Kymatio: Scattering Transforms in Python

Mathieu Andreux  
Tomás Angles  
Georgios Exarchakis  
Roberto Leonarduzzi  
Gaspar Rochette  
Louis Thiry  
John Zarka  
Stephane Mallat  
Joakim Andén  
Eugene Belilovsky  
Joan Bruna  
Vincent Lostanlen  
Muawiz Chaudhary  
Matthew J. Hirn  
Edouard Oyallon  
Sixin Zhang  
Carmine Cella  
Michael Eickenberg

Abstract

The wavelet scattering transform is an invariant and stable signal representation suitable for many signal processing and machine learning applications. We present the Kymatio software package, an easy-to-use, high-performance Python implementation of the scattering transform in 1D, 2D, and 3D that is compatible with modern deep learning frameworks, including PyTorch and TensorFlow/Keras. The transforms are implemented on both CPUs.


License: CC-BY 4.0, see https://creativecommons.org/licenses/by/4.0/. Attribution requirements are provided at http://jmlr.org/papers/v21/19-047.html.
1. Introduction

Many classification and regression tasks have a degree of invariance to translations and deformations, such as those relating to images, audio recordings, and electronic densities. The scattering transform was introduced in Mallat (2012) to build a signal representation that is invariant to such transformations while preserving as much as possible the information relevant to the task at hand. It is defined as a convolutional network whose filters are fixed to be wavelet and lowpass averaging filters coupled with modulus nonlinearities. It has many favorable theoretical properties (Mallat, 2012; Bruna et al., 2015; Waldspurger, 2017) and enjoys considerable success as a powerful tool in modern signal processing (Adel et al., 2017; Bruna and Mallat, 2013; Andén and Mallat, 2014; Chudáček et al., 2014; Sifre and Mallat, 2013; Eickenberg et al., 2017). It is also effective in combination with modern representation learning approaches (Oyallon et al., 2018; Sainath et al., 2014; Zeghidour et al., 2016).

This article presents Kymatio, a scattering transform implementation that is user-friendly, well-documented, fast, and compatible with existing automatic differentiation libraries. It brings together transforms in 1D, 2D, and 3D under a unified application programming interface (API). The scattering network is also traversed depth-first to reduce memory requirements, enabling efficient processing in limited-memory environments, such as GPUs.

2. Implementing the Scattering Transform

Definition We consider signals defined on a grid of size $N_1 \times \cdots \times N_d$ for $d = 1, 2, 3$. Given two signals $x[n]$ and $y[n]$ on this grid, we denote their periodic convolution by $x \ast y[n]$. The second-order scattering transform is defined using two wavelet filter banks $\{\psi^{(1)}_\lambda[n]\}_{\lambda \in \Lambda_1}$ and $\{\psi^{(2)}_\lambda[n]\}_{\lambda \in \Lambda_2}$, where $\lambda_1$ and $\lambda_2$ are frequency indices in the sets $\Lambda_1$ and $\Lambda_2$. It also includes a lowpass filter $\phi_J[n]$, where the integer $J > 0$ specifies the averaging scale $2^J$ of the filter. Together with a non-linearity $\rho(t)$, these filters define the scattering transform.

The zeroth-order scattering coefficient $S_0 x[n] = x \ast \phi_J[n]$. Convolving $x[n]$ with the first-order wavelet filter bank $\{\psi^{(1)}_\lambda[n]\}_{\lambda \in \Lambda_1}$, applying $\rho(t)$, and convolving with $\phi_J[n]$, we obtain the first-order scattering coefficients

$$S_1 x[n, \lambda_1] = \rho\left( x \ast \psi^{(1)}_{\lambda_1} \right) \ast \phi_J[n], \quad \lambda_1 \in \Lambda_1. \quad (1)$$

The modulus of the first wavelet transform acts as a demodulation, shifting its energy to the low frequencies. However, only some of these frequencies are covered by the low-pass filter $\phi_J$. We recover the remaining frequencies by decomposing $\rho(x \ast \psi^{(1)}_{\lambda_1}[n])$ using the second filter bank, but this is done only for a subset $\Lambda_2(\lambda_1)$ of $\Lambda_2$ since $\rho(x \ast \psi^{(1)}_{\lambda_1}[n])$ is a low-frequency signal. Typically, we have $\Lambda_2(\lambda_1) = \{\lambda \in \Lambda_2, |\lambda| > |\lambda_1|\}$. The result is then passed through $\rho(t)$ and averaged, yielding the second-order coefficients

$$S_2 x[n, \lambda_1, \lambda_2] = \rho\left( \rho\left( x \ast \psi^{(1)}_{\lambda_1} \right) \ast \psi^{(2)}_{\lambda_2} \right) \ast \phi_J[n], \quad \lambda_1 \in \Lambda_1, \lambda_2 \in \Lambda_2(\lambda_1). \quad (2)$$
Kymatio: Scattering Transforms in Python

<table>
<thead>
<tr>
<th>Package</th>
<th>Dimension</th>
<th>GPU</th>
<th>Diff.</th>
<th>Core Devs.</th>
<th>License</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>ScatNet</td>
<td>1D, 2D</td>
<td></td>
<td></td>
<td>5</td>
<td>Apache 2.0</td>
<td>MATLAB</td>
</tr>
<tr>
<td>ScatNetLight</td>
<td>2D</td>
<td></td>
<td></td>
<td>2</td>
<td>GPLv2</td>
<td>MATLAB</td>
</tr>
<tr>
<td>PyScatWave</td>
<td>2D</td>
<td>✓</td>
<td></td>
<td>3</td>
<td>BSD-3</td>
<td>Python</td>
</tr>
<tr>
<td>Scattering,m</td>
<td>1D</td>
<td></td>
<td>✓</td>
<td>1</td>
<td>GPLv3</td>
<td>MATLAB</td>
</tr>
<tr>
<td>PyScatHarm</td>
<td>3D</td>
<td>✓</td>
<td></td>
<td>1</td>
<td>BSD-3</td>
<td>Python</td>
</tr>
<tr>
<td>Wavelet Toolbox</td>
<td>1D</td>
<td></td>
<td>✓</td>
<td>N/A</td>
<td>Proprietary</td>
<td>MATLAB</td>
</tr>
<tr>
<td>Kymatio</td>
<td>1D, 2D, 3D</td>
<td>✓</td>
<td>✓</td>
<td>15</td>
<td>BSD-3</td>
<td>Python</td>
</tr>
</tbody>
</table>

Table 1: Comparison to existing scattering transform packages.

The energy of higher-order scattering coefficients is typically small and does not greatly influence results (Waldspurger, 2017; Bruna and Mallat, 2013; Andén and Mallat, 2014). On the other hand they can be computationally intensive. We have thus chosen to restrict our scope to second-order coefficients, which is what is used in most works.

Implementation  Signals obtained by filtering and applying $\rho(t)$ are low-frequency, so intermediate results are downsampled to reduce computational load as in Andén et al. (2014). In 1D and 2D, we use Morlet wavelets which are close to analytic (i.e., complex-valued with low energy in the negative frequencies) and the non-linearity is the complex modulus $\rho(t) = |t|$ for $t \in \mathbb{C}$ (Andén and Mallat, 2014; Bruna and Mallat, 2013). The 3D transform is calculated using solid harmonic wavelets $\psi_{\lambda} = \psi_{j,\ell,m}$, where $j$ indexes the scale, and $\ell, m$ are the azimuthal and magnetic quantum numbers. In this case the non-linearity $\rho : \mathbb{C}^{2\ell+1} \rightarrow \mathbb{R}$ is defined, with a slight abuse of notation, as $\rho(x \ast \psi_{j,\ell}) = \sqrt{\sum_m |x \ast \psi_{j,\ell,m}|^2}$ (Eickenberg et al., 2017). Following Oyallon et al. (2018), we compute the scattering transform in a depth-first manner, reducing the number of intermediate signals stored at a given time. Since convolutions are all periodic, they may be efficiently calculated using fast Fourier transforms.

3. Project vision

Code quality  Adopting the philosophy of scikit-learn (Pedregosa et al., 2011), the goal of Kymatio is not to maximize the number of features, but to provide a stable and easy-to-use framework. To this end, we make heavy use of unit tests, minimize the number of dependencies, and strive for intuitive interfaces inspired by modern deep learning paradigms. Kymatio also provides an extensive user guide, including an API reference, a tutorial, installation instructions, and easy-to-understand examples, several of which feature real-world applications.

Community and bug tracking  Kymatio is free and open-source software with a 3-clause BSD license. The members of its core development team all have experience implementing scattering transforms in other packages. A key goal of Kymatio is to combine these efforts and foster a community effort in order to produce high-quality software and maintain a critical mass of contributors for its maintenance. The package was released publicly on GitHub November 17th, 2018. The main communication channel is the GitHub page for questions, bug reports, and feature requests. There is also a dedicated Slack channel.
**Relation to previous software** Aside from the emphasis on code quality and usability, *Kymatio* provides several improvements over previous scattering implementations:

- *Python* is the *de facto* standard for data science software, but most existing scattering packages are implemented in MATLAB. In contrast, *Kymatio* provides a completely Pythonic implementation, enabling integration with the scientific Python ecosystem.

- *GPU compatibility* is critical to many data science workloads. *Kymatio* offers an easy-to-use GPU implementation for scattering transforms in 1D, 2D, and 3D.

- *Frontends* are provided for many frameworks, including *NumPy*, *scikit-learn*, *PyTorch*, and *TensorFlow/Keras*, allowing for seamless integrating scattering transforms in a variety of pipelines. In particular, the *PyTorch* and *TensorFlow/Keras* frontends allow for inclusion into many deep learning workflows.

- *Differentiability* of the scattering transform simplifies applications in reconstruction and generative modeling, among others.

Table 1 provides a detailed comparison of existing implementations: ScatNet (Andén et al., 2014), ScatNetLight (Oyallon and Mallat, 2015), PyScatWave (Oyallon et al., 2018), Scattering.m (Lostanlen and Mallat, 2015), PyScatHarm (Eickenberg et al., 2018), and the scattering transform implemented in the MATLAB Wavelet Toolbox.

### 4. User Interface and Documentation

**Interface** The interface is designed to be flexible and consistent across inputs and frontends. Let us consider the *PyTorch* frontend. We first create a scattering object by specifying the averaging scale $J$ and the input signal shape.

```python
from kymatio.torch import Scattering1D, Scattering2D, HarmonicScattering3D

S = Scattering1D(J, shape=(length,))
S = Scattering2D(J, shape=(height, width))
S = HarmonicScattering3D(J, shape=(height, width, depth))
```

The resulting object $S$ acts like a *nn.Module* object in *PyTorch*. The scattering transform $S$ is applied through calls of the form

```python
x = torch.randn((28, 28))
output = S(x)
```

Switching from GPU or CPU functionality also follows the API of *nn.module*.

```python
S.cuda()  # Run on GPU
S.cpu()   # Run on CPU
```

**Documentation and examples** Several examples are provided with the code, illustrating the power of *Kymatio*. These include image reconstruction and generation from scattering (Angles and Mallat, 2018), hybrid scattering and CNN training on CIFAR and MNIST (Oyallon et al., 2018), regression of molecular properties on QM7/QM9 using solid harmonic scattering (Eickenberg et al., 2017), and classifying recordings of spoken digits.
5. Conclusion

*Kymatio* provides a well documented, user-friendly, and fast implementation for the scattering transform. It can be used with the *PyTorch* and *TensorFlow/Keras* deep learning frameworks and supports a variety of applications that have been previously inaccessible to non-experts including hybrid deep learning, generative modeling, and 3D chemistry applications. Future work includes further optimization for speed, flexibility, and backend support.

Acknowledgments

We thank Laurent Sifre, Sergey Zagoruyko and Gabriel Huang for their helpful comments. The project was supported by ERC InvariantClass 320959. EB is funded by a Google Focused Research Award and IVADO. MJH is partially supported by Alfred P. Sloan Fellowship #FG-2016-6607, DARPA Young Faculty Award #D16AP00117, and NSF grant #1620216. The Flatiron Institute is a division of the Simons Foundation.

References


