Journal of Machine Learning Research 27 (2026) 1-8 Submitted 7/25; Revised 2/26; Published 3/26

py/cuTAGI: An Open-Source Library for Tractable Approximate Gaussian Inference in Bayesian Neural Networks

Luong-Ha Nguyen LUONGHA.NGUYEN@GMAIL.COM
James-A. Goulet JAMES.GOULET@POLYMTL.CA
Miquel Florensa-Montilla MIQUEL.FLORENSA-MONTILLA@POLYMTL.CA
Van-Dai Vuong VAN-DAI.VUONG@POLYMTL.CA
Polytechnique Montréal, Montréal, Canada

Editor: Zeyi Wen

Abstract

This paper introduces pyTAGI, a Python wrapper, and cuTAGI, its high-performance C++/CUDA backend, implementing Tractable Approximate Gaussian Inference (TAGI) for neural networks. TAGI treats all network quantities as Gaussian random variables and derives closed-form expressions for prior/posterior expected values, variances, and covariances, enabling analytic Bayesian learning without relying on gradient descent or backpropagation.

The libraries mimic PyTorch's sequential interface, allowing users to define models by stacking layers in order and performing uncertainty-aware Bayesian inference. Beyond epistemic uncertainty, it also allows quantifying heteroscedastic aleatoric uncertainty. cuTAGI's custom CPU/GPU kernels and distributed-data-parallel support via NCCL/MPI deliver competitive runtimes, while pyTAGI's pip-installable frontend and MIT-licensed GitHub repo facilitate community adoption and extension. Version 0.2.1 already supports a comprehensive suite of layers and activations; future work will add eager execution, further kernel optimizations, attention mechanisms, and advanced covariance factorization. Together, py/cuTAGI offer an efficient, open-source foundation for the analytic treatment of Bayesian deep learning.

Keywords: Bayesian, Neural Networks, Deep learning, TAGI, Approximate Inference

1. Introduction

This paper presents the pyTAGI, a Python wrapper, and cuTAGI, its custom C++/CUDA backend building upon the Tractable Approximate Gaussian Inference (TAGI) method (Goulet et al., 2021; Nguyen and Goulet, 2022). TAGI enables closed-form analytical Bayesian inference for estimating the parameters and latent variables in deep neural networks. It models unknown parameters and hidden states in neural networks as Gaussian random variables. Its principle is to provide closed-form formulations for computing the expected values, variances and covariances, in order to enable propagating uncertainty through common network architectures. In addition, it provides the closed-form formulations that allow performing approximate Bayesian inference using the Gaussian conditional equations, similar to what is done in state-space models (Simon, 2006) and Gaussian process regression

(Williams and Rasmussen, 2006). A first key aspect enabling the tractability of TAGI at scale is the recursive application of the Gaussian conditional equations in order to infer the parameters and hidden units, from the output to the input layer. This allows only storing the covariance matrices for pairs of layers, enabling linear computational complexity and storage with respect to the number of parameters. The second key aspect is related to the usage of diagonal structures for the covariance matrices. As shown by Goulet et al. (2021), although the approximate Gaussian Inference method in TAGI is theoretically applicable for any parameters and hidden units covariance structure, only the diagonal one is scalable to modern deep neural network architectures like gradient backpropagation is.

The TAGI method does not rely on gradient descent and backpropagation (Rumelhart et al., 1986) in order to learn the network parameters. Therefore, most mainstream deep-learning frameworks such as PyTorch (Paszke et al., 2019), TensorFlow (Abadi et al., 2016) or those building on JAX (Google JAX Team, 2018) are not suited for it. TAGI is also uniquely different from other Bayesian neural network methods (Papamarkou et al., 2024) relying on either sampling (Izmailov et al., 2021), variational inference (Wu et al., 2019; Louizos and Welling, 2016), stochastic gradient descent (Welling and Teh, 2011) or natural gradient (Khan and Rue, 2023), ensembling (Pearce et al., 2020; Lakshminarayanan et al., 2017), Dropout (Gal and Ghahramani, 2016), or the Laplace Approximation (Ritter et al., 2018; Daxberger et al., 2021). Besides these methods, Wagner et al. (2023), have reused the same recursive Gaussian inference framework behind TAGI under a variant named Kalman Bayesian Neural Networks. The py/cuTAGI libraries are thus intended to provide an efficient foundation for the community to use the TAGI method and build upon it. Because TAGI involves operations, not on scalars, but on random variables described by expected values, variances and covariances, it requires specialized kernels that cannot be efficiently implemented as simple matrix additions and multiplications. This paper is organized as follows: Section 2 presents the fundamental concepts behind the library's design, section 3 presents the neural network layers available in the libraries along with their main capabilities, and finally, 4 presents the state of current developments along with future outlook.

2. Library's Design

The library consists of a C++/CUDA backend (cuTAGI) and a Python wrapper (pyTAGI) that are integrated using pybind11. Its structure has been designed to mimic PyTorch in order to facilitate interoperability, development and user interactions. The backend has been implemented in order to minimize the dependencies on external libraries. It contains custom-build performance-oriented CPU and GPU kernels that are tailored to maximize the efficiency of the TAGI method. pyTAGI adopts PyTorch's sequential model building as displayed in Listing 1. Calling `m_pred, v_pred = net(x)` sequentially computes the prior/posterior predictive for each layer from the input `x`, and returns the output's expected values and variances. Calling `net.backward()` recursively computes the quantities `delta_mu` & `delta_var` that need to be added to the prior

Listing 1: Example of sequential network definition

```
net = Sequential(
    Conv2d(1, 16, 4,
        padding=1,
        in_width=28,
        in_height=28),
    ReLU(),
    AvgPool2d(3, 2),
    Conv2d(16, 32, 5),
    ReLU(),
    AvgPool2d(3, 2),
    Linear(32*4*4, 100),
    ReLU(),
    Linear(100, 11))
```

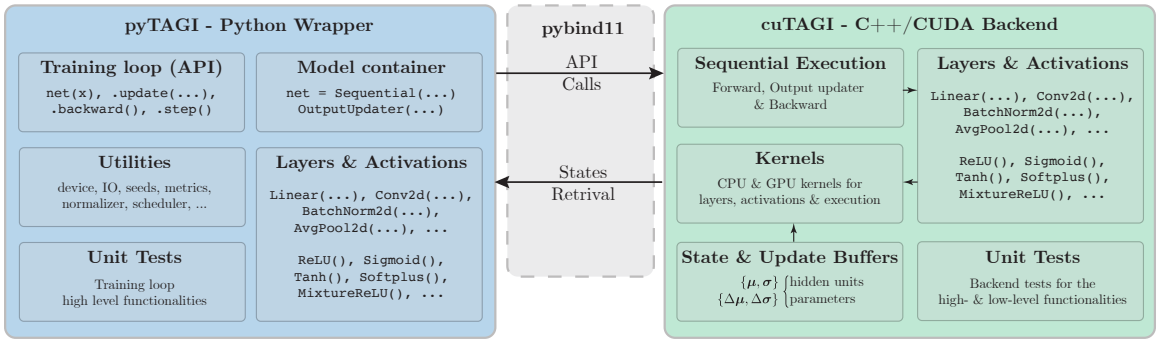

Figure 1: High-level overview of py/cuTAGI library design where the Python wrapper is connected to the C++/CUDA backend using pybind11.

in order to obtain the posterior expected values and variances for hidden layers, their parameters, and latent variables. The backward operation is applied from the output to the input and omits the explicit computation and storage of the posterior for hidden units in order to maximize efficiency. Calling `net.step()` updates the posterior expected values and variances for parameters.

Figure 1 presents a high-level representation of the library's design, where the Python wrapper allows controlling the training loop using a model container to define the architecture by stacking layers, along with various utilities. API calls control the sequential execution in the backend, where layers are defined with custom-build C++/CUDA kernels. The state and update buffers are handled in the backend, whereas their values can be retrieved to the Python interface. Both the wrapper and the backend have their own sets of unit tests for low- and high-level functionalities.

The library can be installed via `pip install pytagi` from the Python Package Index (PyPI). Developers can access the source code released under MIT licence, which is hosted on GitHub (`https://github.com/lhnguyen102/cuTAGI`) with detailed instructions for installing it locally, and for contributing to it. The repository hosts the documentation for the API (`https://www.tagiml.com`), along with a collection of tutorials, examples, and unit tests involving the various layers that are available.

3. Neural Network Layers and Capabilities

The current version 0.2.1 includes the implementation of several core layer formulations for standard neural network architectures. The library includes the linear, convolutional (CNN) & transposed convolutional (Nguyen and Goulet, 2021), average pooling, max pooling, batch & layer normalization, residual layers, as well as long short-term memory (LSTM, Vuong et al. (2025)). All these specialized layers are built by inheriting from the `BaseLayer` class. The same strategy should be leveraged by developers in order to implement new architectures. The library includes the prior initialization strategies for the expected values and variances of parameters developed by Goulet et al. (2021) in order to adapt the deterministic initialization procedures from Xavier (Glorot and Bengio, 2010) and He (He et al., 2015) to the probabilistic setup where we need to define a prior for the parameters of each layer. Users have the possibility of overwriting the default initialization setup through gain hyperparameters that

are accessible in the Python API. In addition to the various layers, it includes common activation functions, i.e., ReLU, Sigmoid, Tanh, Softplus, LeakyReLU and Exponential.

py/cuTAGI trains deep neural networks using closed-form approximate Bayesian inference without using gradient backpropagation so that it inherently enables quantifying the epistemic uncertainty associated with the parameters and latent variables. This uncertainty can be leveraged through the prior/posterior predictives that are the standard outputs with TAGI. In addition, the Approximate Gaussian Variance Inference (AGVI) method (Deka et al., 2024, 2023) is implemented in the library in order to enable the quantification of heteroscedastic aleatoric uncertainties associated with the network's output. This enables quantifying the variability in the discrepancy between predicted and observed values as a function of the neural network's inputs, while maintaining the explicit separation between epistemic and aleatoric uncertainties.

In order to maximize the scalability, the default strategy is to rely on diagonal structures for the covariance matrices of parameters and hidden layers. Nevertheless, for research purposes, the library also has the capability of considering a full-covariance structure for hidden units of the linear layers (CPU only). This feature is useful in order to enable accurate uncertainty propagation from the input to the output layer. Nevertheless, it is inevitably accompanied by a substantially increased computational demand associated with handling full rather than diagonal matrices. Comparative studies of different covariance structures for hidden layers and parameters has already been detailed by Goulet et al. (2021). Another feature of the library is its support Distributed Data Parallel (DDP) for multi-GPU setups via NVIDIA Collective Communications Library (NCCL) and Message Passing Interface (MPI). In the current version, the multi-GPU parallelization is only supported over batch processing.

4. Current Developments & Outlook

The development of the py/cuTAGI has been ongoing since 2022. The initial TAGI implementation was intended to establish a proof of concept (Goulet et al., 2021; Nguyen and Goulet, 2022) and its slow runtime and limited GPU support have been the bottleneck in the scalability of the method. As shown in table 1, the current pyTAGI release has runtime and memory requirements that are now comparable to PyTorch for GPU computation (The experiment details are provided in Appendix A). For networks involving highly optimized layers, such as the linear or convolutional one, the pyTAGI's runtime is on par with PyTorch.For more complex architectures such as ResNet for which the residual layer's computation kernels have not yet been extensively optimized, the runtime and memory requirements are still

Table 1: Comparison of runtime and memory requirements between PyTorch and pyTAGI.

		Runtime in ms/10 iterations (forward & backward)				Maximum amount of memory in MB			
		CPU		GPU		CPU		GPU	
Model	Dataset	PyTorch	pyTAGI	PyTorch	pyTAGI	PyTorch	PyTAGI	PyTorch	pyTAGI
MLP 128	Protein	0.4 ± 0.0	0.7 ± 0.0	2.6 ± 0.4	0.3 ± 0.0	347	307	282	214
MLP 2-4096	MNIST	68.5 ± 4.2	9421.6 ± 785.9	36.2 ± 1.5	32.7 ± 1.2	1158	687	694	917
2 CNN	MNIST	34.9 ± 1.4	206.5 ± 9.5	30.3 ± 0.8	41.6 ± 0.9	421	483	330	306
3 CNN-BN	CIFAR10	134.8 ± 5.7	1563.9 ± 58.8	95.8 ± 2.6	84.5 ± 0.6	1149	1264	442	512
ResNet-18	CIFAR10	2428 ± 17	103323 ± 15854	98 ± 1.7	285.7 ± 1.3	1952	3341	1358	3390

within a factor three from PyTorch, all of it while performing analytical Bayesian learning rather than relying on gradient backpropagation.

The future developments that are envisioned include four main aspects. The first is the refinement of the computation kernels; besides the linear layer that is already highly optimized and that is on par with PyTorch's runtime, there is still room for improving the performance of other kernels involved in the CNN, residual, and other layers. The second development involves the introduction of eager execution in order to facilitate debugging and the development of complex neural network architectures. The third aspect is related to the development and implementation of additional standard deep learning architectures involving attention mechanisms; In addition to requiring the development of the analytical formulations for the moments (expected values, variances, covariances) involved in the internal operations, it also requires the development of an analytically tractable probabilistic Softmax function that allows sending information forward and backward. Finally the library will be further extended with the ongoing research carrying on advanced prior initialization procedures as well as alternative factorization of the covariance matrices for the parameters.

Acknowledgements

The authors acknowledge the contribution of David Wardan and Zhanwen Xin, for their contribution to the development of the LSTM layer and Lucas Alric in the derivation of the analytical formulation for the mixture-based activation function implemented in py/cuTAGI. The development of the library was supported by research grants from the Natural Sciences and Engineering Research Council of Canada (NSERC), Hydro-Quebec and the Quebec Transportation Ministry (MTQ).

Contribution Statement

Luong-Ha Nguyen is the main architect and developer behind py/cuTAGI. James-A. Goulet has contributed to the methodological developments and implementation reviews, Miquel Florensa has contributed with the Heteroscedastic aleatoric uncertainty quantification, and Van Dai Vuong with the LSTM layer.

References

M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, Mi. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: a system for large-scale machine learning. In *USENIX Symposium on Operating Systems Design and Implementation*, page 265–283, 2016.

E. Daxberger, A. Kristiadi, A. Immer, R. Eschenhagen, M. Bauer, and P. Hennig. Laplace redux-effortless Bayesian deep learning. In *Advances in Neural Information Processing Systems*, volume 34, pages 20089–20103, 2021.

B. Deka, L.H. Nguyen, and J.-A. Goulet. Approximate Gaussian variance inference for state-space models. *International Journal of Adaptive Control and Signal Processing*, 37

(11):2934–2962, 2023.

B. Deka, L.H. Nguyen, and J.-A Goulet. Analytically tractable heteroscedastic uncertainty quantification in Bayesian neural networks for regression tasks. *Neurocomputing*, 572: 127183, 2024.

Y. Gal and Z. Ghahramani. Dropout as a Bayesian approximation: Representing model uncertainty in deep learning. In *International Conference on Machine Learning*, pages 1050–1059. PMLR, 2016.

X. Glorot and Y. Bengio. Understanding the difficulty of training deep feedforward neural networks. In *International conference on artificial intelligence and statistics*, pages 249–256, 2010.

Google JAX Team. JAX: composable transformations of Python+NumPy. `https://github.com/google/jax`, 2018.

J.-A. Goulet, L.H. Nguyen, and S. Amiri. Tractable approximate Gaussian inference for Bayesian neural networks. *Journal of Machine Learning Research*, 22(251):1–23, 2021.

K. He, X. Zhang, S. Ren, and J. Sun. Delving deep into rectifiers: Surpassing human-level performance on Imagenet classification. In *IEEE International Conference on Computer Vision*, pages 1026–1034, 2015.

P. Izmailov, S. Vikram, M.D. Hoffman, and A.G. Wilson. What are Bayesian neural network posteriors really like? In *International Conference on Machine Learning*, pages 4629–4640. PMLR, 2021.

M.E. Khan and H. Rue. The Bayesian learning rule. *Journal of Machine Learning Research*, 24(281):1–46, 2023.

B. Lakshminarayanan, A. Pritzel, and C. Blundell. Simple and scalable predictive uncertainty estimation using deep ensembles. In *International Conference on Neural Information Processing Systems*, pages 6405–6416, 2017.

C. Louizos and M. Welling. Structured and efficient variational deep learning with matrix Gaussian posteriors. In *International Conference on Machine Learning*, pages 1708–1716. PMLR, 2016.

L.H. Nguyen and J.-A. Goulet. Analytically tractable inference in deep neural networks. *arXiv preprint arXiv:2103.05461*, 2021.

L.H. Nguyen and J.-A. Goulet. Analytically tractable hidden-states inference in Bayesian neural networks. *Journal of Machine Learning Research*, 23(50):1–33, 2022.

T. Papamarkou, M. Skoularidou, K. Palla, L. Aitchison, J. Arbel, D. Dunson, M. Filippone, V. Fortuin, P. Hennig, J.M. Hernández-Lobato, A. Hubin, A. Immer, T. Karaletsos, M.E. Khan, A. Kristiadi, Y. Li, S. Mandt, C. Nemeth, M.A. Osborne, T.G.J. Rudner, D. Rügamer, Y.W. Teh, M. Welling, A.G. Wilson, and R. Zhang. Position: Bayesian deep learning is needed in the age of large-scale AI. In *International Conference on Machine Learning*, volume 235, pages 39556–39586, 2024.

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*, pages 8024–8035. 2019.

T. Pearce, F. Leibfried, and A. Brintrup. Uncertainty in neural networks: Approximately Bayesian ensembling. In *International Conference on Artificial Intelligence and Statistics*, pages 234–244. PMLR, 2020.

H. Ritter, A. Botev, and D. Barber. A scalable Laplace approximation for neural networks. In *International Conference on Learning Representations*, volume 6, 2018.

D.E. Rumelhart, G.E. Hinton, and R.J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

D. Simon. *Optimal state estimation: Kalman, H infinity, and nonlinear approaches*. Wiley, 2006.

V.D. Vuong, L.H. Nguyen, and J.-A. Goulet. Coupling LSTM neural networks and state-space models through analytically tractable inference. *International Journal of Forecasting*, 41 (1):128–140, 2025.

P. Wagner, X. Wu, and M.F. Huber. Kalman Bayesian neural networks for closed-form online learning. In *AAAI Conference on Artificial Intelligence*, volume 37, pages 10069–10077, 2023.

M. Welling and Y.W. Teh. Bayesian learning via stochastic gradient Langevin dynamics. In *International Conference on Machine Learning*, pages 681–688, 2011.

C.K. Williams and C.E. Rasmussen. *Gaussian processes for machine learning*, volume 2. MIT press, 2006.

A. Wu, S. Nowozin, E. Meeds, R.E. Turner, J.M. Hernandez-Lobato, and A.L. Gaunt. Deterministic variational inference for robust Bayesian neural networks. In *International Conference on Learning Representations*, 2019.

Appendix A. Time and memory Comparison Results

Table 1 compares the runtime and memory requirements between PyTorch and pyTAGI. The runs on CPU were made on an M2 Macbook Air 16GB of memory with 8 threads and the GPU runs on a Nvidia RTX 4070 TI Super. All experiments were run using a batch size of 128 and the first iteration of each run is not considered for the computation time in order to avoid counting network initialization. The first three experiments involve 10 repetitions for 10 iterations (forward and backward pass) and the last two, 5 repetitions for 10 iterations. The maximum amount of memory used in each configuration is measured using `htop` in the case of CPU and `nvidia-smi` in GPU in order to not interfere with the execution time. The datasets and architectures used in the comparison are:

- **MLP 128:** 1 linear layer of 128 units on the UCI Protein dataset.
- **MLP 2-4096:** 2 linear layers of 4096 units on the MNIST dataset.
- **2 CNN:** 2 CNN layers with 16 & 32 output channels and average pooling on the MNIST dataset.
- **3 CNN:** 3 CNN layers with 32, 32 and 64 output channels and average pooling on the CIFAR10 dataset.
- **ResNet-18:** ResNet on CIFAR-10 with MixtureReLU on the CIFAR10 dataset.

All experiments use ReLU activation functions.