The Journal of Open Source Software

Sparsity-preserving gradient utility tools for PyTorch

Theodore Barfoot [1], **Ben Glocker** [2], **and Tom Vercauteren** [1]

1 King's College London, London, UK **2** Imperial College London, London, UK

DOI: 10.xxxxxx/draft

Software

- Review
- Repository
- Archive

Editor: Abhishek Tiwari

Reviewers:

- @yewentao256

Submitted: 17 September 2025
Published: unpublished

Summary

The `torchsparsegradutils` package provides differentiable sparse linear-algebra utilities for PyTorch (Paszke et al., 2019) that preserve sparsity for returned gradients during backpropagation. While PyTorch directly supports sparse tensors, its default semantics treat sparse layouts as storage optimisations rather than a mathematical structure that results in optimising directly for that sparse subspace. Gradients resulting from PyTorch native functions are often dense and incompatible with end-to-end training of models that require fixed sparsity patterns (e.g., sparse covariance/precision structures).

To address this limitation, we introduce `torchsparsegradutils`. Key features include: (1) memory-efficient sparse-dense matrix multiplication with sparse gradient preservation, (2) sparse triangular and generic linear system solvers, enabling sparse gradients during backpropagation, and multiple algorithmic backends (BICGSTAB, CG, LSMR, MINRES), (3) cross-platform sparse solver wrappers for CuPy (Okuta et al., 2017) and JAX (Bradbury et al., 2018), (4) sparse multivariate normal distributions with $\boldsymbol{LL}^T$ and $\boldsymbol{LDL}^T$ sparse covariance and precision matrix parameterisations with reparameterised sampling methods, and (5) specialised encoders for spatial neighbourhood relationships in N-dimensional data.

Statement of need

Many scientific machine learning models benefit from representing large linear operators (e.g., neighbourhood couplings, precision matrices, sparse Jacobians) using sparse tensors to reduce memory and compute complexity. In high-dimensional settings, such as volumetric medical imaging, dense covariance or precision parameterisations are typically intractable, motivating sparse end-to-end parameterisations.

However, learning these models with gradient-based optimisation requires backpropagation through sparse linear algebra (matrix products, triangular solves, and linear system solves). PyTorch's default sparse semantics are not designed to preserve user-imposed sparsity structure during differentiation (PyTorch issue #87448), which can lead to memory blow-ups and prevent end-to-end optimisation of sparse probabilistic models.

`torchsparsegradutils` addresses this gap by implementing custom autograd functions for key sparse operators that return gradients only for stored nonzeros, enabling practical optimisation of models that rely on fixed sparse structure, such as sparse multivariate normal distributions with sparse covariance/precision factors.

State of the field

PyTorch (Paszke et al., 2019) exposes sparse layouts (COO, CSR, and related formats) and implements a growing set of sparse operations. However, PyTorch's design goal is *dense-equivalent semantics* for sparse layouts: a guiding invariant is that applying an operation in

sparse form should match applying it in dense form after conversion, including the backward function (PyTorch issue #87448). This makes it difficult to learn parameters that are intended to remain structurally sparse, because gradients may be produced for implicit zeros, or intermediate computations may densify.

PyTorch also provides `MaskedTensor`, distringuishing specified and unspecified elements in tensors and is conceptually closer to the constrained-subspace interpretation of sparsity. However, `MaskedTensor` remains at prototype stage with incomplete operator coverage, and storing a full boolean mask incurs a significant memory overhead, partially negating the memory benefits of sparse index-based representations for large-scale problems.

Other libraries provide efficient sparse kernels but do not directly solve "sparsity-preserving gradients in PyTorch": SciPy (Virtanen et al., 2020) provides mature sparse linear algebra but no automatic differentiation; CuPy (Okuta et al., 2017) and JAX (Bradbury et al., 2018) provide sparse solvers in their respective ecosystems but are not drop-in components for PyTorch autograd/training loops. GPyTorch (Gardner et al., 2018) targets scalable Gaussian process inference via kernel structure and approximations (e.g., inducing/structured methods) rather than arbitrary user-specified sparse covariance/precision factors. PyTorch Geometric's torch_sparse (Fey & Lenssen, 2019) focuses on graph message-passing primitives rather than sparse covariance/precision modelling and differentiable sparse solves for probabilistic models.

Software design

`torchsparsegradutils` is built around `torch.autograd.Function` operators that wrap PyTorch's forward sparse kernels but override the backward pass to preserve sparsity for selected inputs. This design keeps the user-facing API close to standard PyTorch code while making sparsity preservation an explicit, opt-in choice.

Two design trade-offs shaped the implementation. First, the package targets *structure-preserving learning* over maximal operator coverage, as only a focused set of operations (sparse matrix products, triangular solves, generic sparse solvers) are implemented, but these are sufficient to support sparse multivariate normal sampling and sparse solver-based models. Second, for broad device/backend compatibility, the package combines native PyTorch implementations (iterative Krylov solvers: CG, BiCGSTAB, LSMR, MINRES) with optional wrappers to external libraries (CuPy, JAX), allowing users to trade off portability versus performance.

Build vs. contribute justification. PyTorch's current semantics treat sparse layouts as performance optimisations and prioritise the dense-equivalence invariant (PyTorch issue #87448). In contrast, this package intentionally provides *structure-preserving* backward passes for specific operators to enable learning with fixed sparsity patterns (e.g., sparse triangular factors for covariance/precision). This difference is semantic (not just implementation), so the functionality is better delivered as an opt-in external library rather than changing PyTorch's default behaviour.

Research impact statement

This software provides an opt-in path to sparsity-preserving gradients for sparse linear algebra in PyTorch, enabling research prototypes that would otherwise be limited by dense gradients or densification. The package is currently being used in active research projects for medical image segmentation, though publications resulting from this work are still in preparation.

The codebase demonstrates community-readiness through comprehensive infrastructure: documentation with quickstart guides and API references, extensive test coverage across all modules, CI/CD pipelines for automated testing, and an open contribution process via GitHub issues and pull requests. The codebase has been developed openly over multiple

years with public commit history, releases, and issue tracking. Benchmark suites comparing solver performance across problem sizes and sparsity patterns provide reproducible reference materials.

Given the broad applicability of sparse structured Gaussians—spanning medical imaging, spatial statistics, geostatistics, and large-scale probabilistic modelling, we anticipate growing adoption as the research community increasingly requires memory-efficient optimisation of high-dimensional probabilistic models.

Mathematics

Sparse Matrix Operations

Sparse Matrix Multiplication

The package implements sparse-dense matrix multiplication $\mathbf{C} = \mathbf{AB}$ where $\mathbf{A} \in \mathbb{R}^{m \times n}$ is sparse and $\mathbf{B} \in \mathbb{R}^{n \times p}$ is dense. The forward pass uses PyTorch's native `torch.sparse.mm`, while the backward pass is reimplemented to preserve sparsity patterns in the gradients.

Given upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{C}} \in \mathbb{R}^{m \times p}$ from some scalar objective function $\mathcal{L}$, the chain rule gives:

Gradient wrt $\mathbf{B}$ (dense):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{B}} = \mathbf{A}^\top \frac{\partial \mathcal{L}}{\partial \mathbf{C}}.$$

Gradient wrt $\mathbf{A}$ (sparse): For a nonzero entry $\mathbf{A}_{ij}$,

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}_{ij}} = \sum_{k=1}^{p} \left(\frac{\partial \mathcal{L}}{\partial \mathbf{C}_{ik}} \right) \mathbf{B}_{jk}.$$

Sparse Linear System Solvers

The package provides multiple approaches for solving sparse linear systems

$$\mathbf{Ax} = \mathbf{B},$$

where $\mathbf{A} \in \mathbb{R}^{n \times n}$ is sparse, $\mathbf{B} \in \mathbb{R}^{n \times p}$ is dense (with $p = 1$ for a single right-hand side), and $\mathbf{x} \in \mathbb{R}^{n \times p}$ is the dense solution. We support both direct triangular solves and iterative solvers (CG, BiCGSTAB, LSMR, MINRES). All are differentiable via the implicit function theorem.

Given upstream gradient $\frac{\partial \mathcal{L}}{\partial \mathbf{x}} \in \mathbb{R}^{n \times p}$:

Gradient wrt $\mathbf{b}$ (dense):

$$\frac{\partial \mathcal{L}}{\partial \mathbf{b}} = \mathbf{A}^{-\top} \frac{\partial \mathcal{L}}{\partial \mathbf{x}}.$$

Gradient wrt $\mathbf{A}$ (sparse): The dense form would be

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}} = -\left(\mathbf{A}^{-\top} \frac{\partial \mathcal{L}}{\partial \mathbf{x}} \right) \mathbf{x}^\top.$$

For a nonzero entry $\mathbf{A}_{ij}$ this becomes

$$\frac{\partial \mathcal{L}}{\partial \mathbf{A}_{ij}} = -\sum_{k=1}^{p} \left(\mathbf{A}^{-\top} \frac{\partial \mathcal{L}}{\partial \mathbf{x}} \right)_{ik} \mathbf{x}_{jk}$$

Sparse Multivariate Normal Distributions

The package implements sparse multivariate normal distributions $\boldsymbol{\eta} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \equiv \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Omega}^{-1})$ with two parameterisations for efficient sampling. Both methods transform standard normal samples $\boldsymbol{\epsilon} \sim \mathcal{N}(\mathbf{0}, \boldsymbol{I})$ into samples from the desired multivariate normal distribution:

$\boldsymbol{LL}^\top$ **Parameterisation**: Sparse lower triangular matrices $\boldsymbol{L}$ with positive diagonals:

$$\boldsymbol{\eta}_{\boldsymbol{\Sigma}} = \boldsymbol{\mu} + \boldsymbol{L}_{\Sigma}\boldsymbol{\epsilon}, \quad \boldsymbol{\eta}_{\boldsymbol{\Omega}} = \boldsymbol{\mu} + \boldsymbol{L}_{\Omega}^{-T}\boldsymbol{\epsilon}$$

$\boldsymbol{LDL}^\top$ **Parameterisation**: Sparse unit lower triangular matrices $\boldsymbol{L}$ and diagonal $\boldsymbol{D}$:

$$\boldsymbol{\eta}_{\boldsymbol{\Sigma}} = \boldsymbol{\mu} + \boldsymbol{L}\boldsymbol{D}^{1/2}\boldsymbol{\epsilon}, \quad \boldsymbol{\eta}_{\boldsymbol{\Omega}} = \boldsymbol{\mu} + \boldsymbol{L}_{\Omega}^{-\top}\boldsymbol{D}_{\Omega}^{-1/2}\boldsymbol{\epsilon}$$

We store $\boldsymbol{L}$ without its diagonal (strictly lower), and treat it as unit lower-triangular at use time. The $\boldsymbol{LDL}^\top$ parameterisation provides superior numerical stability for precision matrices by avoiding strict positive definiteness constraints.

AI usage disclosure

Generative AI tools were used during development of this software and manuscript. Various large language models were used to assist with code generation, refactoring, and test scaffolding for portions of the codebase, and AI assistance was used to draft and edit parts of the documentation and this manuscript. The repository was initiated prior to widespread AI coding assistant adoption, with AI tools incorporated during later development phases. All AI-assisted outputs were reviewed, edited, and validated by the human authors, who take responsibility for the final software and paper.

Acknowledgements

We thank the PyTorch development team for foundational sparse tensor support. We also acknowledge upstream solver implementations and references used as starting points for iterative methods (pykrylov, cornellius-gp/linear_operator, pytorch-minimize) (Saad, 2003). We thank Floris Laporte for his excellent tutorial on implementing sparse linear system solvers in PyTorch (Laporte, 2020), which provided valuable insights for gradient computation strategies. This work was supported by the Wellcome/EPSRC Centre for Interventional and Surgical Sciences and the School of Biomedical Engineering & Imaging Sciences, King's College London.

References

Bradbury, J., Frostig, R., Hawkins, P., Johnson, M. J., Leary, C., Maclaurin, D., Necula, G., Paszke, A., VanderPlas, J., Wanderman-Milne, S., & Zhang, Q. (2018). *JAX: Composable transformations of Python+NumPy programs*. https://github.com/jax-ml/jax

Fey, M., & Lenssen, J. E. (2019). Fast graph representation learning with PyTorch Geometric. *ICLR Workshop on Representation Learning on Graphs and Manifolds*. https://arxiv.org/abs/1903.02428

Gardner, J. R., Pleiss, G., Bindel, D., Weinberger, K. Q., & Wilson, A. G. (2018). GPyTorch: Blackbox matrix-matrix gaussian process inference with GPU acceleration. *Advances in Neural Information Processing Systems*, *31*. https://doi.org/10.5555/3327757.3327857

Laporte, F. (2020). *Solving sparse linear systems in PyTorch*. https://blog.flaport.net/solving-sparse-linear-systems-in-pytorch.html.

Okuta, R., Unno, Y., Nishino, D., Hido, S., & Loomis, C. (2017). CuPy: A NumPy-compatible library for NVIDIA GPU calculations. *Proceedings of the Workshop on Machine Learning Systems (LearningSys) at the 31st Conference on Neural Information Processing Systems*, 1–7. https://learningsys.org/nips17/assets/papers/paper_16.pdf

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., & others. (2019). PyTorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, *32*, 8024–8035. https://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library

Saad, Y. (2003). *Iterative methods for sparse linear systems* (2nd ed.). Society for Industrial; Applied Mathematics. https://doi.org/10.1137/1.9780898718003

Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., & others. (2020). SciPy 1.0: Fundamental algorithms for scientific computing in Python. *Nature Methods*, *17*(3), 261–272. https://doi.org/10.1038/s41592-019-0686-2