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, United Kingdom **2** Imperial College London, London, United Kingdom

DOI: 10.21105/joss.10072

Software

- Review
- Repository
- Archive

Editor: Abhishek Tiwari

Reviewers:

- @yewentao256
- @idoby

Submitted: 17 September 2025
Published: 06 July 2026

Summary

The `torchsparsegradutils` package provides differentiable sparse linear-algebra utilities for PyTorch (Paszke et al., 2019) that preserve sparsity in returned gradients during backpropagation. While PyTorch supports sparse tensors, its default dense-equivalent backward semantics can densify gradients and make it difficult to optimise models with fixed sparsity patterns, such as sparse covariance or precision parameterisations.

The package provides sparse-dense matrix multiplication with sparse-gradient preservation, sparse triangular and generic linear system solvers (including BiCGSTAB, CG, LSMR, and MINRES backends), optional CuPy (Okuta et al., 2017) and JAX (Bradbury et al., 2018) solver wrappers, sparse multivariate normal distributions with $\boldsymbol{LL}^\top$ and $\boldsymbol{LDL}^\top$ parameterisations, and specialised encoders for spatial neighbourhood relationships in N-dimensional data.

The source code is available on GitHub at https://github.com/cai4cai/torchsparsegradutils, with full documentation hosted at https://torchsparsegradutils.readthedocs.io.

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 (Vercauteren, 2022), 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.

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 (Vercauteren, 2022). 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`, which distinguishes specified and unspecified elements and is conceptually closer to the constrained-subspace interpretation of sparsity. However, PyTorch classifies the API as prototype stage, operator support is limited, and each `MaskedTensor` stores a separate mask tensor matching the data tensor's size and storage format, introducing additional mask-storage overhead (PyTorch Contributors, 2026b, 2026a).

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 keeps the 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, focusing on sparse matrix products and sparse solves that support sparse multivariate normal sampling and related models. Second, it combines native PyTorch implementations (including CG, BiCGSTAB, LSMR, and MINRES) with optional CuPy and JAX wrappers so users can trade off portability and performance.

Build vs. contribute justification. PyTorch's current sparse semantics prioritise dense-equivalent behaviour (Vercauteren, 2022). In contrast, this package intentionally provides structure-preserving backward passes for specific operators to enable learning with fixed sparsity patterns. Because that is a semantic choice rather than just an implementation detail, the functionality is better delivered as an opt-in external library than as a change to PyTorch defaults.

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 already being used in ongoing medical-image segmentation projects, and the public repository provides tests, documentation, benchmarks, and issue tracking to support reuse and extension.

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 with respect to $\mathbf{B}$ (dense):

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

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

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

Sparse Linear System Solvers

The package provides multiple approaches for solving sparse linear systems

$$\mathbf{A}\mathbf{X} = \mathbf{B},$$

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 (including CG, BiCGSTAB, LSMR, and 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 with respect to $\mathbf{B}$ (dense):

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

Gradient with respect to $\mathbf{A}$ (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:

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

$\boldsymbol{LDL}^\top$ **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 is practically more robust for learned precision matrices because triangular solves use a unit-diagonal factor, avoiding failure modes caused by learned diagonal entries that become zero or nearly zero. The remaining scale is isolated in $\boldsymbol{D}$ and inverted elementwise, avoiding the stricter positive-diagonal and full-rank requirements of a Cholesky-style $\boldsymbol{LL}^\top$ factor.

Usage Examples

Short examples are shown below; fuller worked examples are available in the Read the Docs quickstart.

```
import torch
from torchsparsegradutils import sparse_mm, sparse_generic_solve
from torchsparsegradutils.distributions import SparseMultivariateNormal as SMVN
from torchsparsegradutils.utils import linear_cg, rand_sparse, rand_sparse_tri

n = 100
A = rand_sparse((n, n), 500).requires_grad_()
sparse_mm(A, torch.randn(n, 8, requires_grad=True)).sum().backward()

A = torch.eye(n).to_sparse().requires_grad_()
sparse_generic_solve(A, torch.randn(n), solve=linear_cg).sum().backward()

L = rand_sparse_tri((n, n), 300, upper=False, strict=True).requires_grad_()
SMVN(torch.zeros(n), torch.ones(n), scale_tril=L).rsample().sum().backward()
```

Benchmarks

On the SuiteSparse Rothberg/cfd2 matrix (Davis & Hu, 2011; Rothberg, 1997) ($123{,}440 \times 123{,}440$, 3.1 million nonzeros), dense baselines and PyTorch's native COO backward pass ran out of memory, whereas `torchsparsegradutils` completed sparse matrix-multiplication backward in about 75 ms using 5.1 GB on one tested RTX 4090 setup (results vary by hardware). On the same setup, native COO iterative solvers were up to about 40× faster than CuPy wrappers because they avoid sparse-format conversion overhead; full benchmark scripts and hardware-specific results are available in the repository and Read the Docs benchmark documentation.

AI usage disclosure

Generative AI tools were used intermittently during later development and maintainence of the software and preparation of the documentation and manuscript. This is a best-effort retrospective disclosure because exact provider-side model snapshots were not recorded contemporaneously for every historical interaction. Based on the authors' records and recollection, and on commit dates relative to public model-release dates, OpenAI ChatGPT with GPT-4, GPT-4o, and GPT-5-series models was used for code suggestions, refactoring, debugging, test scaffolding, and documentation drafting when those models were the latest available ChatGPT models. The original unit-test suite and foundational sparse operators were developed before the authors began using generative AI. Later maintenance, tests, and documentation may include AI-assisted contributions, and OpenAI Codex is also used for release-pipeline management.

All AI-assisted outputs were reviewed, edited, and validated by the human authors. Software changes were checked using code review, automated tests, continuous integration, and comparisons with reference behaviour where applicable, and all pull requests were reviewed entirely manually by human reviewers. The human authors made the primary mathematical, architectural, scientific, and editorial decisions and take responsibility for the final software and manuscript.

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, including PyKrylov (Orban & Developers, 2014), cornellius-gp/linear_operator (Gardner et al., 2017), and pytorch-minimize (Feinman, 2021), alongside the iterative-methods literature (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

Davis, T. A., & Hu, Y. (2011). The University of Florida Sparse Matrix Collection. *ACM Transactions on Mathematical Software*, *38*(1), 1:1–1:25. https://doi.org/10.1145/2049662.2049663

Feinman, R. (2021). *PyTorch-minimize: A library for numerical optimization with autograd*. GitHub. https://github.com/rfeinman/pytorch-minimize

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://arxiv.org/abs/1809.11165

Gardner, J. R., Pleiss, G., & Contributors, G. (2017). *LinearOperator: Structured linear algebra for PyTorch*. GitHub repository; GitHub. https://github.com/cornellius-gp/linear_operator

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

Orban, D., & Developers, P. (2014). *PyKrylov: Krylov methods in pure Python*. GitHub repository; GitHub. https://github.com/PythonOptimizers/pykrylov

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

PyTorch Contributors. (2026a). *MaskedTensor sparsity*. PyTorch tutorial. https://docs.pytorch.org/tutorials/unstable/maskedtensor_sparsity.html

PyTorch Contributors. (2026b). *torch.masked: MaskedTensor documentation*. PyTorch documentation. https://docs.pytorch.org/docs/stable/masked.html

Rothberg, E. (1997). *Rothberg/cfd2: CFD, symmetric pressure matrix* (T. A. Davis, Ed.).

SuiteSparse Matrix Collection. https://sparse.tamu.edu/Rothberg/cfd2

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

Vercauteren, T. (2022). *Semantics of sparse operations clarification: Sparsity of the gradient with respect to a sparse tensor input*. PyTorch GitHub issue #87448. https://github.com/pytorch/pytorch/issues/87448

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