Fast geometric learning with symbolic matrices

Jean Feydy\textsuperscript{1,}\textsuperscript{*}  Joan A. Glaunès\textsuperscript{2,}\textsuperscript{*}  Benjamin Charlier\textsuperscript{3,}\textsuperscript{*}  Michael M. Bronstein\textsuperscript{1,}\textsuperscript{4}.


\textsuperscript{1}Imperial College London, \textsuperscript{2}Université de Paris, \textsuperscript{3}Université de Montpellier, \textsuperscript{4}Twitter

* Equal contribution
Machine learning libraries represent most objects as tensors.

\[ M[i, j] \]

**Dense matrix**
Coefficients only

**Dense** matrices – large, contiguous **arrays** of numbers:

+ **Convenient** and well supported.
- Heavy load on the **memories** of our GPUs, with **time-consuming transfers** taking place between layers of CUDA registers.
Machine learning libraries represent most objects as tensors

\[ M[i, j] \]

- **Dense matrix**: Coefficients only
- **Sparse matrix**: Coordinates + coeffs

**Sparse** matrices – tensors that have **few non-zero entries**:

- Represent **large tensors** with a small memory footprint.
- Outside of **graph** processing, few objects are **sparse enough** to really benefit from this representation.
Machine learning libraries represent most objects as tensors

\[ M[i, j] \]

\[ (i_n, j_n, M_n) \]

\[ F(x_i, y_j) \]

**Dense matrix**
- Coefficients only

**Sparse matrix**
- Coordinates + coeffs

**Symbolic matrix**
- Formula + data

**Distance** and **kernel** matrices, **point** convolutions, **attention** layers:

- **Linear** memory usage: no more memory overflows.
- We can optimize the use of registers for a \( \times 10 - \times 100 \) speed-up vs. a standard PyTorch GPU baseline.
We provide support for this “new abstraction” on the GPU

Our library comes with all the perks of a deep learning toolbox:

+ Transparent array-like interface.
+ Full support for automatic differentiation.
+ Comprehensive collection of tutorials, available online.

Under the hood: combines an optimized C++ engine with high-level binders for PyTorch, NumPy, Matlab and R (thanks to Ghislain Durif).
(We welcome contributors for JAX, Julia and other frameworks!)

To get started:

$\Rightarrow$ pip install pykeops $\Leftarrow$

www.kernel-operations.io
A first example: efficient nearest neighbor search in dimension 50

Create large point clouds using **standard PyTorch syntax**:

```python
import torch
N, M, D = 10**6, 10**6, 50
x = torch.rand(N, 1, D).cuda()  # (1M, 1, 50) array
y = torch.rand(1, M, D).cuda()  # (1, 1M, 50) array
```

Turn **dense** arrays into **symbolic** matrices:

```python
from pykeops.torch import LazyTensor
x_i, y_j = LazyTensor(x), LazyTensor(y)
```

Create a large **symbolic matrix** of squared distances:

```python
D_ij = ((x_i - y_j)**2).sum(dim=2)  # (1M, 1M) symbolic
```

Use an **.argmin()** reduction to perform a nearest neighbor query:

```python
indices_i = D_ij.argmin(dim=1)  # -> standard torch tensor
```
The KeOps library combines performance with flexibility

Script of the previous slide = efficient nearest neighbor query, on par with the bruteforce CUDA scheme of the FAISS library...

And can be used with any metric!

\[
\begin{align*}
D_{ij} &= ((x_i - x_j) ** 2).sum(dim=2) \quad \# \text{Euclidean} \\
M_{ij} &= (x_i - x_j).abs().sum(dim=2) \quad \# \text{Manhattan} \\
C_{ij} &= 1 - (x_i \mid x_j) \quad \# \text{Cosine} \\
H_{ij} &= D_{ij} / (x_i[...,...,0] \ast x_j[...,...,0]) \quad \# \text{Hyperbolic}
\end{align*}
\]

KeOps supports arbitrary formulas and variables with:

- **Reductions**: sum, log-sum-exp, K-min, matrix-vector product, etc.
- **Operations**: $+$, $\times$, sqrt, exp, neural networks, etc.
- **Advanced schemes**: batch processing, block sparsity, etc.
- **Automatic differentiation**: seamless integration with PyTorch.
KeOps lets users work with millions of points at a time

Benchmark of a matrix-vector product with a N-by-N Gaussian kernel matrix between 3D point clouds.

We run NumPy, PyTorch and KeOps on a RTX 2080 Ti GPU.
KeOps lets users experiment freely with advanced methods

KeOps provides a **fast backend for research codes:**

- Interfaces well with **standard libraries:** SciPy, GPytorch, etc.

- Speeds up Gaussian process regression: see e.g. *Kernel methods through the roof: handling billions of points efficiently*, by G. Meanti, L. Carratino, L. Rosasco, A. Rudi (NeurIPS 2020).

- Speeds up **optimal transport** solvers and **point cloud convolutions** by one or two orders of magnitude.

- Much more in the **paper**!
Strengths and limitations of our library

KeOps **symbolic** tensors:

+ Have a negligible **memory** footprint.
+ Provide a sizeable **speed-up** for geometric computations.

– Always rely on **bruteforce** computations.
– Are less interesting when the formula $F(x_i, y_j)$ is **too large**.

Our top priority for **early 2021** is to mitigate these weaknesses: we will add support for **Tensor cores** and standard **approximation strategies** – e.g. using trees or the Nyström method.
Symbolic matrices are to geometric ML what sparse matrices are to graph processing.

We believe that KeOps will stimulate research on:

- **Clustering** methods: fast K-Means and EM iterations.
- Data **representation**: UMAP, fast KNN graphs with any metric.
- **Kernel** methods: kernel matrices.
- **Gaussian** processes: covariance matrices.
- **Geometric** deep learning: point convolutions.
- **Natural language** processing: transformer networks?

We’ll be happy to **discuss** these questions with you!
Documentation and tutorials are available online

www.kernel-operations.io

www.jeanfeydy.com/geometric_data_analysis.pdf