com/deepmind/enn

[15] ——, “Epistemic Neural Networks,” *arXiv:2107.08924*, 2021.