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:

Submitted: 17 September 2025
Published: unpublished

Summary

The `torchsparsegradutils` package provides gradient-preserving sparse tensor operations for PyTorch (Paszke et al., 2019), addressing the critical limitation that PyTorch's native sparse operations do not support sparse gradients during backpropagation. This package enables memory-efficient optimisation of high-dimensional models by maintaining sparsity patterns throughout the entire forward and backward pass computation, supporting both coordinate list (COO) and compressed sparse row (CSR) formats.

Key features include: (1) memory-efficient sparse 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 volumetric data.

The package addresses PyTorch limitation of dense gradients resulting in memory errors, such as issue #41128, by implementing custom autograd functions that preserve sparsity in gradients, enabling practical training of models with sparse covariance and precision structures on high-dimensional data where dense alternatives become computationally intractable.

Statement of need

Sparse tensors are essential for computationally tractable machine learning on high-dimensional data, yet PyTorch's sparse tensor operations suffer from a critical limitation: gradients are computed in dense format even when the forward pass maintains sparsity. This results in prohibitive memory usage that scales quadratically with problem dimension rather than linearly with the number of non-zero elements (nnz).

For applications requiring sparse covariance modelling—such as medical imaging with millions of voxels, spatial statistics, and large-scale Gaussian processes—dense gradient computation renders optimisation infeasible. A sparse covariance matrix with 1 million dimensions and 0.1% sparsity contains 1 billion non-zero elements, but dense gradients would require storing 1 trillion parameters.

This package solves this fundamental limitation by implementing custom autograd functions that preserve sparsity patterns throughout both forward and backward passes. Our sparse multivariate normal distribution enables optimisation of million-dimensional Gaussian models with memory usage scaling as $O(nnz)$ rather than $O(n^2)$, where n is the dimension of the multivariate distribution.

Beyond memory efficiency, the package addresses the algorithmic challenge of sparse linear system solving by providing multiple iterative solver backends with automatic differentiation

support. This enables end-to-end optimisation of complex probabilistic models that would be computationally intractable with existing PyTorch sparse operations.

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:

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

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

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

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.

Acknowledgements

The authors acknowledge the PyTorch development team for providing the foundational sparse tensor infrastructure. We thank the SciPy (Virtanen et al., 2020), CuPy (Okuta et al., 2017), and JAX (Bradbury et al., 2018) communities for high-performance sparse linear algebra implementations. Algorithm implementations adapt and extend methods from pykrylov (BICGSTAB), cornellius-gp/linear_operator (CG, MINRES), and pytorch-minimize (LSMR) (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

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 NeurIPS*. 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.

Saad, Y. (2003). *Iterative methods for sparse linear systems* (2nd ed.). SIAM. https://www-users.cse.umn.edu/~saad/IterMethBook_2ndEd.pdf

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